Cay Horstmann
Big Java
6/e
Early Objects
Includes Java 8 coverage

www.ebook3000.com
Class Declaration
public class CashRegister {
  private int itemCount;
  private double totalPrice;
  
  public void addItem(double price) {
    totalPrice = totalPrice + price;
    itemCount ++;
  }
  
  public double getTotalPrice() {
    return totalPrice;
  }
  
  public int getItemCount() {
    return itemCount;
  }
}

Variable and Constant Declarations
Type Name Initial value
int cansPerPack = 6;

Method Declaration
public static double cubeVolume(double sideLength) {
  return sideLength * sideLength * sideLength;
}

Selected Operators and Their Precedence

Mathematical Operations
- Math.pow(base, exponent)
- Math.sqrt(base)
- Math.abs(base)
- Math.sin(base)
- Math.cos(base)
- Math.tan(base)

Loop Statements
- do { } while (condition)
- for (variable = initial; condition; variable = update) {
  }
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:

**Start objects early, teach object orientation gradually.**
In Chapter 2, students learn how to use objects and classes from the standard library. Chapter 3 shows the mechanics of implementing classes from a given specification. Students then use simple objects as they master branches, loops, and arrays. Object-oriented design starts in Chapter 8. This gradual approach allows students to use objects throughout their study of the core algorithmic topics, without teaching bad habits that must be un-learned later.

**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. “How To” guides help students with common programming tasks. Additional Worked 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
- Thinking Recursively
- 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 automatically-graded programming exercises and skill-oriented multiple-choice questions, are available online.
Preface

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.

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.

Reinforce sound engineering practices. A multitude of useful tips on software quality and common errors encourage the development of good programming habits. The optional testing track focuses on test-driven development, encouraging students to test their programs systematically.

Provide an optional graphics track. Graphical shapes are splendid examples of objects. Many students enjoy writing programs that create drawings or use graphical user interfaces. If desired, these topics can be integrated into the course by using the materials at the end of Chapters 2, 3, and 10.

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 chapter on interfaces and the sections that cover sorting have been updated to make these innovations optionally available. A new chapter covers the Java 8 stream library and its applications for “big data” processing.

In addition, Java 7 features such as the try-with-resources statement are now integrated into the text. Chapter 21 covers the utilities provided by the Paths and Files classes.

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/bjeo6interactivities.

“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

The book can be naturally grouped into four parts, as illustrated by Figure 1 on page vi. The organization of chapters offers the same flexibility as the previous edition; dependencies among the chapters are also shown in the figure.

Part A: Fundamentals (Chapters 1–7)
Chapter 1 contains a brief introduction to computer science and Java programming. Chapter 2 shows how to manipulate objects of predefined classes. In Chapter 3, you will build your own simple classes from given specifications. Fundamental data types, branches, loops, and arrays are covered in Chapters 4–7.
Part B: Object-Oriented Design (Chapters 8–12)
Chapter 8 takes up the subject of class design in a systematic fashion, and it introduces a very simple subset of the UML notation. The discussion of polymorphism and inheritance is split into two chapters. Chapter 9 covers inheritance and polymorphism, whereas Chapter 10 covers interfaces. Exception handling and basic file input/output are covered in Chapter 11. The exception hierarchy gives a useful example for
inheritance. Chapter 12 contains an introduction to object-oriented design, including two significant case studies.

**Part C: Data Structures and Algorithms (Chapters 13–19)**

Chapters 13 through 19 contain an introduction to algorithms and data structures, covering recursion, sorting and searching, linked lists, binary trees, and hash tables. These topics may be outside the scope of a one-semester course, but can be covered as desired after Chapter 7 (see Figure 1). 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. Each data structure is presented in the context of the standard Java collections library. You will learn the essential abstractions of the standard library (such as iterators, sets, and maps) as well as the performance characteristics of the various collections. Chapter 18 introduces Java generics. This chapter is suitable for advanced students who want to implement their own generic classes and methods. Finally, Chapter 19 introduces the Java 8 streams library and shows how it can be used to analyze complex real-world data.

**Part D: Applied Topics (Chapters 20–26)**

Chapters 20 through 26 cover Java programming techniques that definitely go beyond a first course in Java (21–26 are on the book’s companion site). Although, as already mentioned, a comprehensive coverage of the Java library would span many volumes, many instructors prefer that a textbook should give students additional reference material valuable beyond their first course. Some institutions also teach a second-semester course that covers more practical programming aspects such as database and network programming, rather than the more traditional in-depth material on data structures and algorithms. This book can be used in a two-semester course to give students an introduction to programming fundamentals and broad coverage of applications. Alternatively, the material in the final chapters can be useful for student projects. The applied topics include graphical user-interface design, advanced file handling, multithreading, and those technologies that are of particular interest to server-side programming: networking, databases, XML, and web applications. The Internet has made it possible to deploy many useful applications on servers, often accessed by nothing more than a browser. This server-centric approach to application development was in part made possible by the Java language and libraries, and today, much of the industrial use of Java is in server-side programming.

**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 F–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

*Big Java* 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.

*Big Java* 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 *Big Java* adds even more value by integrating a wealth of interactive exercises into the eBook. See [http://wiley.com/go/bjo6interactivities](http://wiley.com/go/bjo6interactivities) 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](http://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](http://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, plus additional example programs.
- Worked Examples that apply the problem-solving steps in the book to other realistic examples.
- 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.

**Pointers in the book**

Pointers in the book describe what students will find on the Web.

**WORKED EXAMPLE 6.3**

A Sample Debugging Session

Learn how to find bugs in an algorithm for counting the syllables of a word. Go to [wiley.com/go/bjo6examples](http://wiley.com/go/bjo6examples) and download Worked Example 6.3.

**FULL CODE EXAMPLE**

Go to [wiley.com/go/bjo6code](http://wiley.com/go/bjo6code) to download a program that demonstrates variables and assignments.
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:

```java
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 6.2):

```java
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 is used when a value runs from a starting point to an ending point with a constant increment or decrement.

**Syntax 6.2**

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

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

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

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

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

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.

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

Memorable photos reinforce analogies and help students remember the concepts.

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.

Memorable photos reinforce analogies and help students remember the concepts.

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 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.

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

### Table 1 Variable Declarations in Java

<table>
<thead>
<tr>
<th>Variable Name</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>int width = 20;</code></td>
<td>Declares an integer variable and initializes it with 20.</td>
</tr>
<tr>
<td><code>int perimeter = 4 * width;</code></td>
<td>The initial value need not be a fixed value. (Of course, <code>width</code> must have been previously declared.)</td>
</tr>
<tr>
<td><code>String greeting = &quot;Hi!&quot;;</code></td>
<td>This variable has the type <code>String</code> and is initialized with the string &quot;Hi.&quot;</td>
</tr>
<tr>
<td><code>int height = 30;</code></td>
<td>Error: The type is missing. This statement is not a declaration but an assignment of a new value to an existing variable—see Section 2.2.5.</td>
</tr>
<tr>
<td><code>int width = &quot;20&quot;;</code></td>
<td>Error: You cannot initialize a number with the string &quot;20.&quot; (Note the quotation marks.)</td>
</tr>
<tr>
<td><code>int width;</code></td>
<td>Error: You cannot initialize a number with the string &quot;20.&quot; (Note the quotation marks.)</td>
</tr>
<tr>
<td><code>int width, height;</code></td>
<td>Declares two integer variables in a single statement. In this book, we will declare each variable in a separate statement.</td>
</tr>
</tbody>
</table>
A class to monitor the growth of an investment that accumulates interest at a fixed annual rate.

```java
public class Investment {
    private double balance;
    private double rate;
    private int year;

    public Investment(double aBalance, double aRate) {
        balance = aBalance;
        rate = aRate;
        year = 0;
    }

    public void accumulateInterest() {
        // Calculate interest
        double interest = balance * rate;
        balance += interest;
        year += 1;
    }

    // Method to reach target balance
    public void reachTargetBalance(double targetBalance) {
        // Loop until target balance is reached
        while (balance < targetBalance) {
            accumulateInterest();
        }
    }
}
```

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).

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

- The initialization is executed once, before the loop is entered.
- The condition is checked before each iteration.
- The update is executed after each iteration.

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.

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

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.

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

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

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.
Many thanks to Bryan Gambrel, Don Fowley, Jenny Welter, Jessy Moor, Jennifer Lartz, Billy Ray, and Tim Lindner at John Wiley & Sons, and Vickie Piercey at Publishing Services 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.


Many thanks to the individuals who reviewed the manuscript for this edition, made valuable suggestions, and brought an embarrassingly large number of errors and omissions to my attention. They include:

- Robin Carr, *Drexel University*
- Gerald Cohen, *The Richard Stockton College of New Jersey*
- Aaron Keen, *California Polytechnic State University, San Luis Obispo*
- Aurelia Smith, *Columbus State University*
- Aakash Taneja, *The Richard Stockton College of New Jersey*
- Craig Tanis, *University of Tennessee at Chattanooga*
- Katherine Winters, *University of Tennessee at Chattanooga*

Every new edition builds on the suggestions and experiences of prior reviewers and users. I am grateful for the invaluable contributions these individuals have made:

- Eric Aaron, *Wesleyan University*
- James Agnew, *Anne Arundel Community College*
- Tim Andersen, *Boise State University*
- Ivan Bajic, *San Diego State University*
- Greg Ballinger, *Miami Dade College*
- Ted Bangay, *Sheridan Institute of Technology*
- Ian Barland, *Radford University*
- George Basham, *Franklin University*
- Jon Beck, *Truman State University*
- Sambit Bhattacharya, *Fayetteville State University*
- Rick Birney, *Arizona State University*
- Paul Bladek, *Edmonds Community College*
- Matt Boutell, *Rose-Hulman Institute of Technology*
- Joseph Bowbeer, *Vizrea Corporation*
- Timothy A. Budd, *Oregon State University*
- John Bundy, *DeVry University Chicago*
- Robert P. Burton, *Brigham Young University*
- Frank Butt, *IBM*
- Jerry Cain, *Stanford University*
- Adam Cannon, *Columbia University*
- Michael Carney, *Finger Lakes Community College*
- Christopher Cassa, *Massachusetts Institute of Technology*
- Nancy Chase, *Gonzaga University*
- Dr. Suchindran S. Chatterjee, *Arizona State University*
- Archana Chidanandan, *Rose-Hulman Institute of Technology*
- Vincent Cicirello, *The Richard Stockton College of New Jersey*
- Teresa Cole, *Boise State University*
- Deborah Coleman, *Rochester Institute of Technology*
- Tina Comston, *Franklin University*
- Lennie Cooper, *Miami Dade College*
- Jose Cordova, *University of Louisiana, Monroe*
- Valentino Crespi, *California State University, Los Angeles*
- Jim Cross, *Auburn University*
- Russell Deaton, *University of Arkansas*
- Geoffrey Decker, *Northern Illinois University*
- H. E. Dunsmore, *Purdue University*
- Robert Duvall, *Duke University*
- Sherif Elsayoumy, *University of North Florida*
- Eman El-Sheikh, *University of West Florida*
- Henry A. Etlinger, *Rochester Institute of Technology*
- John Fendrich, *Bradley University*
- David Freer, *Miami Dade College*
- John Fulton, *Franklin University*
- David Geary, *Sabreware, Inc.*
- Margaret Geroch, *Wheeling Jesuit University*
Acknowledgments

Ahmad Ghaifarian, North Georgia College & State University
Rick Giles, Acadia University
Stacey Grasso, College of San Mateo
Jianchao Han, California State University, Dominguez Hills
Lisa Hansen, Western New England College
Elliott Harold
Eileen Head, Binghamton University
Cecily Heiner, University of Utah
Guy Helmer, Iowa State University
Ed Holdren, Rochester Institute of Technology
Brian Howard, Depauw University
Lubomir Ivanov, Iona College
Norman Jacobson, University of California, Irvine
Steven Janke, Colorado College
Curt Jones, Bloomsburg University
Mark Jones, Lock Haven University of Pennsylvania
Dr. Mustafa Kamal, University of Central Missouri
Mugdha Khaladkar, New Jersey Institute of Technology
Gary J. Koehler, University of Florida
Elliot Koffman, Temple University
Ronald Krawitz, DeVry University
Norm Krumpe, Miami University of Ohio
Jim Leone, Rochester Institute of Technology
Kevin Lillis, St. Ambrose University
Darren Lim, Siena College
Hong Lin, DeVry University
Kathy Liszka, University of Akron
Hunter Lloyd, Montana State University
Youmin Lu, Bloomsburg University
Kuber Maharanj, Purdue University, College of Technology at Columbus
John S. Mallozzi, Iona College
John Martin, North Dakota State University
Jeanna Matthews, Clarkson University
Patricia McDermott-Wells, Florida International University
Scott McElfresh, Carnegie Mellon University
Joan McGrory, Christian Brothers University
Carolyn Miller, North Carolina State University
Sandep R. Mitra, State University of New York, Brockport
Teng Moh, San Jose State University
Bill Mongan, Drexel University
John Moore, The Citadel
Jose-Arturo Mora-Soto, Jesica Rivero-Espinosa, and Julio-Angel Cano-Romero, University of Madrid
Faye Navabi, Arizona State University
Parviz Partow-Navid, California State University, Los Angeles
George Novacky, University of Pittsburgh
Kevin O’Gorman, California Polytechnic State University, San Luis Obispo
Michael Olan, Richard Stockton College
Mimi Opkins, California State University Long Beach
Derek Pao, City University of Hong Kong
Kevin Parker, Idaho State University
Jim Perry, Ulster County Community College
Cornel Pokorny, California Polytechnic State University, San Luis Obispo
Roger Priebke, University of Texas, Austin
C. Robert Putnam, California State University, Northridge
Kai Qian, Southern Polytechnic State University
Cyndi Rader, Colorado School of Mines
Neil Rankin, Worcester Polytechnic Institute
Brad Rippe, Fullerton College
Pedro I. Rivera Vega, University of Puerto Rico, Mayaguez
Daniel Rogers, SUNY Brockport
Chaman Lal Sabharwal, Missouri University of Science and Technology
Katherine Salch, Illinois Central College
John Santore, Bridgewater State College
Javad Shakib, DeVry University
Carolyn Schauble, Colorado State University
Brent Seales, University of Kentucky
Christian Shin, SUNY Geneseo
Charlie Shu, Franklin University
Jeffrey Six, University of Delaware
Don Slater, Carnegie Mellon University
Ken Slonneger, University of Iowa
Donald Smith, Columbia College
Joslyn A. Smith, Florida International University
Stephanie Smullen, University of Tennessee, Chattanooga
Robert Strader, Stephen F. Austin State University
Monica Sweat, Georgia Institute of Technology
Peter Stanchev, Kettering University
Shannon Tauro, University of California, Irvine
Ron Taylor, Wright State University
Russell Tessier, University of Massachusetts, Amherst
Jonathan L. Tolstedt, North Dakota State University
David Vineyard, Kettering University
Joseph Vybihal, McGill University
Xiaoming Wei, Iona College
Jonathan S. Weissman, Finger Lakes Community College
Todd Whittaker, Franklin University
Robert Willhoft, Roberts Wesleyan College
Lea Wittie, Bucknell University
David Womack, University of Texas at San Antonio
David Woolbright, Columbus State University
Tom Wulf, University of Cincinnati
Catherine Wyman, DeVry University
Arthur Yanushka, Christian Brothers University
Qi Yu, Rochester Institute of Technology
Salih Yurttas, Texas A&M University
CONTENTS

PREFACE  iii
SPECIAL FEATURES  xxiv

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
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

2  USING OBJECTS  31
2.1  Objects and Classes  32
   Using Objects  32
   Classes  33
2.2  Variables  34
   Variable Declarations  34
   Types  36
   Names  37
   Comments  38
   Assignment  38
2.3  Calling Methods  41
   The Public Interface of a Class  41
   Method Arguments  42
   Return Values  43
   Method Declarations  45
2.4  Constructing Objects  46
2.5  Accessor and Mutator Methods  48
2.6  The API Documentation  50
   Browsing the API Documentation  50
   Packages  52
2.7  Implementing a Test Program  53
   ST1  Testing Classes in an Interactive Environment  54
   WE1  How Many Days Have You Been Alive?  55
   WE2  Working with Pictures  55
2.8  Object References  55
2.9  Graphical Applications  59
   Frame Windows  59
   Drawing on a Component  60
   Displaying a Component in a Frame  63
2.10  Ellipses, Lines, Text, and Color  64
   Ellipses and Circles  64
   Lines  65
   Drawing Text  65
   Colors  66

3  IMPLEMENTING CLASSES  79
3.1  Instance Variables and Encapsulation  80
   Instance Variables  80
   The Methods of the Counter Class  82
   Encapsulation  82
3.2  Specifying the Public Interface of a Class  84
   Specifying Methods  84
   Specifying Constructors  85
   Using the Public Interface  87
   Commenting the Public Interface  87
3.3  Providing the Class Implementation  91
   Providing Instance Variables  91
   Providing Constructors  92
   Providing Methods  93
   HT1  Implementing a Class  96
   WE1  Making a Simple Menu  97
3.4  Unit Testing  100
3.5 PROBLEM SOLVING Tracing Objects 103
3.6 Local Variables 105
3.7 The this Reference 107
ST1 Calling One Constructor from Another 110
3.8 Shape Classes 110
HT2 Drawing Graphical Shapes 114

4 FUNDAMENTAL DATA TYPES 129
4.1 Numbers 130
Number Types 130
Constants 132
ST1 Big Numbers 136
4.2 Arithmetic 137
Arithmetic Operators 137
Increment and Decrement 138
Integer Division and Remainder 138
Powers and Roots 139
Converting Floating-Point Numbers to Integers 140
J81 Avoiding Negative Remainders 143
ST2 Combining Assignment and Arithmetic 143
ST3 Instance Methods and Static Methods 143
4.3 Input and Output 145
Reading Input 145
Formatted Output 146
HT1 Carrying Out Computations 149
WE1 Computing the Volume and Surface Area of a Pyramid
4.4 PROBLEM SOLVING First Do it By Hand 152
WE2 Computing Travel Time
4.5 Strings 154
The String Type 154
Concatenation 155
String Input 155
Escape Sequences 156
Strings and Characters 156
Substrings 157
ST4 Using Dialog Boxes for Input and Output 160

5 DECISIONS 177
5.1 The if Statement 178
ST1 The Conditional Operator 182
5.2 Comparing Values 183
Relational Operators 184
Comparing Floating-Point Numbers 185
Comparing Strings 186
Comparing Objects 187
Testing for null 187
HT1 Implementing an if Statement 190
WE1 Extracting the Middle
5.3 Multiple Alternatives 193
ST2 The switch Statement 196
5.4 Nested Branches 196
ST3 Block Scope 201
ST4 Enumeration Types 203
5.5 PROBLEM SOLVING Flowcharts 203
5.6 PROBLEM SOLVING Selecting Test Cases 206
ST5 Logging 208
5.7 Boolean Variables and Operators 209
ST6 Short-Circuit Evaluation of Boolean Operators 213
ST7 De Morgan's Law 213
5.8 APPLICATION Input Validation 214

6 LOOPS 237
6.1 The while Loop 238
6.2 PROBLEM SOLVING Hand-Tracing 245
6.3 The for Loop 250
ST1 Variables Declared in a for Loop Header 257
6.4 The do Loop 258
6.5 APPLICATION Processing Sentinel Values 259
ST2 Redirection of Input and Output 262
ST3 The “Loop and a Half” Problem 262
ST4 The break and continue Statements 263
6.6 PROBLEM SOLVING Storyboards 265
6.7 Common Loop Algorithms 268
Sum and Average Value 268
Counting Matches 268
7 ARRAYS AND ARRAY LISTS 307

7.1 Arrays 308
Declaring and Using Arrays 308
Array References 311
Using Arrays with Methods 312
Partially Filled Arrays 312
ST1 Methods with a Variable Number of Arguments 315

7.2 The Enhanced for Loop 317

7.3 Common Array Algorithms 318
Filling 318
Sum and Average Value 319
Maximum and Minimum 319
Element Separators 319
Linear Search 320
Removing an Element 320
Inserting an Element 321
Swapping Elements 322
Copying Arrays 323
Reading Input 324
ST2 Sorting with the Java Library 327

7.4 PROBLEM SOLVING Adapting Algorithms 327
HT1 Working with Arrays 330
WE1 Rolling the Dice

7.5 PROBLEM SOLVING Discovering Algorithms by Manipulating Physical Objects 332

7.6 Two-Dimensional Arrays 336
Declaring Two-Dimensional Arrays 336
Accessing Elements 337
Locating Neighboring Elements 338
Accessing Rows and Columns 338
WE2 A World Population Table
ST3 Two-Dimensional Arrays with Variable Row Lengths 341
ST4 Multidimensional Arrays 343

7.7 Array Lists 343
Declaring and Using Array Lists 344
Using the Enhanced for Loop with Array Lists 345
Copying Array Lists 346
Wrappers and Auto-boxing 347
Using Array Algorithms with Array Lists 348
Storing Input Values in an Array List 348
Removing Matches 348
Choosing Between Array Lists and Arrays 349
ST5 The Diamond Syntax 352

7.8 Regression Testing 352

8 DESIGNING CLASSES 375

8.1 Discovering Classes 376

8.2 Designing Good Methods 377
Providing a Cohesive Public Interface 377
Minimizing Dependencies 378
Separating Accessors and Mutators 379
Minimizing Side Effects 380
ST1 Call by Value and Call by Reference 382

8.3 PROBLEM SOLVING Patterns for Object Data 386
Keeping a Total 386
Counting Events 387
Collecting Values 387
Managing Properties of an Object 388
Modeling Objects with Distinct States 388
Describing the Position of an Object 389

8.4 Static Variables and Methods 391
ST2 Alternative Forms of Instance and Static Variable Initialization 394
ST3 Static Imports 395

8.5 PROBLEM SOLVING Solve a Simpler Problem First 395
8.6 Packages 400
Organizing Related Classes into Packages 400
Importing Packages 401
Package Names 401
Packages and Source Files 402
ST4 Package Access 403
HT1 Programming with Packages 404

8.7 Unit Test Frameworks 405

9 INHERITANCE 423

9.1 Inheritance Hierarchies 424
9.2 Implementing Subclasses 428
9.3 Overriding Methods 433
ST1 Calling the Superclass Constructor 438
9.4 Polymorphism 439
ST2 Dynamic Method Lookup and the Implicit Parameter 442
ST3 Abstract Classes 443
ST4 Final Methods and Classes 444
ST5 Protected Access 444
HT1 Developing an Inheritance Hierarchy 445
WE1 Implementing an Employee Hierarchy for Payroll Processing 445

9.5 Object: The Cosmic Superclass 450
Overriding the toString Method 450
The equals Method 452
The instanceof Operator 453
ST6 Inheritance and the toString Method 455
ST7 Inheritance and the equals Method 456

10 INTERFACES 465

10.1 Using Interfaces for Algorithm Reuse 466
Discovering an Interface Type 466
Declaring an Interface Type 467
Implementing an Interface Type 469
Comparing Interfaces and Inheritance 471
ST1 Constants in Interfaces 473
J81 Static Methods in Interfaces 473
J82 Default Methods 473
J83 Conflicting Default Methods 474

10.2 Working with Interface Variables 475
Converting from Classes to Interfaces 475
Invoking Methods on Interface Variables 476
Casting from Interfaces to Classes 476
WE1 Investigating Number Sequences 477

10.3 The Comparable Interface 477
ST2 The clone Method and the Cloneable Interface 479

10.4 Using Interfaces for Callbacks 482
J84 Lambda Expressions 485
ST3 Generic Interface Types 486

10.5 Inner Classes 487
ST4 Anonymous Classes 488

10.6 Mock Objects 489

10.7 Event Handling 490
Listening to Events 491
Using Inner Classes for Listeners 493
J85 Lambda Expressions for Event Handling 496

10.8 Building Applications with Buttons 496

10.9 Processing Timer Events 499

10.10 Mouse Events 502
ST5 Keyboard Events 506
ST6 Event Adapters 506

11 INPUT/OUTPUT AND EXCEPTION HANDLING 519

11.1 Reading and Writing Text Files 520
ST1 Reading Web Pages 523
ST2 File Dialog Boxes 523
ST3 Character Encodings 524

11.2 Text Input and Output 525
Reading Words 525
Reading Characters 526
Classifying Characters 526
Reading Lines 527
Scanning a String 528
Converting Strings to Numbers 528
Avoiding Errors When Reading Numbers 529
Mixing Number, Word, and Line Input 529
Formatting Output 530
ST4 Regular Expressions 532
ST5 Reading an Entire File 533
## 11.3 Command Line Arguments 533

**HT1** Processing Text Files 536

**WE1** Analyzing Baby Names © Alex Slobodkin/iStockphoto.

## 11.4 Exception Handling 540

- Throwing Exceptions 540
- Catching Exceptions 542
- Checked Exceptions 543
- Closing Resources 545
- Designing Your Own Exception Types 546

**ST6** Assertions 549

**ST7** The try/finally Statement 549

## 11.5 APPLICATION Handling Input Errors 549

## 12 OBJECT-ORIENTED DESIGN 565

### 12.1 Classes and Their Responsibilities 566

- Discovering Classes 566
- The CRC Card Method 567

### 12.2 Relationships Between Classes 569

- Dependency 569
- Aggregation 570
- Inheritance 571

**HT1** Using CRC Cards and UML Diagrams in Program Design 572

**ST1** Attributes and Methods in UML Diagrams 573

**ST2** Multiplicities 574

**ST3** Aggregation, Association, and Composition 574

### 12.3 APPLICATION Printing an Invoice 575

- Requirements 575
- CRC Cards 576
- UML Diagrams 578
- Method Documentation 579
- Implementation 581

**WE1** Simulating an Automatic Teller Machine © Alex Slobodkin/iStockphoto.

## 13 RECURSION 593

### 13.1 Triangle Numbers 594

**HT1** Thinking Recursively 599

**WE1** Finding Files © Alex Slobodkin/iStockphoto.

### 13.2 Recursive Helper Methods 602

### 13.3 The Efficiency of Recursion 604

### 13.4 Permutations 609

### 13.5 Mutual Recursion 614

### 13.6 Backtracking 620

**WE2** Towers of Hanoi © Alex Slobodkin/iStockphoto.

## 14 SORTING AND SEARCHING 635

### 14.1 Selection Sort 636

### 14.2 Profiling the Selection Sort Algorithm 639

### 14.3 Analyzing the Performance of the Selection Sort Algorithm 642

**ST1** Oh, Omega, and Theta 644

**ST2** Insertion Sort 645

### 14.4 Merge Sort 647

### 14.5 Analyzing the Merge Sort Algorithm 650

**ST3** The Quicksort Algorithm 652

### 14.6 Searching 654

- Linear Search 654
- Binary Search 655

### 14.7 PROBLEM SOLVING Estimating the Running Time of an Algorithm 659

- Linear Time 659
- Quadratic Time 660
- The Triangle Pattern 661

- Logarithmic Time 662

### 14.8 Sorting and Searching in the Java Library 664

- Sorting 664
- Binary Search 664
- Comparing Objects 665

**ST4** The Comparator Interface 666

**J81** Comparators with Lambda Expressions 667

**WE1** Enhancing the Insertion Sort Algorithm

## 15 THE JAVA COLLECTIONS FRAMEWORK 677

### 15.1 An Overview of the Collections Framework 678
16 BASIC DATA STRUCTURES 721

16.1 Implementing Linked Lists 722
   The Node Class 722
   Adding and Removing the First Element 723
   The Iterator Class 724
   Advancing an Iterator 725
   Removing an Element 726
   Adding an Element 728
   Setting an Element to a Different Value 729
   Efficiency of Linked List Operations 729
   ST1 Static Classes 736
   WE1 Implementing a Doubly-Linked List 737

16.2 Implementing Array Lists 737
   Getting and Setting Elements 737
   Removing or Adding Elements 738
   Growing the Internal Array 739

16.3 Implementing Stacks and Queues 741
   Stacks as Linked Lists 741
   Stacks as Arrays 743
   Queues as Linked Lists 743
   Queues as Circular Arrays 744

16.4 Implementing a Hash Table 747
   Hash Codes 747
   Hash Tables 747
   Finding an Element 749
   Adding and Removing Elements 749
   Iterating over a Hash Table 750
   ST2 Open Addressing 755

17 TREE STRUCTURES 765

17.1 Basic Tree Concepts 766
17.2 Binary Trees 770
   Binary Tree Examples 770
   Balanced Trees 772
   A Binary Tree Implementation 773
   WE1 Building a Huffman Tree 775
17.3 Binary Search Trees 775
   The Binary Search Property 775
   Insertion 776
   Removal 778
   Efficiency of the Operations 780
17.4 Tree Traversal 784
   Inorder Traversal 784
   Preorder and Postorder Traversals 785
   The Visitor Pattern 786
   Depth-First and Breadth-First Search 787
   Tree Iterators 789
17.5 Red-Black Trees 790
   Basic Properties of Red-Black Trees 790
   Insertion 792
   Removal 793
   WE2 Implementing a Red-Black Tree 795
17.6 Heaps 797
17.7 The Heapsort Algorithm 808

18 GENERIC CLASSES 823

18.1 Generic Classes and Type Parameters 824
18.2 Implementing Generic Types 825
18.3 Generic Methods 829
18.4 Constraining Type Parameters 831
  ST1 Wildcard Types 834
18.5 Type Erasure 835
  ST2 Reflection 838
  WE1 Making a Generic Binary Search Tree Class

19 STREAM PROCESSING 845

19.1 The Stream Concept 846
19.2 Producing Streams 848
19.3 Collecting Results 850
  ST1 Infinite Streams 851
19.4 Transforming Streams 852
19.5 Lambda Expressions 855
  ST2 Method and Constructor References 857
  ST3 Higher-Order Functions 858
  ST4 Higher-Order Functions and Comparators 859
19.6 The Optional Type 859
19.7 Other Terminal Operations 862
19.8 Primitive-Type Streams 863
  Creating Primitive-Type Streams 864
  Mapping a Primitive-Type Stream 864
  Processing Primitive-Type Streams 864
19.9 Grouping Results 866
19.10 Common Algorithms Revisited 868
  Filling 868
  Sum, Average, Maximum, and Minimum 869
  Counting Matches 869
  Element Separators 869
  Linear Search 870
  Comparing Adjacent Values 870
  HT1 Working with Streams 871
  WE1 Word Properties
  WE2 A Movie Database

20 GRAPHICAL USER INTERFACES 883

20.1 Layout Management 884
  Using Layout Managers 884

21 ADVANCED INPUT/OUTPUT (WEB ONLY) 907

21.1 Readers, Writers, and Input/Output Streams
21.2 Binary Input and Output
21.3 Random Access
21.4 Object Input and Output Streams
  HT1 Choosing a File Format
21.5 File and Directory Operations
  Paths
  Creating and Deleting Files and Directories
  Useful File Operations
  Visiting Directories

22 MULTITHREADING (WEB ONLY) 911

22.1 Running Threads
  ST1 Thread Pools
22.2 Terminating Threads
22.3 Race Conditions
22.4 Synchronizing Object Access
22.5 Avoiding Deadlocks
  ST2 Object Locks and Synchronized Methods
  ST3 The Java Memory Model
22.6 APPLICATION Algorithm Animation
## 23 INTERNET NETWORKING (WEB ONLY)

- 23.1 The Internet Protocol
- 23.2 Application Level Protocols
- 23.3 A Client Program
- 23.4 A Server Program
- 23.5 URL Connections

## 24 RELATIONAL DATABASES (WEB ONLY)

- 24.1 Organizing Database Information
  - Database Tables
  - Linking Tables
  - Implementing Multi-Valued Relationships
  - Primary Keys and Indexes
- 24.2 Queries
  - Simple Queries
  - Selecting Columns
  - Selecting Subsets
  - Calculations
  - Joins
  - Updating and Deleting Data
- 24.3 Installing a Database
- 24.4 Database Programming in Java
  - Connecting to the Database
  - Executing SQL Statements
  - Analyzing Query Results
  - Result Set Metadata
- 24.5 APPLICATION Entering an Invoice
  - Transactions
  - Primary-Relational Mapping
  - Programming a Bank Database

## 25 XML (WEB ONLY)

- 25.1 XML Tags and Documents
  - Advantages of XML
  - Differences Between XML and HTML
  - The Structure of an XML Document
  - Designing an XML Document Format
- 25.2 Parsing XML Documents
- 25.3 Creating XML Documents
  - Writing an XML Document
  - Grammars, Parsers, and Compilers
- 25.4 Validating XML Documents
  - Document Type Definitions
  - Specifying a DTD in an XML Document
  - Parsing and Validation
  - Writing a DTD
  - Schema Languages
  - Other XML Technologies

## 26 WEB APPLICATIONS (WEB ONLY)

- 26.1 The Architecture of a Web Application
- 26.2 The Architecture of a JSF Application
  - JSF Pages
  - Managed Beans
  - Separation of Presentation and Business Logic
  - Deploying a JSF Application
  - Session State and Cookies
- 26.3 JavaBeans Components
- 26.4 Navigation Between Pages
  - Designing a Managed Bean
- 26.5 JSF Components
- 26.6 APPLICATION A Three-Tier Application
  - AJAX

**Appendices**

- 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 JAVA LANGUAGE CODING GUIDELINES A-39
- APPENDIX F TOOL SUMMARY
- APPENDIX G NUMBER SYSTEMS
- APPENDIX H UML SUMMARY
- APPENDIX I JAVA SYNTAX SUMMARY
- APPENDIX J HTML SUMMARY

**Glossary** G-1

**Index** I-1

**Credits** C-1
**ALPHABETICAL LIST OF SYNTAX BOXES**

<table>
<thead>
<tr>
<th>Syntax Box</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Arrays</td>
<td>309</td>
</tr>
<tr>
<td>Array Lists</td>
<td>343</td>
</tr>
<tr>
<td>Assignment</td>
<td>39</td>
</tr>
<tr>
<td>Calling a Superclass Method</td>
<td>433</td>
</tr>
<tr>
<td>Cast</td>
<td>141</td>
</tr>
<tr>
<td>Catching Exceptions</td>
<td>542</td>
</tr>
<tr>
<td>Class Declaration</td>
<td>87</td>
</tr>
<tr>
<td>Comparisons</td>
<td>184</td>
</tr>
<tr>
<td>Constant Declaration</td>
<td>134</td>
</tr>
<tr>
<td>Constructor with Superclass Initializer</td>
<td>438</td>
</tr>
<tr>
<td>Declaring a Generic Class</td>
<td>826</td>
</tr>
<tr>
<td>Declaring a Generic Method</td>
<td>830</td>
</tr>
<tr>
<td>Declaring an Interface</td>
<td>468</td>
</tr>
<tr>
<td>for Statement</td>
<td>250</td>
</tr>
<tr>
<td>if Statement</td>
<td>180</td>
</tr>
<tr>
<td>Implementing an Interface</td>
<td>469</td>
</tr>
<tr>
<td>Importing a Class from a Package</td>
<td>52</td>
</tr>
<tr>
<td>Input Statement</td>
<td>145</td>
</tr>
<tr>
<td>Instance Variable Declaration</td>
<td>81</td>
</tr>
<tr>
<td>Java Program</td>
<td>12</td>
</tr>
<tr>
<td>Lambda Expressions</td>
<td>855</td>
</tr>
<tr>
<td>Object Construction</td>
<td>47</td>
</tr>
<tr>
<td>Package Specification</td>
<td>401</td>
</tr>
<tr>
<td>Subclass Declaration</td>
<td>430</td>
</tr>
<tr>
<td>The Enhanced for Loop</td>
<td>318</td>
</tr>
<tr>
<td>The instanceof Operator</td>
<td>453</td>
</tr>
<tr>
<td>The throws Clause</td>
<td>545</td>
</tr>
<tr>
<td>The try-with-resources Statement</td>
<td>545</td>
</tr>
<tr>
<td>Throwing an Exception</td>
<td>540</td>
</tr>
<tr>
<td>Two-Dimensional Array Declaration</td>
<td>337</td>
</tr>
<tr>
<td>while Statement</td>
<td>239</td>
</tr>
<tr>
<td>Variable Declaration</td>
<td>35</td>
</tr>
</tbody>
</table>
# CHAPTER 1 Introduction

<table>
<thead>
<tr>
<th>Common Errors</th>
<th>How Tos and Worked Examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>Omitting Semicolons</td>
<td>Describing an Algorithm</td>
</tr>
<tr>
<td></td>
<td>with Pseudocode</td>
</tr>
<tr>
<td>Misspelling Words</td>
<td>Writing an Algorithm</td>
</tr>
<tr>
<td></td>
<td>for Tiling a Floor</td>
</tr>
</tbody>
</table>

# CHAPTER 2 Using Objects

<table>
<thead>
<tr>
<th>Common Errors</th>
<th>How Tos and Worked Examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>Using Undeclared or Uninitialized Variables</td>
<td>How Many Days Have You</td>
</tr>
<tr>
<td></td>
<td>Been Alive?</td>
</tr>
<tr>
<td>Confusing Variable Declarations and</td>
<td>Working with Pictures</td>
</tr>
<tr>
<td>Assignment Statements</td>
<td></td>
</tr>
<tr>
<td>Trying to Invoke a Constructor Like a Method</td>
<td></td>
</tr>
</tbody>
</table>

# CHAPTER 3 Implementing Classes

<table>
<thead>
<tr>
<th>Common Errors</th>
<th>How Tos and Worked Examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>Declaring a Constructor as void</td>
<td>Implementing a Class</td>
</tr>
<tr>
<td>Ignoring Parameter Variables</td>
<td>Making a Simple Menu</td>
</tr>
<tr>
<td>Duplicating Instance Variables in Local</td>
<td>Drawing Graphical Shapes</td>
</tr>
<tr>
<td>Variables</td>
<td></td>
</tr>
<tr>
<td>Providing Unnecessary Instance Variables</td>
<td></td>
</tr>
<tr>
<td>Forgetting to Initialize Object References</td>
<td></td>
</tr>
<tr>
<td>in a Constructor</td>
<td></td>
</tr>
</tbody>
</table>

# CHAPTER 4 Fundamental Data Types

<table>
<thead>
<tr>
<th>Common Errors</th>
<th>How Tos and Worked Examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unintended Integer Division</td>
<td>Carrying out Computations</td>
</tr>
<tr>
<td>Unbalanced Parentheses</td>
<td>Computing the Volume and</td>
</tr>
<tr>
<td></td>
<td>Surface Area of a Pyramid</td>
</tr>
<tr>
<td></td>
<td>Computing Travel Time</td>
</tr>
</tbody>
</table>

# CHAPTER 5 Decisions

<table>
<thead>
<tr>
<th>Common Errors</th>
<th>How Tos and Worked Examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>A Semicolon After the if Condition</td>
<td>Implementing an if Statement</td>
</tr>
<tr>
<td>Using == to Compare Strings</td>
<td>Extracting the Middle</td>
</tr>
<tr>
<td>The Dangling else Problem</td>
<td></td>
</tr>
<tr>
<td>Combining Multiple Relational Operators</td>
<td></td>
</tr>
<tr>
<td>Confusing &amp;&amp; and</td>
<td></td>
</tr>
</tbody>
</table>

# CHAPTER 6 Loops

<table>
<thead>
<tr>
<th>Common Errors</th>
<th>How Tos and Worked Examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>Don’t Think “Are We There Yet?”</td>
<td>Writing a Loop</td>
</tr>
<tr>
<td>Infinite Loops</td>
<td>Credit Card Processing</td>
</tr>
<tr>
<td>Off-by-One Errors</td>
<td>Manipulating the Pixels</td>
</tr>
<tr>
<td></td>
<td>in an Image</td>
</tr>
<tr>
<td></td>
<td>Debugging</td>
</tr>
<tr>
<td></td>
<td>A Sample Debugging Session</td>
</tr>
</tbody>
</table>

Available online at www.wiley.com/college/horstmann.
<table>
<thead>
<tr>
<th>Programming Tips</th>
<th>Special Topics and Java 8 Notes</th>
<th>Computing &amp; Society</th>
</tr>
</thead>
<tbody>
<tr>
<td>Backup Copies</td>
<td>10</td>
<td>Computers Are Everywhere 5</td>
</tr>
<tr>
<td>Choose Descriptive Variable Names</td>
<td>41</td>
<td>Testing Classes in an Interactive Environment 54</td>
</tr>
<tr>
<td>Learn By Trying</td>
<td>45</td>
<td>Computer Monopoly 58</td>
</tr>
<tr>
<td>Don't Memorize—Use Online Help</td>
<td>53</td>
<td></td>
</tr>
<tr>
<td>The javadoc Utility</td>
<td>90</td>
<td>Calling One Constructor from Another 110</td>
</tr>
<tr>
<td>Do Not Use Magic Numbers</td>
<td>137</td>
<td>Big Numbers 136</td>
</tr>
<tr>
<td>Spaces in Expressions</td>
<td>143</td>
<td>Avoiding Negative Remainders 143</td>
</tr>
<tr>
<td>Reading Exception Reports</td>
<td>160</td>
<td>Combining Assignment and Arithmetic 143</td>
</tr>
<tr>
<td>Brace Layout</td>
<td>181</td>
<td>Instance Methods and Static Methods 143</td>
</tr>
<tr>
<td>Always Use Braces</td>
<td>181</td>
<td>Using Dialog Boxes for Input and Output 160</td>
</tr>
<tr>
<td>Tabs</td>
<td>182</td>
<td></td>
</tr>
<tr>
<td>Avoid Duplication in Branches</td>
<td>183</td>
<td></td>
</tr>
<tr>
<td>Hand-Tracing</td>
<td>200</td>
<td></td>
</tr>
<tr>
<td>Make a Schedule and Make Time for Unexpected Problems</td>
<td>208</td>
<td></td>
</tr>
<tr>
<td>Use for Loops for Their Intended Purpose Only</td>
<td>255</td>
<td>Variables Declared in a for Loop Header 257</td>
</tr>
<tr>
<td>Choose Loop Bounds That Match Your Task</td>
<td>256</td>
<td>Redirection of Input and Output 262</td>
</tr>
<tr>
<td>Count Iterations</td>
<td>256</td>
<td>The Loop-and-a-Half Problem 262</td>
</tr>
<tr>
<td>Flowcharts for Loops</td>
<td>259</td>
<td>The break and continue Statements 263</td>
</tr>
</tbody>
</table>

Available online at www.wiley.com/college/horstmann.
# 7 Arrays and Array Lists

<table>
<thead>
<tr>
<th>Common Errors</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bounds Errors</td>
<td>314</td>
</tr>
<tr>
<td>Uninitialized and Unfilled Arrays</td>
<td>314</td>
</tr>
<tr>
<td>Underestimating the Size of a Data Set</td>
<td>327</td>
</tr>
<tr>
<td>Length and Size</td>
<td>352</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>How Tos and Worked Examples</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Working with Arrays</td>
<td>330</td>
</tr>
<tr>
<td>Rolling the Dice</td>
<td></td>
</tr>
<tr>
<td>A World Population Table</td>
<td></td>
</tr>
</tbody>
</table>

# 8 Designing Classes

<table>
<thead>
<tr>
<th>Common Errors</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Trying to Access Instance Variables in Static Methods</td>
<td>394</td>
</tr>
<tr>
<td>Confusing Dots</td>
<td>403</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>How Tos and Worked Examples</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Programming with Packages</td>
<td>404</td>
</tr>
</tbody>
</table>

# 9 Inheritance

<table>
<thead>
<tr>
<th>Common Errors</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Replicating Instance Variables from the Superclass</td>
<td>432</td>
</tr>
<tr>
<td>Confusing Super- and Subclasses</td>
<td>432</td>
</tr>
<tr>
<td>Accidental Overloading</td>
<td>437</td>
</tr>
<tr>
<td>Forgetting to Use super When Invoking a Superclass Method</td>
<td>437</td>
</tr>
<tr>
<td>Don’t Use Type Tests</td>
<td>454</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>How Tos and Worked Examples</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Developing an Inheritance Hierarchy</td>
<td>445</td>
</tr>
<tr>
<td>Implementing an Employee Hierarchy for Payroll Processing</td>
<td></td>
</tr>
</tbody>
</table>

# 10 Interfaces

<table>
<thead>
<tr>
<th>Common Errors</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Forgetting to Declare Implementing Methods as Public</td>
<td>472</td>
</tr>
<tr>
<td>Trying to Instantiate an Interface</td>
<td>472</td>
</tr>
<tr>
<td>Modifying Parameter Types in the Implementing Method</td>
<td>495</td>
</tr>
<tr>
<td>Trying to Call Listener Methods</td>
<td>495</td>
</tr>
<tr>
<td>Forgetting to Attach a Listener</td>
<td>498</td>
</tr>
<tr>
<td>Forgetting to Repaint</td>
<td>502</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>How Tos and Worked Examples</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Investigating Number Sequences</td>
<td></td>
</tr>
</tbody>
</table>

# 11 Input/Output and Exception Handling

<table>
<thead>
<tr>
<th>Common Errors</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Backslashes in File Names</td>
<td>523</td>
</tr>
<tr>
<td>Constructing a Scanner with a String</td>
<td>523</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>How Tos and Worked Examples</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Processing Text Files</td>
<td>536</td>
</tr>
<tr>
<td>Analyzing Baby Names</td>
<td></td>
</tr>
</tbody>
</table>

Available online at www.wiley.com/college/horstmann.
<table>
<thead>
<tr>
<th>Programming Tips</th>
<th>Special Topics and Java 8 Notes</th>
<th>Computing &amp; Society</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Use Arrays for Sequences of Related Items</strong></td>
<td>Methods with a Variable Number of Arguments</td>
<td><strong>Computer Viruses</strong> 316</td>
</tr>
<tr>
<td><strong>Make Parallel Arrays into Arrays of Objects</strong></td>
<td>Sorting with the Java Library</td>
<td>The Therac-25 Incidents 355</td>
</tr>
<tr>
<td><strong>Batch Files and Shell Scripts</strong></td>
<td>Two-Dimensional Arrays with Variable Row Lengths</td>
<td><strong>Consistency</strong> 381</td>
</tr>
<tr>
<td></td>
<td>Multidimensional Arrays</td>
<td><strong>Minimize the Use of Static Methods</strong> 393</td>
</tr>
<tr>
<td></td>
<td>The Diamond Syntax</td>
<td><strong>Call by Value and Call by Reference</strong> 382</td>
</tr>
<tr>
<td></td>
<td></td>
<td><strong>Alternative Forms of Instance and Static Variable Initialization</strong> 394</td>
</tr>
<tr>
<td></td>
<td></td>
<td><strong>Static Imports</strong> 395</td>
</tr>
<tr>
<td></td>
<td></td>
<td><strong>Package Access</strong> 403</td>
</tr>
<tr>
<td><strong>Consistency</strong> 381</td>
<td><strong>Use a Single Class for Variation in Values, Inheritance for Variation in Behavior</strong> 428</td>
<td><strong>Who Controls the Internet?</strong> 456</td>
</tr>
<tr>
<td><strong>Minimize the Use of Static Methods</strong> 393</td>
<td><strong>Calling the Superclass Constructor</strong> 438</td>
<td></td>
</tr>
<tr>
<td></td>
<td><strong>Dynamic Method Lookup and the Implicit Parameter</strong> 442</td>
<td></td>
</tr>
<tr>
<td></td>
<td><strong>Abstract Classes</strong> 443</td>
<td></td>
</tr>
<tr>
<td></td>
<td><strong>Final Methods and Classes</strong> 444</td>
<td></td>
</tr>
<tr>
<td></td>
<td><strong>Protected Access</strong> 444</td>
<td></td>
</tr>
<tr>
<td></td>
<td><strong>Inheritance and the toString Method</strong> 455</td>
<td></td>
</tr>
<tr>
<td></td>
<td><strong>Inheritance and the equals Method</strong> 456</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td><strong>Comparing Integers and Floating-Point Numbers</strong> 478</td>
</tr>
<tr>
<td></td>
<td></td>
<td><strong>Don't Use a Container as a Listener</strong> 499</td>
</tr>
<tr>
<td></td>
<td><strong>Constants in Interfaces</strong> 473</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td><strong>Static Methods in Interfaces</strong> 473</td>
</tr>
<tr>
<td></td>
<td></td>
<td><strong>Default Methods</strong> 473</td>
</tr>
<tr>
<td></td>
<td></td>
<td><strong>Conflicting Default Methods</strong> 474</td>
</tr>
<tr>
<td></td>
<td></td>
<td>The clone Method and the Cloneable Interface 479</td>
</tr>
<tr>
<td></td>
<td></td>
<td><strong>Lambda Expressions</strong> 485</td>
</tr>
<tr>
<td></td>
<td></td>
<td><strong>Generic Interface Types</strong> 486</td>
</tr>
<tr>
<td></td>
<td></td>
<td><strong>Anonymous Classes</strong> 488</td>
</tr>
<tr>
<td></td>
<td></td>
<td><strong>Lambda Expressions for Event Handling</strong> 496</td>
</tr>
<tr>
<td></td>
<td></td>
<td><strong>Keyboard Events</strong> 506</td>
</tr>
<tr>
<td></td>
<td></td>
<td><strong>Event Adapters</strong> 506</td>
</tr>
<tr>
<td></td>
<td></td>
<td><strong>Throw Early, Catch Late</strong> 548</td>
</tr>
<tr>
<td></td>
<td></td>
<td><strong>Do Not Squelch Exceptions</strong> 548</td>
</tr>
<tr>
<td></td>
<td></td>
<td><strong>Do Throw Specific Exceptions</strong> 548</td>
</tr>
<tr>
<td></td>
<td><strong>Reading Web Pages</strong> 523</td>
<td><strong>Encryption Algorithms</strong> 539</td>
</tr>
<tr>
<td></td>
<td><strong>File Dialog Boxes</strong> 523</td>
<td>The Ariane Rocket Incident 554</td>
</tr>
<tr>
<td></td>
<td><strong>Character Encodings</strong> 524</td>
<td></td>
</tr>
<tr>
<td></td>
<td><strong>Regular Expressions</strong> 532</td>
<td></td>
</tr>
<tr>
<td></td>
<td><strong>Reading an Entire File</strong> 533</td>
<td></td>
</tr>
<tr>
<td></td>
<td><strong>Assertions</strong> 549</td>
<td></td>
</tr>
<tr>
<td></td>
<td><strong>The try/finally Statement</strong> 549</td>
<td></td>
</tr>
<tr>
<td>Chapter</td>
<td>Common Errors</td>
<td>How Tos and Worked Examples</td>
</tr>
<tr>
<td>---------</td>
<td>---------------</td>
<td>----------------------------</td>
</tr>
<tr>
<td><strong>12</strong> Object-Oriented Design</td>
<td>Using CRC Cards and UML Diagrams in Program Design 572</td>
<td>Simulating an Automatic Teller Machine</td>
</tr>
<tr>
<td><strong>13</strong> Recursion</td>
<td>Infinite Recursion 598</td>
<td>Thinking Recursively 599</td>
</tr>
<tr>
<td></td>
<td>Tracing Through Recursive Methods 598</td>
<td>Finding Files</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Towers of Hanoi</td>
</tr>
<tr>
<td><strong>14</strong> Sorting and Searching</td>
<td>The compareTo Method Can Return Any Integer, Not Just –1, 0, and 1 666</td>
<td>Enhancing the Insertion Sort Algorithm</td>
</tr>
<tr>
<td><strong>15</strong> The Java Collections Framework</td>
<td>Choosing a Collection 694</td>
<td>Word Frequency</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Simulating a Queue of Waiting Customers</td>
</tr>
<tr>
<td><strong>16</strong> Basic Data Structures</td>
<td>Implementing a Doubly-Linked List</td>
<td></td>
</tr>
<tr>
<td><strong>17</strong> Tree Structures</td>
<td>Building a Huffman Tree</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Implementing a Red-Black Tree</td>
<td></td>
</tr>
<tr>
<td><strong>18</strong> Generic Classes</td>
<td>Genericity and Inheritance 833</td>
<td>Making a Generic Binary Search Tree Class</td>
</tr>
<tr>
<td></td>
<td>The Array Store Exception 833</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Using Generic Types in a Static Context 838</td>
<td></td>
</tr>
<tr>
<td><strong>19</strong> Stream Processing</td>
<td>Don't Use a Terminated Stream 854</td>
<td>Working with Streams 871</td>
</tr>
<tr>
<td></td>
<td>Optional Results Without Values 861</td>
<td>Word Properties</td>
</tr>
<tr>
<td></td>
<td>Don't Apply Mutations in Parallel Stream Operations 863</td>
<td>A Movie Database</td>
</tr>
<tr>
<td><strong>20</strong> Graphical User Interfaces</td>
<td>By Default, Components Have Zero Width and Height 887</td>
<td>Laying Out a User Interface 901</td>
</tr>
<tr>
<td></td>
<td>Programming a Working Calculator</td>
<td></td>
</tr>
<tr>
<td>Programming Tips</td>
<td>Special Topics and Java 8 Notes</td>
<td>Computing &amp; Society</td>
</tr>
<tr>
<td>------------------</td>
<td>---------------------------------</td>
<td>---------------------</td>
</tr>
<tr>
<td>Attributes and Methods in UML Diagrams</td>
<td>Databases and Privacy 586</td>
<td></td>
</tr>
<tr>
<td>Multiplicities</td>
<td>The Limits of Computation 612</td>
<td></td>
</tr>
<tr>
<td>Aggregation, Association, and Composition</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Oh, Omega, and Theta</td>
<td>The First Programmer 658</td>
<td></td>
</tr>
<tr>
<td>Insertion Sort</td>
<td></td>
<td></td>
</tr>
<tr>
<td>The Quicksort Algorithm</td>
<td></td>
<td></td>
</tr>
<tr>
<td>The Comparator Interface</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Comparators with Lambda Expressions</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Use Interface References to Manipulate Data Structures 691</td>
<td>Updating Map Entries 694</td>
<td></td>
</tr>
<tr>
<td>Hash Functions</td>
<td>Standardization 686</td>
<td></td>
</tr>
<tr>
<td>Reverse Polish Notation</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Static Classes</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Open Addressing</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Wildcard Types</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Reflection</td>
<td></td>
<td></td>
</tr>
<tr>
<td>One Stream Operation Per Line Keep Lambda Expressions Short 851 856</td>
<td>Infinite Streams 851</td>
<td></td>
</tr>
<tr>
<td>Method and Constructor References</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Higher-Order Functions</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Higher-Order Functions and Comparators</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Use a GUI Builder 904</td>
<td>Adding the main Method to the Frame Class 888</td>
<td></td>
</tr>
<tr>
<td>Chapter</td>
<td>Title</td>
<td>Common Errors</td>
</tr>
<tr>
<td>------------</td>
<td>--------------------------------------------</td>
<td>--------------------------------------------</td>
</tr>
<tr>
<td>21</td>
<td>Advanced Input/Output (WEB ONLY)</td>
<td>Negative byte Values</td>
</tr>
<tr>
<td>22</td>
<td>Multithreading (WEB ONLY)</td>
<td>Calling await Without Calling signalAll</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Calling signalAll Without Locking the Object</td>
</tr>
<tr>
<td>23</td>
<td>Internet Networking (WEB ONLY)</td>
<td></td>
</tr>
<tr>
<td>24</td>
<td>Relational Databases (WEB ONLY)</td>
<td>Joining Tables Without Specifying a Link Condition</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Constructing Queries from Arbitrary Strings</td>
</tr>
<tr>
<td>25</td>
<td>XML (WEB ONLY)</td>
<td>XML Elements Describe Objects, Not Classes</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>26</td>
<td>Web Applications (WEB ONLY)</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Programming Tips</th>
<th>Special Topics and Java 8 Notes</th>
<th>Computing &amp; Society</th>
</tr>
</thead>
<tbody>
<tr>
<td>Use the Runnable Interface</td>
<td>Thread Pools</td>
<td></td>
</tr>
<tr>
<td>Check for Thread Interruptions in the <code>run</code> Method of a Thread</td>
<td>Object Locks and Synchronized Methods</td>
<td>The Java Memory Model</td>
</tr>
<tr>
<td>Use High-Level Libraries</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Stick with the Standard</td>
<td>Primary Keys and Indexes</td>
<td></td>
</tr>
<tr>
<td>Avoid Unnecessary Data Replication</td>
<td>Transactions</td>
<td></td>
</tr>
<tr>
<td>Don’t Replicate Columns in a Table</td>
<td>Object-Relational Mapping</td>
<td></td>
</tr>
<tr>
<td>Don’t Hardwire Database Connection Parameters into Your Program</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Let the Database Do the Work</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Prefer XML Elements over Attributes</td>
<td>Grammars, Parsers, and Compilers</td>
<td></td>
</tr>
<tr>
<td>Avoid Children with Mixed Elements and Text</td>
<td>Schema Languages</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Other XML Technologies</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Session State and Cookies</td>
<td></td>
</tr>
<tr>
<td></td>
<td>AJAX</td>
<td></td>
</tr>
</tbody>
</table>
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

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

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
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

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.

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) or a solid-state drive, provides slower and less expensive storage that persists without

Figure 1 Central Processing Unit

Figure 2 A Hard Disk
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 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.
1.3 The Java Programming Language

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.

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

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

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

1.4 Becoming Familiar with Your Programming Environment

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.
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:

```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.
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).

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
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.

---

**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:

- **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.
1.5 Analyzing Your First Program

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

```
section_5/HelloPrinter.java

public class HelloPrinter
{
    public static void main(String[] args)
    {
        // Display a greeting in the console window
        System.out.println("Hello, World!");
    }
}
```

The line
```
public class HelloPrinter
```
indicates the declaration of a class called HelloPrinter.

Every Java program consists of one or more classes. We will discuss classes in more detail in Chapters 2 and 3.

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 class ClassName
{
    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 3 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 11. At this time, simply consider
```
public class ClassName
{
    public static void main(String[] args)
    {
        ...
    }
}
```
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.
Every Java program contains at least one class. Choose a class name that describes the program action.

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

Each statement ends in a semicolon. See page 13.

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!");
    }
}

Every Java program contains a main method with this header.
The statements inside the main method are executed when the program runs.

Be sure to match the opening and closing braces.

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!"

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 main, 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.

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

```java
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

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

is the single line

```
007
```

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?

```java
System.out.println("Hello");
```

14. What does the following set of statements print?

```java
System.out.print("My lucky number is");
System.out.println(3 + 4 + 5);
```

15. What do the following statements print?

```java
System.out.println("Hello");
System.out.println("\n");
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.

---

**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

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

a single statement, as if you had written

```java
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

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

```java
System.out.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

```java
Hello, Word!
```

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

```java
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—
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.

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 print line 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.

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:

```java
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

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.
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:

<table>
<thead>
<tr>
<th>year</th>
<th>interest</th>
<th>balance</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td>$10,000.00</td>
</tr>
<tr>
<td>1</td>
<td>$10,000.00 x 0.05 = 500.00</td>
<td>$10,000.00 + 500.00 = $10,500.00</td>
</tr>
<tr>
<td>2</td>
<td>$10,500.00 x 0.05 = 525.00</td>
<td>$10,500.00 + 525.00 = $11,025.00</td>
</tr>
<tr>
<td>3</td>
<td>$11,025.00 x 0.05 = 551.25</td>
<td>$11,025.00 + 551.25 = $11,576.25</td>
</tr>
<tr>
<td>4</td>
<td>$11,576.25 x 0.05 = 578.81</td>
<td>$11,576.25 + 578.81 = $12,155.06</td>
</tr>
</tbody>
</table>

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:

Start with a year value of 0, a column for the interest, and a balance of $10,000.

<table>
<thead>
<tr>
<th>year</th>
<th>interest</th>
<th>balance</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td>$10,000.00</td>
</tr>
</tbody>
</table>

Repeat the following steps while the balance is less than $20,000

Add 1 to the year value.

Compute the interest as balance x 0.05 (i.e., 5 percent interest).

Add the interest to the balance.

<table>
<thead>
<tr>
<th>year</th>
<th>interest</th>
<th>balance</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td>$10,000.00</td>
</tr>
<tr>
<td>1</td>
<td>$500.00</td>
<td>$10,500.00</td>
</tr>
<tr>
<td>14</td>
<td>$942.82</td>
<td>$19,799.32</td>
</tr>
<tr>
<td>15</td>
<td>$989.96</td>
<td>$20,789.28</td>
</tr>
</tbody>
</table>

Report the final year value 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.
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:
  
  \[
  \text{total cost} = \text{purchase price} + \text{operating cost} \\
  \text{Multiply the balance value by 1.05.} \\
  \text{Remove the first and last character from the word.}
  \]

- Describe decisions and repetitions as follows:
  
  \[
  \text{If total cost 1 < total cost 2} \\
  \text{While the balance is less than $20,000} \\
  \text{For each picture in the sequence}
  \]

  Use indentation to indicate which statements should be selected or repeated:
  
  \[
  \text{For each car} \\
  \text{operating cost} = 10 \times \text{annual fuel cost} \\
  \text{total cost} = \text{purchase price} + \text{operating cost}
  \]

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

- Indicate results with statements such as:
  
  \[
  \text{Choose car1.} \\
  \text{Report the final year value as the answer.}
  \]

### 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
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:

   Pick the first photo and call it "the best so far".
   For each photo in the sequence
      If it is more attractive than the "best so far"
         Discard "the best so far".
         Call this photo "the best so far".
   The photo called "the best so far" is 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 ○●○●? 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?
Step 1  Determine the inputs and outputs.

In our sample problem, we have these inputs:
- **purchase price**\textsubscript{1} and **fuel efficiency**\textsubscript{1}, the price and fuel efficiency (in mpg) of the first car
- **purchase price**\textsubscript{2} and **fuel efficiency**\textsubscript{2}, 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 \(10 \times \text{annual fuel cost}\).

The annual fuel cost is \(\text{price per gallon \times annual fuel consumed}\).

The annual fuel consumed is \(\text{annual miles driven} / \text{fuel efficiency}\). 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

\[
\text{total cost} = \text{purchase price} + \text{operating cost}
\]

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** \(\times\) **annual fuel consumed**
- **operating cost** = \(10 \times \text{annual fuel cost}\)
- **total cost** = **purchase price** + **operating cost**

If total cost\textsubscript{1} < total cost\textsubscript{2}
  Choose car\textsubscript{1}.
Else
  Choose car\textsubscript{2}.

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:

\[
\text{annual fuel consumed} = \frac{\text{annual miles driven}}{\text{fuel efficiency}} = \frac{15000}{50} = 300
\]

\[
\text{annual fuel cost} = \text{price per gallon} \times \text{annual fuel consumed} = 4 \times 300 = 1200
\]

\[
\text{operating cost} = 10 \times \text{annual fuel cost} = 10 \times 1200 = 12000
\]

\[
\text{total cost} = \text{purchase price} + \text{operating cost} = 25000 + 12000 = 37000
\]

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/bjeo6examples. 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 (length × 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 \( \frac{\text{width}}{4} \) tiles.

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 the following steps:
Repeat this step \( \frac{\text{width}}{4} - 1 \) 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.

**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.
• 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?

```java
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.

```java
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?

```java
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.

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, $1 + 2 + \cdots + 10$.

E1.3 Write a program that prints the product of the first ten positive integers, $1 \times 2 \times \cdots \times 10$. (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!.”

```java
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.

```java
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.

```java
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.

<table>
<thead>
<tr>
<th>State</th>
<th>Sales Tax Rate</th>
</tr>
</thead>
<tbody>
<tr>
<td>Alaska</td>
<td>0%</td>
</tr>
<tr>
<td>Hawaii</td>
<td>4%</td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
</tbody>
</table>

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.

<table>
<thead>
<tr>
<th>List of Phrases to Translate</th>
</tr>
</thead>
<tbody>
<tr>
<td>Good morning.</td>
</tr>
<tr>
<td>It is a pleasure to meet you.</td>
</tr>
<tr>
<td>Please call me tomorrow.</td>
</tr>
<tr>
<td>Have a nice day!</td>
</tr>
</tbody>
</table>
Chapter 1  Introduction

**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 $\pi$ can be computed according to the following formula:

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

Write an algorithm to compute $\pi$. 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 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.
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.

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.

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).

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.
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. No. The compiler would look for an item whose name is Hello. You need to enclose Hello in quotation marks:

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

13. This is a compile-time error. The compiler will complain that it does not know the meanings of the words Hello and World.

14. The printout is My lucky number is12. It would be a good idea to add a space after the is.

15. a blank line

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 println.

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.

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 it 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. Pick the first photo and call it "the most expensive so far". For each photo in the sequence
    If it is more expensive than "the most expensive so far"
      Discard "the most expensive so far".
      Call this photo "the most expensive so far".
The photo called "the most expensive so far" is the most expensive photo in the sequence.

25. The first black marble that is preceded by a white one is marked in blue:

```
  O O O O O
```

Switching the two yields

```
  O O O O O
```

The next black marble to be switched is

```
  O O O O O
```
yielding

```
  O O O O O
```
The next steps are

```
  O O O O O
  O O O O O
  O O O O O
  O O O O O
```
Now the sequence is sorted.

26. The sequence doesn’t terminate. Consider the input O O O O O. The first two marbles keep getting switched.
CHAPTER 2
USING OBJECTS

CHAPTER GOALS

To learn about variables
To understand the concepts of classes and objects
To be able to call methods
To learn about arguments and return values
To be able to browse the API documentation
To implement test programs
To understand the difference between objects and object references
To write programs that display simple shapes

CHAPTER CONTENTS

2.1 OBJECTS AND CLASSES 32
2.2 VARIABLES 34
   SYN Variable Declaration 35
   SYN Assignment 39
   CE1 Using Undeclared or Uninitialized Variables 40
   CE2 Confusing Variable Declarations and Assignment Statements 40
   PT1 Choose Descriptive Variable Names 41
2.3 CALLING METHODS 41
   PT2 Learn By Trying 45
2.4 CONSTRUCTING OBJECTS 46
   SYN Object Construction 47
   CE3 Trying to Invoke a Constructor Like a Method 48
2.5 ACCESSOR AND MUTATOR METHODS 48

2.6 THE API DOCUMENTATION 50
   SYN Importing a Class from a Package 52
   PT3 Don’t Memorize—Use Online Help 53
2.7 IMPLEMENTING A TEST PROGRAM 53
   ST1 Testing Classes in an Interactive Environment 54
   WE1 How Many Days Have You Been Alive? 54
   WE2 Working with Pictures
2.8 OBJECT REFERENCES 55
   C&S Computer Monopoly 58
2.9 GRAPHICAL APPLICATIONS 59
2.10 ELLIPSES, LINES, TEXT, AND COLOR 64
Most useful programs don’t just manipulate numbers and strings. Instead, they deal with data items that are more complex and that more closely represent entities in the real world. Examples of these data items include bank accounts, employee records, and graphical shapes.

The Java language is ideally suited for designing and manipulating such data items, or objects. In Java, you implement classes that describe the behavior of these objects. In this chapter, you will learn how to manipulate objects that belong to classes that have already been implemented. This will prepare you for the next chapter, in which you will learn how to implement your own classes.

2.1 Objects and Classes

When you write a computer program, you put it together from certain “building blocks”. In Java, you build programs from objects. Each object has a particular behavior, and you can manipulate it to achieve certain effects.

As an analogy, think of a home builder who constructs a house from certain parts: doors, windows, walls, pipes, a furnace, a water heater, and so on. Each of these elements has a particular function, and they work together to fulfill a common purpose. Note that the home builder is not concerned with how to build a window or a water heater. These elements are readily available, and the builder’s job is to integrate them into the house.

Of course, computer programs are more abstract than houses, and the objects that make up a computer program aren’t as tangible as a window or a water heater. But the analogy holds well: A programmer produces a working program from elements with the desired functionality—the objects. In this chapter, you will learn the basics about using objects written by other programmers.

2.1.1 Using Objects

An object is an entity that you can manipulate by calling one or more of its methods. A method consists of a sequence of instructions that can access the internal data of an object. When you call the method, you do not know exactly what those instructions are, or even how the object is organized internally. However, the behavior of the method is well defined, and that is what matters to us when we use it.
2.1 Objects and Classes

33

2.1.1 Objects

Figure 1  Representation of the System.out Object

For example, you saw in Chapter 1 that System.out refers to an object. You manipulate it by calling the println method. When the println method is called, some activities occur inside the object, and the ultimate effect is that text appears in the console window. You don’t know how that happens, and that’s OK. What matters is that the method carries out the work that you requested.

Figure 1 shows a representation of the System.out object. The internal data is symbolized by a sequence of zeroes and ones. Think of each method (symbolized by the gears) as a piece of machinery that carries out its assigned task.

In general, think of an object as an entity that can do work for you when you call its methods. How the work is done is not important to the programmer using the object.

In the remainder of this chapter, you will see other objects and the methods that they can carry out.

A method is a sequence of instructions that accesses the data of an object.

A class describes a set of objects with the same behavior.

2.1.2 Classes

In Chapter 1, you encountered two objects:

• System.out
• "Hello, World!"

Each of these objects belongs to a different class. The System.out object belongs to the PrintStream class. The "Hello, World!" object belongs to the String class. Of course, there are many more String objects, such as "Goodbye" or "Mississippi". They all have something in common—you can invoke the same methods on all strings. You will see some of these methods in Section 2.3.

As you will see in Chapter 11, you can construct objects of the PrintStream class other than System.out. Those objects write data to files or other destinations instead of the console. Still, all PrintStream objects share common behavior. You can invoke the println and print methods on any PrintStream object, and the printed values are sent to their destination.
Of course, the objects of the PrintStream class have a completely different behavior than the objects of the String class. You could not call println on a String object. A string wouldn’t know how to send itself to a console window or file.

As you can see, different classes have different responsibilities. A string knows about the letters that it contains, but it does not know how to display them to a human or to save them to a file.

1. In Java, objects are grouped into classes according to their behavior. Would a window object and a water heater object belong to the same class or to different classes? Why?

2. Some light bulbs use a glowing filament, others use a fluorescent gas. If you consider a light bulb a Java object with an “illuminate” method, would you need to know which kind of bulb it is?

3. What actually happens when you try to call the following?
   "Hello, World".println(System.out)

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

2.2 Variables

Before we continue with the main topic of this chapter—the behavior of objects—we need to go over some basic programming terminology. In the following sections, you will learn about the concepts of variables, types, and assignment.

2.2.1 Variable Declarations

When your program manipulates objects, you will want to store the objects and the values that their methods return, so that you can use them later. In a Java program, you use variables to store values. The following statement declares a variable named width:

```java
int width = 20;
```
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 “J053”), and it can hold a vehicle. A variable has a name (such as `width`), and it can hold a value (such as 20). 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:

```java
int width = 20;
```

The variable `width` is initialized with the value 20.

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 `width` variable is an **integer**, a whole number without a fractional part. In Java, this type is called `int`.

Note that the type comes before the variable name:

```java
int width = 20;
```

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

```java
int width = 20;
System.out.println(width);
int area = width * width;
```

Table 1 shows several examples of variable declarations.
### Table 1 Variable Declarations in Java

<table>
<thead>
<tr>
<th>Variable Name</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>int width = 20;</td>
<td>Declares an integer variable and initializes it with 20.</td>
</tr>
<tr>
<td>int perimeter = 4 * width;</td>
<td>The initial value need not be a fixed value. (Of course, width must have been previously declared.)</td>
</tr>
<tr>
<td>String greeting = &quot;Hi&quot;;</td>
<td>This variable has the type String and is initialized with the string “Hi”.</td>
</tr>
<tr>
<td>height = 30;</td>
<td>Error: The type is missing. This statement is not a declaration but an assignment of a new value to an existing variable—see Section 2.2.5.</td>
</tr>
<tr>
<td>int width = &quot;20&quot;;</td>
<td>Error: You cannot initialize a number with the string “20”. (Note the quotation marks.)</td>
</tr>
<tr>
<td>int width;</td>
<td>Declares an integer variable without initializing it. This can be a cause for errors—see Common Error 2.1 on page 40.</td>
</tr>
<tr>
<td>int width, height;</td>
<td>Declares two integer variables in a single statement. In this book, we will declare each variable in a separate statement.</td>
</tr>
</tbody>
</table>

### 2.2.2 Types

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, suppose you count the number of cars in a parking lot. The counter must be an integer number—you cannot have a fraction of a car.

When a fractional part is required (such as in the number 22.5), we use *floating-point numbers*. The most commonly used type for floating-point numbers in Java is called `double`. Here is the declaration of a floating-point variable:

```java
double milesPerGallon = 22.5;
```

You can combine numbers with the `+` and `-` operators, as in `width + 10` or `width - 1`. To multiply two numbers, use the `*` operator. For example, `2 × width` is written as `2 * width`. Use the `/` operator for division, such as `width / 2`.

As in mathematics, the `*` and `/` operator bind more strongly than the `+` and `-` operators. That is, `width + height * 2` means the sum of `width` and the product `height * 2`. If you want to multiply the sum by 2, use parentheses: `(width + height) * 2`.

Not all types are number types. For example, the value "Hello" has the type `String`. You need to specify that type when you define a variable that holds a string:

```java
String greeting = "Hello";
```

A type specifies the operations that can be carried out with its values.

Types are important because they indicate what you can do with a variable. For example, consider the variable `width`. It’s type is `int`. Therefore, you can multiply the value that it holds with another number. But the type of `greeting` is `String`. You can’t multiply a string with another number. (You will see in Section 2.3.1 what you can do with strings.)
2.2 Variables

2.2.3 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 `milesPerGallon`, than a terse name, such as `mpg`.

In Java, there are a few simple rules for the names of variables, methods, and classes:

1. 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 `milesPerGallon`. 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. Names are case sensitive, that is, `milesPerGallon` and `milespergallon` 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.)

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

Table 2 shows examples of legal and illegal variable names in Java.

<table>
<thead>
<tr>
<th>Variable Name</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>distance_1</td>
<td>Names consist of letters, numbers, and the underscore character.</td>
</tr>
<tr>
<td>x</td>
<td>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 41).</td>
</tr>
<tr>
<td>CanVolume</td>
<td><strong>Caution:</strong> 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.</td>
</tr>
<tr>
<td>6pack</td>
<td><strong>Error:</strong> Names cannot start with a number.</td>
</tr>
<tr>
<td>can volume</td>
<td><strong>Error:</strong> Names cannot contain spaces.</td>
</tr>
<tr>
<td>double</td>
<td><strong>Error:</strong> You cannot use a reserved word as a name.</td>
</tr>
<tr>
<td>miles/gal</td>
<td><strong>Error:</strong> You cannot use symbols such as / in names.</td>
</tr>
</tbody>
</table>
2.2.4 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 to initialize a variable:

```java
double milesPerGallon = 35.5; // The average fuel efficiency of new U.S. cars in 2013
```

This comment explains the significance of the value 35.5 to a human reader. The compiler does not process comments at all. It ignores everything from a // delimiter to the end of the line.

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,

```java
/*
   In most countries, fuel efficiency is measured in liters per hundred kilometer. Perhaps that is more useful—it tells you how much gas you need to purchase to drive a given distance. Here is the conversion formula.
*/
double fuelEfficiency = 235.214583 / milesPerGallon;
```

2.2.5 Assignment

You can change the value of a variable with the assignment operator (=). For example, consider the variable declaration

```java
int width = 10;  
```

If you want to change the value of the variable, simply assign the new value:

```java
width = 20;  
```

The assignment replaces the original value of the variable (see Figure 2).

![Figure 2](image1.png)

**Figure 2**
Assigning a New Value to a Variable

It is an error to use a variable that has never had a value assigned to it. For example, the following assignment statement has an error:

```java
int height;
int width = height;  // ERROR—uninitialized variable height
```

The compiler will complain about an “uninitialized variable” when you use a variable that has never been assigned a value. (See Figure 3.)

![Figure 3](image2.png)

**Figure 3**
An Uninitialized Variable


### Syntax 2.2 Assignment

**Syntax**

```
variableName = value;
```

- **This is a variable declaration.**
  - `double width = 20;`
  - `width = 30;`
- **This is an assignment statement.**
  - `width = width + 10;`

The value of this variable is changed.

The new value of the variable

The same name can occur on both sides. See Figure 4.

---

The remedy is to assign a value to the variable before you use it:

```
int height = 20;
int width = height; // OK
```

The right-hand side of the `=` symbol can be a mathematical expression. For example,

```
width = height + 10;
```

This means “compute the value of `height + 10` and store that value in the variable `width`”.

In the Java programming language, the `=` operator denotes an action, namely to replace the value of a variable. This usage differs from the mathematical usage of the `=` symbol as a statement about equality. For example, in Java, the following statement is entirely legal:

```
width = width + 10;
```

This means “compute the value of `width + 10` and store that value in the variable `width`” (see Figure 4).

In Java, it is not a problem that the variable `width` is used on both sides of the `=` symbol. Of course, in mathematics, the equation `width = width + 10` has no solution.

---

**Figure 4**

Executing the Statement `width = width + 10`

1. **Compute the value of the right-hand side**
   - `width = 30`
   - `width + 10 = 40`

2. **Store the value in the variable**
   - `width = 40`
4. What is wrong with the following variable declaration?
   ```java
   int miles per gallon = 39.4
   ```

5. Declare and initialize two variables, `unitPrice` and `quantity`, to contain the unit price of a single item and the number of items purchased. Use reasonable initial values.

6. Use the variables declared in Self Check 5 to display the total purchase price.

7. What are the types of the values 0 and "0"?

8. Which number type would you use for storing the area of a circle?

9. Which of the following are legal identifiers?
   - Greeting1
   - g
   - void
   - 101dalmatians
   - Hello, World
   - <greeting>

10. Declare a variable to hold your name. Use camel case in the variable name.

11. Is `12 = 12` a valid expression in the Java language?

12. How do you change the value of the `greeting` variable to "Hello, Nina!"?

13. How would you explain assignment using the parking space analogy?

**Practice It**  Now you can try these exercises at the end of the chapter: R2.4, R2.5, R2.7.

---

**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:

```java
int perimeter = 4 * width; // ERROR: width not yet declared
int width = 20;
```

In your program, the statements are compiled in order. When the compiler reaches the first statement, it does not know that `width` 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:

```java
int width;
int perimeter = 4 * width; // ERROR: width 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.

---

**Confusing Variable Declarations and Assignment Statements**

Suppose your program declares a variable as follows:

```java
int width = 20;
```

If you want to change the value of the variable, you use an assignment statement:

```java
width = 30;
```

It is a common error to accidentally use another variable declaration:

```java
int width = 30; // ERROR—starts with int and is therefore a declaration
```
2.3 Calling Methods

A program performs useful work by calling methods on its objects. In this section, we examine how to supply values in a method, and how to obtain the result of the method.

2.3.1 The Public Interface of a Class

You use an object by calling its methods. All objects of a given class share a common set of methods. For example, the PrintStream class provides methods for its objects (such as println and print). Similarly, the String class provides methods that you can apply to String objects. One of them is the length method. The length method counts the number of characters in a string. You can apply that method to any object of type String. For example, the sequence of statements:

```java
String greeting = "Hello, World!";
int numberOfCharacters = greeting.length();
```

sets numberOfCharacters to the length of the String object "Hello, World!". After the instructions in the length method are executed, numberOfCharacters is set to 13. (The quotation marks are not part of the string, and the length method does not count them.)

When calling the length method, you do not supply any values inside the parentheses. Also note that the length method does not produce any visible output. It returns a value that is subsequently used in the program.

Let’s look at another method of the String class. When you apply the toUpperCase method to a String object, the method creates another String object that contains the characters of the original string, with lowercase letters converted to uppercase. For example, the sequence of statements

```java
String river = "Mississippi";
String bigRiver = river.toUpperCase();
```

sets bigRiver to the String object "MISSISSIPPI".

Choose Descriptive Variable Names

In algebra, variable names are usually just one letter long, such as \( p \) or \( A \), maybe with a subscript such as \( p_1 \). You might be tempted to save yourself a lot of typing by using short variable names in your Java programs:

```java
int a = w * h;
```

Compare that statement with the following one:

```java
int area = width * height;
```

The advantage is obvious. Reading width is much easier than reading \( w \) and then figuring out that it must mean “width”.

In practical programming, descriptive variable names are particularly important when programs are written by more than one person. It may be obvious to you that \( w \) stands for width, but is it obvious to the person who needs to update your code years later? For that matter, will you yourself remember what \( w \) means when you look at the code a month from now?

Programming Tip 2.1

© Eric Isselé/iStockphoto.
The String class declares many other methods besides the length and toUpperCase methods—you will learn about many of them in Chapter 4. Collectively, the methods form the public interface of the class, telling you what you can do with the objects of the class. A class also declares a private implementation, describing the data inside its objects and the instructions for its methods. Those details are hidden from the programmers who use objects and call methods.

Figure 5 shows two objects of the String class. Each object stores its own data (drawn as boxes that contain characters). Both objects support the same set of methods—the public interface that is specified by the String class.

2.3.2 Method Arguments

Most methods require values that give details about the work that the method needs to do. For example, when you call the println method, you must supply the string that should be printed. Computer scientists use the technical term argument for method inputs. We say that the string greeting is an argument of the method call

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

Figure 6 illustrates passing the argument to the method.
At this tailor shop, the customer's measurements and the fabric are the arguments of the sew method. The return value is the finished garment.

Some methods require multiple arguments; others don’t require any arguments at all. An example of the latter is the length method of the String class (see Figure 7). All the information that the length method requires to do its job—namely, the character sequence of the string—is stored in the object that carries out the method.

2.3.3 Return Values

Some methods, such as the println method, carry out an action for you. Other methods compute and return a value. For example, the length method returns a value, namely the number of characters in the string. You can store the return value in a variable:

```java
int numberOfCharacters = greeting.length();
```

You can also use the return value of one method as an argument of another method:

```java
System.out.println(greeting.length());
```

The method call greeting.length() returns a value—the integer 13. The return value becomes an argument of the println method. Figure 8 shows the process.
Not all methods return values. One example is the `println` method. The `println` method interacts with the operating system, causing characters to appear in a window. But it does not return a value to the code that calls it.

Let us analyze a more complex method call. Here, we will call the `replace` method of the `String` class. The `replace` method carries out a search-and-replace operation, similar to that of a word processor. For example, the call

```java
river.replace("issipp", "our")
```

constructs a new string that is obtained by replacing all occurrences of "issipp" in "Mississippi" with "our". (In this situation, there was only one replacement.) The method returns the `String` object "Missouri". You can save that string in a variable:

```java
river = river.replace("issipp", "our");
```

Or you can pass it to another method:

```java
System.out.println(river.replace("issipp", "our"));
```

As Figure 9 shows, this method call

- Is invoked on a `String` object: "Mississippi"
- Has two arguments: the strings "issipp" and "our"
- Returns a value: the string "Missouri"

![Figure 9](Calling the replace Method)

### Table 3 Method Arguments and Return Values

<table>
<thead>
<tr>
<th>Example</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>System.out.println(greeting)</code></td>
<td>greeting is an argument of the <code>println</code> method.</td>
</tr>
<tr>
<td><code>greeting.replace(&quot;e&quot;,&quot;3&quot;)</code></td>
<td>The <code>replace</code> method has two arguments, in this case &quot;e&quot; and &quot;3&quot;.</td>
</tr>
<tr>
<td><code>greeting.length()</code></td>
<td>The <code>length</code> method has no arguments.</td>
</tr>
<tr>
<td><code>int n = greeting.length();</code></td>
<td>The <code>length</code> method returns an integer value.</td>
</tr>
<tr>
<td><code>System.out.println(n);</code></td>
<td>The <code>println</code> method returns no value. In the API documentation, its return type is <code>void</code>.</td>
</tr>
<tr>
<td><code>System.out.println(greeting.length());</code></td>
<td>The return value of one method can become the argument of another.</td>
</tr>
</tbody>
</table>
2.3.4 Method Declarations

When a method is declared in a class, the declaration specifies the types of the arguments and the return value. For example, the String class declares the length method as

```java
public int length()
```

That is, there are no arguments, and the return value has the type int. (For now, all the methods that we consider will be “public” methods—see Chapter 9 for more restricted methods.)

The replace method is declared as

```java
public String replace(String target, String replacement)
```

To call the replace method, you supply two arguments, target and replacement, which both have type String. The returned value is another string.

When a method returns no value, the return type is declared with the reserved word void. For example, the PrintStream class declares the println method as

```java
public void println(String output)
```

Occasionally, a class declares two methods with the same name and different argument types. For example, the PrintStream class declares a second method, also called println, as

```java
public void println(int output)
```

That method is used to print an integer value. We say that the println name is overloaded because it refers to more than one method.

14. How can you compute the length of the string "Mississippi"?
15. How can you print out the uppercase version of "Hello, World!"?
16. Is it legal to call river.println()? Why or why not?
17. What are the arguments in the method call river.replace("p", "s")?
18. What is the result of the call river.replace("p", "s")?
19. What is the result of the call greeting.replace("World", "Dave").length()?
20. How is the toUpperCase method declared in the String class?

Practice It Now you can try these exercises at the end of the chapter: R2.8, R2.9, R2.10.

Learn By Trying

When you learn about a new method, write a small program to try it out. For example, you can go right now to your Java development environment and run this program:

```java
public class ReplaceDemo
{
    public static void main(String[] args)
    {
        String river = "Mississippi";
        System.out.println(river.replace("issipp", "our"));
    }
}
```
Then you can see with your own eyes what the `replace` method does. Also, you can run experiments. Does `replace` change every match, or only the first one? Try it out:

```java
System.out.println(river.replace("i", "x"));
```

Set up your work environment to make this kind of experimentation easy and natural. Keep a file with the blank outline of a Java program around, so you can copy and paste it when needed. Alternatively, some development environments will automatically type the class and `main` method. Find out if yours does. Some environments even let you type commands into a window and show you the result right away, without having to make a `main` method to call `System.out.println` (see Figure 10).

### 2.4 Constructing Objects

Generally, when you want to use objects in your program, you need to specify their initial properties by **constructing** them.

To learn about object construction, we need to go beyond `String` objects and the `System.out` object. Let us turn to another class in the Java library: the `Rectangle` class. Objects of type `Rectangle` describe rectangular shapes. These objects are useful for a variety of purposes. You can assemble rectangles into bar charts, and you can program simple games by moving rectangles inside a window.

Note that a `Rectangle` object isn’t a rectangular shape—it’s an object that contains a set of numbers. The numbers describe the rectangle (see Figure 11). Each rectangle is described by the x- and y-coordinates of its top-left corner, its width, and its height.
2.4 Constructing Objects

It is very important that you understand this distinction. In the computer, a Rectangle object is a block of memory that holds four numbers, for example $x = 5$, $y = 10$, $width = 20$, $height = 30$. In the imagination of the programmer who uses a Rectangle object, the object describes a geometric figure.

To make a new rectangle, you need to specify the $x$, $y$, $width$, and $height$ values. Then *invoke the new operator*, specifying the name of the class and the argument(s) required for constructing a new object. For example, you can make a new rectangle with its top-left corner at $(5, 10)$, width 20, and height 30 as follows:

```java
new Rectangle(5, 10, 20, 30)
```

Here is what happens in detail:

1. The `new` operator makes a `Rectangle` object.
2. It uses the arguments (in this case, 5, 10, 20, and 30) to initialize the object’s data.
3. It returns the object.

The process of creating a new object is called *construction*. The four values 5, 10, 20, and 30 are called the *construction arguments*.

The `new` expression yields an object, and you need to store the object if you want to use it later. Usually you assign the output of the `new` operator to a variable. For example,

```java
Rectangle box = new Rectangle(5, 10, 20, 30);
```

**Syntax 2.3 Object Construction**

```java
new ClassName(arguments)
```

The `new` expression yields an object.

Construction arguments

```
Rectangle box = new Rectangle(5, 10, 20, 30);
```

System.out.println(new Rectangle());

You can also pass a constructed object to a method.

Supply the parentheses even when there are no arguments.

Usually, you save the constructed object in a variable.
Some classes let you construct objects in multiple ways. For example, you can also obtain a Rectangle object by supplying no construction arguments at all (but you must still supply the parentheses):

```java
new Rectangle()
```

This expression constructs a (rather useless) rectangle with its top-left corner at the origin (0, 0), width 0, and height 0.

21. How do you construct a square with center (100, 100) and side length 20?
22. Initialize the variables box and box2 with two rectangles that touch each other.
23. The getWidth method returns the width of a Rectangle object. What does the following statement print?
   ```java
   System.out.println(new Rectangle().getWidth());
   ```
24. The PrintStream class has a constructor whose argument is the name of a file. How do you construct a PrintStream object with the construction argument "output.txt"?
25. Write a statement to save the object that you constructed in Self Check 24 in a variable.

Practice It
Now you can try these exercises at the end of the chapter: R2.13, R2.16, R2.18.

**Common Error 2.3**
Constructors are not methods. You can only use a constructor with the `new` operator, not to reinitialize an existing object:

```java
box.Rectangle(20, 35, 20, 30); // Error—can’t reinitialize object
```

The remedy is simple: Make a new object and overwrite the current one stored by box.

```java
box = new Rectangle(20, 35, 20, 30); // OK
```

### 2.5 Accessor and Mutator Methods

In this section we introduce a useful terminology for the methods of a class. A method that accesses an object and returns some information about it, without changing the object, is called an **accessor method**. In contrast, a method whose purpose is to modify the internal data of an object is called a **mutator method**.

For example, the `length` method of the `String` class is an accessor method. It returns information about a string, namely its length. But it doesn’t modify the string at all when counting the characters.

The `Rectangle` class has a number of accessor methods. The `getX`, `getY`, `getWidth`, and `getHeight` methods return the x- and y-coordinates of the top-left corner, the width, and the height values. For example,

```java
double width = box.getWidth();
```
Now let us consider a mutator method. Programs that manipulate rectangles frequently need to move them around, for example, to display animations. The Rectangle class has a method for that purpose, called translate. (Mathematicians use the term “translation” for a rigid motion of the plane.) This method moves a rectangle by a certain distance in the $x$- and $y$-directions. The method call,

```java
box.translate(15, 25);
```

moves the rectangle by 15 units in the $x$-direction and 25 units in the $y$-direction (see Figure 12). Moving a rectangle doesn’t change its width or height, but it changes the top-left corner. Afterward, the rectangle that had its top-left corner at (5, 10) now has it at (20, 35).

This method is a mutator because it modifies the object on which the method is invoked.

![Figure 12 Using the translate Method to Move a Rectangle](https://www.wiley.com/go/bjeo6code)

26. What does this sequence of statements print?

```java
Rectangle box = new Rectangle(5, 10, 20, 30);
System.out.println("Before: " + box.getX());
box.translate(25, 40);
System.out.println("After: " + box.getX());
```

27. What does this sequence of statements print?

```java
Rectangle box = new Rectangle(5, 10, 20, 30);
System.out.println("Before: " + box.getWidth());
box.translate(25, 40);
System.out.println("After: " + box.getWidth());
```

28. What does this sequence of statements print?

```java
String greeting = "Hello";
System.out.println(greeting.toUpperCase());
System.out.println(greeting);
```

29. Is the `toUpperCase` method of the `String` class an accessor or a mutator?

30. Which call to `translate` is needed to move the rectangle declared by `Rectangle box = new Rectangle(5, 10, 20, 30)` so that its top-left corner is the origin $(0, 0)$?

**Practice It** Now you can try these exercises at the end of the chapter: R2.19, E2.7, E2.9.
2.6 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 `PrintStream` and `Rectangle` are *system programmers*.

You can find the API documentation on the Web. Point your web browser to [http://docs.oracle.com/javase/8/docs/api/index.html](http://docs.oracle.com/javase/8/docs/api/index.html). An abbreviated version of the API documentation is provided in Appendix D that may be easier to use at first, but you should eventually move on to the real thing.

### 2.6.1 Browsing the API Documentation

The API documentation documents all classes in the Java library—there are thousands of them (see Figure 13, top). Most of the classes are rather specialized, and only a few are of interest to the beginning programmer.

Locate the `Rectangle` link in the left pane, preferably by using the search function of your browser. Click on the link, and the right pane shows all the features of the `Rectangle` class (see Figure 13, bottom).
The API documentation for each class starts out with a section that describes the purpose of the class. Then come summary tables for the constructors and methods (see Figure 14, top). Click on a method’s link to get a detailed description (see Figure 14, bottom).

The detailed description of a method shows

- The action that the method carries out.  
- The types and names of the parameter variables that receive the arguments when the method is called. 
- The value that it returns (or the reserved word \texttt{void} if the method doesn’t return any value).

As you can see, the \texttt{Rectangle} class has quite a few methods. While occasionally intimidating for the beginning programmer, this is a strength of the standard library. If you ever need to do a computation involving rectangles, chances are that there is a method that does all the work for you.

For example, suppose you want to change the width or height of a rectangle. If you browse through the API documentation, you will find a \texttt{setSize} method with the description “Sets the size of this \texttt{Rectangle} to the specified width and height.” The method has two arguments, described as

- \texttt{width} - the new width for this \texttt{Rectangle}
- \texttt{height} - the new height for this \texttt{Rectangle}
We can use this information to change the box object so that it is a square of side length 40. The name of the method is `setSize`, and we supply two arguments: the new width and height:

```java
box.setSize(40, 40);
```

## 2.6.2 Packages

The API documentation contains another important piece of information about each class. The classes in the standard library are organized into packages. A package is a collection of classes with a related purpose. The `Rectangle` class belongs to the package `java.awt` (where `awt` is an abbreviation for “Abstract Windowing Toolkit”), which contains many classes for drawing windows and graphical shapes. You can see the package name `java.awt` in Figure 13, just above the class name.

To use the `Rectangle` class from the `java.awt` package, you must `import` the package. Simply place the following line at the top of your program:

```java
import java.awt.Rectangle;
```

Why don’t you have to import the `System` and `String` classes? Because the `System` and `String` classes are in the `java.lang` package, and all classes from this package are automatically imported, so you never need to import them yourself.

### Syntax 2.4 Importing a Class from a Package

**Syntax**

```java
import packageName.ClassName;
```

- **Package name**
- **Class name**

**Import statements must be at the top of the source file.**

You can look up the package name in the API documentation.

### SELF CHECK

31. Look at the API documentation of the `String` class. Which method would you use to obtain the string "hello, world!" from the string "Hello, World!"?

32. In the API documentation of the `String` class, look at the description of the `trim` method. What is the result of applying `trim` to the string " Hello, Space ! "? (Note the spaces in the string.)

33. Look into the API documentation of the `Rectangle` class. What is the difference between the methods `void translate(int x, int y)` and `void setLocation(int x, int y)`?

34. The `Random` class is declared in the `java.util` package. What do you need to do in order to use that class in your program?
35. In which package is the BigInteger class located? Look it up in the API documentation.

**Practice It** Now you can try these exercises at the end of the chapter: R2.20, E2.5, E2.12.

**Don’t Memorize—Use Online Help**

The Java library has thousands of classes and methods. It is neither necessary nor useful trying to memorize them. Instead, you should become familiar with using the API documentation. Because you will need to use the API documentation all the time, it is best to download and install it onto your computer, particularly if your computer is not always connected to the Internet. You can download the documentation from [http://www.oracle.com/technetwork/java/javase/downloads/index.html](http://www.oracle.com/technetwork/java/javase/downloads/index.html).

### 2.7 Implementing a Test Program

In this section, we discuss the steps that are necessary to implement a test program. The purpose of a test program is to verify that one or more methods have been implemented correctly. A test program calls methods and checks that they return the expected results. Writing test programs is a very important skill.

In this section, we will develop a simple program that tests a method in the Rectangle class using these steps:

1. Provide a tester class.
2. Supply a main method.
3. Inside the main method, construct one or more objects.
4. Apply methods to the objects.
5. Display the results of the method calls.
6. Display the values that you expect to get.

Our sample test program tests the behavior of the translate method. Here are the key steps (which have been placed inside the main method of the MoveTester class).

```java
Rectangle box = new Rectangle(5, 10, 20, 30);
// Move the rectangle
box.translate(15, 25);
// Print information about the moved rectangle
System.out.print("x: ");
System.out.println(box.getX());
System.out.println("Expected: 20");
```

We print the value that is returned by the getX method, and then we print a message that describes the value we expect to see.

This is a very important step. You want to spend some time thinking about the expected result 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. Finding and fixing errors early is a very effective strategy that can save you a great deal of time.
In our case, the rectangle has been constructed with the top-left corner at (5, 10). The x-direction is moved by 15, so we expect an x-value of 5 + 15 = 20 after the move.

Here is the program that tests the moving of a rectangle:

```
section_7/MoveTester.java
1  import java.awt.Rectangle;
2  
3  public class MoveTester
4  {
5      public static void main(String[] args)
6      {
7          Rectangle box = new Rectangle(5, 10, 20, 30);
8          // Move the rectangle
9          box.translate(15, 25);
10         // Print information about the moved rectangle
11         System.out.print("x: ");
12         System.out.println(box.getX());
13         System.out.println("Expected: 20");
14         System.out.print("y: ");
15         System.out.println(box.getY());
16         System.out.println("Expected: 35");
17      }
18  }
```

**Program Run**

```
x: 20
Expected: 20
y: 35
Expected: 35
```

**SELF CHECK**

36. Suppose we had called `box.translate(25, 15)` instead of `box.translate(15, 25)`. What are the expected outputs?

37. Why doesn’t the `MoveTester` program need to print the width and height of the rectangle?

**Practice It**

Now you can try these exercises at the end of the chapter: E2.1, E2.8, E2.14.

---

**Testing Classes in an Interactive Environment**

Some development environments are specifically designed to help students explore objects without having to provide tester classes. These environments can be very helpful for gaining insight into the behavior of objects, and for promoting object-oriented thinking. The BlueJ environment (shown in the figure) displays objects as blobs on a workbench.

You can construct new objects, put them on the workbench, invoke methods, and see the return values, all without writing a line of code. You can download BlueJ at no charge from [www.bluej.org](http://www.bluej.org). Another excellent environment for interactively exploring objects is Dr. Java at [drjava.sourceforge.net](http://drjava.sourceforge.net).
In Java, an object variable (that is, 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 15.
There is a reason for this behavior. Objects can be very large. It is more efficient to store only the memory location instead of the entire object.

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

```java
Rectangle box = new Rectangle(5, 10, 20, 30);
```

the variable `box` refers to the `Rectangle` 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 `box` variable.

It is very important that you remember that the `box` variable does not contain the object. It refers to the object. Two object variables can refer to the same object:

```java
Rectangle box2 = box;
```

Now you can access the same `Rectangle` object as `box` and as `box2`, as shown in Figure 16.

In Java, numbers are not objects. Number variables actually store numbers. When you declare

```java
int luckyNumber = 13;
```

then the `luckyNumber` variable holds the number 13, not a reference to the number (see Figure 17). The reason is again efficiency. Because numbers require little storage, it is more efficient to store them directly in a variable.

You can see the difference between number 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 18):

```java
int luckyNumber = 13;  // 1
int luckyNumber2 = luckyNumber;  // 2
luckyNumber2 = 12;  // 3
```

Now the variable `luckyNumber` contains the value 13, and `luckyNumber2` contains 12.
Now consider the seemingly analogous code with `Rectangle` objects (see Figure 19).

```java
Rectangle box = new Rectangle(5, 10, 20, 30);  // 1
Rectangle box2 = box;  // 2
box2.translate(15, 25);  // 3
```

Because `box` and `box2` refer to the same rectangle after step 2, both variables refer to the moved rectangle after the call to the `translate` method.

---

**Figure 18** Copying Numbers

1. `luckyNumber = 13`
2. `luckyNumber = 13`
   `luckyNumber2 = 12`
3. `luckyNumber = 13`
   `luckyNumber2 = 12`

**Figure 19** Copying Object References
You need not worry too much about the difference between objects and object references. Much of the time, you will have the correct intuition when you think of the “object box” rather than the technically more accurate “object reference stored in variable box”. The difference between objects and object references only becomes apparent when you have multiple variables that refer to the same object.

**SELF CHECK**

38. What is the effect of the assignment `String greeting2 = greeting`?

39. After calling `greeting2.toUpperCase()`, what are the contents of `greeting` and `greeting2`?

**Practice It**

Now you can try these exercises at the end of the chapter: R2.17, R2.21.

### Computing & Society 2.1  Computer Monopoly

When International Business Machines Corporation (IBM), a successful manufacturer of punched-card equipment for tabulating data, first turned its attention to designing computers in the early 1950s, its planners assumed that there was a market for perhaps 50 such devices, for installation by the government, the military, and a few of the country’s largest corporations. Instead, they sold about 1,500 machines of their System 650 model and went on to build and sell more powerful computers.

These computers, called mainframes, were huge. They filled rooms, which had to be climate-controlled to protect the delicate equipment. IBM was not the first company to build mainframe computers; that honor belongs to the Univac Corporation. However, IBM soon became the major player, partially because of its technical excellence and attention to customer needs and partially because it exploited its strengths and structured its products and services in a way that made it difficult for customers to mix them with those of other vendors.

As all of IBM’s competitors fell on hard times, the U.S. government brought an antitrust suit against IBM in 1969. In the United States, it is legal to be a monopoly supplier, but it is not legal to use one’s monopoly in one market to gain supremacy in another. IBM was accused of forcing customers to buy bundles of computers, software, and peripherals, making it impossible for other vendors of software and peripherals to compete.

The suit went to trial in 1975 and dragged on until 1982, when it was abandoned, largely because new waves of smaller computers had made it irrelevant.

In fact, when IBM offered its first personal computers, its operating system was supplied by an outside vendor, Microsoft, which became so dominant that it too was sued by the U.S. government for abusing its monopoly position in 1998. Microsoft was accused of bundling its web browser with its operating system. At the time, Microsoft allegedly threatened hardware makers that they would not receive a Windows license if they distributed the competing Netscape browser. In 2000, the company was found guilty of antitrust violations, and the judge ordered it broken up into an operating systems unit and an applications unit. The breakup was reversed on appeal, and a settlement in 2001 was largely unsuccessful in establishing alternatives for desktop software.

Now the computing landscape is shifting once again, toward mobile devices and cloud computing. As you observe that change, you may well see new monopolies in the making. When a software vendor needs the permission of a hardware vendor in order to place a product into an “app store”, or when a maker of a digital book reader tries to coerce publishers into a particular pricing structure, the question arises whether such conduct is illegal exploitation of a monopoly position.
2.9 Graphical Applications

The following optional sections teach you how to write graphical applications: applications that display drawings inside a window. The drawings are made up of shape objects: rectangles, ellipses, and lines. The shape objects provide another source of examples, and many students enjoy the visual feedback.

2.9.1 Frame Windows

A graphical application shows information inside a frame: a window with a title bar, as shown in Figure 20. In this section, you will learn how to display a frame. In Section 2.9.2, you will learn how to create a drawing inside the frame.

To show a frame, carry out the following steps:

1. Construct an object of the JFrame class:
   
   ```java
   JFrame frame = new JFrame();
   ```

2. Set the size of the frame:
   
   ```java
   frame.setSize(300, 400);
   ```

   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.)

3. If you’d like, set the title of the frame:
   
   ```java
   frame.setTitle("An empty frame");
   ```

   If you omit this step, the title bar is simply left blank.

4. Set the “default close operation”:
   
   ```java
   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:
   
   ```java
   frame.setVisible(true);
   ```

The simple program below shows all of these steps. It produces the empty frame shown in Figure 20.

The JFrame class is a part of the javax.swing package. Swing is the nickname for the graphical user interface library in Java. The “x” in javax denotes the fact that Swing started out as a Java extension before it was added to the standard library.
We will go into much greater detail about Swing programming in Chapters 3, 10, and 20. For now, consider this program to be the essential plumbing that is required to show a frame.

**section_9_1/EmptyFrameViewer.java**

```java
import javax.swing.JFrame;

public class EmptyFrameViewer {

    public static void main(String[] args) {
        JFrame frame = new JFrame();
        frame.setSize(300, 400);
        frame.setTitle("An empty frame");
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.setVisible(true);
    }
}
```

### 2.9.2 Drawing on a Component

In this section, you will learn how to make shapes appear inside a frame window. The first drawing will be exceedingly modest: just two rectangles (see Figure 21). You'll soon see how to produce more interesting drawings. The purpose of this example is to show you the basic outline of a program that creates a drawing.

You cannot draw directly onto a frame. Instead, drawing happens in a component object. In the Swing toolkit, the JComponent class represents a blank component.

Because we don’t want to add a blank component, we have to modify the JComponent class and specify how the component should be painted. The solution is to declare a new class that extends the JComponent class. You will learn about the process of extending classes in Chapter 9.
For now, simply use the following code as a template:

```java
public class RectangleComponent extends JComponent {
    public void paintComponent(Graphics g) {
        // Drawing instructions.
    }
}
```

The extends reserved word indicates that our component class, `RectangleComponent`, can be used like a `JComponent`. However, the `RectangleComponent` class will be different from the plain `JComponent` class in one respect: Its `paintComponent` method will contain instructions to draw the rectangles.

When the component is shown for the first time, the `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` as its argument. The `Graphics` object stores the graphics state—the current color, font, and so on—that are used for drawing operations. However, the `Graphics` class is not very useful. When programmers clamored for a more object-oriented approach to drawing graphics, the designers of Java created the `Graphics2D` class, which extends the `Graphics` class. Whenever the Swing toolkit calls the `paintComponent` method, it actually passes an object of type `Graphics2D` as the argument. Because we want to use the more sophisticated methods to draw two-dimensional graphics objects, we need to use the `Graphics2D` class. This is accomplished by using a `cast`:

```java
public class RectangleComponent extends JComponent {
    public void paintComponent(Graphics g) {
        // Recover Graphics2D
        Graphics2D g2 = (Graphics2D) g;
        ...
    }
}
```
Chapter 9 has more information about casting. For now, you should simply include the cast at the top of your paintComponent methods.

Now you are ready to draw shapes. The draw method of the Graphics2D class can draw shapes, such as rectangles, ellipses, line segments, polygons, and arcs. Here we draw a rectangle:

```java
public class RectangleComponent extends JComponent {
    public void paintComponent(Graphics g) {
        // Recover Graphics2D
        Graphics2D g2 = (Graphics2D) g;
        // New Rectangle and Draw
        Rectangle box = new Rectangle(5, 10, 20, 30);
        g2.draw(box);
    }
}
```

When positioning the shapes, you need to pay attention to the coordinate system. It 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.

![Coordinate System Diagram]

Following is the source code for the RectangleComponent class. Note that the paintComponent method of the RectangleComponent class draws two rectangles. As you can see from the import statements, the Graphics and Graphics2D classes are part of the java.awt package.

```java
import java.awt.Graphics;
import java.awt.Graphics2D;
import java.awt.Rectangle;
import javax.swing.JComponent;

/**
 * A component that draws two rectangles.
 */
public class RectangleComponent extends JComponent {
    public void paintComponent(Graphics g) {
        // Recover Graphics2D
        Graphics2D g2 = (Graphics2D) g;
        // New Rectangle and Draw
        Rectangle box = new Rectangle(5, 10, 20, 30);
        g2.draw(box);
    }
}
```
2.9.3 Displaying a Component in a Frame

In a graphical application, you need a frame to show the application, and you need a component for the drawing. In this section, you will see how to combine the two. Follow these steps:

1. Construct a frame object and configure it.
2. Construct an object of your component class:
   
   ```java
   RectangleComponent component = new RectangleComponent();
   ```

3. Add the component to the frame:
   
   ```java
   frame.add(component);
   ```

4. Make the frame visible.

The following listing shows the complete process.

**section_9_3/RectangleViewer.java**

```java
import javax.swing.JFrame;

public class RectangleViewer {
    public static void main(String[] args) {
        JFrame frame = new JFrame();
        frame.setSize(300, 400);
        frame.setTitle("Two rectangles");
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        RectangleComponent component = new RectangleComponent();
        frame.add(component);
        frame.setVisible(true);
    }
}
```

Note that the rectangle drawing program consists of two classes:

- The RectangleComponent class, whose paintComponent method produces the drawing.
- The RectangleViewer class, whose main method constructs a frame and a RectangleComponent, adds the component to the frame, and makes the frame visible.
40. How do you display a square frame with a title bar that reads “Hello, World!”?

41. How can a program display two frames at once?

42. How do you modify the program to draw two squares?

43. How do you modify the program to draw one rectangle and one square?

44. What happens if you call `g.draw(box)` instead of `g2.draw(box)`?

Practice It  Now you can try these exercises at the end of the chapter: R2.22, R2.26, E2.18.

2.10 Ellipses, Lines, Text, and Color

In Section 2.9 you learned how to write a program that draws rectangles. In the following sections, you will learn how to draw other shapes: ellipses and lines. With these graphical elements, you can draw quite a few interesting pictures.

2.10.1 Ellipses and Circles

To draw an ellipse, you specify its bounding box (see Figure 22) 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.

However, there is no simple `Ellipse` class that you can use. Instead, you must use one of the two classes `Ellipse2D.Float` and `Ellipse2D.Double`, depending on whether you want to store the ellipse coordinates as single- or double-precision floating-point values. Because the latter are more convenient to use in Java, we will always use the `Ellipse2D.Double` class.

Here is how you construct an ellipse:

```java
Ellipse2D.Double ellipse = new Ellipse2D.Double(x, y, width, height);
```

The class name `Ellipse2D.Double` looks different from the class names that you have encountered up to now. It consists of two class names `Ellipse2D` and `Double` separated...
by a period (\(\cdot\)). This indicates that \texttt{Ellipse2D.Double} is a so-called \textbf{inner class} inside \texttt{Ellipse2D}. When constructing and using ellipses, you don’t actually need to worry about the fact that \texttt{Ellipse2D.Double} is an inner class—just think of it as a class with a long name. However, in the \texttt{import} statement at the top of your program, you must be careful that you import only the outer class:

\begin{verbatim}
import java.awt.geom.Ellipse2D;
\end{verbatim}

Drawing an ellipse is easy: Use exactly the same \texttt{draw} method of the \texttt{Graphics2D} class that you used for drawing rectangles.

\begin{verbatim}
g2.draw(ellipse);
\end{verbatim}

To draw a circle, simply set the width and height to the same values:

\begin{verbatim}
Ellipse2D.Double circle = new Ellipse2D.Double(x, y, diameter, diameter);
g2.draw(circle);
\end{verbatim}

Notice that \((x, y)\) is the top-left corner of the bounding box, not the center of the circle.

### 2.10.2 Lines

To draw a line, use an object of the \texttt{Line2D.Double} class. A line is constructed by specifying its two end points. You can do this in two ways. Give the \(x\) - and \(y\)-coordinates of both end points:

\begin{verbatim}
Line2D.Double segment = new Line2D.Double(x1, y1, x2, y2);
\end{verbatim}

Or specify each end point as an object of the \texttt{Point2D.Double} class:

\begin{verbatim}
Point2D.Double from = new Point2D.Double(x1, y1);
Point2D.Double to = new Point2D.Double(x2, y2);

Line2D.Double segment = new Line2D.Double(from, to);
\end{verbatim}

The second option is more object-oriented and is often more useful, particularly if the point objects can be reused elsewhere in the same drawing.

### 2.10.3 Drawing Text

You often want to put text inside a drawing, for example, to label some of the parts. Use the \texttt{drawString} method of the \texttt{Graphics2D} 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 23). For example,

\begin{verbatim}
g2.drawString(\"Message\", 50, 100);
\end{verbatim}

![Message](message.png)

**Figure 23**  Basepoint and Baseline
2.10.4 Colors

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,

```java
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.

For your convenience, a variety of colors have been declared in the Color class. Table 4 shows those colors and their RGB values. For example, Color.PINK has been declared 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 Graphics2D object, then call the draw method:

```java
g2.setColor(Color.RED);
g2.draw(circle); // Draws the shape in red
```

If you want to color the inside of the shape, use the fill method instead of the draw method. For example,

```java
g2.fill(circle);
```

fills the inside of the circle with the current color.

<table>
<thead>
<tr>
<th>Color</th>
<th>RGB Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>Color.BLACK</td>
<td>0, 0, 0</td>
</tr>
<tr>
<td>Color.BLUE</td>
<td>0, 0, 255</td>
</tr>
<tr>
<td>Color.CYAN</td>
<td>0, 255, 255</td>
</tr>
<tr>
<td>Color.GRAY</td>
<td>128, 128, 128</td>
</tr>
<tr>
<td>Color.DARK_GRAY</td>
<td>64, 64, 64</td>
</tr>
<tr>
<td>Color.LIGHT_GRAY</td>
<td>192, 192, 192</td>
</tr>
<tr>
<td>Color.GREEN</td>
<td>0, 255, 0</td>
</tr>
<tr>
<td>Color.MAGENTA</td>
<td>255, 0, 255</td>
</tr>
<tr>
<td>Color.ORANGE</td>
<td>255, 200, 0</td>
</tr>
<tr>
<td>Color.PINK</td>
<td>255, 175, 175</td>
</tr>
<tr>
<td>Color.RED</td>
<td>255, 0, 0</td>
</tr>
<tr>
<td>Color.WHITE</td>
<td>255, 255, 255</td>
</tr>
<tr>
<td>Color.YELLOW</td>
<td>255, 255, 0</td>
</tr>
</tbody>
</table>

When you set a new color in the graphics context, it is used for subsequent drawing operations.
The following program puts all these shapes to work, creating a simple drawing (see Figure 24).

**section_10/FaceComponent.java**

```java
import java.awt.Color;
import java.awt.Graphics;
import java.awt.Graphics2D;
import java.awt.Rectangle;
import java.awt.geom.Ellipse2D;
import java.awt.geom.Line2D;
import javax.swing.JComponent;

/**
 * A component that draws an alien face.
 */
public class FaceComponent extends JComponent {
    public void paintComponent(Graphics g) {
        // Recover Graphics2D
        Graphics2D g2 = (Graphics2D) g;

        // Draw the head
        Ellipse2D.Double head = new Ellipse2D.Double(5, 10, 100, 150);
        g2.draw(head);

        // Draw the eyes
        g2.setColor(Color.GREEN);
        Rectangle eye = new Rectangle(25, 70, 15, 15);
        g2.fill(eye);
        eye.translate(50, 0);
        g2.fill(eye);

        // Draw the mouth
        Line2D.Double mouth = new Line2D.Double(30, 110, 80, 110);
        g2.setColor(Color.RED);
        g2.draw(mouth);

        // Draw the greeting
        g2.setColor(Color.BLUE);
        g2.drawString("Hello, World!", 5, 175);
    }
}
```

*Figure 24*  An Alien Face
section_10/FaceViewer.java

```java
import javax.swing.JFrame;

public class FaceViewer {
    public static void main(String[] args) {
        JFrame frame = new JFrame();
        frame.setSize(150, 250);
        frame.setTitle("An Alien Face");
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        FaceComponent component = new FaceComponent();
        frame.add(component);
        frame.setVisible(true);
    }
}
```

**SELF CHECK**

45. Give instructions to draw a circle with center (100, 100) and radius 25.

46. Give instructions to draw a letter “V” by drawing two line segments.

47. Give instructions to draw a string consisting of the letter “V”.

48. What are the RGB color values of Color.BLUE?

49. How do you draw a yellow square on a red background?

**Practice It** Now you can try these exercises at the end of the chapter: R2.27, E2.19, E2.20.

**CHAPTER SUMMARY**

**Identify objects, methods, and classes.**

- Objects are entities in your program that you manipulate by calling methods.
- A method is a sequence of instructions that accesses the data of an object.
- A class describes a set of objects with the same behavior.

**Write variable declarations and assignments.**

- 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.
- Numbers can be combined by arithmetic operators such as +, -, and *.
- By convention, variable names should start with a lowercase letter.
- Use comments to add explanations for humans who read your code. The compiler ignores comments.
• Use the assignment operator (=) to change the value of a variable.
• All variables must be initialized before you access them.
• The assignment operator = does not denote mathematical equality.

**Recognize arguments and return values of methods.**

• The public interface of a class specifies what you can do with its objects. The hidden implementation describes how these actions are carried out.
• An argument is a value that is supplied in a method call.
• The return value of a method is a result that the method has computed.

**Use constructors to construct new objects.**

• Use the new operator, followed by a class name and arguments, to construct new objects.

**Classify methods as accessor and mutator methods.**

• An accessor method does not change the internal data of the object on which it is invoked. A mutator method changes the data.

**Use the API documentation for finding method descriptions and packages.**

• The API (Application Programming Interface) documentation lists the classes and methods of the Java library.
• Java classes are grouped into packages. Use the import statement to use classes that are declared in other packages.

**Write programs that test the behavior of methods.**

• A test program verifies that methods behave as expected.
• Determining the expected result in advance is an important part of testing.

**Describe how multiple object references can refer to the same object.**

• An object reference describes the location of an object.
• Multiple object variables can contain references to the same object.
• Number variables store numbers. Object variables store references.

**Write programs that display frame windows.**

• To show a frame, construct a JFrame object, set its size, and make it visible.
• In order to display a drawing in a frame, declare 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.
• Use a cast to recover the `Graphics2D` object from the `Graphics` argument of the `paintComponent` method.

Use the Java API for drawing simple figures.

• The `Ellipse2D.Double` and `Line2D.Double` classes describe graphical 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.

STANDARD LIBRARY ITEMS INTRODUCED IN THIS CHAPTER

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>getHeight</td>
<td>setColor</td>
<td>setSize</td>
</tr>
<tr>
<td>getWidth</td>
<td>draw</td>
<td>translate</td>
</tr>
<tr>
<td>setSize</td>
<td>drawString</td>
<td>java.lang.String</td>
</tr>
<tr>
<td>setVisible</td>
<td>fill</td>
<td>length</td>
</tr>
<tr>
<td>java.awt.Frame</td>
<td>java.awt.Rectangle</td>
<td></td>
</tr>
<tr>
<td>setTitle</td>
<td>getX</td>
<td>replace</td>
</tr>
<tr>
<td>java.awt.geom.Ellipse2D.Double</td>
<td>getY</td>
<td>toLowerCase</td>
</tr>
<tr>
<td>java.awt.geom.Line2D.Double</td>
<td>getHeight</td>
<td>toUpperCase</td>
</tr>
<tr>
<td>java.awt.geom.Point2D.Double</td>
<td>getWidth</td>
<td>javax.swing.JComponent</td>
</tr>
<tr>
<td>java.awt.Frame</td>
<td>java.awt.Rectangle</td>
<td>paintComponent</td>
</tr>
<tr>
<td>java.awt.geom.Ellipse2D.Double</td>
<td>getX</td>
<td>javax.swing.JFrame</td>
</tr>
<tr>
<td>java.awt.geom.Line2D.Double</td>
<td>getY</td>
<td>setDefaultCloseOperation</td>
</tr>
</tbody>
</table>

REVIEW EXERCISES

• **R2.1** Explain the difference between an object and a class.
• **R2.2** Give three examples of objects that belong to the `String` class. Give an example of an object that belongs to the `PrintStream` class. Name two methods that belong to the `String` class but not the `PrintStream` class. Name a method of the `PrintStream` class that does not belong to the `String` class.
• **R2.3** What is the public interface of a class? How does it differ from the implementation of a class?
• **R2.4** Declare and initialize variables for holding the price and the description of an article that is available for sale.
• **R2.5** What is the value of mystery after this sequence of statements?
  ```java
  int mystery = 1;
  mystery = 1 - 2 * mystery;
  mystery = mystery + 1;
  ```
• **R2.6** What is wrong with the following sequence of statements?
  ```java
  int mystery = 1;
  mystery = mystery + 1;
  int mystery = 1 - 2 * mystery;
  ```
R2.7 Explain the difference between the = symbol in Java and in mathematics.

R2.8 Give an example of a method that has an argument of type int. Give an example of a method that has a return value of type int. Repeat for the type String.

R2.9 Write Java statements that initialize a string message with "Hello" and then change it to "HELLO". Use the toUpperCase method.

R2.10 Write Java statements that initialize a string message with "Hello" and then change it to "hello". Use the replace method.

R2.11 Write Java statements that initialize a string message with a message such as "Hello, World" and then remove punctuation characters from the message, using repeated calls to the replace method.

R2.12 Explain the difference between an object and an object variable.

R2.13 Give the Java code for constructing an object of class Rectangle, and for declaring an object variable of class Rectangle.

R2.14 Give Java code for objects with the following descriptions:
   a. A rectangle with center (100, 100) and all side lengths equal to 50
   b. A string with the contents “Hello, Dave”

Create objects, not object variables.

R2.15 Repeat Exercise R2.14, but now declare object variables that are initialized with the required objects.

R2.16 Write a Java statement to initialize a variable square with a rectangle object whose top-left corner is (10, 20) and whose sides all have length 40. Then write a statement that replaces square with a rectangle of the same size and top-left corner (20, 20).

R2.17 Write Java statements that initialize two variables square1 and square2 to refer to the same square with center (20, 20) and side length 40.

R2.18 Find the errors in the following statements:
   a. Rectangle r = (5, 10, 15, 20);
   b. double width = Rectangle(5, 10, 15, 20).getWidth();
   c. Rectangle r;
      r.translate(15, 25);
   d. r = new Rectangle();
      r.translate("far, far away!");

R2.19 Name two accessor methods and two mutator methods of the Rectangle class.

R2.20 Consult the API documentation to find methods for
   • Concatenating two strings, that is, making a string consisting of the first string, followed by the second string.
   • Removing leading and trailing white space of a string.
   • Converting a rectangle to a string.
   • Computing the smallest rectangle that contains two given rectangles.
   • Returning a random floating-point number.

For each method, list the class in which it is defined, the return type, the method name, and the types of the arguments.
Chapter 2  Using Objects

- R2.21 Explain the difference between an object and an object reference.
- Graphics R2.22 What is the difference between a console application and a graphical application?
- Graphics R2.23 Who calls the paintComponent method of a component? When does the call to the paintComponent method occur?
- Graphics R2.24 Why does the argument of the paintComponent method have type Graphics and not Graphics2D?
- Graphics R2.25 What is the purpose of a graphics context?
- Graphics R2.26 Why are separate viewer and component classes used for graphical programs?
- Graphics R2.27 How do you specify a text color?

PRACTICE EXERCISES

- Testing E2.1 Write an AreaTester program that constructs a Rectangle object and then computes and prints its area. Use the getWidth and getHeight methods. Also print the expected answer.
- Testing E2.2 Write a PerimeterTester program that constructs a Rectangle object and then computes and prints its perimeter. Use the getWidth and getHeight methods. Also print the expected answer.
- E2.3 Write a program that initializes a string with "Mississippi". Then replace all "i" with "ii" and print the length of the resulting string. In that string, replace all "ss" with "s" and print the length of the resulting string.
- E2.4 Write a program that constructs a rectangle with area 42 and a rectangle with perimeter 42. Print the widths and heights of both rectangles.
- Testing E2.5 Look into the API documentation of the Rectangle class and locate the method
  ```java
  void add(int newx, int newy)
  ```
  Read through the method documentation. Then determine the result of the following statements:
  ```java
  Rectangle box = new Rectangle(5, 10, 20, 30);
  box.add(0, 0);
  ```
  Write a program AddTester that prints the expected and actual location, width, and height of box after the call to add.
- Testing E2.6 Write a program ReplaceTester that encodes a string by replacing all letters "i" with "!" and all letters "s" with "$". Use the replace method. Demonstrate that you can correctly encode the string "Mississippi". Print both the actual and expected result.
- E2.7 Write a program HollePrinter that switches the letters "e" and "o" in a string. Use the replace method repeatedly. Demonstrate that the string "Hello, World!" turns into "Holle, Werld!"
- Testing E2.8 The StringBuilder class has a method for reversing a string. In a ReverseTester class, construct a StringBuilder from a given string (such as "desserts"), call the reverse method followed by the toString method, and print the result. Also print the expected value.
• E2.9 In the Java library, a color is specified by its red, green, and blue components between 0 and 255 (see Table 4 on page 66). Write a program BrighterDemo that constructs a Color object with red, green, and blue values of 50, 100, and 150. Then apply the brighter method of the Color class and print the red, green, and blue values of the resulting color. (You won’t actually see the color—see Exercise E2.10 on how to display the color.)

• Graphics E2.10 Repeat Exercise E2.9, but place your code into the following class. Then the color will be displayed.

```java
import java.awt.Color;
import javax.swing.JFrame;

public class BrighterDemo {
    public static void main(String[] args) {
        JFrame frame = new JFrame();
        frame.setSize(200, 200);
        Color myColor = ...;
        frame.getContentPane().setBackground(myColor);
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.setVisible(true);
    }
}
```

• E2.11 Repeat Exercise E2.9, but apply the darker method of the Color class twice to the object Color.RED. Call your class DarkerDemo.

• E2.12 The Random class implements a random number generator, which produces sequences of numbers that appear to be random. To generate random integers, you construct an object of the Random class, and then apply the nextInt method. For example, the call `generator.nextInt(6)` gives you a random number between 0 and 5.

Write a program DieSimulator that uses the Random class to simulate the cast of a die, printing a random number between 1 and 6 every time that the program is run.

• E2.13 Write a program RandomPrice that prints a random price between $10.00 and $19.95 every time the program is run.

• Testing E2.14 Look at the API of the Point class and find out how to construct a Point object. In a PointTester program, construct two points with coordinates (3, 4) and (−3, −4). Find the distance between them, using the distance method. Print the distance, as well as the expected value. (Draw a sketch on graph paper to find the value you will expect.)

• E2.15 Using the Day class of Worked Example 2.1, write a DayTester program that constructs a Day object representing today, adds ten days to it, and then computes the difference between that day and today. Print the difference and the expected value.

• E2.16 Using the Picture class of Worked Example 2.2, write a HalfSizePicture program that loads a picture and shows it at half the original size, centered in the window.

• E2.17 Using the Picture class of Worked Example 2.2, write a DoubleSizePicture program that loads a picture, doubles its size, and shows the center of the picture in the window.

• Graphics E2.18 Write a graphics program that draws two squares, both with the same center. Provide a class TwoSquareViewer and a class TwoSquareComponent.
Chapter 2  Using Objects

**Graphics E2.19** 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.

**Graphics E2.20** Write a graphics program that draws your name in red, contained inside a blue rectangle. Provide a class NameViewer and a class NameComponent.

**PROGRAMMING PROJECTS**

**P2.1** Write a program called FourRectanglePrinter that constructs a Rectangle object, prints its location by calling System.out.println(box), and then translates and prints it three more times, so that, if the rectangles were drawn, they would form one large rectangle, as shown at right.

Your program will not produce a drawing. It will simply print the locations of the four rectangles.

**P2.2** Write a GrowSquarePrinter program that constructs a Rectangle object square representing a square with top-left corner (100, 100) and side length 50, prints its location by calling System.out.println(square), applies the translate and grow methods, and calls System.out.println(square) again. The calls to translate and grow should modify the square so that it has twice the size and the same top-left corner as the original. If the squares were drawn, they would look like the figure at right.

Your program will not produce a drawing. It will simply print the locations of square before and after calling the mutator methods. Look up the description of the grow method in the API documentation.

**P2.3** Write a CenteredSquaresPrinter program that constructs a Rectangle object square representing a square with top-left corner (100, 100) and side length 200, prints its location by calling System.out.println(square), applies the grow and translate methods, and calls System.out.println(square) again. The calls to grow and translate should modify the square so that it has half the width and is centered in the original square. If the squares were drawn, they would look like the figure at right. Your program will not produce a drawing. It will simply print the locations of square before and after calling the mutator methods. Look up the description of the grow method in the API documentation.

**P2.4** The intersection method computes the intersection of two rectangles—that is, the rectangle that would be formed by two overlapping rectangles if they were drawn, as shown at right.

You call this method as follows:

```java
Rectangle r3 = r1.intersection(r2);
```

Write a program IntersectionPrinter that constructs two rectangle objects, prints them as described in Exercise P2.1, and then prints the rectangle object that describes the intersection. Then the program should print the result of the intersection method when the rectangles do not overlap. Add a comment to your program that explains how you can tell whether the resulting rectangle is empty.
Graphics P2.5  In this exercise, you will explore a simple way of visualizing a Rectangle object. The setBounds method of the JFrame class moves a frame window to a given rectangle. Complete the following program to visually show the translate method of the Rectangle class:

```java
import java.awt.Rectangle;
import javax.swing.JFrame;
import javax.swing.JOptionPane;

public class TranslateDemo
{
  public static void main(String[] args)
  {
    // Construct a frame and show it
    JFrame frame = new JFrame();
    frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    frame.setVisible(true);

    // Your work goes here: Construct a rectangle and set the frame bounds
    JOptionPane.showMessageDialog(frame, "Click OK to continue");

    // Your work goes here: Move the rectangle and set the frame bounds again
  }
}
```

P2.6  Write a program LotteryPrinter that picks a combination in a lottery. In this lottery, players can choose 6 numbers (possibly repeated) between 1 and 49. Construct an object of the Random class (see Exercise E2.12) and invoke an appropriate method to generate each number. (In a real lottery, repetitions aren’t allowed, but we haven’t yet discussed the programming constructs that would be required to deal with that problem.) Your program should print out a sentence such as “Play this combination—it’ll make you rich!”, followed by a lottery combination.

P2.7  Using the Day class of Worked Example 2.1, write a program that generates a Day object representing February 28 of this year, and three more such objects that represent February 28 of the next three years. Advance each object by one day, and print each object. Also print the expected values:

- 2016-02-29
- Expected: 2016-02-29
- 2017-03-01
- Expected: 2017-03-01
- ...

P2.8  The GregorianCalendar class describes a point in time, as measured by the Gregorian calendar, the standard calendar that is commonly used throughout the world today. You construct a GregorianCalendar object from a year, month, and day of the month, like this:

```java
GregorianCalendar cal = new GregorianCalendar(); // Today's date
GregorianCalendar eckertsBirthday = new GregorianCalendar(1919, Calendar.APRIL, 9);
```

Use the values Calendar.JANUARY . . . Calendar.DECEMBER to specify the month. The add method can be used to add a number of days to a GregorianCalendar object:

```java
cal.add(Calendar.DAY_OF_MONTH, 10); // Now cal is ten days from today
```

This is a mutator method—it changes the cal object.
Chapter 2  Using Objects

The get method can be used to query a given GregorianCalendar object:

```java
int dayOfMonth = cal.get(Calendar.DAY_OF_MONTH);
int month = cal.get(Calendar.MONTH);
int year = cal.get(Calendar.YEAR);
int weekday = cal.get(Calendar.DAY_OF_WEEK);
// 1 is Sunday, 2 is Monday, ..., 7 is Saturday
```

Your task is to write a program that prints:
- The date and weekday that is 100 days from today.
- The weekday of your birthday.
- The date that is 10,000 days from your birthday.

Use the birthday of a computer scientist if you don’t want to reveal your own.

**Hint:** The GregorianCalendar class is complex, and it is a really good idea to write a few test programs to explore the API before tackling the whole problem. Start with a program that constructs today’s date, adds ten days, and prints out the day of the month and the weekday.

**P2.9** In Java 8, the LocalDate class describes a calendar date that does not depend on a location or time zone. You construct a date like this:

```java
LocalDate today = LocalDate.now(); // Today’s date
LocalDate eckertsBirthday = LocalDate.of(1919, 4, 9);
```

The plusDays method can be used to add a number of days to a LocalDate object:

```java
LocalDate later = today.plusDays(10); // Ten days from today
```

This method does not mutate the today object, but it returns a new object that is a given number of days away from today.

To get the year of a day, call

```java
int year = today.getYear();
```

To get the weekday of a LocalDate, call

```java
String weekday = today.getDayOfWeek().toString();
```

Your task is to write a program that prints
- The weekday of “Pi day”, that is, March 14, of the current year.
- The date and weekday of “Programmer’s day” in the current year; that is, the 256th day of the year. (The number 256, or 2^8, is useful for some programming tasks.)
- The date and weekday of the date that is 10,000 days earlier than today.

**Testing P2.10** Write a program LineDistanceTester that constructs a line joining the points (100, 100) and (200, 200), then constructs points (100, 200), (150, 150), and (250, 50). Print the distance from the line to each of the three points, using the ptSegDist method of the Line2D class. Also print the expected values. (Draw a sketch on graph paper to find what values you expect.)

**Graphics P2.11** Repeat Exercise P2.10, but now write a graphical application that shows the line and the points. Draw each point as a tiny circle. Use the drawString method to draw each distance next to the point, using calls

```java
g2.drawString("Distance: " + distance, p.getX(), p.getY());
```

**Graphics P2.12** Write a graphics program that draws 12 strings, one each for the 12 standard colors (except Color.WHITE), each in its own color. Provide a class ColorNameViewer and a class ColorNameComponent.
**Answers to Self-Check Questions**

1. Objects with the same behavior belong to the same class. A window lets in light while protecting a room from the outside wind and heat or cold. A water heater has completely different behavior. It heats water. They belong to different classes.

2. When one calls a method, one is not concerned with how it does its job. As long as a light bulb illuminates a room, it doesn’t matter to the occupant how the photons are produced.

3. When you compile the program, you get an error message that the `String` class doesn’t have a `println` method.

4. 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.

5. `double unitPrice = 1.95;`  
   `int quantity = 2;`

6. `System.out.print("Total price: ");`  
   `System.out.println(unitPrice * quantity);`

7. `int` and `String`

8. `double`

9. Only the first two are legal identifiers.

10. `String myName = "John Q. Public";`  
    `String greeting = "Hello, Nina!";`  
    Note that  
    `String greeting = "Hello, Nina!";`  
    is not the right answer—that statement declares a new variable.

11. Assignment would occur when one car is replaced by another in the parking space.

12. `river.length()` or "Mississippi".length()
15. System.out.println(greeting.toUpperCase());
   or
   System.out.println(
     "Hello, World!").toUpperCase();
16. It is not legal. The variable river has type String. The println method is not a method of the String class.
17. The arguments are the strings “p” and “s”.
18. "Mississippi"
19. 12
20. As public String toUpperCase(), with no argument and return type String.
21. new Rectangle(90, 90, 20, 20)
22. Rectangle box = new Rectangle(5, 10, 20, 30);
    Rectangle box2 = new Rectangle(25, 10, 20, 30);
23. 0
24. new PrintStream("output.txt");
25. PrintStream out = new PrintStream("output.txt");
26. Before: 5
    After: 30
27. Before: 20
    After: 20
    Moving the rectangle does not affect its width or height. You can change the width and height with the setSize method.
28. HELLO
    hello
    Note that calling toUpperCase doesn’t modify the string.
29. An accessor—it doesn’t modify the original string but returns a new string with uppercase letters.
30. box.translate(-5, -10), provided the method is called immediately after storing the new rectangle into box.
31. toLowerCase
32. "Hello, Space !"—only the leading and trailing spaces are trimmed.
33. The arguments of the translate method tell how far to move the rectangle in the x- and y-directions. The arguments of the setLocation method indicate the new x- and y-values for the top-left corner.
    For example, box.translate(1, 1) moves the box one pixel down and to the right. box.setLocation(1, 1) moves box to the top-left corner of the screen.
34. Add the statement import java.util.Random; at the top of your program.
35. In the java.math package.
36. x: 30, y: 25
37. Because the translate method doesn’t modify the shape of the rectangle.
38. Now greeting and greeting2 both refer to the same string object.
39. Both variables still refer to the same string, and the string has not been modified. Recall that the toUpperCase method constructs a new string that contains uppercase characters, leaving the original string unchanged.
40. Modify the EmptyFrameViewer program as follows:
    frame.setSize(300, 300);
    frame.setTitle("Hello, World!");
41. Construct two JFrame objects, set each of their sizes, and call setVisible(true) on each of them.
42. Change line 17 of RectangleComponent to
    Rectangle box = new Rectangle(5, 10, 20, 20);
43. Replace the call to box.translate(15, 25) with
    box = new Rectangle(20, 35, 20, 20);
44. The compiler complains that g doesn’t have a draw method.
45. g2.draw(new Ellipse2D.Double(75, 75, 50, 50));
46. Line2D.Double segment1
    = new Line2D.Double(0, 0, 10, 30);
    g2.draw(segment1);
Line2D.Double segment2
    = new Line2D.Double(10, 30, 20, 0);
    g2.draw(segment2);
47. g2.drawString("V", 0, 30);
48. 0, 0, 255
49. First fill a big red square, then fill a small yellow square inside:
    g2.setColor(Color.RED);
    g2.fill(new Rectangle(0, 0, 200, 200));
    g2.setColor(Color.YELLOW);
    g2.fill(new Rectangle(50, 50, 100, 100));
How Many Days Have You Been Alive?

Many programs need to process dates such as “February 15, 2010.” The worked_example_1 directory of this chapter’s companion code contains a Day class that was designed to work with calendar days.

The Day class knows about the intricacies of our calendar, such as the fact that January has 31 days and February has 28 or sometimes 29. The Julian calendar, instituted by Julius Caesar in the first century BCE, introduced the rule that every fourth year is a leap year. In 1582, Pope Gregory XIII ordered the implementation of the calendar that is in common use throughout the world today, called the Gregorian calendar. It refines the leap year rule by specifying that years divisible by 100 are not leap years, unless they are divisible by 400. Thus, the year 1900 was not a leap year but the year 2000 was. All of these details are handled by the internals of the Day class.

The Day class lets you answer questions such as

- How many days are there between now and the end of the year?
- What day is 100 days from now?

**Problem Statement**  Your task is to write a program that determines how many days you have been alive. You should not look inside the internal implementation of the Day class. Use the API documentation by pointing your browser to the file index.html in the ch02/worked_example_1/api subdirectory.

As you can see from the API documentation (see figure on next page), you construct a Day object from a given year, month, and day, like this:

```java
Day jamesGoslingsBirthday = new Day(1955, 5, 19);
```

There is a method for adding days to a given day, for example:

```java
Day later = jamesGoslingsBirthday.addDays(100);
```

You can then find out what the result is, by applying the `getYear/getMonth/getDate` methods:

```java
System.out.println(later.getYear());
System.out.println(later.getMonth());
System.out.println(later.getDate());
```

However, that approach does not solve our problem (unless you are willing to replace `100` with other values until, by trial and error, you obtain today’s date). Instead, use the `daysFrom` method. According to the API documentation, we need to supply another day. That is, the method is called like this:

```java
int daysAlive = day1.daysFrom(day2);
```

In our situation, one of the Day objects is `jamesGoslingsBirthday`, and the other is today’s date. This can be obtained with the constructor that has no arguments:

```java
Day today = new Day();
```

We have two candidates on which the `daysFrom` method could be invoked, yielding the call

```java
int daysAlive = jamesGoslingsBirthday.daysFrom(today);
or int daysAlive = today.daysFrom(jamesGoslingsBirthday);
```
Which is the right choice? Fortunately, the author of the `Day` class has anticipated this question. The detail comment of the `daysFrom` method contains this statement:

```
Returns: the number of days that this day is away from the other
(larger than 0 if this day comes later than other)
```

We want a positive result. Therefore, the second form is the correct one.

Here is the program that solves our problem (see `ch02/worked_example_1` in your source code):

```
worked_example_1/DaysAlivePrinter.java

1 public class DaysAlivePrinter
2 {
3     public static void main(String[] args)
4     {
5         Day jamesGoslingsBirthday = new Day(1955, 5, 19);
6         Day today = new Day();
7         System.out.print("Today: ");
8         System.out.println(today.toString());
9         int daysAlive = today.daysFrom(jamesGoslingsBirthday);
```
10 System.out.print("Days alive: ");
11 System.out.println(daysAlive);
12 }
13 }

Program Run

Today: 2015-02-09
Days alive: 21826
WORKED EXAMPLE 2.2  Working with Pictures

Problem Statement  Edit and display image files in the Picture class found in the ch02/worked_example_2 directory of this chapter’s companion code. For example, the following program simply shows the image given below:

```java
public class PictureDemo {
    public static void main(String[] args) {
        Picture pic = new Picture();
        pic.load("queen-mary.png");
    }
}
```

Your task is to write a program that reads in an image, shrinks it, and adds a border. Shrink it sufficiently so that there is a transparent border inside the black border, as in the figure below.
You should *not* look inside the internal implementation of the `Picture` class. Instead, use the API documentation by pointing your browser to the file `index.html` in the `worked_example_2/picture/api` subdirectory.

The API contains a number of methods that are unrelated to the task, but two of the methods are clearly useful:

```java
public void scale(int newWidth, int newHeight)
public void border(int width)
```

If the method comments are not clear, it is a good idea to write a couple of simple test programs to see their effect. For example, this program demonstrates the `scale` method:

```java
public class PictureScaleDemo
{
    public static void main(String[] args)
    {
        Picture pic = new Picture();
        pic.load("queen-mary.png");
        pic.scale(200, 200);
    }
}
```
Here is the result:

As you can see, the picture has been resized to a 200 \times 200\ pixel square.

That’s not quite what we want. We want the picture to be a bit smaller than the original. Let’s say that the black border is 10\ pixels thick, and we want another transparent border of 10\ pixels. Then the target width and height are 40\ pixels less than the original, leaving 20\ pixels on each side for the borders.

Looking at the API, we find methods for obtaining the original width and height. Therefore, we will call

\[
\text{int newWidth} = \text{pic.getWidth()} - 40; \\
\text{int newHeight} = \text{pic.getHeight()} - 40; \\
\text{pic.scale(newWidth, newHeight);}
\]

Then we add the border:

\[
\text{pic.border(10)};
\]

The result is

If we can move the picture a bit before applying the border, we are done. Another look at the API reveals a method

\[
\text{public void move(int dx, int dy)}
\]
That’s just what we need. The picture needs to be moved 20 pixels down and to the right. Our final program is

```java
public class BorderMaker {
    public static void main(String[] args) {
        Picture pic = new Picture();
        pic.load("queen-mary.png");
        int newWidth = pic.getWidth() - 40;
        int newHeight = pic.getHeight() - 40;
        pic.scale(newWidth, newHeight);
        pic.move(20, 20);
        pic.border(10);
    }
}
```

Couldn’t we have achieved the same result with an image editing program such as Photoshop or GIMP? Yes, but it is an easy matter to extend this program so that it can automatically apply a border to any number of images.
CHAPTER 3
IMPLEMENTING CLASSES

CHAPTER GOALS

To become familiar with the process of implementing classes
To be able to implement and test simple methods
To understand the purpose and use of constructors
To understand how to access instance variables and local variables
To be able to write javadoc comments
To implement classes for drawing graphical shapes

CHAPTER CONTENTS

3.1 INSTANCE VARIABLES AND ENCAPSULATION 80
SYN Instance Variable Declaration 81

3.2 SPECIFYING THE PUBLIC INTERFACE OF A CLASS 84
SYN Class Declaration 87
CE1 Declaring a Constructor as void 90
PT1 The javadoc Utility 90

3.3 PROVIDING THE CLASS IMPLEMENTATION 91
CE2 Ignoring Parameter Variables 96
HT1 Implementing a Class 96
WE1 Making a Simple Menu

3.4 UNIT TESTING 100
C&S Electronic Voting Machines 102

3.5 PROBLEM SOLVING: TRACING OBJECTS 103
CE3 Duplicating Instance Variables in Local Variables 106
CE4 Providing Unnecessary Instance Variables 106
CE5 Forgetting to Initialize Object References in a Constructor 107

3.6 LOCAL VARIABLES 105

3.7 THE THIS REFERENCE 107
ST1 Calling One Constructor from Another 110

3.8 SHAPE CLASSES 110
HT2 Drawing Graphical Shapes 114
In this chapter, you will learn how to implement your own classes. You will start with a given design that specifies the public interface of the class—that is, the methods through which programmers can manipulate the objects of the class. Then you will learn the steps to completing the class—creating the internal “workings” like the inside of an air conditioner shown here. You need to implement the methods, which entails finding a data representation for the objects and supplying the instructions for each method. You need to document your efforts so that other programmers can understand and use your creation. And you need to provide a tester to validate that your class works correctly.

3.1 Instance Variables and Encapsulation

In Chapter 1, you learned how to use objects from existing classes. In this chapter, you will start implementing your own classes. We begin with a very simple example that shows you how objects store their data, and how methods access the data of an object. 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).

3.1.1 Instance Variables

Whenever the operator clicks the button of a tally counter, the counter value advances by one. We model this operation with a click method of a Counter class. A physical counter has a display to show the current value. In our simulation, we use a getValue method to get the current value. For example,

```
Counter tally = new Counter();
tally.click();
tally.click();
int result = tally.getValue(); // Sets result to 2
```

When implementing the Counter class, you need to determine the data that each counter object contains. In this simple example, that is very straightforward. Each counter needs a variable that keeps track of the number of simulated button clicks.

An object stores its data 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;
    . . .
}
```
Syntax 3.1 Instance Variable Declaration

Syntax

```
public class ClassName
{
    private typeName variableName;
    ...
}
```

Each object of this class has a separate copy of this instance variable.

Instance variables should always be private.

Type of the variable

An instance variable declaration consists of the following parts:

- An access specifier (private)
- The type of the instance variable (such as int)
- The name of the instance variable (such as value)

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 3.3, the instance variable value is set to 0 when a Counter object is constructed.

Figure 2 Instance Variables

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.
3.1.2 The Methods of the Counter Class

In this section, we will look at the implementation of the methods of the Counter class. The click method advances the counter value by 1. You have seen the method header syntax in Chapter 2. Now, focus on the body of the method inside the braces.

```java
public void click()
{
    value = value + 1;
}
```

Note how the click method accesses the instance variable value. Which instance variable? The one belonging to the object on which the method is invoked. For example, consider the call

```java
concertCounter.click();
```

This call advances the value variable of the concertCounter object.

The getValue method returns the current value:

```java
public int getValue()
{
    return value;
}
```

The return statement is a special statement that terminates the method call and returns a result (the return value) to the method’s caller.

Instance variables are generally declared with the access specifier private. That specifier means that they can be accessed only by the methods of the same class, not by any other method. For example, the value variable can be accessed by the click and getValue methods of the Counter class but not by a method of another class. Those other methods need to use the Counter class methods if they want to manipulate a counter’s internal data.

3.1.3 Encapsulation

In the preceding section, you learned that you should hide instance variables by making them private. Why would a programmer want to hide something?

The strategy of information hiding is not unique to computer programming—it is used in many engineering disciplines. Consider the thermostat that you find in your home. It is a device that allows a user to set temperature preferences and that controls the furnace and the air conditioner. If you ask your contractor what is inside the thermostat, you will likely get a shrug.

The thermostat is a black box, something that magically does its thing. A contractor would never open the control module—it contains electronic parts that can only be serviced at the factory. In general, engineers use the term “black box” to describe any device whose inner workings are hidden. Note that a black box is not totally mysterious. Its interface with the outside world is well-defined. For example, the contractor understands how the thermostat must be connected with the furnace and air conditioner.

The thermostat is a black box, something that magically does its thing. A contractor would never open the control module—it contains electronic parts that can only be serviced at the factory. In general, engineers use the term “black box” to describe any device whose inner workings are hidden. Note that a black box is not totally mysterious. Its interface with the outside world is well-defined. For example, the contractor understands how the thermostat must be connected with the furnace and air conditioner.

The process of hiding implementation details while publishing an interface is called encapsulation. In Java, the class construct provides encapsulation. The public methods of a class are the interface through which the private implementation is manipulated.
Why do contractors use prefabricated components such as thermostats and furnaces? These “black boxes” greatly simplify the work of the contractor. In ancient times, builders had to know how to construct furnaces from brick and mortar, and how to produce some rudimentary temperature controls. Nowadays, a contractor just makes a trip to the hardware store, without needing to know what goes on inside the components.

Similarly, a programmer using a class is not burdened by unnecessary detail, as you know from your own experience. In Chapter 2, you used classes for strings, streams, and windows without worrying how these classes are implemented.

Encapsulation also helps with diagnosing errors. A large program may consist of hundreds of classes and thousands of methods, but if there is an error with the internal data of an object, you only need to look at the methods of one class. Finally, encapsulation makes it possible to change the implementation of a class without having to tell the programmers who use the class.

In Chapter 2, you learned to be an object user. You saw how to obtain objects, how to manipulate them, and how to assemble them into a program. In that chapter, you treated objects as black boxes. Your role was roughly analogous to the contractor who installs a new thermostat.

In this chapter, you will move on to implementing classes. In these sections, your role is analogous to the hardware manufacturer who puts together a thermostat from buttons, sensors, and other electronic parts. You will learn the necessary Java programming techniques that enable your objects to carry out the desired behavior.

section_1/Counter.java

```java
/**
 * 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 click() {
        value = value + 1;
    }
}
```
Chapter 3  Implementing Classes

/**
 * Resets the value of this counter to 0.
 */
public void reset()
{
    value = 0;
}

1. Supply the body of a method public void unclick() that undoes an unwanted button click.

2. Suppose you use a class Clock with private instance variables hours and minutes. How can you access these variables in your program?

3. Consider the Counter class. A counter’s value starts at 0 and is advanced by the click method, so it should never be negative. Suppose you found a negative value variable during testing. Where would you look for the error?

4. In Chapters 1 and 2, you used System.out as a black box to cause output to appear on the screen. Who designed and implemented System.out?

5. Suppose you are working in a company that produces personal finance software. You are asked to design and implement a class for representing bank accounts. Who will be the users of your class?

Practice It  Now you can try these exercises at the end of the chapter: R3.1, R3.3, E3.1.

3.2 Specifying the Public Interface of a Class

In the following sections, we will discuss the process of specifying the public interface of a class. Imagine that you are a member of a team that works on banking software. A fundamental concept in banking is a bank account. Your task is to design a BankAccount class that can be used by other programmers to manipulate bank accounts. What methods should you provide? What information should you give the programmers who use this class? You will want to settle these questions before you implement the class.

3.2.1 Specifying Methods

You need to know exactly what operations of a bank account need to be implemented. Some operations are essential (such as taking deposits), whereas others are not important (such as giving a gift to a customer who opens a bank account). Deciding which operations are essential is not always an easy task. We will revisit that issue in Chapters 8 and 12. For now, we will assume that a competent designer has decided that the following are considered the essential operations of a bank account:

- Deposit money
- Withdraw money
- Get the current balance
In Java, you call a method when you want to apply an operation to an object. To figure out the exact specification of the method calls, imagine how a programmer would carry out the bank account operations. We’ll assume that the variable `harrysChecking` contains a reference to an object of type `BankAccount`. We want to support method calls such as the following:

```java
harrysChecking.deposit(2240.59);
harrysChecking.withdraw(500);
double currentBalance = harrysChecking.getBalance();
```

The first two methods are mutators. They modify the balance of the bank account and don’t return a value. The third method is an accessor. It returns a value that you store in a variable or pass to a method.

From the sample calls, we decide the `BankAccount` class should declare three methods:

- `public void deposit(double amount)`
- `public void withdraw(double amount)`
- `public double getBalance()`

Recall from Chapter 12 that `double` denotes the double-precision floating-point type, and `void` indicates that a method does not return a value.

Here we only give the method headers. When you declare a method, you also need to provide the method body, which consists of statements that are executed when the method is called.

```java
public void deposit(double amount)
{
    // method body—implementation filled in later
}
```

We will supply the method bodies in Section 3.3.

Note that the methods have been declared as `public`, indicating that all other methods in a program can call them. Occasionally, it can be useful to have private methods. They can only be called from other methods of the same class.

Some people like to fill in the bodies so that they compile, like this:

```java
public double getBalance()
{
    // TODO: fill in implementation
    return 0;
}
```

That is a good idea if you compose your method specification in your development environment—you won’t get warnings about incorrect code.

### 3.2.2 Specifying Constructors

As you know from Chapter 2, constructors are used to initialize objects. In Java, a constructor is very similar to a method, with two important differences:

- The name of the constructor is always the same as the name of the class (e.g., `BankAccount`).
- Constructors have no return type (not even `void`).

We want to be able to construct bank accounts that initially have a zero balance, as well as accounts that have a given initial balance.
For this purpose, we specify two constructors:

- `public BankAccount()`
- `public BankAccount(double initialBalance)`

They are used as follows:

```java
BankAccount harrysChecking = new BankAccount();
BankAccount momsSavings = new BankAccount(5000);
```

Don’t worry about the fact that there are two constructors with the same name—all constructors of a class have the same name, that is, the name of the class. The compiler can tell them apart because they take different arguments. The first constructor takes no arguments at all. Such a constructor is called a **no-argument constructor**. The second constructor takes an argument of type `double`.

Just like a method, a constructor also has a body—a sequence of statements that is executed when a new object is constructed.

```java
public BankAccount()
{
    constructor body—implementation filled in later
}
```

The statements in the constructor body will set the instance variables of the object that is being constructed—see Section 3.3.

When declaring a class, you place all constructor and method declarations inside, like this:

```java
public class BankAccount
{
    private instance variables—filled in later

    // Constructors
    public BankAccount()
    {
        implementation—filled in later
    }

    public BankAccount(double initialBalance)
    {
        implementation—filled in later
    }

    // Methods
    public void deposit(double amount)
    {
        implementation—filled in later
    }

    public void withdraw(double amount)
    {
        implementation—filled in later
    }

    public double getBalance()
    {
        implementation—filled in later
    }
}
```
### Syntax 3.2  Class Declaration

Syntax 3.2  Class Declaration

```
Syntax  accessSpecifier  class  ClassName
{  
    instance variables
    constructors
    methods
}
```

```
public class Counter  
{  
    private  int  value;
    
    public Counter(int initialValue)  
    {  
        value = initialValue;
    }
    
    public void click()  
    {  
        value = value + 1;
    }
    
    public int getValue()  
    {  
        return value;
    }
}
```

The public constructors and methods of a class form the **public interface** of the class. These are the operations that any programmer can use to create and manipulate BankAccount objects.

#### 3.2.3  Using the Public Interface

Our BankAccount class is simple, but it allows programmers to carry out all of the important operations that commonly occur with bank accounts. For example, consider this program segment, authored by a programmer who uses the BankAccount class.

These statements transfer an amount of money from one bank account to another:

```java
// Transfer from one account to another
double transferAmount = 500;
momsSavings.withdraw(transferAmount);
harrysChecking.deposit(transferAmount);
```

And here is a program segment that adds interest to a savings account:

```java
double interestRate = 5;  // 5 percent interest
double interestAmount = momsSavings.getBalance() * interestRate / 100;
momsSavings.deposit(interestAmount);
```

As you can see, programmers can use objects of the BankAccount class to carry out meaningful tasks, without knowing how the BankAccount objects store their data or how the BankAccount methods do their work.

Of course, as implementors of the BankAccount class, we will need to supply the private implementation. We will do so in Section 3.3. First, however, an important step remains: documenting the public interface. That is the topic of the next section.

#### 3.2.4  Commenting the Public Interface

When you implement classes and methods, you should get into the habit of thoroughly **commenting** their behaviors. In Java there is a very useful standard form for
documentation comments. If you use this form in your classes, a program called javadoc can automatically generate a neat set of HTML pages that describe them. (See Programming Tip 3.1 on page 90 for a description of this utility.)

A documentation comment is placed before the class or method declaration that is being documented. It starts with a /**, a special comment delimiter used by the javadoc utility. Then you describe the method’s purpose. Then, for each argument, you supply a line that starts with @param, followed by the name of the variable that holds the argument (which is called a parameter variable). Supply a short explanation for each argument after the variable name. Finally, you supply a line that starts with @return, describing the return value. You omit the @param tag for methods that have no arguments, and you omit the @return tag for methods whose return type is void.

The javadoc utility copies the first sentence of each comment to a summary table in the HTML documentation. Therefore, it is best to write that first sentence with some care. It should start with an uppercase letter and end with a period. It does not have to be a grammatically complete sentence, but it should be meaningful when it is pulled out of the comment and displayed in a summary.

Here are two typical examples:

```java
/**
   * Withdraws money from the bank account.
   * @param amount the amount to withdraw
   */
   public void withdraw(double amount)
   {
      implementation—filled in later
   }

/**
   * Gets the current balance of the bank account.
   * @return the current balance
   */
   public double getBalance()
   {
      implementation—filled in later
   }
```

The comments you have just seen explain individual methods. Supply a brief comment for each class, too, explaining its purpose. Place the documentation comment above the class declaration:

```java
/**
   * A bank account has a balance that can be changed by deposits and withdrawals.
   */
   public class BankAccount
   {
      ...
   }
```

Your first reaction may well be “Whoa! Am I supposed to write all this stuff?” Sometimes, documentation comments seem pretty repetitive, but in most cases, they are informative. Even with seemingly repetitive comments, you should take the time to write them.

It is always a good idea to write the method comment first, before writing the code in the method body. This is an excellent test to see that you firmly understand what
you need to program. If you can’t explain what a class or method does, you aren’t ready to implement it.

What about very simple methods? You can easily spend more time pondering whether a comment is too trivial to write than it takes to write it. In practical programming, very simple methods are rare. It is harmless to have a trivial method overcommented, whereas a complicated method without any comment can cause real grief to future maintenance programmers. According to the standard Java documentation style, every class, every method, every parameter variable, and every return value should have a comment.

The javadoc utility formats your 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 the comment is used for a summary table of all methods of your class (see Figure 3). The @param and @return comments are neatly formatted in the detail description of each method (see Figure 4). If you omit any of the comments, then javadoc generates documents that look strangely empty.
This documentation format should look familiar. The programmers who implement the Java library use javadoc themselves. They too document every class, every method, every parameter variable, and every return value, and then use javadoc to extract the documentation in HTML format.

6. How can you use the methods of the public interface to empty the harrysChecking bank account?

7. What is wrong with this sequence of statements?
   ```java
   BankAccount harrysChecking = new BankAccount(10000);
   System.out.println(harrysChecking.withdraw(500));
   ```

8. Suppose you want a more powerful bank account abstraction that keeps track of an account number in addition to the balance. How would you change the public interface to accommodate this enhancement?

9. Suppose we enhance the BankAccount class so that each account has an account number. Supply a documentation comment for the constructor
   ```java
   public BankAccount(int accountNumber, double initialBalance)
   ```

10. Why is the following documentation comment questionable?
    ```java
    /**
     * Each account has an account number.
     * @return the account number of this account
     */
    public int getAccountNumber()
    ```

Practice It  Now you can try these exercises at the end of the chapter: R3.7, R3.8, R3.9.

---

### Declaring a Constructor as void

Do not use the `void` reserved word when you declare a constructor:

```java
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.

### The javadoc Utility

Always insert documentation comments in your code, whether or not you use javadoc to produce HTML documentation. Most people find the HTML documentation convenient, so it is worth learning how to run `javadoc`. Some programming environments (such as BlueJ) can execute `javadoc` for you. Alternatively, you can invoke the `javadoc` utility from a shell window, by issuing the command

```bash
javadoc MyClass.java
```

or, if you want to document multiple Java files,

```bash
javadoc *.java
```

The `javadoc` utility produces files such as `MyClass.html` in HTML format, which you can inspect in a browser. If you know HTML (see Appendix H), you can embed HTML tags into the
comments to specify fonts or add images. Perhaps most importantly, javadoc automatically provides hyperlinks 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.

3.3 Providing the Class Implementation

Now that you understand the specification of the public interface of the BankAccount class, let’s provide the implementation.

3.3.1 Providing Instance Variables

First, we need to determine the data that each bank account object contains. In the case of our simple bank account class, each object needs to store a single value, the current balance. (A more complex bank account class might store additional data—perhaps an account number, the interest rate paid, the date for mailing out the next statement, and so on.)

```java
public class BankAccount
{
    private double balance;
    // Methods and constructors below
}
```

In general, it can be challenging to find a good set of instance variables. Ask yourself what an object needs to remember so that it can carry out any of its methods.

Like a wilderness explorer who needs to carry all items that may be needed, an object needs to store the data required for its method calls.
3.3.2 Providing Constructors

A constructor has a simple job: to initialize the instance variables of an object. Recall that we designed the BankAccount class to have two constructors. The first constructor simply sets the balance to zero:

```java
public BankAccount()
{
    balance = 0;
}
```

The second constructor sets the balance to the value supplied as the construction argument:

```java
public BankAccount(double initialBalance)
{
    balance = initialBalance;
}
```

To see how these constructors work, let us trace the statement

```java
BankAccount harrysChecking = new BankAccount(1000);
```

one step at a time. Here are the steps that are carried out when the statement executes (see Figure 5):

1. Create a new object of type BankAccount.
2. Call the second constructor (because an argument is supplied in the constructor call).
3. Set the parameter variable initialBalance to 1000.
4. Set the balance instance variable of the newly created object to initialBalance.
5. Return an object reference, that is, the memory location of the object, as the value of the new expression.
6. Store that object reference in the harrysChecking variable.

In general, when you implement constructors, be sure that each constructor initializes all instance variables, and that you make use of all parameter variables (see Common Error 3.2 on page 96).

A constructor is like a set of assembly instructions for an object.
3.3 Providing the Class Implementation

3.3.3 Providing Methods

In this section, we finish implementing the methods of the BankAccount class.
When you implement a method, ask yourself whether it is an accessor or mutator method. A mutator method needs to update the instance variables in some way. An accessor method retrieves or computes a result.

Here is the deposit method. It is a mutator method, updating the balance:

```java
public void deposit(double amount) {
    balance = balance + amount;
}
```

The withdraw method is very similar to the deposit method:

```java
public void withdraw(double amount) {
    balance = balance - amount;
}
```
Table 1 Implementing Classes

<table>
<thead>
<tr>
<th>Example</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>public class BankAccount { ... }</td>
<td>This is the start of a class declaration. Instance variables, methods, and constructors are placed inside the braces.</td>
</tr>
<tr>
<td>private double balance;</td>
<td>This is an instance variable of type double. Instance variables should be declared as private.</td>
</tr>
<tr>
<td>public double getBalance() { ... }</td>
<td>This is a method declaration. The body of the method must be placed inside the braces.</td>
</tr>
<tr>
<td>... { return balance; }</td>
<td>This is the body of the getBalance method. The return statement returns a value to the caller of the method.</td>
</tr>
<tr>
<td>public void deposit(double amount) { ... }</td>
<td>This is a method with a parameter variable (amount). Because the method is declared as void, it has no return value.</td>
</tr>
<tr>
<td>... { balance = balance + amount; }</td>
<td>This is the body of the deposit method. It does not have a return statement.</td>
</tr>
<tr>
<td>public BankAccount() { ... }</td>
<td>This is a constructor declaration. A constructor has the same name as the class and no return type.</td>
</tr>
<tr>
<td>... { balance = 0; }</td>
<td>This is the body of the constructor. A constructor should initialize the instance variables.</td>
</tr>
</tbody>
</table>

There is one method left, getBalance. Unlike the deposit and withdraw methods, which modify the instance variable of the object on which they are invoked, the getBalance method returns a value:

```
public double getBalance()
{
    return balance;
}
```

We have now completed the implementation of the BankAccount class—see the code listing below. There is only one step remaining: testing that the class works correctly. That is the topic of the next section.

section_3/BankAccount.java

```java
/**
 * A bank account has a balance that can be changed by deposits and withdrawals.
 */
public class BankAccount
{
    private double balance;

    /**
     * Constructs a bank account with a zero balance.
     */
    public BankAccount()
    {
        balance = 0;
    }
```
11. Suppose we modify the BankAccount class so that each bank account has an account number. How does this change affect the instance variables?

12. Why does the following code not succeed in robbing mom's bank account?

```java
public class BankRobber
{
    public static void main(String[] args)
    {
        BankAccount momsSavings = new BankAccount(1000);
        momsSavings.balance = 0;
    }
}
```

13. The Rectangle class has four instance variables: x, y, width, and height. Give a possible implementation of the getWidth method.

14. Give a possible implementation of the translate method of the Rectangle class.

Practice It  Now you can try these exercises at the end of the chapter: R3.4, R3.10, E3.7.
Ignoring Parameter Variables

A surprisingly common beginner’s error is to ignore parameter variables of methods or constructors. This usually happens when an assignment gives an example with specific values. For example, suppose you are asked to provide a class Letter with a recipient and a sender, and you are given a sample letter like this:

Dear John:
I am sorry we must part.
I wish you all the best.

Sincerely,
Mary

Now look at this incorrect attempt:

```java
public class Letter
{
    private String recipient;
    private String sender;

    public Letter(String aRecipient, String aSender)
    {
        recipient = "John"; // Error—should use parameter variable
        sender = "Mary"; // Same error
    }
}
```

The constructor ignores the names of the recipient and sender arguments that were provided to the constructor. If a user constructs a

```java
new Letter("John", "Yoko")
```

the sender is still set to "Mary", which is bound to be embarrassing.

The constructor should use the parameter variables, like this:

```java
public Letter(String aRecipient, String aSender)
{
    recipient = aRecipient;
    sender = aSender;
}
```
Step 1  Find out which methods you are asked to supply.
      In a simulation, you won’t have to provide every feature that occurs in the real world—there
      are too many. In the cash register example, we don’t deal with sales tax or credit card payments.
      The assignment tells you which aspects of the self-service cash register your class should simu-
      late. Make a list of them:
      • Process the price of each purchased item.
      • Receive payment.
      • Calculate the amount of change due to the customer.

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:
      
      CashRegister register = new CashRegister();
      register.recordPurchase(29.95);
      register.recordPurchase(9.95);
      register.receivePayment(50);
      double change = register.giveChange();

      Now we have a specific list of methods:
      • public void recordPurchase(double amount)
      • public void receivePayment(double amount)
      • public double giveChange()

      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 the case of the cash register example, we can get by with a single constructor that creates
      an empty register. A more realistic cash register might start out with some coins and bills so
      that we can give exact change, but that is well beyond the scope of our assignment.
      Thus, we add a single constructor:
      • public CashRegister()

Step 3  Document the public interface.
      Here is the documentation, with comments, that describes the class and its methods:
      
      /**
       * A cash register totals up sales and computes change due.
       */
      public class CashRegister
      {
        /**
         * Constructs a cash register with no money in it.
         */
        public CashRegister()
        {
        }

        /**
         * Records the sale of an item.
         * @param amount the price of the item
         */
        public void recordPurchase(double amount)
        {
Chapter 3  Implementing Classes

Step 4  Determine instance variables.

Ask yourself what information an object needs to store to do its job. Remember, the methods can be called in any order. The object needs to have enough internal memory to be able to process every method using just its instance variables and the parameter variables. Go through each method, perhaps starting with a simple one or an interesting one, and ask yourself what you need to carry out the method’s task. Make instance variables to store the information that the method needs.

Just as importantly, don’t introduce unnecessary instance variables (see Common Error 3.3 on page 106). If a value can be computed from other instance variables, it is generally better to compute it on demand than to store it.

In the cash register example, you need to keep track of the total purchase amount and the payment. You can compute the change due from these two amounts.

```java
public class CashRegister
{
    private double purchase;
    private double payment;
    . . .
}
```

Step 5  Implement constructors and methods.

Implement the constructors and methods in your class, one at a time, starting with the easiest ones. Here is the implementation of the `recordPurchase` method:

```java
public void recordPurchase(double amount)
{
    purchase = purchase + amount;
}
```

The `receivePayment` method looks almost the same,

```java
public void receivePayment(double amount)
{
    payment = payment + amount;
}
```

but why does the method add the amount, instead of simply setting `payment = amount`? A customer might provide two separate payments, such as two $10 bills, and the machine must process them both. Remember, methods can be called more than once, and they can be called in any order.
Finally, here is the giveChange method. This method is a bit more sophisticated—it computes the change due, and it also resets the cash register for the next sale.

```java
public double giveChange()
{
    double change = payment - purchase;
    purchase = 0;
    payment = 0;
    return change;
}
```

If you find that you have trouble with the implementation, 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 reflect the state of an object. Don’t hesitate to go back and add or modify instance variables.

You can find the complete implementation in the how_to_1 directory of the book’s companion code.

**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.

```java
public class CashRegisterTester
{
    public static void main(String[] args)
    {
        CashRegister register = new CashRegister();

        register.recordPurchase(29.50);
        register.recordPurchase(9.25);
        register.receivePayment(50);
        double change = register.giveChange();

        System.out.println(change);
        System.out.println("Expected: 11.25");
    }
}
```

The output of this test program is:

```
11.25
Expected: 11.25
```
3.4 Unit Testing

In the preceding section, we completed the implementation of the BankAccount class. What can you do with it? Of course, you can compile the file BankAccount.java. However, you can’t execute the resulting BankAccount.class file. It doesn’t contain a main method. That is normal—most classes don’t contain 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.

To test your class, you have two choices. Some interactive development environments have commands for constructing objects and invoking methods (see Special Topic 2.1). Then you can test a class simply by constructing an object, calling methods, and verifying that you get the expected return values. Figure 6 shows the result of calling the getBalance method on a BankAccount 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. As discussed in Section 2.7, 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.
The `MoveTester` class in Section 2.7 is a good example of a tester class. That class runs methods of the `Rectangle` class—a class in the Java library.

Following is a class to run methods of the `BankAccount` class. The `main` method constructs an object of type `BankAccount`, invokes the `deposit` and `withdraw` methods, and then displays the remaining balance on the console.

We also print the value that we expect to see. In our sample program, we deposit $2,000 and withdraw $500. We therefore expect a balance of $1,500.

```java
subsection_4/BankAccountTester.java
1 /**
2  * A class to test the `BankAccount` class.
3  */
4 public class BankAccountTester
5 {
6  /**
7   * Tests the methods of the `BankAccount` class.
8   * @param args not used
9  */
10  public static void main(String[] args)
11  {
12    BankAccount harrysChecking = new BankAccount();
13    harrysChecking.deposit(2000);
14    harrysChecking.withdraw(500);
15    System.out.println(harrysChecking.getBalance());
16    System.out.println("Expected: 1500");
17  }
18 }
```

**Program Run**

```
1500
Expected: 1500
```

To produce a program, you need to combine the `BankAccount` and the `BankAccountTester` 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 `BankAccount` class describes objects that compute bank balances. The `BankAccountTester` class runs a test that puts a `BankAccount` object through its paces.

**SELF CHECK**

**15.** When you run the `BankAccountTester` program, how many objects of class `BankAccount` are constructed? How many objects of type `BankAccountTester`?

**16.** Why is the `BankAccountTester` 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: E3.6, E3.13.
Computing & Society 3.1 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.

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://verifiablevoting.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 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.
Researchers have studied why some students have an easier time learning how to program than others. One important skill of successful programmers is the ability to simulate the actions of a program with pencil and paper. In this section, you will see how to develop this skill by tracing method calls on objects.

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:

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 on the back.

When an object is constructed, fill in the initial values of the instance variables:

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 recordPurchase method:
If you have more than one object in your program, you will have multiple cards, one for each object:

<table>
<thead>
<tr>
<th>reg1.purchase</th>
<th>reg1.payment</th>
</tr>
</thead>
<tbody>
<tr>
<td>19.95</td>
<td>0.0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>reg2.purchase</th>
<th>reg2.payment</th>
</tr>
</thead>
<tbody>
<tr>
<td>29.50</td>
<td>9.25</td>
</tr>
<tr>
<td>50.00</td>
<td>0.0</td>
</tr>
</tbody>
</table>

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 methods recordTaxablePurchase and 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 value that can be used in a computation must be

- An instance variable.
- A method argument.
- A static variable (uncommon; see Section 8.4).

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.

```java
CashRegister reg3 = new CashRegister(7.5); // 7.5 percent sales tax
reg3.recordPurchase(3.95); // Not taxable
reg3.recordTaxablePurchase(19.95); // Taxable
```

When you record the effect on a card, it looks like this:

<table>
<thead>
<tr>
<th>reg3.purchase</th>
<th>reg3.taxablePurchase</th>
<th>reg3.payment</th>
<th>reg3.taxRate</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.95</td>
<td>0</td>
<td>0</td>
<td>7.5</td>
</tr>
<tr>
<td>19.95</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

With this information, we can compute the tax. It is `taxablePurchase * taxRate / 100`. Tracing the object helped us understand the need for additional instance variables.

**SELF CHECK**

17. 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.
18. Trace the following method calls:
   
   ```java
   Car myCar = new Car(25);
   myCar.addGas(20);
   myCar.drive(100);
   myCar.drive(200);
   myCar.addGas(5);
   ```

19. 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’s result.

20. Trace the methods of Self Check 18, updating the instance variable that you added in Self Check 19.

**Practice It**  
Now you can try these exercises at the end of the chapter: R3.19, R3.20, R3.21.

### 3.6 Local Variables

In this section, we discuss the behavior of *local* variables. A **local variable** is a variable that is declared in the body of a method. For example, the `giveChange` method in How To 3.1 declares a local variable change:

```java
public double giveChange()
{
    double change = payment - purchase;
    purchase = 0;
    payment = 0;
    return change;
}
```

Parameter variables are similar to local variables, but they are declared in method headers. For example, the following method declares a parameter variable `amount`:

```java
public void receivePayment(double amount)
```

Local and parameter variables belong to methods. When a method runs, its local and parameter variables come to life. When the method exits, they are removed immediately. For example, if you call `register.giveChange()`, then a variable `change` is created. When the method exits, that variable is removed.

In contrast, instance variables belong to objects, not methods. When an object is constructed, its instance variables are created. The instance variables stay alive until no method uses the object any longer. (The Java virtual machine contains an agent called a **garbage collector** that periodically reclaims objects when they are no longer used.)

An important difference between instance variables and local variables is initialization. You must **initialize** all local variables. If you don’t initialize a local variable, the compiler complains when you try to use it. (Note that parameter variables are initialized when the method is called.)

Instance variables are initialized with a default value before a constructor is invoked. Instance variables that are numbers are initialized to 0. Object references are set to a special value called `null`. If an object reference is `null`, then it refers to no object at all. We will discuss the `null` value in greater detail in Section 5.2.5.
21. What do local variables and parameter variables have in common? In which essential aspect do they differ?

22. Why was it necessary to introduce the local variable change in the giveChange method? That is, why didn’t the method simply end with the statement
   return payment - purchase;

23. Consider a CashRegister object reg1 whose payment instance variable has the value 20 and whose purchase instance variable has the value 19.5. Trace the call
   reg1.giveChange(). Include the local variable change. Draw an X in its column
   when the variable ceases to exist.

Practice It  Now you can try these exercises at the end of the chapter: R3.15, R3.16.

Duplicating Instance Variables in Local Variables

Beginning programmers commonly add types to assignment statements, thereby changing them into local variable declarations. For example,

```java
public double giveChange()
{
    double change = payment - purchase;
    double purchase = 0; // ERROR! This declares a local variable.
    double payment = 0;  // ERROR! The instance variable is not updated.
    return change;
}
```

Another common error is to declare a parameter variable with the same name as an instance variable. For example, consider this BankAccount constructor:

```java
public BankAccount(double balance)
{
    balance = balance; // ERROR! Does not set the instance variable
}
```

This constructor simply sets the parameter variable to itself, leaving it unchanged. A simple remedy is to come up with a different name for the parameter variable:

```java
public BankAccount(double initialBalance)
{
    balance = initialBalance; // OK
}
```

Providing Unnecessary Instance Variables

A common beginner’s mistake is to use instance variables when local variables would be more appropriate. For example, consider the change variable of the giveChange method. It is not needed anywhere else—that’s why it is local to the method. But what if it had been declared as an instance variable?

```java
public class CashRegister {
    private double purchase;
    private double payment;
    private double change; // Not appropriate
```
3.7 The this Reference

When you call a method, you pass two kinds of inputs to the method:

- The object on which you invoke the method
- The method arguments
For example, when you call

```java
momsSavings.deposit(500);
```

the deposit method needs to know the account object (momsSavings) as well as the amount that is being deposited (500).

When you implement the method, you provide a parameter variable for each argument. But you don’t need to provide a parameter variable for the object on which the method is being invoked. That object is called the **implicit parameter**. All other parameter variables (such as the amount to be deposited in our example) are called **explicit parameters**.

Look again at the code of the deposit method:

```java
public void deposit(double amount)
{
    balance = balance + amount;
}
```

Here, `amount` is an explicit parameter. You don’t see the implicit parameter—that is why it is called “implicit”. But consider what `balance` means exactly. After all, our program may have multiple `BankAccount` objects, and each of them has its own balance.

Because we are depositing the money into `momsSavings`, `balance` must mean `momsSavings.balance`. In general, when you refer to an instance variable inside a method, it means the instance variable of the implicit parameter.

In any method, you can access the implicit parameter—the object on which the method is called—with the reserved word `this`. For example, in the preceding method invocation, `this` refers to the same object as `momsSavings` (see Figure 7).

The statement

```java
balance = balance + amount;
```

actually means

```java
this.balance = this.balance + amount;
```

When you refer to an instance variable in a method, the compiler automatically applies it to the `this` reference. Some programmers actually prefer to manually insert the `this` reference before every instance variable because they find it makes the code clearer. Here is an example:

```java
public BankAccount(double initialBalance)
{
    this.balance = initialBalance;
}
```

You may want to try it out and see if you like that style.

---

**Figure 7** The Implicit Parameter of a Method Call
The `this` reference can also be used to distinguish between instance variables and local or parameter variables. Consider the constructor:

```java
public BankAccount(double balance) {
    this.balance = balance;
}
```

The expression `this.balance` clearly refers to the `balance` instance variable. However, the expression `balance` by itself seems ambiguous. It could denote either the parameter variable or the instance variable. The Java language specifies that in this situation the local variable wins out. It “shadows” the instance variable. Therefore,

```java
this.balance = balance;
```

means: “Set the instance variable `balance` to the parameter variable `balance`”.

There is another situation in which it is important to understand implicit parameters. Consider the following modification to the `BankAccount` class. We add a method to apply the monthly account fee:

```java
public class BankAccount {
    . . .
    public void monthlyFee() {
        withdraw(10); // Withdraw $10 from this account
    }
}
```

That means to withdraw from the same bank account object that is carrying out the `monthlyFee` operation. In other words, the implicit parameter of the `withdraw` method is the (invisible) implicit parameter of the `monthlyFee` method.

If you find it confusing to have an invisible parameter, you can use the `this` reference to make the method easier to read:

```java
public class BankAccount {
    . . .
    public void monthlyFee() {
        this.withdraw(10); // Withdraw $10 from this account
    }
}
```

You have now seen how to use objects and implement classes, and you have learned some important technical details about variables and method parameters. The remainder of this chapter continues the optional graphics track. In the next chapter, you will learn more about the most fundamental data types of the Java language.

24. How many implicit and explicit parameters does the `withdraw` method of the `BankAccount` class have, and what are their names and types?

25. In the `deposit` method, what is the meaning of `this.amount`? Or, if the expression has no meaning, why not?

26. How many implicit and explicit parameters does the `main` method of the `BankAccountTester` class have, and what are they called?

Practice It   Now you can try these exercises at the end of the chapter: R3.11, R3.13.
### Calling One Constructor from Another

Consider the `BankAccount` class. It has two constructors: a no-argument constructor 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:

```java
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 the same class.

### 3.8 Shape Classes

In this section, we continue the optional graphics track by discussing how to organize complex drawings in a more object-oriented fashion.

When you produce a drawing that has multiple shapes, or parts made of multiple shapes, such as the car in Figure 8, it is a good idea to make a separate class for each part. The class should have a `draw` method that draws the shape, and a constructor to set the position of the shape. For example, here is the outline of the `Car` class:

```java
public class Car
{
  public Car(int x, int y)
  {
    Remember position.
    ...
  }

  public void draw(Graphics2D g2)
  {
    Drawing instructions.
    ...
  }
}
```

You will find the complete class declaration at the end of this section. The `draw` method contains a rather long sequence of instructions for drawing the body, roof, and tires.
The coordinates of the car parts seem a bit arbitrary. To come up with suitable values, draw the image on graph paper and read off the coordinates (Figure 9).

The program that produces Figure 8 is composed of three classes.

- The Car class is responsible for drawing a single car. Two objects of this class are constructed, one for each car.
- The CarComponent class displays the drawing.
- The CarViewer class shows a frame that contains a CarComponent.

Let us look more closely at the CarComponent class. The paintComponent method draws two cars. We place one car in the top-left corner of the window, and the other car in the bottom-right corner. To compute the bottom-right position, we call the getWidth and getHeight methods of the JComponent class. These methods return the dimensions of the component. We subtract the dimensions of the car to determine the position of car2:

```java
Car car1 = new Car(0, 0);
int x = getWidth() - 60;
int y = getHeight() - 30;
Car car2 = new Car(x, y);
```
Pay close attention to the call to `getWidth` inside the `paintComponent` method of `CarComponent`. The method call has no implicit parameter, which means that the method is applied to the same object that executes the `paintComponent` method. The component simply obtains its own width.

Run the program and resize the window. Note that the second car always ends up at the bottom-right corner of the window. Whenever the window is resized, the `paintComponent` method is called and the car position is recomputed, taking the current component dimensions into account.

**section_8/CarComponent.java**

```java
import java.awt.Graphics;
import java.awt.Graphics2D;
import javax.swing.JComponent;

/**
 * This component draws two car shapes.
 */
public class CarComponent extends JComponent {
    public void paintComponent(Graphics g) {
        Graphics2D g2 = (Graphics2D) g;
        Car car1 = new Car(0, 0);
        int x = getWidth() - 60;
        int y = getHeight() - 30;
        Car car2 = new Car(x, y);
        car1.draw(g2);
        car2.draw(g2);
    }
}
```

**section_8/Car.java**

```java
import java.awt.Graphics2D;
import java.awt.Rectangle;
import java.awt.geom.Ellipse2D;
import java.awt.geom.Line2D;
import java.awt.geom.Point2D;

/**
 * A car shape that can be positioned anywhere on the screen.
 */
public class Car {
    private int xLeft;
    private int yTop;

    /**
     * Constructs a car with a given top left corner.
     * @param x the x-coordinate of the top-left corner
     * @param y the y-coordinate of the top-left corner
     */
    public Car(int x, int y) {
        
    }
```
```java
section_8/CarViewer.java

import javax.swing.JFrame;

public class CarViewer {
    public static void main(String[] args) {
        JFrame frame = new JFrame();
        frame.setSize(300, 400);
        frame.setTitle("Two cars");
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        CarComponent component = new CarComponent();
        frame.add(component);
        frame.setVisible(true);
    }
}
```

```java
public class CarComponent {
    // The bottom of the front windshield
    Point2D.Double r1 = new Point2D.Double(xLeft + 10, yTop + 10);
    // The front of the roof
    Point2D.Double r2 = new Point2D.Double(xLeft + 20, yTop);
    // The rear of the roof
    Point2D.Double r3 = new Point2D.Double(xLeft + 40, yTop);
    // The bottom of the rear windshield
    Point2D.Double r4 = new Point2D.Double(xLeft + 50, yTop + 10);
    Line2D.Double frontWindshield = new Line2D.Double(r1, r2);
    Line2D.Double roofTop = new Line2D.Double(r2, r3);
    Line2D.Double rearWindshield = new Line2D.Double(r3, r4);

    draw(Graphics2D g2) {
        Rectangle body = new Rectangle(xLeft, yTop + 10, 60, 10);
        Ellipse2D.Double frontTire = new Ellipse2D.Double(xLeft + 10, yTop + 20, 10, 10);
        Ellipse2D.Double rearTire = new Ellipse2D.Double(xLeft + 40, yTop + 20, 10, 10);

        g2.draw(body);
        g2.draw(frontTire);
        g2.draw(rearTire);
        g2.draw(frontWindshield);
        g2.draw(roofTop);
        g2.draw(rearWindshield);
    }
}
```
27. Which class needs to be modified to have the two cars positioned next to each other?

28. Which class needs to be modified to have the car tires painted in black, and what modification do you need to make?

29. How do you make the cars twice as big?

Practice It  Now you can try these exercises at the end of the chapter: E3.19, E3.24.

HOW TO 3.2 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 that draws 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 flags consist of three equally wide sections of different colors, side by side.

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 start and end points.
- 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 · 2 / 3 = 67, which seems more awkward.)

Now you can compute the coordinates of all the important points of the shape:

![Diagram of coordinates for flag shapes]

Step 3  Write Java statements to draw the shapes.

In our example, there are two rectangles and two lines:

```java
Rectangle leftRectangle = new Rectangle(100, 100, 30, 60);
Rectangle rightRectangle = new Rectangle(160, 100, 30, 60);
Line2D.Double topLine = new Line2D.Double(130, 100, 160, 100);
Line2D.Double bottomLine = new Line2D.Double(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:

```java
Rectangle leftRectangle = new Rectangle(
xLeft, yTop,
width / 3, width * 2 / 3);
Rectangle rightRectangle = new Rectangle(
xLeft + 2 * width / 3, yTop,
width / 3, width * 2 / 3);
Line2D.Double topLine = new Line2D.Double(
xLeft + width / 3, yTop,
xLeft + width * 2 / 3, yTop);
Line2D.Double bottomLine = new Line2D.Double(
xLeft + width / 3, yTop + width * 2 / 3,
xLeft + width * 2 / 3, yTop + width * 2 / 3);
```
Now you need to fill the rectangles and draw the lines. For the flag of Italy, the left rectangle is green and the right rectangle is red. Remember to switch colors before the filling and drawing operations:

```java
g2.setColor(Color.GREEN);
g2.fill(leftRectangle);
g2.setColor(Color.RED);
g2.fill(rightRectangle);
g2.setColor(Color.BLACK);
g2.draw(topLine);
g2.draw(bottomLine);
```

**Step 4** Combine the drawing statements with the component “plumbing”.

```java
public class MyComponent extends JComponent
{
    public void paintComponent(Graphics g)
    {
        Graphics2D g2 = (Graphics2D) g;
        // Drawing instructions.
        ...
    }
}
```

In our simple example, you could add all shapes and drawing instructions inside the `paintComponent` method:

```java
public class ItalianFlagComponent extends JComponent
{
    public void paintComponent(Graphics g)
    {
        Graphics2D g2 = (Graphics2D) g;
        Rectangle leftRectangle = new Rectangle(100, 100, 30, 60);
        //...
        g2.setColor(Color.GREEN);
        g2.fill(leftRectangle);
        //...
    }
}
```

That approach is acceptable for simple drawings, but it is not very object-oriented. After all, a flag is an object. It is better to make a separate class for the flag. Then you can draw different flags at different positions. Specify the sizes in a constructor and supply a `draw` method:

```java
public class ItalianFlag
{
    private int xLeft;
    private int yTop;
    private int width;

    public ItalianFlag(int x, int y, int aWidth)
    {
        xLeft = x;
        yTop = y;
        width = aWidth;
    }

    public void draw(Graphics2D g2)
    {
        Rectangle leftRectangle = new Rectangle(  
            xLeft, yTop,  
            width / 3, width * 2 / 3);
    }
```
.. .

g2.setColor(Color.GREEN);
g2.fill(leftRectangle);

}

} You still need a separate class for the component, but it is very simple:

public class ItalianFlagComponent extends JComponent
{
    public void paintComponent(Graphics g)
    {
        Graphics2D g2 = (Graphics2D) g;
        ItalianFlag flag = new ItalianFlag(100, 100, 90);
        flag.draw(g2);
    }

}

Step 5 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);

        ItalianFlagComponent component = new ItalianFlagComponent();
        frame.add(component);
        frame.setVisible(true);
    }

}

CHAPTER SUMMARY

Understand instance variables and the methods that access them.

- An object’s instance variables store the data required for executing its methods.
- Each object of a class has its own set of instance variables.
- Private instance variables can only be accessed by methods of the same class.
- Encapsulation is the process of hiding implementation details and providing methods for data access.
- Encapsulation allows a programmer to use a class without having to know its implementation.
- Information hiding makes it simpler for the implementor of a class to locate errors and change implementations.
Write method and constructor headers that describe the public interface of a class.

- In order to implement a class, you first need to know which methods are required.
- Constructors set the initial data for objects.
- The constructor name is always the same as the class name.
- Use documentation comments to describe the classes and public methods of your programs.
- Provide documentation comments for every class, every method, every parameter variable, and every return value.

Implement a class.

- The private implementation of a class consists of instance variables, and the bodies of constructors and methods.

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.

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.

Compare initialization and lifetime of instance, local, and parameter variables.

- Local variables are declared in the body of a method.
- When a method exits, its local variables are removed.
- Instance variables are initialized to a default value, but you must initialize local variables.

Recognize the use of the implicit parameter in method declarations.

- Use of an instance variable name in a method denotes the instance variable of the implicit parameter.
- The this reference denotes the implicit parameter.
- A local variable shadows an instance variable with the same name. You can access the instance variable name through the this reference.
- A method call without an implicit parameter is applied to the same object.

Implement classes that draw graphical shapes.

- It is a good idea to make a class for any part of a drawing that can occur more than once.
- To figure out how to draw a complex shape, make a sketch on graph paper.
• R3.1 What is the public interface of the Counter class in Section 3.1? How does it differ from the implementation of the class?

• R3.2 What is encapsulation? Why is it useful?

• R3.3 Instance variables are a part of the hidden implementation of a class, but they aren’t actually hidden from programmers who have the source code of the class. Explain to what extent the private reserved word provides information hiding.

• R3.4 Consider a class Grade that represents a letter grade, such as A+ or B. Give two choices of instance variables that can be used for implementing the Grade class.

• R3.5 Consider a class Time that represents a point in time, such as 9 A.M. or 3:30 P.M. Give two different sets of instance variables that can be used for implementing the Time class.

• R3.6 Suppose the implementor of the Time class of Exercise R3.5 changes from one implementation strategy to another, keeping the public interface unchanged. What do the programmers who use the Time class need to do?

• R3.7 You can read the value instance variable of the Counter class with the getValue accessor method. Should there be a setValue mutator method to change it? Explain why or why not.

• R3.8 a. Show that the BankAccount(double initialBalance) constructor is not strictly necessary. That is, if we removed that constructor from the public interface, how could a programmer still obtain BankAccount objects with an arbitrary balance?

   b. Conversely, could we keep only the BankAccount(double initialBalance) constructor and remove the BankAccount() constructor?

• R3.9 Why does the BankAccount class not have a reset method?

• R3.10 What happens in our implementation of the BankAccount class when more money is withdrawn from the account than the current balance?

• R3.11 What is the this reference? Why would you use it?

• R3.12 Which of the methods in the CashRegister class of Worked Example 3.1 are accessor methods? Which are mutator methods?

• R3.13 What does the following method do? Give an example of how you can call the method.

```java
public class BankAccount
{
    public void mystery(BankAccount that, double amount)
    {
        this.balance = this.balance - amount;
        that.balance = that.balance + amount;
    }
    . . . // Other bank account methods
}
```

• R3.14 Suppose you want to implement a class TimeDepositAccount. A time deposit account has a fixed interest rate that should be set in the constructor, together with the initial
balance. Provide a method to get the current balance. Provide a method to add the earned interest to the account. This method should have no arguments because the interest rate is already known. It should have no return value because you already provided a method for obtaining the current balance. It is not possible to deposit additional funds into this account. Provide a withdraw method that removes the entire balance. Partial withdrawals are not allowed.

R3.15 Consider the following implementation of a class Square:

```java
public class Square {
    private int sideLength;
    private int area; // Not a good idea

    public Square(int length) {
        sideLength = length;
    }

    public int getArea() {
        area = sideLength * sideLength;
        return area;
    }
}
```

Why is it not a good idea to introduce an instance variable for the area? Rewrite the class so that area is a local variable.

R3.16 Consider the following implementation of a class Square:

```java
public class Square {
    private int sideLength;
    private int area;

    public Square(int initialLength) {
        sideLength = initialLength;
        area = sideLength * sideLength;
    }

    public int getArea() { return area; }
    public void grow() { sideLength = 2 * sideLength; }
}
```

What error does this class have? How would you fix it?

Testing R3.17 Provide a unit test class for the Counter class in Section 3.1.

Testing R3.18 Read Exercise E3.12, but do not implement the Car class yet. Write a tester class that tests a scenario in which gas is added to the car, the car is driven, more gas is added, and the car is driven again. Print the actual and expected amount of gas in the tank.

R3.19 Using the object tracing technique described in Section 3.5, trace the program at the end of Section 3.4.

R3.20 Using the object tracing technique described in Section 3.5, trace the program in How To 3.1.

R3.21 Using the object tracing technique described in Section 3.5, trace the program in Worked Example 3.1.
*** R3.22  Design a modification of the BankAccount class 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 3.5, trace a scenario that shows how the fees are computed over two months.

>>> Graphics R3.23  Suppose you want to extend the car viewer program in Section 3.8 to show a suburban scene, with several cars and houses. Which classes do you need?

>>> Graphics R3.24  Explain why the calls to the getWidth and getHeight methods in the CarComponent class have no explicit parameter.

>>> Graphics R3.25  How would you modify the Car class in order to show cars of varying sizes?

---

PRACTICE EXERCISES

- E3.1  We want to add a button to the tally counter in Section 3.1 that allows an operator to undo an accidental button click. Provide a method
  
  ```java
  public void undo()
  ```
  that simulates such a button. As an added precaution, make sure that clicking the undo button more often than the click button has no effect. (*Hint:* The call `Math.max(n, 0)` returns `n` if `n` is greater than zero, zero otherwise.)

- E3.2  Simulate a tally counter that can be used to admit a limited number of people. First, the limit is set with a call
  
  ```java
  public void setLimit(int maximum)
  ```
  If the click button is clicked more often than the limit, it has no effect. (*Hint:* The call `Math.min(n, limit)` returns `n` if `n` is less than `limit`, and `limit` otherwise.)

- E3.3  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
  
  ```java
  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()
  ```

- Testing E3.4  Write a CircuitTester class that tests all switch combinations in Exercise E3.3, printing out actual and expected states for the switches and lamps.

- E3.5  Change the public interface of the circuit class of Exercise E3.3 so that it has the following methods:
  
  ```java
  public int getSwitchState(int switch)
  public int getLampState()  
  public void toggleSwitch(int switch)
  ```
  Provide an implementation using only language features that have been introduced. The challenge is to find a data representation from which to recover the switch states.

- Testing E3.6  Write a BankAccountTester class whose main method constructs a bank account, deposits $1,000, withdraws $500, withdraws another $400, and then prints the remaining balance. Also print the expected result.
• E3.7 Add a method
  
  ```java
  public void addInterest(double rate)
  ```
  
  to the BankAccount class that adds interest at the given rate. For example, after the statements
  
  ```java
  BankAccount momsSavings = new BankAccount(1000);
  momsSavings.addInterest(10); // 10 percent interest
  ```
  
  the balance in momsSavings is $1,100. Also supply a BankAccountTester class that prints the actual and expected balance.

• E3.8 Write a class SavingsAccount that is similar to the BankAccount class, except that it has an added instance variable interest. Supply a constructor that sets both the initial balance and the interest rate. Supply a method addInterest (with no explicit parameter) that adds interest to the account. Write a SavingsAccountTester class that constructs a savings account with an initial balance of $1,000 and an interest rate of 10 percent. Then apply the addInterest method and print the resulting balance. Also compute the expected result by hand and print it.

•• E3.9 Add a method printReceipt to the CashRegister class. The method should print the prices of all purchased items and the total amount due. Hint: You will need to form a string of all prices. Use the concat method of the String class to add additional items to that string. To turn a price into a string, use the call String.valueOf(price).

• E3.10 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 reset that resets any counters and totals so that the next day’s sales start from zero.

•• E3.11 Implement a class Employee. An employee has a name (a string) and a salary (a double). Provide a constructor with two arguments

  ```java
  public Employee(String employeeName, double currentSalary)
  ```

  and methods

  ```java
  public String getName()
  public double getSalary()
  public void raiseSalary(double byPercent)
  ```

  These methods return the name and salary, and raise the employee’s salary by a certain percentage. Sample usage:

  ```java
  Employee harry = new Employee("Hacker, Harry", 50000);
  harry.raiseSalary(10); // Harry gets a 10 percent raise
  ```

  Supply an EmployeeTester class that tests all methods.

•• E3.12 Implement a class Car with the following properties. A car has a certain fuel efficiency (measured in miles/gallon or liters/km—pick one) 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 amount of gasoline in the fuel tank. Also supply methods getGasInTank, returning the current amount of gasoline in the fuel tank, and addGas, to add gasoline to the fuel tank. Sample usage:

  ```java
  Car myHybrid = new Car(50); // 50 miles per gallon
  myHybrid.addGas(20); // Tank 20 gallons
  ```
myHybrid.drive(100);  // Drive 100 miles
double gasLeft = myHybrid.getGasInTank();  // Get gas remaining in tank

You may assume that the drive method is never called with a distance that consumes more than the available gas. Supply a CarTester class that tests all methods.

**E3.13** Implement a class Product. A product has a name and a price, for example new Product("Toaster", 29.95). Supply methods getName, getPrice, and reducePrice. Supply a program ProductPrinter that makes two products, prints each name and price, reduces their prices by $5.00, and then prints the prices again.

**E3.14** 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 class LetterPrinter 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.  

*Hints:* (1) Use the concat method to form a longer string from two shorter strings. (2) The special string "\n" represents a new line. For example, the statement

```
body = body.concat("Sincerely,\n").concat("\n");
```

adds a line containing the string "Sincerely," to the body.

**E3.15** 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)
```
Implementing Classes and methods

```java
public void turn()
public void move()
public int getPosition()
```

Sample usage:

```java
Bug bugsy = new Bug(10);
bugsy.move(); // Now the position is 11
bugsy.turn();
bugsy.move(); // Now the position is 10
```

Your BugTester should construct a bug, make it move and turn a few times, and print the actual and expected position.

**E3.16** Implement a class Moth that models a moth flying along a straight line. The moth has a position, which is 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

```java
public Moth(double initialPosition)
```

and methods

```java
public void moveToLight(double lightPosition)
public double getPosition()
```

Your MothTester should construct a moth, move it toward a couple of light sources, and check that the moth’s position is as expected.

**Graphics E3.17** 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. Call the `getWidth` and `getHeight` methods of the `JComponent` class in the `paintComponent` method.

**Graphics E3.18** Draw a shooting target—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 Target, TargetComponent, and TargetViewer.

**Graphics E3.19** Write a program that draws a picture of a house. It could be as simple as the accompanying figure, or if you like, more elaborate (3-D, skyscraper, marble columns in the entryway, whatever). Implement a class House and supply a method `draw(Graphics2D g2)` that draws the house.

**Graphics E3.20** Extend Exercise E3.19 by supplying a House constructor for specifying the position and size. Then populate your screen with a few houses of different sizes.

**Graphics E3.21** Change the car viewer program in Section 3.8 to make the cars appear in different colors. Each Car object should store its own color. Supply modified Car and CarComponent classes.

**Graphics E3.22** Change the Car class so that the size of a car can be specified in the constructor. Change the CarComponent class to make one of the cars appear twice the size of the original example.

**Graphics E3.23** 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.
**Graphics E3.24** Write a program that displays the Olympic rings. Color the rings in the Olympic colors. Provide classes `OlympicRing`, `OlympicRingViewer`, and `OlympicRingComponent`.

**Graphics E3.25** Make a bar chart to plot the following data set. Label each bar. Make the bars horizontal for easier labeling. Provide a class `BarChartViewer` and a class `BarChartComponent`.

<table>
<thead>
<tr>
<th>Bridge Name</th>
<th>Longest Span (ft)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Golden Gate</td>
<td>4,200</td>
</tr>
<tr>
<td>Brooklyn</td>
<td>1,595</td>
</tr>
<tr>
<td>Delaware Memorial</td>
<td>2,150</td>
</tr>
<tr>
<td>Mackinac</td>
<td>3,800</td>
</tr>
</tbody>
</table>

**P3.1** Enhance the `CashRegister` class so that it counts the purchased items. Provide a `getCount` method that returns the count.

**P3.2** Support computing sales tax in the `CashRegister` class. The tax rate should be supplied when constructing a `CashRegister` object. Add `recordTaxablePurchase` and `getTotalTax` methods. (Amounts added with `recordPurchase` are not taxable.) The `giveChange` method should correctly reflect the sales tax that is charged on taxable items.

**P3.3** Implement a class `Balloon`. A balloon starts out with radius 0. Supply a method

```java
public void inflate(double amount)
```

that increases the radius by the given amount. Supply a method

```java
public double getVolume()
```

that returns the current volume of the balloon. Use `Math.PI` for the value of \( \pi \). To compute the cube of a value \( r \), just use \( r \times r \times r \).

**P3.4** 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 ...”.

**P3.5** 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

```java
public void befriend(Person p)
public void unfriend(Person p)
public String getFriendNames()
```

**P3.6** Add a method

```java
public int getFriendCount()
```

to the `Person` class of Exercise P3.5.
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 average, you also need to store the number of quizzes that the student took. Supply a `StudentTester` class that tests all methods.

Write a class `Battery` that models a rechargeable battery. A battery has a constructor

```java
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

```java
public void drain(double amount)
```

drains the capacity of the battery by the given amount. The method

```java
public void charge()
```

charges the battery to its original capacity. The method

```java
public double getRemainingCapacity()
```

gets the remaining capacity of the battery.

Write a program that draws three stars like the one at right. Use classes `Star`, `StarComponent`, and `StarViewer`.

Implement a class `RoachPopulation` that simulates the growth of a roach population. The constructor takes the size of the initial roach population. The `breed` method simulates a period in which the roaches breed, which doubles their population. The `spray(double percent)` method simulates spraying with insecticide, which reduces the population by the given percentage. The `getRoaches` method returns the current number of roaches. A program called `RoachSimulation` simulates a population that starts out with 10 roaches. Breed, spray to reduce the population by 10 percent, and print the roach count. Repeat three more times.

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.

In this project, you will enhance the `BankAccount` class and see how abstraction and encapsulation enable evolutionary changes to software. Begin with a simple enhancement: charging a fee for every deposit and withdrawal. Supply a mechanism for setting the fee and modify the `deposit` and `withdraw` methods so that the fee is levied. Test your class and check that the fee is computed correctly.

Now make a more complex change. The bank will allow a fixed number of free transactions (deposits or withdrawals) every month, and charge for transactions exceeding the free allotment. The charge is not levied immediately but at the end of the month.

Supply a new method `deductMonthlyCharge` to the `BankAccount` class that deducts the monthly charge and resets the transaction count. (*Hint: Use `Math.max(actual transaction count, free transaction count)` in your computation.*)

Produce a test program that verifies that the fees are calculated correctly over several months.
In this project, you will explore an object-oriented alternative to the “Hello, World” program in Chapter 1.

Begin with a simple `Greeter` class that has a single method, `sayHello`. That method should `return` a string, not print it. Create two objects of this class and invoke their `sayHello` methods. Of course, both objects return the same answer.

Enhance the `Greeter` class so that each object produces a customized greeting. For example, the object constructed as `new Greeter("Dave")` should say "Hello, Dave". (Use the `concat` method to combine strings to form a longer string, or peek ahead at Section 4.5 to see how you can use the `+` operator for the same purpose.)

Add a method `sayGoodbye` to the `Greeter` class.

Finally, add a method `refuseHelp` to the `Greeter` class. It should return a string such as "I am sorry, Dave. I am afraid I can't do that."

If you use BlueJ, place two `Greeter` objects on the workbench (one that greets the world and one that greets Dave) and invoke methods on them. Otherwise, write a tester program that constructs these objects, invokes methods, and prints the results.

---

**Answers to Self-Check Questions**

1. public void unclick()  
   {
     value = value - 1;
   }

2. You can only access them by invoking the methods of the `Clock` class.

3. In one of the methods of the `Counter` class.

4. The programmers who designed and implemented the Java library.

5. Other programmers who work on the personal finance application.

6. `harrysChecking.withdraw(harrysChecking.getBalance());`

7. The `withdraw` method has return type `void`. It doesn't return a value. Use the `getBalance` method to obtain the balance after the withdrawal.

8. Add an `accountNumber` parameter variable to the constructors, and add a `getAccountNumber` method. There is no need for a `setAccountNumber` method—the account number never changes after construction.

9. /**<
   Constructs a new bank account with a given initial balance.
   @param accountNumber the account number for this account
   @param initialBalance the initial balance for this account
   */

10. The first sentence of the method description should describe the method—it is displayed in isolation in the summary table.

11. An instance variable needs to be added to the class:
    ```java
    private int accountNumber;
    ```

12. Because the balance instance variable is accessed from the main method of `BankRobber`. The compiler will report an error because `main` is not a method of the `BankAccount` class and has no access to `BankAccount` instance variables.

13. public int getWidth()
    {
       return width;
    }

14. There is more than one correct answer. One possible implementation is as follows:
    ```java
    public void translate(int dx, int dy)
    {
      int newx = x + dx;
      x = newx;
      int newy = y + dy;
      y = newy;
    }
    ```

15. One `BankAccount` object, no `BankAccountTester` object. The purpose of the `BankAccountTester` class is merely to hold the `main` method.

16. In those environments, you can issue interactive commands to construct `BankAccount`
objects, invoke methods, and display their return values.

17. 

```
Car myCar
    Car(mpg)
    addGas(amount)
    drive(distance)
    getGasLeft
```

18. 

<table>
<thead>
<tr>
<th>gasLeft</th>
<th>milesPerGallon</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>25</td>
</tr>
</tbody>
</table>

19. 

<table>
<thead>
<tr>
<th>gasLeft</th>
<th>milesPerGallon</th>
<th>totalMiles</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>25</td>
<td>0</td>
</tr>
</tbody>
</table>

20. 

<table>
<thead>
<tr>
<th>gasLeft</th>
<th>milesPerGallon</th>
<th>totalMiles</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>25</td>
<td>0</td>
</tr>
<tr>
<td>20</td>
<td>100</td>
<td></td>
</tr>
<tr>
<td>-16</td>
<td>300</td>
<td></td>
</tr>
<tr>
<td>-8</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

21. Variables of both categories belong to methods—they come alive when the method is called, and they die when the method exits. They differ in their initialization. Parameter variables are initialized with the values supplied as arguments in the call; local variables must be explicitly initialized.

22. After computing the change due, `payment` and `purchase` were set to zero. If the method returned `payment - purchase`, it would always return zero.

23. 

<table>
<thead>
<tr>
<th>reg1.purchase</th>
<th>reg1.payment</th>
<th>change</th>
</tr>
</thead>
<tbody>
<tr>
<td>-19.5</td>
<td>-20</td>
<td>0.5</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>X</td>
</tr>
</tbody>
</table>

24. One implicit parameter, called `this`, of type `BankAccount`, and one explicit parameter, called `amount`, of type `double`.

25. It is not a legal expression. `this` is of type `BankAccount` and the `BankAccount` class has no instance variable named `amount`.

26. No implicit parameter—the `main` method is not invoked on any object—and one explicit parameter, called `args`.

27. `CarComponent`

28. In the `draw` method of the `Car` class, call 
```
g2.fill(frontTire);
g2.fill(rearTire);
```

29. Double all measurements in the `draw` method of the `Car` class.
WORKED EXAMPLE 3.1  Making a Simple Menu

Problem Statement  Your task is to design a class 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

The numbers should be supplied automatically when options are added to the menu.

Step 1  Find out which methods you are asked to supply.

The problem description lists two tasks:

   Display the menu.
   Add an option to the menu.

Step 2  Specify the public interface.

Here we turn the list in Step 1 into a set of methods, with specific types for the parameter variables and the return values. As recommended in How To 3.1, we start by writing out sample code:

   mainMenu.addOption("Open new account");
   mainMenu.addOption("Log into existing account");
   mainMenu.display();

Now we have a specific list of methods:

   public void addOption(String option)
   public void display()

To complete the public interface, we need to specify the constructors. We have two choices:

   • Supply a constructor Menu(String firstOption) that makes a menu with one option.
   • Supply a constructor Menu() that makes a menu with no options.

Either choice will work fine. If we decide in favor of the second choice, the user of the class needs to call addOption to add the first option—after all, there is no sense in having a menu with no options. At first glance, that seems like a burden for the programmer using the class. But actually, it is usually conceptually simpler if an API has no special cases (such as having to supply the first option in the constructor). Therefore, we decide that “simplest is best” and specify the constructor

   public Menu() 

Step 3  Document the public interface.

Here is the documentation, with comments, that describes the class and its methods:

   /**
    * A menu that is displayed on a console.
    */
   public class Menu
   {
   /**
    * 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 on the console.
   */
   public void display() {

   }

Step 4 Determine instance variables.
What data does a Menu option need to keep in order to fulfill its responsibilities? Of course, in order to display the menu, it needs to store the menu text. Now consider the addOption method. That method adds a number and the option to the menu text. Where does the number come from? The menu object needs to store it too, so that it can increment whenever addOption is called.
Therefore, our instance variables are

public class Menu {
    private String menuText;
    private int optionCount;
    . . .
}

Step 5 Implement constructors and methods.
We now implement the constructors and methods in the class, one at a time, in the order that is most convenient. The constructor seems pretty easy:

public Menu() {
    menuText = "";
    optionCount = 0;
}

The display method is easy as well:

public void display() {
    System.out.println(menuText);
}

The addOption method requires a bit more thought. Here is the pseudocode:

Increment the option count.
Add the following to the menu text:
The option count
A } symbol
The option to be added
A "newline" character that causes the next option to appear on a new line
How do you add something to a string? If you look at the API of the String class, you will find a method concat. For example, the call

```
menuText.concat(option)
```

creates a string consisting of the strings menuText and option. You can then store that string back into the menuText variable:

```
menuText = menuText.concat(option);
```

As you will learn in Chapter 4, you can achieve the same effect with the + operator:

```
menuText = menuText + option;
```

We use the + operator in our solution because it is so convenient. Our method then becomes

```
public void addOption(String option)
{
  optionCount = optionCount + 1;
  menuText = menuText + optionCount + ") " + option + "\n";
}
```

**Step 6** Test your class.

Here is a short program that demonstrates all methods in the public interface of the Menu class:

```
public class MenuDemo
{
  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");
    mainMenu.display();
  }
}
```

**Program Run**

1) Open new account
2) Log into existing account
3) Help
4) Quit
CHAPTER 4

FUNDAMENTAL DATA TYPES

CHAPTER GOALS

To understand integer and floating-point numbers
To recognize the limitations of the numeric types
To become aware of causes for overflow and roundoff errors
To understand the proper use of constants
To write arithmetic expressions in Java
To use the String type to manipulate character strings
To write programs that read input and produce formatted output

CHAPTER CONTENTS

4.1 NUMBERS 130
   SYN Constant Declaration 134
   ST1 Big Numbers 136
   PT1 Do Not Use Magic Numbers 137

4.2 ARITHMETIC 137
   SYN Cast 141
   CE1 Unintended Integer Division 142
   CE2 Unbalanced Parentheses 142
   PT2 Spaces in Expressions 143
   JB1 Avoiding Negative Remainders 143
   ST2 Combining Assignment and Arithmetic 143
   ST3 Instance Methods and Static Methods 143
   C&S The Pentium Floating-Point Bug 144

4.3 INPUT AND OUTPUT 145
   SYN Input Statement 145
   HT1 Carrying Out Computations 149
   WE1 Computing the Volume and Surface Area of a Pyramid

4.4 PROBLEM SOLVING: FIRST DO IT BY HAND 152
   WE2 Computing Travel Time

4.5 STRINGS 154
   PT3 Reading Exception Reports 160
   ST4 Using Dialog Boxes for Input and Output 160
   C&S International Alphabets and Unicode 161
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. We also cover the important topic of input and output, which enables you to implement interactive programs.

4.1 Numbers

We start this chapter with information about numbers. The following sections tell you how to choose the most appropriate number types for your numeric values, and how to work with constants—numeric values that do not change.

4.1.1 Number Types

In Java, every value is either a reference to an object, or it belongs to one of the eight primitive types shown in Table 1.

Six of the primitive types are number types; four of them for integers and two for floating-point numbers.

Each of the number types has a different range. Appendix G explains why the range limits are related to powers of two. The largest number that can be represented in an int is denoted by `Integer.MAX_VALUE`. Its value is about 2.14 billion. Similarly, the smallest integer is `Integer.MIN_VALUE`, about –2.14 billion.

<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
<th>Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>int</td>
<td>The integer type, with range –2,147,483,648 (Integer.MIN_VALUE) . . . 2,147,483,647 (Integer.MAX_VALUE, about 2.14 billion)</td>
<td>4 bytes</td>
</tr>
<tr>
<td>byte</td>
<td>The type describing a single byte, with range –128 . . . 127</td>
<td>1 byte</td>
</tr>
<tr>
<td>short</td>
<td>The short integer type, with range –32,768 . . . 32,767</td>
<td>2 bytes</td>
</tr>
<tr>
<td>long</td>
<td>The long integer type, with range –9,223,372,036,854,775,808 . . . 9,223,372,036,854,775,807</td>
<td>8 bytes</td>
</tr>
<tr>
<td>double</td>
<td>The double-precision floating-point type, with a range of about ±10^308 and about 15 significant decimal digits</td>
<td>8 bytes</td>
</tr>
<tr>
<td>float</td>
<td>The single-precision floating-point type, with a range of about ±10^38 and about 7 significant decimal digits</td>
<td>4 bytes</td>
</tr>
<tr>
<td>char</td>
<td>The character type, representing code units in the Unicode encoding scheme (see Computing &amp; Society 4.2 on page 161)</td>
<td>2 bytes</td>
</tr>
<tr>
<td>boolean</td>
<td>The type with the two truth values false and true (see Chapter 5)</td>
<td>1 bit</td>
</tr>
</tbody>
</table>
4.1 Numbers

Table 2 Number Literals in Java

<table>
<thead>
<tr>
<th>Number</th>
<th>Type</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>int</td>
<td>An integer has no fractional part.</td>
</tr>
<tr>
<td>-6</td>
<td>int</td>
<td>Integers can be negative.</td>
</tr>
<tr>
<td>0</td>
<td>int</td>
<td>Zero is an integer.</td>
</tr>
<tr>
<td>0.5</td>
<td>double</td>
<td>A number with a fractional part has type double.</td>
</tr>
<tr>
<td>1.0</td>
<td>double</td>
<td>An integer with a fractional part .0 has type double.</td>
</tr>
<tr>
<td>1E6</td>
<td>double</td>
<td>A number in exponential notation: $1 \times 10^6$ or 1000000. Numbers in exponential notation always have type double.</td>
</tr>
<tr>
<td>2.96E-2</td>
<td>double</td>
<td>Negative exponent: $2.96 \times 10^{-2} = 2.96 / 100 = 0.0296$</td>
</tr>
<tr>
<td>100000L</td>
<td>long</td>
<td>The L suffix indicates a long literal.</td>
</tr>
<tr>
<td>100,000</td>
<td>Error</td>
<td>Do not use a comma as a decimal separator.</td>
</tr>
<tr>
<td>100_000</td>
<td>int</td>
<td>You can use underscores in number literals.</td>
</tr>
<tr>
<td>3 1/2</td>
<td>Error</td>
<td>Do not use fractions; use decimal notation: 3.5</td>
</tr>
</tbody>
</table>

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.

Generally, you will use the int type for integer quantities. Occasionally, however, calculations involving integers can overflow. This happens if the result of a computation exceeds the range for the number type. For example,

```java
int n = 100000;
System.out.println(n * n);  // Prints –727379968, which is clearly wrong
```

The product $n \times n$ is $10^{12}$, which is larger than the largest integer (about $2 \cdot 10^9$). The result is truncated to fit into an int, yielding a value that is completely wrong. Unfortunately, there is no warning when an integer overflow occurs.

If you run into this problem, the simplest remedy is to use the long type. Special Topic 4.1 on page 136 shows you how to use the BigInteger type in the unlikely event that even the long type overflows.

Overflow is not usually a problem for double-precision floating-point numbers. The double type has a range of about $\pm10^{308}$. Floating-point numbers have a different problem—limited precision. The double type has about 15 significant digits, and there are many numbers that cannot be accurately represented as double values.

When a value cannot be represented exactly, it is rounded to the nearest match. Consider this example:

```java
double f = 4.35;
System.out.println(100 * f);  // Prints 434.99999999999994
```

If a computation yields an integer that is larger than the largest int value (about 2.14 billion), it overflows.
Floating-point numbers have limited precision. Not every value can be represented precisely, and roundoff errors can occur.

The problem arises because computers represent numbers in the binary number system. In the binary number system, there is no exact representation of the fraction 1/10, just as there is no exact representation of the fraction 1/3 = 0.33333 in the decimal number system. (See Appendix G for more information.)

For this reason, the double type is not appropriate for financial calculations. In this book, we will continue to use double values for bank balances and other financial quantities so that we keep our programs as simple as possible. However, professional programs need to use the BigDecimal type for this purpose—see Special Topic 4.1.

In Java, it is legal to assign an integer value to a floating-point variable:

```java
int dollars = 100;
double balance = dollars; // OK
```

But the opposite assignment is an error: You cannot assign a floating-point expression to an integer variable.

```java
double balance = 13.75;
int dollars = balance; // Error
```

You will see in Section 4.2.5 how to convert a value of type double into an integer.

In this book, we do not use the float type. It has less than 7 significant digits, which greatly increases the risk of roundoff errors. Some programmers use float to save on memory if they need to store a huge set of numbers that do not require much precision.

### 4.1.2 Constants

In many programs, you need to use numerical constants—values that do not change and that have a special significance for a computation.

A typical example for the use of constants is a computation that involves coin values, such as the following:

```java
payment = dollars + quarters * 0.25 + dimes * 0.1 + nickels * 0.05 + pennies * 0.01;
```

Most of the code is self-documenting. However, the four numeric quantities, 0.25, 0.1, 0.05, and 0.01 are included in the arithmetic expression without any explanation. Of course, in this case, you know that the value of a nickel is five cents, which explains the 0.05, and so on. However, the next person who needs to maintain this code may live in another country and may not know that a nickel is worth five cents.

Thus, it is a good idea to use symbolic names for all values, even those that appear obvious. Here is a clearer version of the computation of the total:

```java
double quarterValue = 0.25;
double dimeValue = 0.1;
double nickelValue = 0.05;
double pennyValue = 0.01;
```
payment = dollars + quarters * quarterValue + dimes * dimeValue + nickels * nickelValue + pennies * pennyValue;

There is another improvement we can make. There is a difference between the nickels and nickelValue variables. The nickels variable can truly vary over the life of the program, as we calculate different payments. But nickelValue is always 0.05.

In Java, constants are identified with the reserved word final. A variable tagged as final can never change after it has been set. If you try to change the value of a final variable, the compiler will report an error and your program will not compile.

Many programmers use all-uppercase names for constants (final variables), such as NICKEL_VALUE. That way, it is easy to distinguish between variables (with mostly lowercase letters) and constants. We will follow this convention in this book. However, this rule is a matter of good style, not a requirement of the Java language. The compiler will not complain if you give a final variable a name with lowercase letters.

Here is an improved version of the code that computes the value of a payment.

```java
final double QUARTER_VALUE = 0.25;
final double DIME_VALUE = 0.1;
final double NICKEL_VALUE = 0.05;
final double PENNY_VALUE = 0.01;
payment = dollars + quarters * QUARTER_VALUE + dimes * DIME_VALUE + nickels * NICKEL_VALUE + pennies * PENNY_VALUE;
```

Frequently, constant values are needed in several methods. Then you should declare them together with the instance variables of a class and tag them as static and final. As before, final indicates that the value is a constant. The static reserved word means that the constant belongs to the class—this is explained in greater detail in Chapter 8.)

```java
public class CashRegister
{
    // Constants
    public static final double QUARTER_VALUE = 0.25;
    public static final double DIME_VALUE = 0.1;
    public static final double NICKEL_VALUE = 0.05;
    public static final double PENNY_VALUE = 0.01;

    // Instance variables
    private double purchase;
    private double payment;

    // Methods
    . . .
}
```

We declared the constants as public. There is no danger in doing this because constants cannot be modified. Methods of other classes can access a public constant by first specifying the name of the class in which it is declared, then a period, then the name of the constant, such as CashRegister.NICKEL_VALUE.

The Math class from the standard library declares a couple of useful constants:

```java
public class Math
{
    public static final double E = 2.7182818284590452354;
    public static final double PI = 3.14159265358979323846;
}
```

You can refer to these constants as Math.PI and Math.E in any method. For example,

```java
double circumference = Math.PI * diameter;
```
The sample program below puts constants to work. The program shows a refinement of the `CashRegister` class of How To 3.1. The public interface of that class has been modified in order to solve a common business problem.

Busy cashiers sometimes make mistakes totaling up coin values. Our `CashRegister` class features a method whose inputs are the coin counts. For example, the call

```java
register.receivePayment(1, 2, 1, 1, 4);
```

processes a payment consisting of one dollar, two quarters, one dime, one nickel, and four pennies. The `receivePayment` method figures out the total value of the payment, $1.69. As you can see from the code listing, the method uses named constants for the coin values.

```java
/**
 * A cash register totals up sales and computes change due.
 */
public class CashRegister {

   public static final double QUARTER_VALUE = 0.25;
   public static final double DIME_VALUE = 0.1;
   public static final double NICKEL_VALUE = 0.05;
   public static final double PENNY_VALUE = 0.01;

   private double purchase;
   private double payment;

   /**
    * Constructs a cash register with no money in it.
    */
   public CashRegister() {
      purchase = 0;
      payment = 0;
   }
```
public void recordPurchase(double amount) {
    purchase = purchase + amount;
}

public void receivePayment(int dollars, int quarters, int dimes, int nickels, int pennies) {
    payment = dollars + quarters * QUARTER_VALUE + dimes * DIME_VALUE + nickels * NICKEL_VALUE + pennies * PENNY_VALUE;
}

public double giveChange() {
    double change = payment - purchase;
    purchase = 0;
    payment = 0;
    return change;
}

public class CashRegisterTester {
    public static void main(String[] args) {
        CashRegister register = new CashRegister();
        register.recordPurchase(0.75);
        register.recordPurchase(1.50);
        register.receivePayment(2, 0, 5, 0, 0);
        System.out.print("Change: ");
        System.out.println(register.giveChange());
        System.out.println("Expected: 0.25");
        register.recordPurchase(2.25);
        register.recordPurchase(19.25);
        register.receivePayment(23, 2, 0, 0, 0);
System.out.print("Change: ");
System.out.println(register.giveChange());
System.out.println("Expected: 2.0");
}

Program Run
Change: 0.25
Expected: 0.25
Change: 2.0
Expected: 2.0

1. Which are the most commonly used number types in Java?

2. Suppose you want to write a program that works with population data from various countries. Which Java data type should you use?

3. Which of the following initializations are incorrect, and why?
   a. int dollars = 100.0;
   b. double balance = 100;

4. What is the difference between the following two statements?
   final double CM_PER_INCH = 2.54;
   and
   public static final double CM_PER_INCH = 2.54;

5. What is wrong with the following statement sequence?
   double diameter = . . .;
   double circumference = 3.14 * diameter;

Practice It Now you can try these exercises at the end of the chapter: R4.1, R4.27, E4.21.

Special Topic 4.1
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 n = new BigInteger("1000000");
BigInteger r = n.multiply(n);
System.out.println(r); // Prints 1000000000000
```

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

```
BigDecimal d = new BigDecimal("4.35");
BigDecimal e = new BigDecimal("100");
BigDecimal f = d.multiply(e);
System.out.println(f); // Prints 435.00
```
4.2 Arithmetic

In this section, you will learn how to carry out arithmetic calculations in Java.

4.2.1 Arithmetic Operators

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

You must write \( \times \) to denote multiplication. Unlike in mathematics, you cannot write \( \times \), \( \cdot \), or \( \times \). Similarly, division is always indicated with the \( / \) operator, 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 (see Appendix B).

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.
4.2.2 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 1):

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

Similarly, the -- operator decrements a variable:

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

![Figure 1 Incrementing a Variable](image)

4.2.3 Integer Division and Remainder

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 integer 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 4.1.

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

```c
  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:

```c
  int pennies = 1729;
```

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

```c
  int dollars = pennies / 100;  // Sets dollars to 17
```
4.2 Arithmetic

Table 3 Integer Division and Remainder

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

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

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

See Table 3 for additional examples.

4.2.4 Powers and Roots

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

\[b \times \text{Math.pow}(1 + r / 100, n)\]

Figure 2 shows how to analyze such an expression. Table 4 shows additional mathematical methods.

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

Figure 2
Analyzing an Expression
<table>
<thead>
<tr>
<th>Method</th>
<th>Returns</th>
<th>Method</th>
<th>Returns</th>
</tr>
</thead>
<tbody>
<tr>
<td>Math.sqrt(x)</td>
<td>Square root of x (≥0)</td>
<td>Math.abs(x)</td>
<td>Absolute value</td>
</tr>
<tr>
<td>Math.pow(x, y)</td>
<td>x^y (x &gt; 0, or x = 0 and y &gt; 0, or x &lt; 0 and y is an integer)</td>
<td>Math.max(x, y)</td>
<td>The larger of x and y</td>
</tr>
<tr>
<td>Math.sin(x)</td>
<td>Sine of x (x in radians)</td>
<td>Math.min(x, y)</td>
<td>The smaller of x and y</td>
</tr>
<tr>
<td>Math.cos(x)</td>
<td>Cosine of x</td>
<td>Math.exp(x)</td>
<td>e^x</td>
</tr>
<tr>
<td>Math.tan(x)</td>
<td>Tangent of x</td>
<td>Math.log(x)</td>
<td>Natural log (ln(x), x &gt; 0)</td>
</tr>
<tr>
<td>Math.round(x)</td>
<td>Closest integer to x (as a long)</td>
<td>Math.log10(x)</td>
<td>Decimal log (log_{10}(x), x &gt; 0)</td>
</tr>
<tr>
<td>Math.ceil(x)</td>
<td>Smallest integer ≥ x (as a double)</td>
<td>Math.floor(x)</td>
<td>Largest integer ≤ x (as a double)</td>
</tr>
<tr>
<td>Math.toRadians(x)</td>
<td>Convert x degrees to radians (i.e., returns x \cdot \pi/180)</td>
<td>Math.toDegrees(x)</td>
<td>Convert x radians to degrees (i.e., returns x \cdot 180/\pi)</td>
</tr>
</tbody>
</table>

### 4.2.5 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
```

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 convert 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);
```

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.
4.2 Arithmetic

Syntax 4.2  Cast

 Syntax (typeName) expression

This is the type of the expression after casting.

(int) (balance * 100)

These parentheses are a part of the cast operator.

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

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

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

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

6. 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.

7. In Java, how do you compute the side length of a square whose area is stored in the variable area?

8. The volume of a sphere is given by the formula at right. If the radius is given by a variable radius of type double, write a Java expression for the volume.

9. What is the value of 1729 / 100 and 1729 % 100?

10. If \( n \) is a positive number, what is \( (n / 10) \% 10 \)?

Practice It  Now you can try these exercises at the end of the chapter: R4.4, R4.8, E4.4, E4.24.
**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:

```java
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

\[
\frac{score1 + score2 + score3}{3}
\]

Here, however, the `/` does not mean division in the mathematical sense. It denotes integer division because both 3 and the sum of score1 + score2 + score3 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:

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

or

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

---

**Unbalanced Parentheses**

Consider the expression:

\[
((a + b) \times t) / (2 \times (1 - t))
\]

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:

\[
(a + b) \times t) / (2 \times (1 - t)
\]

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

\[
(a + b) \times t) / (2 \times (1 - t)
\]

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

\[
(a + b) \times t) / (2 \times (1 - t)
\]

1 0 1

and you would find the error.
Spaces in Expressions

It is easier to read
\[ x_1 = \frac{-b + \text{Math.sqrt}(b \times b - 4 \times a \times c)}{2 \times a}; \]
than
\[ x_1=\frac{-b+\text{Math.sqrt}(b*b-4*a*c)}{2*a}; \]

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).

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 \( \text{direction} + \text{turn} \) \% 360 because you might get a negative result (see Exercise R4.7). In Java 8, you can instead call
\[ \text{Math.floorMod(direction} + \text{turn}, 360) \]
to compute the correct remainder. The result of Math.floorMod(m, n) is always positive when n is positive.

Combining Assignment and Arithmetic

In Java, you can combine arithmetic and assignment. For example, the instruction
\[ \text{balance} += \text{amount}; \]
is a shortcut for \[ \text{balance} = \text{balance} + \text{amount}; \]
Similarly,
\[ \text{total} *= 2; \]
is another way of writing \[ \text{total} = \text{total} \times 2; \]
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.

Instance Methods and Static Methods

In the preceding section, you encountered the Math class, which contains a collection of helpful methods for carrying out mathematical computations. These methods do not operate on an object. That is, you don’t call
\[ \text{double root} = 2.\text{sqrt}(); \]
\[ \text{// Error} \]
In Java, numbers are not objects, so you can never invoke a method on a number. Instead, you pass a number as an argument (explicit parameter) to a method, enclosing the number in parentheses after the method name:
\[ \text{double root} = \text{Math.sqrt}(2); \]
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.)

Static methods do not operate on objects, but they are still declared inside classes. When calling the method, you specify the class to which the `sqrt` method belongs:

```
The name of the class        The name of the static method
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 will learn more about the distinction between static and instance methods in Chapter 8.

---

### Computing & Society 4.1  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 roundoff behavior of floating-point calculations has been standardized by the Institute for Electrical and Electronics 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,

\[
4,195,835 - \left( \frac{4,195,835}{3,145,727} \right) \times 3,145,727
\]

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.
4.3 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.

4.3.1 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 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);
```

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 4.3 Input Statement

- Include this line so you can use the `Scanner` class.
  ```java
  import java.util.Scanner;
  ```
- Create a `Scanner` object to read keyboard input.
  ```java
  Scanner in = new Scanner(System.in);
  ```
- Display a prompt in the console window.
  ```java
  System.out.print("Please enter the number of bottles: ");
  ```
- Define a variable to hold the input value.
  ```java
  int bottles = in.nextInt();
  ```

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

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

The `Scanner` class belongs to the package `java.util`. When using the `Scanner` class, import it by placing the following declaration at the top of your program file:

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

### 4.3.2 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:

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

You can also specify a **field width**:

```java
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 6 for examples.

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

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

prints

```
Price per liter:      1.22
```

Use the `printf` method to specify how values should be formatted.
Table 6 Format Specifier Examples

<table>
<thead>
<tr>
<th>Format String</th>
<th>Sample Output</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;%d&quot;</td>
<td>24</td>
<td>Use d with an integer.</td>
</tr>
<tr>
<td>&quot;%5d&quot;</td>
<td>24</td>
<td>Spaces are added so that the field width is 5.</td>
</tr>
<tr>
<td>&quot;Quantity:%5d&quot;</td>
<td>Quantity: 24</td>
<td>Characters inside a format string but outside a format specifier appear in the output.</td>
</tr>
<tr>
<td>&quot;%f&quot;</td>
<td>1.21997</td>
<td>Use f with a floating-point number.</td>
</tr>
<tr>
<td>&quot;%.2f&quot;</td>
<td>1.22</td>
<td>Prints two digits after the decimal point.</td>
</tr>
<tr>
<td>&quot;%7.2f&quot;</td>
<td>1.22</td>
<td>Spaces are added so that the field width is 7.</td>
</tr>
<tr>
<td>&quot;%s&quot;</td>
<td>Hello</td>
<td>Use s with a string.</td>
</tr>
<tr>
<td>&quot;%d %.2f&quot;</td>
<td>24 1.22</td>
<td>You can format multiple values at once.</td>
</tr>
</tbody>
</table>

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

```java
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 the next output to be on a separate line, you can call `System.out.println()`. Alternatively, Section 4.5.4 shows you how to add a newline character to the format string.

Our next example program will prompt for the price of a six-pack of soda and a two-liter bottle, and then print out the price per liter for both. The program puts to work what you just learned about reading input and formatting output.

What is the better deal? A six-pack of 12-ounce cans or a two-liter bottle?
### section_3/Volume.java

```java
import java.util.Scanner;

/**
 * This program prints the price per liter for a six-pack of cans and
 * a two-liter bottle.
 */
public class Volume {
    public static void main(String[] args) {
        // Read price per pack
        Scanner in = new Scanner(System.in);
        System.out.print("Please enter the price for a six-pack: ");
        double packPrice = in.nextDouble();

        // Read price per bottle
        System.out.print("Please enter the price for a two-liter bottle: ");
        double bottlePrice = in.nextDouble();

        final double CANS_PER_PACK = 6;
        final double CAN_VOLUME = 0.355; // 12 oz. = 0.355 l
        final double BOTTLE_VOLUME = 2;

        // Compute and print price per liter
        double packPricePerLiter = packPrice / (CANS_PER_PACK * CAN_VOLUME);
        double bottlePricePerLiter = bottlePrice / BOTTLE_VOLUME;

        System.out.printf("Pack price per liter: %8.2f", packPricePerLiter);
        System.out.println();
        System.out.printf("Bottle price per liter: %8.2f", bottlePricePerLiter);
        System.out.println();
    }
}
```

### Program Run

```
Please enter the price for a six-pack: 2.95
Please enter the price for a two-liter bottle: 2.85
Pack price per liter: 1.38
Bottle price per liter: 1.43
```

### SELF CHECK

11. Write statements to prompt for and read the user’s age using a Scanner variable named `in`.

12. What is wrong with the following statement sequence?

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

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

15. What is the output of the following statement sequence?
   int volume = 10;
   System.out.printf("The volume is %5d", volume);

16. 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: R4.13, E4.6, E4.7.

HOW TO 4.1  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:

\[
\text{change due} = 100 \times \text{bill value} - \text{item price in pennies}
\]

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

\[
\text{dollar coins} = \text{change due} / 100 \text{ (without remainder)}
\]

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

\[
\text{change due} = \text{change due} \mod 100
\]

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

\[
\text{change due} = \text{change due} - 100 \times \text{dollar coins}
\]

To get the quarters due, divide by 25:

\[
\text{quarters} = \text{change due} / 25
\]

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 \text{PENNIES\_PER\_DOLLAR} and \text{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 \text{changeDue} and \text{PENNIES\_PER\_DOLLAR} are of type \text{int} because the computation of \text{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.

\[
\text{changeDue} = \text{PENNIES\_PER\_DOLLAR} \times \text{billValue} - \text{itemPrice};
\]

\[
\text{dollarCoins} = \text{changeDue} / \text{PENNIES\_PER\_DOLLAR};
\]

\[
\text{changeDue} = \text{changeDue} \mod \text{PENNIES\_PER\_DOLLAR};
\]

\[
\text{quarters} = \text{changeDue} / \text{PENNIES\_PER\_QUARTER};
\]

Step 6  Provide input and output.

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

\[
\text{System.out.print("Enter bill value (1 = $1 bill, 5 = $5 bill, etc.): ");}
\]

\[
\text{billValue = in.nextInt();}
\]

\[
\text{System.out.print("Enter item price in pennies: ");}
\]

\[
\text{itemPrice = in.nextInt();}
\]

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

\[
\text{System.out.printf("Dollar coins: %d", dollarCoins);}
\]

\[
\text{System.out.printf("Quarters: %d", quarters);}
\]
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.

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:

```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();
    }
}
```

A vending machine takes bills and gives change in coins.
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.

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:

\[
\begin{array}{ccccccc}
\text{ } & \text{ } & \text{ } & \text{ } & \text{ } & \text{ } & \\
\text{ } & \text{ } & \text{ } & \text{ } & \text{ } & \text{ } & \\
\text{ } & \text{ } & \text{ } & \text{ } & \text{ } & \text{ } & \\
\text{ } & \text{ } & \text{ } & \text{ } & \text{ } & \text{ } & \\
\text{ } & \text{ } & \text{ } & \text{ } & \text{ } & \text{ } & \\
\text{ } & \text{ } & \text{ } & \text{ } & \text{ } & \text{ } & \\
\text{ } & \text{ } & \text{ } & \text{ } & \text{ } & \text{ } & \\
\end{array}
\]

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 because 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 × 5 = 95 inches, leaving a total gap of 100 − 19 × 5 = 5 inches.

The gap should be evenly distributed at both ends. At each end, the gap is (100 − 19 × 5) / 2 = 2.5 inches.

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

\[
\begin{align*}
\text{number of pairs} &= \text{integer part of } (\text{total width} - \text{tile width}) / (2 \times \text{tile width}) \\
\text{number of tiles} &= 1 + 2 \times \text{number of pairs} \\
\text{gap at each end} &= (\text{total width} - \text{number of tiles} \times \text{tile width}) / 2
\end{align*}
\]

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

17. Translate the pseudocode for computing the number of tiles and the gap width into Java.

18. Suppose the architect specifies a pattern with black, gray, and white tiles, like this:

\[
\begin{array}{cccccccc}
\text{ } & \text{ } & \text{ } & \text{ } & \text{ } & \text{ } & \text{ } & \\
\text{ } & \text{ } & \text{ } & \text{ } & \text{ } & \text{ } & \text{ } & \\
\text{ } & \text{ } & \text{ } & \text{ } & \text{ } & \text{ } & \text{ } & \\
\text{ } & \text{ } & \text{ } & \text{ } & \text{ } & \text{ } & \text{ } & \\
\text{ } & \text{ } & \text{ } & \text{ } & \text{ } & \text{ } & \text{ } & \\
\text{ } & \text{ } & \text{ } & \text{ } & \text{ } & \text{ } & \text{ } & \\
\text{ } & \text{ } & \text{ } & \text{ } & \text{ } & \text{ } & \text{ } & \\
\end{array}
\]

Again, the first and last tile should be black. How do you need to modify the algorithm?

19. 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.

20. 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 \( n \).

21. 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 \( h_3 \).
Chapter 4  Fundamental Data Types

Using the formulas for the volume of a cylinder, \( V = \pi r^2 h \), and a cone section,
\[
V = \pi \left( \frac{r_1^2 + r_1 r_2 + r_2^2}{3} \right) h,
\]
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: R4.18, R4.22, R4.23.

## 4.5 Strings

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.

### 4.5.1 The String Type

You can declare variables that hold strings.

```java
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. As you saw in Section 2.3, you can compute the length of a string with the `length` method.

```java
int n = name.length();
```

A string of length 0 is called the **empty string**. It contains no characters and is written as "".
4.5.2 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,

```java
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:

```java
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. For example, consider this code:

```java
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

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

to the single call

```java
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.

4.5.3 String Input

You can read a string from the console:

```java
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.
4.5.4 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

```java
System.out.print("*\n**\n***\n");
```

prints the characters

```
*  
**  
*** 
```
on three separate lines.

You often want to add a newline character to the end of the format string when you use `System.out.printf`:

```java
System.out.printf("Price: %10.2f\n", price);
```

4.5.5 Strings and Characters

Strings are sequences of Unicode characters (see Computing & Society 4.2). 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 it 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`.

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

```java
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'.

### 4.5.6 Substrings

Once you have a string, you can extract substrings by using the `substring` method. The method call

```java
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:

```java
String greeting = "Hello, World!";
String sub = greeting.substring(0, 5); // sub is "Hello"
```

Here the `substring` operation makes a string that consists of the first five characters taken from the string `greeting`.

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

```java
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,

```java
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.
Chapter 4  Fundamental Data Types

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 3.)

```
import java.util.Scanner;

/**
 * This program prints a pair of initials.
 */

public class Initials {
    public static void main(String[] args) {
        Scanner in = new Scanner(System.in);
        // Get the names of the couple
        System.out.print("Enter your first name: ");
        String first = in.next();
        System.out.print("Enter your significant other's first name: ");
        String second = in.next();
        // Compute and display the inscription
        String initials = first.substring(0, 1) + "&" + second.substring(0, 1);
        System.out.println(initials);
    }
}
```

**Program Run**

```
Enter your first name: Rodolfo
Enter your significant other's first name: Sally
R&S
```
Table 7 String Operations

<table>
<thead>
<tr>
<th>Statement</th>
<th>Result</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>string str = &quot;Ja&quot;; str = str + &quot;va&quot;;</td>
<td>str is set to &quot;Java&quot;</td>
<td>When applied to strings, + denotes concatenation.</td>
</tr>
<tr>
<td>System.out.println(&quot;Please &quot; + &quot; enter your name: &quot;);</td>
<td>Prints Please enter your name:</td>
<td>Use concatenation to break up strings that don’t fit into one line.</td>
</tr>
<tr>
<td>team = 49 + &quot;ers&quot;</td>
<td>team is set to &quot;49ers&quot;</td>
<td>Because &quot;ers&quot; is a string, 49 is converted to a string.</td>
</tr>
<tr>
<td>String first = in.next(); String last = in.next(); (User input: Harry Morgan)</td>
<td>first contains &quot;Harry&quot; last contains &quot;Morgan&quot;</td>
<td>The next method places the next word into the string variable.</td>
</tr>
<tr>
<td>String greeting = &quot;H &amp; S&quot;; int n = greeting.length();</td>
<td>n is set to 5</td>
<td>Each space counts as one character.</td>
</tr>
<tr>
<td>String str = &quot;Sally&quot;; char ch = str.charAt(1);</td>
<td>ch is set to 'a'</td>
<td>This is a char value, not a String. Note that the initial position is 0.</td>
</tr>
<tr>
<td>String str = &quot;Sally&quot;; String str2 = str.substring(1, 4);</td>
<td>str2 is set to &quot;all&quot;</td>
<td>Extracts the substring starting at position 1 and ending before position 4.</td>
</tr>
<tr>
<td>String str = &quot;Sally&quot;; String str2 = str.substring(1);</td>
<td>str2 is set to &quot;ally&quot;</td>
<td>If you omit the end position, all characters from the position until the end of the string are included.</td>
</tr>
<tr>
<td>String str = &quot;Sally&quot;; String str2 = str.substring(1, 2);</td>
<td>str2 is set to &quot;a&quot;</td>
<td>Extracts a String of length 1; contrast with str.charAt(1).</td>
</tr>
<tr>
<td>String last = str.substring( str.length() - 1);</td>
<td>last is set to the string containing the last character in str</td>
<td>The last character has position str.length() - 1.</td>
</tr>
</tbody>
</table>

22. What is the length of the string "Java Program"?

23. Consider this string variable.
   ```java
   String str = "Java Program";
   Give a call to the substring method that returns the substring "gram".
   
   24. Use string concatenation to turn the string variable str from Self Check 23 into "Java Programming".

   25. What does the following statement sequence print?
   ```java
   String str = "Harry";
   int n = str.length();
   String mystery = str.substring(0, 1) + str.substring(n - 1, n);
   System.out.println(mystery);
   ```

26. Give an input statement to read a name of the form “John Q. Public”.

Practice It Now you can try these exercises at the end of the chapter: R4.10, R4.14, E4.15, P4.7.
Reading Exception Reports

You will often have programs that terminate and display an error message, such as

```
Exception in thread "main" java.lang.StringIndexOutOfBoundsException:
  String index out of range: -4
  at java.lang.String.substring(String.java:1444)
  at Homework1.main(Homework1.java:16)
```

If this happens to you, don’t say “it didn’t work” or “my program died”. Instead, read the error message. Admittedly, the format of the exception report is not very friendly. But it is actually easy to decipher it.

When you have a close look at the error message, you will notice two pieces of useful information:

1. The name of the exception, such as `StringIndexOutOfBoundsException`
2. The line number of the code that contained the statement that caused the exception, such as `Homework1.java:16`

The name of the exception is always in the first line of the report, and it ends in `Exception`. If you get a `StringIndexOutOfBoundsException`, then there was a problem with accessing an invalid position in a string. That is useful information.

The line number of the offending code is a little harder to determine. The exception report contains the entire stack trace—that is, the names of all methods that were pending when the exception hit. 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`. Often, the exception was thrown by a method that is in the standard library. Look for the first line in your code that appears in the exception report. For example, skip the line that refers to

```
java.lang.String.substring(String.java:1444)
```

The next line in our example mentions a line number in your code, `Homework1.java`. Once you have the line number in your code, 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.

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.

```
String input = JOptionPane.showInputDialog("Enter price: ");
```

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:

```java
JOptionPane.showMessageDialog(null, "Price: " + price);
```

## Computing & Society 4.2 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.

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.

The Chinese languages as well as Japanese and Korean use Chinese characters. Each character represents an idea or thing. Words are made up of one or more of these ideographic characters. Over 70,000 ideographs are known.

Starting in 1987, a consortium of hardware and software manufacturers developed a uniform encoding scheme 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 (www.unicode.org). 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.

### Choose appropriate types for representing numeric data.

- Java has eight primitive types, including four integer types and two floating-point types.
- A numeric computation overflows if the result falls outside the range for the number type.
- Rounding errors occur when an exact conversion between numbers is not possible.
- A `final` variable is a constant. Once its value has been set, it cannot be changed.
- Use named constants to make your programs easier to read and maintain.
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.

- Use the Scanner class to read keyboard input in a console window.
- Use the printf method to specify how values should be formatted.

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.

STANDARD LIBRARY ITEMS INTRODUCED IN THIS CHAPTER

<table>
<thead>
<tr>
<th>Class/Type</th>
<th>Method/Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>java.io.PrintStream</td>
<td>cos, exp, floor, floorMod, log, log10, max, min, pow, round, sin, sqrt, tan, toDegrees, toRadians</td>
</tr>
<tr>
<td>java.lang.Double</td>
<td></td>
</tr>
<tr>
<td>parseDouble</td>
<td></td>
</tr>
<tr>
<td>java.lang.Integer</td>
<td></td>
</tr>
<tr>
<td>MAX_VALUE</td>
<td></td>
</tr>
<tr>
<td>MIN_VALUE</td>
<td></td>
</tr>
<tr>
<td>parseInt</td>
<td></td>
</tr>
<tr>
<td>java.lang.Math</td>
<td></td>
</tr>
<tr>
<td>PI</td>
<td></td>
</tr>
<tr>
<td>abs, ceil</td>
<td></td>
</tr>
<tr>
<td>java.math.BigDecimal</td>
<td></td>
</tr>
<tr>
<td>add, multiply, subtract</td>
<td></td>
</tr>
<tr>
<td>java.math.BigInteger</td>
<td></td>
</tr>
<tr>
<td>add, multiply, subtract</td>
<td></td>
</tr>
<tr>
<td>java.util.Scanner</td>
<td></td>
</tr>
<tr>
<td>next, nextDouble, nextInt</td>
<td></td>
</tr>
<tr>
<td>javax.swing.JOptionPane</td>
<td></td>
</tr>
<tr>
<td>showInputDialog, showMessageDialog</td>
<td></td>
</tr>
<tr>
<td>javax.swing.JOptionPane</td>
<td></td>
</tr>
<tr>
<td>showInputDialog, showMessageDialog</td>
<td></td>
</tr>
<tr>
<td>java.math.BigDecimal</td>
<td></td>
</tr>
<tr>
<td>add, multiply</td>
<td></td>
</tr>
</tbody>
</table>
R4.1 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

R4.2 What is the value of mystery after this sequence of statements?

```java
int mystery = 1;
mystery = 1 - 2 * mystery;
mystery = mystery + 1;
```

R4.3 What is wrong with the following sequence of statements?

```java
int mystery = 1;
mystery = mystery + 1;
int mystery = 1 - 2 * mystery;
```

R4.4 Write the following Java expressions in mathematical notation.

a. \( dm = m \times \frac{\text{Math.sqrt}(1 + v / c)}{\text{Math.sqrt}(1 - v / c)} - 1 \)

b. \( \text{volume} = \text{Math.PI} \times r \times r \times h \)

c. \( \text{volume} = 4 \times \text{Math.PI} \times \text{Math.pow}(r, 3) / 3 \)

d. \( z = \text{Math.sqrt}(x \times x + y \times y) \)

R4.5 Write the following mathematical expressions in Java.

\[
 s = s_0 + v_0t + \frac{1}{2}gt^2 \quad \quad \quad \quad \quad FV = PV \left(1 + \frac{\text{INT}}{100}\right)^{\text{YRS}}
\]

\[
 G = 4\pi^2 \frac{a^3}{p^2(m_1 + m_2)} \quad \quad \quad \quad \quad c = \sqrt{a^2 + b^2 - 2ab\cos\gamma}
\]

R4.6 Assuming that `a` and `b` are variables of type `int`, fill in the following table:

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>Math.pow(a, b)</th>
<th>Math.max(a, b)</th>
<th>a / b</th>
<th>a % b</th>
<th>Math.floorMod(a, b)</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>-3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>-2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>-3</td>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>-3</td>
<td>-2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
**R4.7** Suppose direction is an integer angle between 0 and 359 degrees. You turn by a given angle and update the direction as

\[ \text{direction} = (\text{direction} + \text{turn}) \mod 360; \]

In which situation do you get the wrong result? How can you fix that without using the \texttt{Math.floorMod} method described in Java 8 Note 4.1?

**R4.8** What are the values of the following expressions? In each line, assume that

\begin{align*}
\text{double } x &= 2.5; \\
\text{double } y &= -1.5; \\
\text{int } m &= 18; \\
\text{int } n &= 4;
\end{align*}

\begin{enumerate}
\item \(x + n \times y - (x + n) \times y\)
\item \(m / n \times m \mod n\)
\item \(5 \times x - n \div 5\)
\item \(1 - (1 - (1 - (1 - (1 - n))))\)
\item \(\text{Math.sqrt(\text{Math.sqrt}(n))}\)
\end{enumerate}

**R4.9** What are the values of the following expressions, assuming that \(n\) is 17 and \(m\) is 18?

\begin{enumerate}
\item \(n \div 10 + n \mod 10\)
\item \(n \mod 2 + m \mod 2\)
\item \((m + n) \div 2\)
\item \((m + n) \div 2.0\)
\item \((\text{int}) (0.5 \times (m + n))\)
\item \((\text{int}) \text{Math.round}(0.5 \times (m + n))\)
\end{enumerate}

**R4.10** What are the values of the following expressions? In each line, assume that

\begin{verbatim}
String s = "Hello";
String t = "World";
\end{verbatim}

\begin{enumerate}
\item \(s.length() + t.length()\)
\item \(s.substring(1, 2)\)
\item \(s.substring(s.length() / 2, s.length())\)
\item \(s + t\)
\item \(t + s\)
\end{enumerate}

**R4.11** Find at least five \textit{compile-time} errors in the following program.

```java
public class HasErrors {
    public static void main(){
    System.out.print("Please enter two numbers: ");
    x = in.readDouble;
    y = in.readDouble;
    System.out.print("The sum is "+ x + y);
    }
}
```

**R4.12** Find three \textit{run-time} errors in the following program.

```java
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);
}
}

> R4.13 Consider the following code:

```java
CashRegister register = new CashRegister();
register.recordPurchase(19.93);
register.receivePayment(20, 0, 0, 0, 0);
System.out.print("Change: ");
System.out.println(register.giveChange());
```

The code segment prints the total as 0.07000000000000028. Explain why. Give a recommendation to improve the code so that users will not be confused.

> R4.14 Explain the differences between 2, 2.0, '2', "2", and "2.0".

> R4.15 Explain what each of the following program segments computes.

a. x = 2;
   y = x + x;

b. s = "2";
   t = s + s;

> R4.16 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.

> R4.17 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).

> R4.18 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: Use % and Math.log10.

> R4.19 Modify the pseudocode for the program in How To 4.1 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.

> R4.20 In Worked Example 4.1, it is easy enough to measure the width of a pyramid. To measure the height without climbing to the top, you can use a theodolite and determine the angle between the ground and the line joining the theodolite’s position and the top of the pyramid. What other information do you need in order to compute the surface area?

> R4.21 Suppose an ancient civilization had constructed circular pyramids. Write a program that determines the surface area from measurements that you can determine from the ground.

> R4.22 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 21 for a cone section. Then develop an algorithm that works for arbitrary dimensions.
**R4.23** You are cutting off a piece of pie like this, where \(c\) is the length of the straight part (called the chord length) and \(b\) is the height of the piece.

There is an approximate formula for the area:

\[
A = \frac{2}{3}cb + \frac{b^3}{2c}
\]

However, \(b\) 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.

**R4.24** The following pseudocode describes how to obtain the name of a day, given the day number (0 = Sunday, 1 = Monday, and so on.)

1. Declare a string called names containing "SunMonTueWedThuFriSat".
2. Compute the starting position as 3x the day number.
3. 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 3.

**R4.25** The following pseudocode describes how to swap two letters in a word.

1. We are given a string str and two positions i and j. (i comes before j)
2. Set first to the substring from the start of the string to the last position before i.
3. Set middle to the substring from positions i + 1 to j - 1.
4. Set last to the substring from position j + 1 to the end of the string.
5. 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 3.

**R4.26** How do you get the first character of a string? The last character? How do you remove the first character? The last character?

**R4.27** 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

**R4.28** Write a program that prints the values

- \(3 \times 1000 \times 1000 \times 1000\)
- \(3.0 \times 1000 \times 1000 \times 1000\)

Explain the results.

**PRACTICE EXERCISES**

**E4.1** 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.
E4.2 Write a program that computes and displays the perimeter of a letter-size (8.5 \times 11 inches) sheet of paper and the length of its diagonal.

E4.3 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.

E4.4 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.

E4.5 Enhance the output of Exercise E4.4 so that the numbers are properly aligned:

```
    Sum:            45
Difference:      -5
Product:         500
Average:         22.50
Distance:        5
Maximum:         25
Minimum:         20
```

E4.6 Write a program that prompts the user for a measurement in meters and then converts it to miles, feet, and inches.

E4.7 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

E4.8 Write a program that asks the user for the lengths of a rectangle's sides. Then print
- The area and perimeter of the rectangle
- The length of the diagonal (use the Pythagorean theorem)

E4.9 Improve the program discussed in How To 4.1 to allow input of quarters in addition to bills.

E4.10 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.

E4.11 Change the Menu class in Worked Example 3.1 so that the menu options are labeled A, B, C, and so on. *Hint:* Make a string of the labels.

E4.12 *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.)
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
```

**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.

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
```

**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.

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.

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

**Writing large letters.** A large letter H can be produced like this:

```
*   *
*   *
*****
*   *
*   *
```
It can be declared as a string literal like this:

```java
final string LETTER_H = "* *\n*\n****\n*\n*\n";
```

(The \n escape sequence denotes a “newline” character that causes subsequent characters to be printed on a new line.) Do the same for the letters E, L, and O. Then write the message

```
H
E
L
L
O
```
in large letters.

**E4.19** 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.*

**E4.20** Write a program that prints a Christmas tree:

```
/\ \
/ \\
/ \\
---------
  
  
  
```

Remember to use escape sequences.

**E4.21** Enhance the `CashRegister` class by adding separate methods `enterDollars`, `enterQuarters`, `enterDimes`, `enterNickels`, and `enterPennies`. Use this tester class:

```java
public class CashRegisterTester {
    public static void main (String[] args) {
        CashRegister register = new CashRegister();
        register.recordPurchase(20.37);
        register.enterDollars(20);
        register.enterQuarters(2);
        System.out.println("Change: " + register.giveChange());
        System.out.println("Expected: 0.13");
    }
}
```

**E4.22** Implement a class `IceCreamCone` with methods `getSurfaceArea()` and `getVolume()`. In the constructor, supply the height and radius of the cone. Be careful when looking up the formula for the surface area—you should only include the outside area along the side of the cone because the cone has an opening on the top to hold the ice cream.

**E4.23** Implement a class `SodaCan` whose constructor receives the height and diameter of the soda can. Supply methods `getVolume()` and `getSurfaceArea()`. Supply a `SodaCanTester` class that tests your class.
E4.24 Implement a class \texttt{Balloon} that models a spherical balloon that is being filled with air. The constructor constructs an empty balloon. Supply these methods:

- void \texttt{addAir(double \textit{amount})} adds the given amount of air
- double \texttt{getVolume()} gets the current volume
- double \texttt{getSurfaceArea()} gets the current surface area
- double \texttt{getRadius()} gets the current radius

Supply a \texttt{BalloonTester} class that constructs a balloon, adds 100 cm$^3$ of air, tests the three accessor methods, adds another 100 cm$^3$ of air, and tests the accessor methods again.

P4.1 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.

P4.2 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 $8 \times b + 13$ by 25 to get a quotient $g$. Ignore the remainder.
6. Divide $19 \times a + b - d - g + 15$ 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 $a + 11 \times h$ by 319 to get a quotient $m$. Ignore the remainder.
9. Divide $2 \times e + 2 \times j - k - h + m + 32$ 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{align*}
a &= 6 & g &= 6 & m &= 0 & n &= 4 \\
b &= 20 & c &= 1 & h &= 18 & r &= 6 & p &= 15 \\
d &= 5 & e &= 0 & j &= 0 & k &= 1
\end{align*}
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.

**P4.3** 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

Implement a `Triangle` class with appropriate methods. Supply a program that prompts a user for the corner point coordinates and produces a nicely formatted table of the triangle properties.

**P4.4** A boat floats in a two-dimensional ocean. It has a position and a direction. It can move by a given distance in its current direction, and it can turn by a given angle.

Provide methods

```java
public double getX()
public double getY()
public double getDirection()
public void turn(double degrees)
public void move(double distance)
```

**P4.5** The `CashRegister` class has an unfortunate limitation: It is closely tied to the coin system in the United States and Canada. Research the system used in most of Europe. Your goal is to produce a cash register that works with euros and cents. Rather than designing another limited `CashRegister` implementation for the European market, you should design a separate `Coin` class and a cash register that can work with coins of all types.

**Business P4.6** 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.

1. Read the total book price and the number of books.
2. Compute the tax (7.5 percent of the total book price).
3. Compute the shipping charge ($2 per book).
4. The price of the order is the sum of the total book price, the tax, and the shipping charge.
5. Print the price of the order.

Translate this pseudocode into a Java program.

**Business P4.7** 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".

1. Take the substring consisting of the first three characters and surround it with "(" and ")". This is the area code.
2. 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 P4.8** 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 P4.9** *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.

**Business P4.10** 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 P4.11** 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 P4.12** Consider the following circuit.

Write a program that reads the resistances of the three resistors and computes the total resistance, using Ohm’s law.
-- Science P4.13 --
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 function $\log$ to compute the natural logarithm.

-- Science P4.14 --
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:

$$R = R_0 e^{\frac{\beta}{T} - \frac{\beta}{T_0}}$$

where $R$ is the resistance (in $\Omega$) at the temperature $T$ (in °K), and $R_0$ is the resistance (in $\Omega$) at the temperature $T_0$ (in °K). $\beta$ 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 $\beta$.

The thermistors used to make the pipe clip temperature sensors have $R_0 = 1075 \ \Omega$ at $T_0 = 85 \ ^\circ \text{C}$, and $\beta = 3969 \ ^\circ \text{K}$. (Notice that $\beta$ has units of °K. Recall that the temperature in °K is obtained by adding 273 to the temperature in °C.) The liquid temperature, in °C, is determined from the resistance $R$, in $\Omega$, using

$$T = \frac{\beta T_0}{T_0 \ln\left(\frac{R}{R_0}\right)} - 273 - \beta$$

Write a Java program that prompts the user for the thermistor resistance $R$ and prints a message giving the liquid temperature in °C.

-- Science P4.15 --
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.
Chapter 4  Fundamental Data Types

\( P \) 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.

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

\[
V_s = \sqrt{\left( V_t + \frac{2RP}{V_t} \right) + \left( \frac{2RP}{pfV_t} \right) \left( 1 - pf^2 \right)}
\]

(\( V_s \) 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.

\[ \textbf{Science P4.16} \] Consider the following tuning circuit connected to an antenna, where \( C \) is a variable capacitor whose capacitance ranges from \( C_{\text{min}} \) to \( C_{\text{max}} \).

The tuning circuit selects the frequency \( f = \frac{2\pi}{\sqrt{LC}} \). To design this circuit for a given frequency, take \( C = \sqrt{C_{\text{min}}C_{\text{max}}} \) and calculate the required inductance \( L \) from \( f \) and \( C \). Now the circuit can be tuned to any frequency in the range \( f_{\text{min}} = \frac{2\pi}{\sqrt{LC_{\text{max}}}} \) to \( f_{\text{max}} = \frac{2\pi}{\sqrt{LC_{\text{min}}}} \).

Write a Java program to design a tuning circuit for a given frequency, using a variable capacitor with given values for \( C_{\text{min}} \) and \( C_{\text{max}} \). (A typical input is \( f = 16.7 \) MHz, \( C_{\text{min}} = 14 \) pF, and \( C_{\text{max}} = 365 \) pF.) The program should read in \( f \) (in Hz), \( C_{\text{min}} \) and
\( C_{\text{max}} \) (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 P4.17** 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
\[
F = \frac{Q_1 Q_2}{4 \pi \varepsilon r^2}
\]
Newtons, where \( \varepsilon = 8.854 \times 10^{-12} \) 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. int and double.
2. The world’s most populous country, China, has about \( 1.2 \times 10^9 \) inhabitants. Therefore, individual population counts could be held in an int. However, the world population is over \( 6 \times 10^9 \). If you compute totals or averages of multiple countries, you can exceed the largest int value. Therefore, double is a better choice. You could also use long, but there is no benefit because the exact population of a country is not known at any point in time.
3. The first initialization is incorrect. The right hand side is a value of type double, and it is not legal to initialize an int variable with a double value. The second initialization is correct—an int value can always be converted to a double.
4. The first declaration is used inside a method, the second inside a class.
5. Two things: You should use a named constant, not the “magic number” 3.14, and 3.14 is not an accurate representation of \( \pi \).
6. double interest = balance * percent / 100;  
7. double sideLength = Math.sqrt(area);  
8. \( 4 \times PI \times Math.pow(radius, 3) / 3 \) or \( (4.0 / 3) \times PI \times Math.pow(radius, 3) \), but not \( (4 / 3) \times PI \times Math.pow(radius, 3) \)
9. 17 and 29
10. 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.
11. System.out.print("How old are you? ");  
12. int age = in.nextInt();  
13. There is no prompt that alerts the program user to enter the quantity.
14. 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”.
15. The total volume is 10  
There are four spaces between is and 10. One space originates from the format string (the space between is and %), and three spaces are added before 10 to achieve a field width of 5.
16. Here is a simple solution:  
System.out.printf("Bottles: %8d\n", bottles);  
System.out.printf("Cans: %8d\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\n%-9s%8d\n",  
"Bottles: ", bottles, "Cans: ", cans);
17. int pairs = (totalWidth - tileWidth) / (2 * tileWidth);  
int tiles = 1 + 2 * pairs;  
double gap = (totalWidth - tiles * tileWidth) / 2.0;  
Be sure that pairs is declared as an int.
18. Now there are groups of four tiles (gray/white/gray/black) following the initial black tile. Therefore, the algorithm is now

\[
\text{number of groups} = \text{integer part of } (\text{total width} - \text{tile width}) / (4 \times \text{tile width}) \\
\text{number of tiles} = 1 + 4 \times \text{number of groups}
\]

The formula for the gap is not changed.

19. 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:

<table>
<thead>
<tr>
<th>Row % 2</th>
<th>Column % 2</th>
<th>Color</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

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:

\[
\text{color} = ((\text{row} \% 2) + (\text{column} \% 2)) \% 2
\]

20. 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,

\[
\text{Repair cost in year } n = 100 + n \times 1400 / 9
\]

21. The pseudocode follows from the equations:

\[
\begin{align*}
\text{bottom volume} &= \pi \times r_1^2 \times h_1 \\
\text{top volume} &= \pi \times r_2^2 \times h_2 \\
\text{middle volume} &= \pi \times (r_1^2 + r_1 \times r_2 + r_2^2) \times h_3 / 3 \\
\text{total volume} &= \text{bottom volume} + \text{top volume} + \text{middle volume}
\end{align*}
\]

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,

- bottom volume = 610.73
- top volume = 31.67
- middle volume = 135.72
- total volume = 778.12

The actual volume is 750 ml, which is close enough to our computation to give confidence that it is correct.

22. The length is 12. The space counts as a character.

23. \str.substring(8, 12) or \str.substring(8)

24. \str = \str + "ming";

25. \Hy

26. String first = in.next();
    String middle = in.next();
    String last = in.next();
Computing the Volume and Surface Area of a Pyramid

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

Make a list of all the values that can vary. It is common for beginners to implement classes 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 implementation, even if the original problem only asked about the great pyramid. It is just as easy—and far more useful—to write a class that describes any pyramid.

In our case, a pyramid is described by its height and base length. The desired outputs are the volume and surface area.

Step 2 Work out examples by hand.

An Internet search yields the following diagram for geometric computations with square-based pyramids:

The volume is straightforward. Consider a pyramid whose base and height are 10 cm each. Then the volume is \( \frac{1}{3} \times 10^2 \times 10 = 333.3 \text{ cm}^3 \), or \( \frac{1}{3} \) of the volume of a cube with side length of 10 cm. That makes sense if you are familiar with Archimedes' famous decomposition of a cube into three pyramids.

The surface area is not so clear. Looking at the formula \( A = 2bs + b^2 \), we note that the formula gives the entire area, including the square bottom. That's what you would need if you wanted to find out how much paint you need for a paper model of a pyramid. But do our researchers care about the bottom square that is not exposed? You would need to check back with them. Let's say they reply that they only want the part above the ground. Then the formula becomes \( A = 2bs \).

Unfortunately, the value \( s \) is not one of our inputs, so we need to compute it. Look at the colored triangle in the figure above. It is a right triangle with sides \( s, b, \) and \( b/2 \). The Pythagorean theorem tells us that \( s^2 = b^2 + (b/2)^2 \).

Now let's try again. If \( h \) and \( b \) are both 10, then \( s^2 = 10^2 + 5^2 = 125 \), and \( s = \sqrt{125} \). Then the area is \( A = 2bs = 20 \times \sqrt{125} \), or about 224. This is plausible because four sides of a cube with side...
length 10 have area 400, and you would expect that area to be somewhat larger than the four sides of our sample pyramid.

Having solved this example by hand, we are now better prepared to implement the necessary computations in Java.

**Step 3**  Design a class that carries out your computations.

According to How To 3.1, we need to determine methods and instance variables for our class. In our case, the problem statement yields the following constructor and methods:

- `public Pyramid(double height, double baseLength)`
- `public double getVolume()`
- `public double getSurfaceArea()`

Determining the instance variables requires some thought. Consider these alternatives:

- A pyramid stores its height and base length. The volume and surface area are computed as needed in the `getVolume` and `getSurfaceArea` methods.
- A pyramid stores its volume and surface area. They are computed in the constructor from the `height` and `baseLength`, which are then discarded.

Both approaches will work for our problem. There is no simple rule as to which design is better. One way of settling the question is to consider how the `Pyramid` class might evolve. More methods for geometrical computations (such as angles) might be added. There might be methods to resize the pyramid. The first alternative makes it easier to accommodate those scenarios. Moreover, it seems more object-oriented. A pyramid is described by its height and base, not by its volume and surface area.

**Step 4**  Write pseudocode for implementing the methods.

As already described, the volume is simply

\[ \text{volume} = \frac{(\text{base} \times \text{base} \times \text{height})}{3} \]

For the surface area, we first need the side length

\[ \text{side length} = \text{square root of} \ (\text{height} \times \text{height} + \text{base} \times \text{base} / 4) \]

Then we have

\[ \text{surface area} = 2 \times \text{base} \times \text{side length} \]

**Step 5**  Implement the class.

As decided in Step 3, we have instance variables for the height and base length:

```java
public class Pyramid
{
    private double height;
    private double baseLength;
    ...;
}
```

The methods for computing the volume and surface area are now straightforward.

```java
public double getVolume()
{
    return height * baseLength * baseLength / 3;
}
```

```java
public double getSurfaceArea()
{
    double sideLength = Math.sqrt(height * height + baseLength * baseLength / 4);
    return 2 * baseLength * sideLength;
}
```
There is a minor issue with the constructor. As described in Step 3, the parameter variables of
the constructor are identical to those of the instance variables:

```java
public Pyramid(double height, double baseLength)
```

One solution is simply to rename the constructor parameter variables:

```java
public Pyramid(double aHeight, double aBaseLength)
{
    height = aHeight;
    baseLength = aBaseLength;
}
```

This approach has a small disadvantage. The awkward parameter variable names leak into the
API documentation:

```java
/**
   * Constructs a pyramid with a given height and base length.
   * @param aHeight the height
   * @param aBaseLength the length of one of the sides of the square base
   */
```

If you prefer, you can avoid that issue by using the `this` reference as follows:

```java
public Pyramid(double height, double baseLength)
{
    this.height = height;
    this.baseLength = baseLength;
}
```

We have now completed the class implementation. You can find the complete program in the
`ch04/worked_example_1` directory of the book’s companion code.

**Step 6**  Test your class.

We can use the computations from Step 2 as a test case:

```java
Pyramid sample = new Pyramid(10, 10);
System.out.println(sample.getVolume());
System.out.println("Expected: 333.33");
System.out.println(sample.getSurfaceArea());
System.out.println("Expected: 224");
```

It is a good idea to have another test case where the height and base are different, to check that
the constructor is taking the parameters in the correct order. An Internet search yields an esti-
mate of about 2,500,000 cubic meters for the Giza pyramid.

```java
Pyramid gizeh = new Pyramid(146, 230);
System.out.println(gizeh.getVolume());
System.out.println("Expected: 2500000");
```

The program output is:

```
333.3333333333333
Expected: 333.33
223.60679774997897
Expected: 224
2574466.6666666665
Expected: 2500000
```

The answers match well, and we decide that the test was successful.
**WORKED EXAMPLE 4.2 Computing Travel Time**

In this Worked Example, we develop a hand calculation to compute travel time that we then use to develop pseudocode and program statements that will perform the calculation.

**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.

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

You will be 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)

You are expected to compute the total travel time.

**Step 2** Work out examples by hand.

To calculate an example by hand, 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. 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.

**Step 3** Write pseudocode for implementing the computation.

Look again at the steps in the hand calculation. The steps didn’t depend on the particular values. Therefore, you can reformulate them as pseudocode by replacing the actual values with their names:

- Time for segment 1 = $l_1 / s_1$
- Length of segment 2 = square root of $(dx - l_1)^2 + dy^2$
- Time for segment 2 = length of segment 2 / $s_2$
- Total time = time for segment 1 + time for segment 2

**Step 4** Translate the pseudocode into Java.

When you do hand calculations, it is convenient to use short variable names such as $dx$ or $s_1$. In your program, you should change them to names that are longer and more descriptive.

Translated into Java, the computations are

```java
double segment1Time = segment1Length / segment1Speed;
double segment2Length = Math.sqrt(
    Math.pow(xDistance - segment1Length, 2) +
    Math.pow(yDistance, 2));
double segment2Time = segment2Length / segment2Speed;
double totalTime = segment1Time + segment2Time;
```

You can find the complete program in the ch04/worked_example_2 directory of the book’s companion code.
# 5
## DECISIONS

### CHAPTER GOALS

To implement decisions using if statements
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

#### 5.1 THE IF STATEMENT

<table>
<thead>
<tr>
<th>SYN</th>
<th>if Statement 180</th>
</tr>
</thead>
<tbody>
<tr>
<td>PT1</td>
<td>Brace Layout 181</td>
</tr>
<tr>
<td>PT2</td>
<td>Always Use Braces 181</td>
</tr>
<tr>
<td>CE1</td>
<td>A Semicolon After the if Condition 182</td>
</tr>
<tr>
<td>PT3</td>
<td>Tabs 182</td>
</tr>
<tr>
<td>ST1</td>
<td>The Conditional Operator 182</td>
</tr>
<tr>
<td>PT4</td>
<td>Avoid Duplication in Branches 183</td>
</tr>
</tbody>
</table>

#### 5.2 COMPARING VALUES

<table>
<thead>
<tr>
<th>SYN</th>
<th>Comparisons 184</th>
</tr>
</thead>
<tbody>
<tr>
<td>CE1</td>
<td>Using == to Compare Strings 189</td>
</tr>
<tr>
<td>HT1</td>
<td>Implementing an if Statement 190</td>
</tr>
<tr>
<td>WE1</td>
<td>Extracting the Middle</td>
</tr>
<tr>
<td>C&amp;S</td>
<td>Denver’s Luggage Handling System 192</td>
</tr>
</tbody>
</table>

#### 5.3 MULTIPLE ALTERNATIVES

| ST2  | The switch Statement 196 |

#### 5.4 NESTED BRANCHES

| PT5  | Hand-Tracing 200 |
| CE3  | The Dangling else Problem 201 |
| ST3  | Block Scope 201 |
| ST4  | Enumeration Types 203 |

#### 5.5 PROBLEM SOLVING: FLOWCHARTS

| PT6  | Make a Schedule and Make Time for Unexpected Problems 208 |
| ST5  | Logging 208 |

#### 5.6 PROBLEM SOLVING: SELECTING TEST CASES

| CE4  | Combining Multiple Relational Operators 212 |
| CE5  | Confusing && and || Conditions 212 |
| ST6  | Short-Circuit Evaluation of Boolean Operators 213 |
| ST7  | De Morgan’s Law 213 |

#### 5.7 BOOLEAN VARIABLES AND OPERATORS

| CE4  | Combining Multiple Relational Operators 212 |
| CE5  | Confusing && and || Conditions 212 |
| ST6  | Short-Circuit Evaluation of Boolean Operators 213 |
| ST7  | De Morgan’s Law 213 |

#### 5.8 APPLICATION: INPUT VALIDATION

| C&S  | Artificial Intelligence 217 |
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.

5.1 The if Statement

The if statement is used to implement a decision (see Syntax 5.1 on page 180). 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. 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 to be 19. Otherwise, it simply uses the supplied floor number.

```java
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 there is nothing to do in the else branch of the statement. In that case, you can omit it entirely, as in this example:

```java
int actualFloor = floor;
if (floor > 13)
{
    actualFloor--;  // No else needed
}
```

See Figure 2 for the flowchart.
The following program puts the if statement to work. This program asks for the desired floor and then prints out the actual floor.

```
import java.util.Scanner;

/**
 * This program simulates an elevator panel that skips the 13th floor.
 */
public class ElevatorSimulation {
    public static void main(String[] args) {
        Scanner in = new Scanner(System.in);
        System.out.print("Floor: ");
        int floor = in.nextInt();

        // Adjust floor if necessary
        int actualFloor;
        if (floor > 13) {
            actualFloor = floor - 1;
        } else {
            actualFloor = floor;
        }

        System.out.println("The elevator will travel to the actual floor "+ actualFloor);
    }
}
```

Program Run
Floor: 20
The elevator will travel to the actual floor 19
Syntax 5.1  if Statement

Syntax  
if (condition)  
{  
  statements  
}  
else  
{  
  statements  
}

A condition that is true or false. Often uses relational operators:  
== != <= >= (See page 184.)

if (floor > 13)  
{  
  actualFloor = floor - 1;  
}  
else  
{  
  actualFloor = floor;  
}

Lining up braces is a good idea. See page 181.

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:

```java
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: R5.5, R5.6, E5.10.

---

**Programming Tip 5.1**

The compiler doesn’t care where you place braces. In this book, we follow the simple rule of making { and } line up.

```java
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:

```java
if (floor > 13) {
    floor--; 
}
```

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 5.2**

When the body of an if statement consists of a single statement, you need not use braces. For example, the following is legal:

```java
if (floor > 13) 
    floor--; 
```

However, it is a good idea to always include the braces:

```java
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.
A Semicolon After the if Condition

The following code fragment has an unfortunate error:

```java
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 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.

Tabs

Block-structured code has the property that nested statements are indented by one or more levels:

```java
public class ElevatorSimulation
{
    public static void main(String[] args)
    {
        int floor;
        . . .
        if (floor > 13)
        {
            floor--;  
        }
        . . .
    }
```

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.

The Conditional Operator

Java has a conditional operator of the form

```java
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

```java
actualFloor = floor > 13 ? floor - 1 : floor;
```

which is equivalent to

```java
if (floor > 13) { actualFloor = floor - 1; } else { actualFloor = floor; }
```
You can use the conditional operator anywhere that a value is expected, for example:

```java
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.

**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:

```java
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. But you can simplify the program by moving the duplicated statement:

```java
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.

## 5.2 Comparing Values

Every if statement contains a condition. In many cases, the condition involves comparing two values. In the following sections, you will learn how to implement comparisons in Java.

_In Java, you use a relational operator to check whether one value is greater than another._
5.2.1 Relational Operators

A relational operator tests the relationship between two values. An example is the > operator that we used in the test `floor > 13`. Java has six relational operators (see Table 1).

<table>
<thead>
<tr>
<th>Table 1</th>
<th>Relational Operators</th>
</tr>
</thead>
<tbody>
<tr>
<td>Java</td>
<td>Math Notation</td>
</tr>
<tr>
<td>&gt;</td>
<td>&gt;</td>
</tr>
<tr>
<td>&gt;=</td>
<td>≥</td>
</tr>
<tr>
<td>&lt;</td>
<td>&lt;</td>
</tr>
<tr>
<td>&lt;=</td>
<td>≤</td>
</tr>
<tr>
<td>==</td>
<td>=</td>
</tr>
<tr>
<td>!=</td>
<td>≠</td>
</tr>
</tbody>
</table>

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 ≥, ≤, or ≠, but the >=, <=, and != operators are easy to remember because they look similar. The == operator is initially confusing to most newcomers to Java.

Syntax 5.2 Comparisons

These quantities are compared.

- `floor > 13`
  - One of: != < <= >= (See Table 1.)
- `floor == 13`
  - Checks for equality.
- String input;
  - if (input.equals("Y"))
  - Use equals to compare strings. (See page 186.)
- `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 page 185.
In Java, = already has a meaning, namely assignment. The == operator denotes equality testing:

```java
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.

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

```java
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.

### 5.2.2 Comparing Floating-Point Numbers

You have to be careful when comparing floating-point numbers in order to cope with roundoff errors. For example, the following code multiplies the square root of 2 by itself and then subtracts 2.

```java
double r = Math.sqrt(2);
double d = r * r - 2;
if (d == 0)
{
    System.out.println("sqrt(2) squared minus 2 is 0");
}
else
{
    System.out.println("sqrt(2) squared minus 2 is not 0 but " + d);
}
```

Even though the laws of mathematics tell us that \((\sqrt{2})^2 - 2\) equals 0, this program fragment prints

```
sqrt(2) squared minus 2 is not 0 but 4.440892098500626E-16
```

Unfortunately, such roundoff errors are unavoidable. It plainly does not make sense in most circumstances to compare floating-point numbers exactly. Instead, test whether they are close enough.

To test whether a number \(x\) is close to zero, you can test whether the absolute value \(|x|\) (that is, the number with its sign removed) is less than a very small threshold number. That threshold value is often called \(\varepsilon\) (the Greek letter epsilon). It is common to set \(\varepsilon\) to \(10^{-14}\) when testing `double` numbers.

Similarly, you can test whether two numbers are approximately equal by checking whether their difference is close to 0.

\[|x - y| \leq \varepsilon\]

In Java, we program the test as follows:

```java
final double EPSILON = 1E-14;
if (Math.abs(x - y) <= EPSILON)
{
    // x is approximately equal to y
}
```
5.2.3 Comparing Strings

To test whether two strings are equal to each other, you must use the method called equals:

```java
if (string1.equals(string2)) . . .
```

Do not use the `==` operator to compare strings. The comparison

```java
if (string1 == string2) // Not useful
```

has an unrelated meaning. It tests whether the two strings are stored in the same memory location. You can have strings with identical contents stored in different locations, so this test never makes sense in actual programming; see Common Error 5.2 on page 189.

If two strings are not identical, 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

```java
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".

Conversely, if

```java
string1.compareTo(string2) > 0
```

then `string1` comes after `string2` in dictionary order.

Finally, if

```java
string1.compareTo(string2) == 0
```

then `string1` and `string2` are equal.

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.

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, compare "cat" with "cart". The first two letters match. Because t comes after r, the string "cat" comes after "cart" in the lexicographic ordering.

To see which of two terms comes first in the dictionary, consider the first letter in which they differ.
5.2.4 Comparing Objects

If you compare two object references with the == operator, you test whether the references refer to the same object. Here is an example:

```
Rectangle box1 = new Rectangle(5, 10, 20, 30);
Rectangle box2 = box1;
Rectangle box3 = new Rectangle(5, 10, 20, 30);
```

The comparison

```
box1 == box2
```

is true. Both object variables refer to the same object. But the comparison

```
box1 == box3
```

is false. The two object variables refer to different objects (see Figure 3). It does not matter that the objects have identical contents.

You can use the equals method to test whether two rectangles have the same contents, that is, whether they have the same upper-left corner and the same width and height. For example, the test

```
box1.equals(box3)
```

is true.

However, you must be careful when using the equals method. It works correctly only if the implementors of the class have supplied it. The Rectangle class has an equals method that is suitable for comparing rectangles.

For your own classes, you need to supply an appropriate equals method. You will learn how to do that in Chapter 9. Until that point, you should not use the equals method to compare objects of your own classes.

![Figure 3](Comparing Object References)

5.2.5 Testing for null

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; // Not set
if ( . . . )
{
    middleInitial = middleName.substring(0, 1);
}
```
Table 2 Relational Operator Examples

<table>
<thead>
<tr>
<th>Expression</th>
<th>Value</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>3 &lt;= 4</td>
<td>true</td>
<td>3 is less than 4; &lt;= tests for “less than or equal”.</td>
</tr>
<tr>
<td>3 ==&lt; 4</td>
<td>Error</td>
<td>The “less than or equal” operator is &lt;=, not =&lt;. The “less than” symbol comes first.</td>
</tr>
<tr>
<td>3 &gt; 4</td>
<td>false</td>
<td>&gt; is the opposite of &lt;=.</td>
</tr>
<tr>
<td>4 &lt; 4</td>
<td>false</td>
<td>The left-hand side must be strictly smaller than the right-hand side.</td>
</tr>
<tr>
<td>4 &lt;= 4</td>
<td>true</td>
<td>Both sides are equal; &lt;= tests for “less than or equal”.</td>
</tr>
<tr>
<td>3 == 5 - 2</td>
<td>true</td>
<td>== tests for equality.</td>
</tr>
<tr>
<td>3 != 5 - 1</td>
<td>true</td>
<td>!= tests for inequality. It is true that 3 is not 5 – 1.</td>
</tr>
<tr>
<td>3 = 6 / 2</td>
<td>Error</td>
<td>Use == to test for equality.</td>
</tr>
<tr>
<td>1.0 / 3.0 == 0.333333333</td>
<td>false</td>
<td>Although the values are very close to one another, they are not exactly equal. See Section 5.2.2.</td>
</tr>
<tr>
<td>“10” &gt; 5</td>
<td>Error</td>
<td>You cannot compare a string to a number.</td>
</tr>
<tr>
<td>“Tomato”.substring(0, 3).equals(“Tom”)</td>
<td>true</td>
<td>Always use the equals method to check whether two strings have the same contents.</td>
</tr>
<tr>
<td>“Tomato”.substring(0, 3) == (“Tom”)</td>
<td>false</td>
<td>Never use == to compare strings; it only checks whether the strings are stored in the same location. See Common Error 5.2 on page 189.</td>
</tr>
</tbody>
</table>

You use the == operator (and not equals) to test whether an object reference is a null reference:

```java
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.

Table 2 summarizes how to compare values in Java.

6. Which of the following conditions are true, provided a is 3 and b is 4?
   - a. a + 1 <= b
   - b. a + 1 >= b
   - c. a + 1 != b
7. Give the opposite of the condition
   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. Give two ways of testing that a string str is the empty string.
11. What is the value of s.length() if s is
   a. the empty string ""?
   b. the string " " containing a space?
   c. null?
12. Which of the following comparisons are syntactically incorrect? Which of them
   are syntactically correct, but logically questionable?
   String a = "1";
   String b = "one";
   double x = 1;
   double y = 3 * (1.0 / 3);
   a. a == "1"
   b. a == null
   c. a.equals(""")
   d. a == b
   e. a == x
   f. x == y
   g. x - y == null
   h. x.equals(y)

**Practice It** Now you can try these exercises at the end of the chapter: R5.4, R5.7, E5.14.

**Common Error 5.2**

**Using == to Compare Strings**

If you write

```java
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:

```java
String nickname = "Rob";
```

```java
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:

```java
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.

### 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.

### 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:

*discounted price = 0.92 x 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 5.4 on page 183.)

In our example, we have two statements of the form

\[ \text{discounted price} = \_\_\_ x \text{original price} \]

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 \times \text{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 \times 100 = 92

When the original price is 200, then the condition \( 200 < 128 \) is false, and

- discount rate = 0.84
- discounted price = 0.84 \times 200 = 168

In both cases, we get the expected answer.

**Step 7** Assemble the if statement in Java.

Type the skeleton

```java
if ()
{
}
else
{
}
```

and fill it in, as shown in Syntax 5.1 on page 180. Omit the else branch if it is not needed.
In our example, the completed statement is

```c
if (originalPrice < 128)
{
    discountRate = 0.92;
}
else
{
    discountRate = 0.84;
}
discountedPrice = discountRate * originalPrice;
```

### WORKED EXAMPLE 5.1  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 www.wiley.com/go/bjeo6examples and download Worked Example 5.1.

### Computing & Society 5.1  Denver’s Luggage Handling 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.

© Alex Slobodkin/iStockphoto.

© Tom Horyn/iStockphoto.

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.
In Section 5.1, 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).

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 4 shows the flowchart for this multiple-branch statement.

You use multiple if statements to implement multiple alternatives, like this:

```java
if (richter >= 8.0)
{
    description = "Most structures fall";
}
else if (richter >= 7.0)
{
    description = "Many buildings destroyed";
}
else if (richter >= 6.0)
{
    description = "Many buildings considerably damaged, some collapse";
}
else if (richter >= 4.5)
{
    description = "Damage to poorly constructed buildings";
}
else
{
    description = "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.
Here you must sort the conditions and test against the largest cutoff first. Suppose we reverse the order of tests:

```java
if (richter >= 4.5) // Tests in wrong order
{
    description = "Damage to poorly constructed buildings";
}
else if (richter >= 6.0)
{
    description = "Many buildings considerably damaged, some collapse";
}
else if (richter >= 7.0)
{
    description = "Many buildings destroyed";
}
else if (richter >= 8.0)
{
    description = "Most structures fall";
}
```
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 \( \text{richter} \geq 8.0 \) is more specific than the condition \( \text{richter} \geq 7.0 \), and the condition \( \text{richter} \geq 4.5 \) 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.

```c
if (richter >= 8.0) // Didn't use else
{
    description = "Most structures fall";
}
if (richter >= 7.0)
{
    description = "Many buildings destroyed";
}
if (richter >= 6.0)
{
    description = "Many buildings considerably damaged, some collapse";
}
if (richter >= 4.5)
{
    "Damage to poorly constructed buildings";
}
```

Now the alternatives are no longer exclusive. If \( \text{richter} \) is 7.1, then the last three tests all match. The description variable is set to three different strings, ending up with the wrong one.

13. In a game program, the scores of players A and B are stored in variables \( \text{scoreA} \) and \( \text{scoreB} \). 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".

14. 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.

15. How could you achieve the task of Self Check 14 with only two branches?

16. Beginners sometimes write statements such as the following:

```c
if (price > 100)
{
    discountedPrice = price - 20;
}
else if (price <= 100)
{
    discountedPrice = price - 10;
}
```

Explain how this code can be improved.

17. Suppose the user enters -1 into the earthquake program. What is printed?

18. 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: R5.23, E5.11, E5.24.
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,

```java
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

```java
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.

5.4 Nested Branches

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 that were 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>
<thead>
<tr>
<th>Table 4 Federal Tax Rate Schedule</th>
</tr>
</thead>
<tbody>
<tr>
<td>If your status is Single and if the taxable income is at most $32,000, the tax is 10% of the amount over $0. Over $32,000, the tax is $3,200 + 25% of the amount over $32,000.</td>
</tr>
<tr>
<td>If your status is Married and if the taxable income is at most $64,000, the tax is 10% of the amount over $0. Over $64,000, the tax is $6,400 + 25% of the amount over $64,000.</td>
</tr>
</tbody>
</table>

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 5 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.

Figure 5 Income Tax Computation
section_4/TaxReturn.java

/*
*/
public class TaxReturn {
    public static final int SINGLE = 1;
    public static final int MARRIED = 2;
    private static final double RATE1 = 0.10;
    private static final double RATE2 = 0.25;
    private static final double RATE1_SINGLE_LIMIT = 32000;
    private static final double RATE1_MARRIED_LIMIT = 64000;
    private double income;
    private int status;

    /**
     * Constructs a TaxReturn object for a given income and marital status.
     * @param anIncome the taxpayer income
     * @param aStatus either SINGLE or MARRIED
     */
    public TaxReturn(double anIncome, int aStatus) {
        income = anIncome;
        status = aStatus;
    }

    public double getTax() {
        double tax1 = 0;
        double tax2 = 0;
        if (status == SINGLE) {
            if (income <= RATE1_SINGLE_LIMIT) {
                tax1 = RATE1 * income;
            } else {
                tax1 = RATE1 * RATE1_SINGLE_LIMIT;
                tax2 = RATE2 * (income - RATE1_SINGLE_LIMIT);
            }
        } else {
            if (income <= RATE1_MARRIED_LIMIT) {
                tax1 = RATE1 * income;
            } else {
                tax1 = RATE1 * RATE1_MARRIED_LIMIT;
                tax2 = RATE2 * (income - RATE1_MARRIED_LIMIT);
            }
        }
        return tax1 + tax2;
5.4 Nested Branches

section_4/TaxCalculator.java

```java
import java.util.Scanner;

/**
   * This program calculates a simple tax return.
   */

public class TaxCalculator
{
    public static void main(String[] args)
    {
        Scanner in = new Scanner(System.in);

        System.out.print("Please enter your income: ");
        double income = in.nextDouble();

        System.out.print("Are you married? (Y/N) ");
        String input = in.next();
        int status;
        if (input.equals("Y"))
        {
            status = TaxReturn.MARRIED;
        }
        else
        {
            status = TaxReturn.SINGLE;
        }
        TaxReturn aTaxReturn = new TaxReturn(income, status);
        System.out.println("Tax: "+ aTaxReturn.getTax());
    }
}
```

Program Run

Please enter your income: 80000
Are you married? (Y/N) Y
Tax: 10400.0

19. What is the amount of tax that a single taxpayer pays on an income of $32,000?
20. Would that amount change if the first nested if statement changed from
    if (income <= RATE1_SINGLE_LIMIT)
    to
    if (income < RATE1_SINGLE_LIMIT)
21. Suppose Harry and Sally each make $40,000 per year. Would they save taxes if they married?
22. 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., Y or N)?
23. 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: R5.10, R5.22, E5.15, E5.18.
**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.

```java
public double getTax()
{
    double tax1 = 0;
    double tax2 = 0;
}
```

Because `status` is not `SINGLE`, we move to the `else` branch of the outer if statement (line 46).

```java
if (status == SINGLE)
{
    if (income <= RATE1_SINGLE_LIMIT)
    {
        tax1 = RATE1 * income;
    }
    else
    {
        tax1 = RATE1 * RATE1_SINGLE_LIMIT;
        tax2 = RATE2 * (income - RATE1_SINGLE_LIMIT);
    }
}
else
{
}
```

Because `income` is not <= 64000, we move to the `else` branch of the inner if statement (line 52).

```java
if (income <= RATE1_MARRIED_LIMIT)
{
    tax1 = RATE1 * income;
}
else
{
    tax1 = RATE1 * RATE1_MARRIED_LIMIT;
    tax2 = RATE2 * (income - RATE1_MARRIED_LIMIT);
}
```

The values of `tax1` and `tax2` are updated.

```java
{  
tax1 = RATE1 * RATE1_MARRIED_LIMIT;
tax2 = RATE2 * (income - RATE1_MARRIED_LIMIT);
}
```

Their sum is returned and the method ends.

```java
return tax1 + tax2;
```

Because the program trace shows the expected return value ($10,400), it successfully demonstrates that this test case works correctly.
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 5.2 on page 181:

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
  }

Block Scope

A block is a sequence of statements that is enclosed in braces. For example, consider this statement:

if (status == TAXABLE)
  {
    double tax = price * TAX_RATE;
    price = price + tax;
  }

The highlighted part is a block. You can declare a variable in a block, such as the tax variable in this example. Such a variable is only visible inside the block.

  
  { 
    double tax = price * TAX_RATE; // Variable declared inside a block
    price = price + tax;
  }

// You can no longer access the tax variable here
In fact, the variable is only created after the program enters the block, and it is removed as soon as the program exits the block. Such a variable is said to have \textit{block scope}. In general, the \textit{scope} of a variable is the part of the program in which the variable can be accessed. A variable with block scope is visible only inside a block.

It is considered good design to minimize the scope of a variable. This reduces the possibility of accidental modification and name conflicts. For example, as long as the \textit{tax} variable is not needed outside the block, it is a good idea to declare it inside the block. However, if you need the variable outside the block, you must define it outside. For example,

\begin{verbatim}
    double tax = 0;
    if (status == TAXABLE)
    {
        tax = price * TAX_RATE;
    }
    price = price + tax;
\end{verbatim}

Here, the \textit{tax} variable is used outside the block of the \texttt{if} statement, and you must declare it outside.

In Java, the scope of a local variable can never contain the declaration of another local variable with the same name. For example, the following is an error:

\begin{verbatim}
    double tax = 0;
    if (status == TAXABLE)
    {
        double tax = price * TAX_RATE;
        // Error: Cannot declare another variable with the same name
        price = price + tax;
    }
\end{verbatim}

However, you can have local variables with identical names if their scopes do not overlap, such as

\begin{verbatim}
    if (Math.random() > 0.5)
    {
        Rectangle r = new Rectangle(5, 10, 20, 30);
        
    } // Scope of \texttt{r} ends here
    else
    {
        int r = 5;
        // OK—it is legal to declare another \texttt{r} here
    }
\end{verbatim}

These variables are independent from each other. You can have local variables with the same name, as long as their scopes don’t overlap.
5.5 Problem Solving: Flowcharts

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 can help to draw a flowchart to visualize the flow of control.

The basic flowchart elements are shown in Figure 6.

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 7).
Each branch of a decision can contain tasks and further decisions.

Never point an arrow inside another branch.

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 8.

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. 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:

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.

Spaghetti code has so many pathways that it becomes impossible to understand.
24. Draw a flowchart for a program that reads a value \( \text{temp} \) and prints “Frozen” if it is less than zero.

25. What is wrong with the flowchart at right?

26. How do you fix the flowchart of Self Check 25?

27. Draw a flowchart for a program that reads a value \( x \). If it is less than zero, print “Error”. Otherwise, print its square root.

28. Draw a flowchart for a program that reads a value \( \text{temp} \). 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: R5.13, R5.14, R5.15.

### 5.6 Problem Solving: Selecting Test Cases

Testing the functionality of a program without consideration of its internal structure is called **black-box testing**. This is an important part of testing, because, after all, the users of a program do not know its internal structure. If a program works perfectly on all inputs, then it surely does its job.

However, it is impossible to ensure absolutely that a program will work correctly on all inputs just by supplying a finite number of test cases. As the famous computer scientist Edsger Dijkstra pointed out, testing can show only the presence of bugs—not their absence. To gain more confidence in the correctness of a program, it is useful to consider its internal structure. Testing strategies that look inside a program are called **white-box testing**. Performing unit tests of each method is a part of white-box testing.

You want to make sure that each part of your program is exercised at least once by one of your test cases. This is called **code coverage**. If some code is never executed by any of your test cases, you have no way of knowing whether that code would perform correctly if it ever were executed by user input. That means that you need to look at every `if/else` branch to see that each of them is reached by some test case. Many conditional branches are in the code only to take care of strange and abnormal inputs, but they still do something. It is a common phenomenon that they end up doing something incorrectly, but those faults are never discovered during testing, because nobody supplied the strange and abnormal inputs. The remedy is to ensure that each part of the code is covered by some test case.

For example, in testing the `getTax` method of the `TaxReturn` class, you want to make sure that every `if` statement is entered for at least one test case. You should test both single and married taxpayers, with incomes in each of the three tax brackets.

When you select test cases, you should make it a habit to include **boundary test cases**: legal values that lie at the boundary of the set of acceptable inputs.
Here is a plan for obtaining a comprehensive set of test cases for the tax program:

- 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 5.8), also test an invalid input, such as a negative income.

Make a list of the test cases and the expected outputs:

<table>
<thead>
<tr>
<th>Test Case</th>
<th>Married</th>
<th>Expected Output</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>30,000</td>
<td>N</td>
<td>3,000</td>
<td>10% bracket</td>
</tr>
<tr>
<td>72,000</td>
<td>N</td>
<td>13,200</td>
<td>3,200 + 25% of 40,000</td>
</tr>
<tr>
<td>50,000</td>
<td>Y</td>
<td>5,000</td>
<td>10% bracket</td>
</tr>
<tr>
<td>104,000</td>
<td>Y</td>
<td>16,400</td>
<td>6,400 + 25% of 40,000</td>
</tr>
<tr>
<td>32,000</td>
<td>N</td>
<td>3,200</td>
<td>boundary case</td>
</tr>
<tr>
<td>0</td>
<td></td>
<td>0</td>
<td>boundary case</td>
</tr>
</tbody>
</table>

When you develop a set of test cases, it is helpful to have a flowchart of your program (see Section 5.5). 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.

29. Using Figure 1 on page 179 as a guide, follow the process described in this section to design a set of test cases for the ElevatorSimulation.java program in Section 5.1.

30. What is a boundary test case for the algorithm in How To 5.1 on page 190? What is the expected output?

31. Using Figure 4 on page 194 as a guide, follow the process described in Section 5.6 to design a set of test cases for the Earthquake.java program in Section 5.3.

32. 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: R5.16, R5.17.
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.

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:

```java
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.getLogger(). (Prior to Java 7, you obtained the global logger as Logger.getLogger("global").) Then call the info method:

```java
Logger.getLogger().info("status is SINGLE");
```

By default, the message is printed. But if you call

```java
Logger.getLogger().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.getLogger().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.

### 5.7 Boolean Variables and Operators

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:

```java
boolean failed = true;
```

You can use the value later in your program to make a decision:

```java
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.

| A  | B  | A && B | A  | B  | A || B | A  |   |
|----|----|--------|----|----|--------|----|---|
| true | true | true  | true | true | true  | true | !A |
| true | false | false | true | false | true  | false | true |
| false | true | false | false | true | true  | false | true |
| false | false | false | false | false | false | false | false |

**Figure 9  Boolean Truth Tables**
At this geyser in Iceland, you can see ice, liquid water, and steam.

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:

```java
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 9).

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

```java
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.

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.

![Flowcharts for and and or Combinations](image-url)
5.7 Boolean Variables and Operators

Table 5 Boolean Operator Examples

<table>
<thead>
<tr>
<th>Expression</th>
<th>Value</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 &lt; 200 &amp;&amp; 200 &lt; 100</td>
<td>false</td>
<td>Only the first condition is true.</td>
</tr>
<tr>
<td>0 &lt; 200</td>
<td></td>
<td>200 &lt; 100</td>
</tr>
<tr>
<td>0 &lt; 200</td>
<td></td>
<td>100 &lt; 200</td>
</tr>
</tbody>
</table>

\[0 < x \&\& x < 100 \mid x == -1\] \[0 < x \&\& (x < 100) \mid x == -1\]

The && operator has a higher precedence than the || operator (see Appendix B).

\[0 < x < 100\]

Error: This expression does not test whether \( x \) is between 0 and 100. The expression \( 0 < x \) is a Boolean value. You cannot compare a Boolean value with the integer 100.

\[x \&\& y > 0\]

Error: This expression does not test whether \( x \) and \( y \) are positive. The left-hand side of && is an integer, \( x \), and the right-hand side, \( y > 0 \), is a Boolean value. You cannot use && with an integer argument.

\!(0 < 200)\]

false 0 < 200 is true, therefore its negation is false.

\[\text{frozen} == \text{true}\]

frozen There is no need to compare a Boolean variable with true.

\[\text{frozen} == \text{false}\]

!frozen It is clearer to use ! than to compare with false.

Use the || (or) operator to combine the expressions:

\[
\text{if (temp} \leq 0 || \text{temp} \geq 100) \{ \text{System.out.println("Not liquid"); } \}
\]

Figure 10 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:

\[
\text{if (!frozen) \{ System.out.println("Not frozen"); } \}
\]

Table 5 illustrates additional examples of evaluating Boolean operators.

33. Suppose \( x \) and \( y \) are two integers. How do you test whether both of them are zero?

34. How do you test whether at least one of them is zero?

35. How do you test whether exactly one of them is zero?

36. What is the value of !!frozen?

37. 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: R5.30, E5.23, E5.24.
Combining Multiple Relational Operators

Consider the expression

```java
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, $0 \leq \text{temp}$, 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:

```java
if (0 <= temp && temp <= 100) . . .
```

Another common error, along the same lines, is to write

```java
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:

```java
if (input == 1 || input == 2) . . .
```

Confusing && and || Conditions

It is a surprisingly common error to confuse 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.

Because 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.
5.7 Boolean Variables and Operators

**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

\[
\text{quantity} > 0 \quad \&\& \quad \text{price / quantity} < 10
\]

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**.

*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.*

**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:

```java
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:

\[
\neg (A \quad \&\& \quad B) \quad \text{is the same as} \quad \neg A \quad || \quad \neg B
\]

\[
\neg (A \quad || \quad B) \quad \text{is the same as} \quad \neg A \quad && \quad \neg B
\]

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”,

\[
\neg (\text{state.equals("AK") || state.equals("HI"))}
\]

is “the state is not Alaska and it is not Hawaii”:

\[
\neg \text{state.equals("AK") && \neg state.equals("HI")}
\]
Now apply the law to our shipping charge computation:

```java
!(country.equals("USA")
 && !state.equals("AK")
 && !state.equals("HI"))
```

is equivalent to

```java
!(country.equals("USA")
 || !state.equals("AK")
 || !state.equals("HI"))
```

Because two ! cancel each other out, the result is the simpler test

```java
!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 in 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.

5.8 Application: Input Validation

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 want to give an error message and exit the program.

It is simple to guard against an input of 13:

```java
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:

```java
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

```java
floor = in.nextInt();
```

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
Call the hasNextInt or hasNextDouble method to ensure that the next input is a number.

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:

```java
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:

```java
import java.util.Scanner;

/**
 * This program simulates an elevator panel that skips the 13th floor, checking for
 * input errors.
 */

class ElevatorSimulation2
{
    public static void main(String[] args)
    {
        Scanner in = new Scanner(System.in);
        System.out.print("Floor: ");
        if (in.hasNextInt())
        {
            // Now we know that the user entered an integer
            int floor = in.nextInt();
            if (floor == 13)
            {
                System.out.println("Error: There is no thirteenth floor.");
            }
            else if (floor <= 0 || floor > 20)
            {
                System.out.println("Error: The floor must be between 1 and 20.");
            }
            else
            {
                // Now we know that the input is valid
                int actualFloor = floor;
                if (floor > 13)
                {
                    actualFloor = floor - 1;
                }
                System.out.println("The elevator will travel to the actual floor "+ actualFloor);
            }
        }
        else
        {
            System.out.println("Error: Not an integer.");
        }
    }
}
Chapter 5  Decisions

Program Run

Floor: 13
Error: There is no thirteenth floor.

38. In the ElevatorSimulation2 program, what is the output when the input is
   a. 100?
   b. -1?
   c. 20?
   d. thirteen?

39. 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");
   }

40. 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.

41. 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: R5.3, R5.33, E5.13.
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 (\(<\leq >\geq == !=\)) to compare numbers.
- Relational operators compare values. The \(==\) operator tests for equality.
- When comparing floating-point numbers, don’t test for equality. Instead, check whether they are close enough.
- Do not use the \(==\) operator to compare strings. Use the \(==\) method instead.
- The \(compareTo\) method compares strings in lexicographic order.
- The \(==\) operator tests whether two object references are identical. To compare the contents of objects, you need to use the \(==\) method.
- The \(null\) reference refers to no object.

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.

- Flow charts 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.

- Black-box testing describes a testing method that does not take the structure of the implementation into account.
- White-box testing uses information about the structure of a program.
- Code coverage is a measure of how many parts of a program have been tested.
- Boundary test cases are test cases that are at the boundary of acceptable inputs.
- 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.awt.Rectangle
- java.lang.String
- java.util.Scanner
- java.util.logging.Level
- java.util.logging.Logger
- getGlobal
- info
- setLevel

### REVIEW EXERCISES

#### R5.1
What is the value of each variable after the if statement?

**a.** int n = 1; int k = 2; int r = n;
   if (k < n) { r = k; }

**b.** int n = 1; int k = 2; int r;
   if (n < k) { r = k; }
   else { r = k + n; }

**c.** int n = 1; int k = 2; int r = k;
   if (r < k) { n = r; }
   else { k = n; }

**d.** int n = 1; int k = 2; int r = 3;
   if (r < n + k) { r = 2 * n; }
   else { k = 2 * r; }

#### R5.2
Explain the difference between

```java
s = 0;
if (x > 0) { s++; }
if (y > 0) { s++; }
```

and

```java
s = 0;
if (x > 0) { s++; }
else if (y > 0) { s++; }
```
**R5.3** Find the errors in the following if statements.

a. if x > 0 then System.out.print(x);

b. if (1 + x > Math.pow(x, Math.sqrt(2))) { y = y + x; }

c. if (x = 1) { y++; }

d. 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"; }

**R5.4** 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); }

**R5.5** 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.

**R5.6** 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.

**R5.7** 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.

**R5.8** 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).

**R5.9** 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 with the statement

```java
count == 0;
```

What does your compiler do when you compile the program?

- **R5.10** Each square on a chess board can be described by a letter and number, such as g5 in the example at right.

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 5.5, trace this pseudocode with input g5.

- **Testing R5.11** Give a set of four test cases for the algorithm of Exercise R5.10 that covers all branches.

- **R5.12** 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 0–23). 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.

- **R5.13** Draw a flow chart for the algorithm in Exercise R5.12.


- **R5.15** Draw a flow chart for the algorithm in Exercise E5.15.

- **Testing R5.16** Develop a set of test cases for the algorithm in Exercise R5.12.
**Testing R5.17** Develop a set of test cases for the algorithm in Exercise E5.15.

**R5.18** 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)

**R5.19** Write pseudocode for a program that assigns letter grades for a quiz, according to the following table:

<table>
<thead>
<tr>
<th>Score</th>
<th>Grade</th>
</tr>
</thead>
<tbody>
<tr>
<td>90-100</td>
<td>A</td>
</tr>
<tr>
<td>80-89</td>
<td>B</td>
</tr>
<tr>
<td>70-79</td>
<td>C</td>
</tr>
<tr>
<td>60-69</td>
<td>D</td>
</tr>
<tr>
<td>&lt; 60</td>
<td>F</td>
</tr>
</tbody>
</table>

**R5.20** 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.

**R5.21** 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", "Car1"
- i. "car", "bar"

**R5.22** Explain the difference between an *if/else if/else* sequence and nested *if* statements. Give an example of each.

**R5.23** 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.

**R5.24** Rewrite the condition in Section 5.3 to use `<` operators instead of `>=` operators. What is the impact on the order of the comparisons?

**Testing R5.25** Give a set of test cases for the tax program in Exercise P5.2. Manually compute the expected results.

**R5.26** 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.

**R5.27** 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.
R5.28 True or false? $A \&\& B$ is the same as $B \&\& A$ for any Boolean conditions $A$ and $B$.

R5.29 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.

R5.30 Suppose the value of $b$ is false and the value of $x$ is 0. What is the value of each of the following expressions?

- a. $b \&\& x == 0$
- b. $b || x == 0$
- c. $!b \&\& x == 0$
- d. $!b || x == 0$
- e. $b \&\& x != 0$
- f. $b || x != 0$
- g. $!b \&\& x != 0$
- h. $!b || x != 0$

R5.31 Simplify the following expressions. Here, $b$ is a variable of type boolean.

- a. $b == true$
- b. $b == false$
- c. $b != true$
- d. $b != false$

R5.32 Simplify the following statements. Here, $b$ is a variable of type boolean and $n$ is a variable of type int.

- a. if ($n == 0$) { $b = true; }$ else { $b = false; }$
  
  (Hint: What is the value of $n == 0$?)
- b. if ($n == 0$) { $b = false; }$ else { $b = true; }$
- c. $b = false; if (n > 1) { if (n < 2) { $b = true; } }$
- d. if ($n < 1$) { $b = true; }$ else { $b = n > 2; }$

R5.33 What is wrong with the following program?

```java
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.");
}
```
**E5.1** Write a program that reads an integer and prints whether it is negative, zero, or positive.

**E5.2** 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.

**E5.3** Write a program that reads an integer and prints how many digits the number has, by checking whether the number is $\geq 10$, $\geq 100$, and so on. (Assume that all integers are less than ten billion.) If the number is negative, first multiply it with –1.

**E5.4** 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.

**E5.5** 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.

**E5.6** Repeat Exercise E5.5, 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.

**E5.7** 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
```

**E5.8** 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
```

**E5.9** 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).

**Business E5.10** 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. In your solution, implement a class `Paycheck`.

**E5.11** 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.
E5.12 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 E5.11 to allow the user to supply the altitude in meters or feet.

E5.13 Add error handling to Exercise E5.12. 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.

E5.14 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.

\[
\text{If } \text{hour}_1 < \text{hour}_2 \\
\quad \text{time}_1 \text{ comes first.} \\
\text{Else if } \text{hour}_1 \text{ and } \text{hour}_2 \text{ are the same} \\
\quad \text{If } \text{minute}_1 < \text{minute}_2 \\
\quad \quad \text{time}_1 \text{ comes first.} \\
\text{Else if } \text{minute}_1 \text{ and } \text{minute}_2 \text{ are the same} \\
\quad \quad \text{time}_1 \text{ and } \text{time}_2 \text{ are the same.} \\
\text{Else} \\
\quad \text{time}_2 \text{ comes first.} \\
\text{Else} \\
\quad \text{time}_2 \text{ comes first.}
\]

Write a program that prompts the user for two points in time and prints the time that comes first, then the other time. In your program, supply a class Time and a method

\[
\text{public int compareTo(Time other)}
\]

that returns –1 if the time comes before the other, 0 if both are the same, and 1 otherwise.

E5.15 The following algorithm yields the season (Spring, Summer, Fall, or Winter) for a given month and day.

\[
\text{If month is 1, 2, or 3, season } = \text{ “Winter”} \\
\text{Else if month is 4, 5, or 6, season } = \text{ “Spring”} \\
\text{Else if month is 7, 8, or 9, season } = \text{ “Summer”} \\
\text{Else if month is 10, 11, or 12, season } = \text{ “Fall”} \\
\text{If month is divisible by 3 and day } \geq 21 \\
\quad \text{If season is “Winter”, season } = \text{ “Spring”} \\
\text{Else if season is “Spring”, season } = \text{ “Summer”} \\
\text{Else if season is “Summer”, season } = \text{ “Fall”} \\
\text{Else season } = \text{ “Winter”}
\]

Write a program that prompts the user for a month and day and then prints the season, as determined by this algorithm. Use a class Date with a method getSeason.

E5.16 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.

Use a class Grade with a method getNumericGrade.
• E5.17} 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.

Use a class `Grade` with a method `getNumericGrade`.

• E5.18} 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.

• E5.19} 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
```

Implement a class `Card` whose constructor takes the card notation string and whose `getDescription` method returns a description of the card. If the notation string is not in the correct format, the `getDescription` method should return the string "Unknown".

• E5.20} 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.
```

• E5.21} Write a program that reads in three strings and sorts them lexicographically.

```
Enter three strings: Charlie Able Baker
Able
Baker
Charlie
```
E5.22 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.

E5.23 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.

E5.24 Write a program that asks the user to enter a month (1 for January, 2 for February, etc.) and then prints the number of days in the month. For February, print “28 days”.

Enter a month: 5
30 days

Use a class Month with a method
public int getLength()

Do not use a separate if/else branch for each month. Use Boolean operators.

Business E5.25 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)

<table>
<thead>
<tr>
<th>Money Spent</th>
<th>Coupon Percentage</th>
</tr>
</thead>
<tbody>
<tr>
<td>Less than $10</td>
<td>No coupon</td>
</tr>
<tr>
<td>From $10 to $60</td>
<td>8%</td>
</tr>
<tr>
<td>More than $60 to $150</td>
<td>10%</td>
</tr>
<tr>
<td>More than $150 to $210</td>
<td>12%</td>
</tr>
<tr>
<td>More than $210</td>
<td>14%</td>
</tr>
</tbody>
</table>

P5.1 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.) Use a class Date with a method getFortune.
P5.2 Write a program that computes taxes for the following schedule.

<table>
<thead>
<tr>
<th>If your status is Single and if the taxable income is over</th>
<th>but not over</th>
<th>the tax is</th>
<th>of the amount over</th>
</tr>
</thead>
<tbody>
<tr>
<td>$0</td>
<td>$8,000</td>
<td>10%</td>
<td>$0</td>
</tr>
<tr>
<td>$8,000</td>
<td>$32,000</td>
<td>$800 + 15%</td>
<td>$8,000</td>
</tr>
<tr>
<td>$32,000</td>
<td>$4,400 + 25%</td>
<td></td>
<td>$32,000</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>If your status is Married and if the taxable income is over</th>
<th>but not over</th>
<th>the tax is</th>
<th>of the amount over</th>
</tr>
</thead>
<tbody>
<tr>
<td>$0</td>
<td>$16,000</td>
<td>10%</td>
<td>$0</td>
</tr>
<tr>
<td>$16,000</td>
<td>$64,000</td>
<td>$1,600 + 15%</td>
<td>$16,000</td>
</tr>
<tr>
<td>$64,000</td>
<td></td>
<td>$8,800 + 25%</td>
<td>$64,000</td>
</tr>
</tbody>
</table>

P5.3 The TaxReturn.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.

P5.4 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
```

P5.5 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.

P5.6 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.

P5.7 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.

P5.8 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. Provide a class Year with a method isLeapYear. Use a single if statement and Boolean operators.
P5.9 Roman numbers. Write a program that converts a positive integer into the Roman number system. The Roman number system has digits

<table>
<thead>
<tr>
<th>Digit</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>I</td>
<td>1</td>
</tr>
<tr>
<td>V</td>
<td>5</td>
</tr>
<tr>
<td>X</td>
<td>10</td>
</tr>
<tr>
<td>L</td>
<td>50</td>
</tr>
<tr>
<td>C</td>
<td>100</td>
</tr>
<tr>
<td>D</td>
<td>500</td>
</tr>
<tr>
<td>M</td>
<td>1,000</td>
</tr>
</tbody>
</table>

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.
- The numbers 1 to 9 are expressed as

<table>
<thead>
<tr>
<th>Number</th>
<th>Roman Numeral</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>I</td>
</tr>
<tr>
<td>4</td>
<td>IV</td>
</tr>
<tr>
<td>5</td>
<td>V</td>
</tr>
<tr>
<td>6</td>
<td>VI</td>
</tr>
<tr>
<td>7</td>
<td>VII</td>
</tr>
<tr>
<td>8</td>
<td>VIII</td>
</tr>
<tr>
<td>9</td>
<td>IX</td>
</tr>
</tbody>
</table>

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.

- 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.

P5.10 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 P5.11 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. Reject transactions that overdraw an account. At the end, print the balances of both accounts.
• **Business P5.12** 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:
  - 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 P5.13** 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.

• **Science P5.14** 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.

<table>
<thead>
<tr>
<th>Electromagnetic Spectrum</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Type</strong></td>
</tr>
<tr>
<td>Radio Waves</td>
</tr>
<tr>
<td>Microwaves</td>
</tr>
<tr>
<td>Infrared</td>
</tr>
<tr>
<td>Visible light</td>
</tr>
<tr>
<td>Ultraviolet</td>
</tr>
<tr>
<td>X-rays</td>
</tr>
<tr>
<td>Gamma rays</td>
</tr>
</tbody>
</table>

• **Science P5.15** Repeat Exercise P5.14, modifying the program so that it prompts for the frequency instead.

• **Science P5.16** Repeat Exercise P5.14, modifying the program so that it first asks the user whether the input will be a wavelength or a frequency.
**Science P5.17** 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 P5.18** Sound level $L$ in units of decibel (dB) is determined by

$$L = 20 \log_{10}\left(\frac{p}{p_0}\right)$$

where $p$ is the sound pressure of the sound (in Pascals, abbreviated Pa), and $p_0$ is a reference sound pressure equal to $20 \times 10^{-6}$ Pa (where $L$ is 0 dB). The following table gives descriptions for certain sound levels.

<table>
<thead>
<tr>
<th>Sound Level</th>
<th>dB</th>
</tr>
</thead>
<tbody>
<tr>
<td>Threshold of pain</td>
<td>130</td>
</tr>
<tr>
<td>Possible hearing damage</td>
<td>120</td>
</tr>
<tr>
<td>Jack hammer at 1 m</td>
<td>100</td>
</tr>
<tr>
<td>Traffic on a busy roadway at 10 m</td>
<td>90</td>
</tr>
<tr>
<td>Normal conversation</td>
<td>60</td>
</tr>
<tr>
<td>Calm library</td>
<td>30</td>
</tr>
<tr>
<td>Light leaf rustling</td>
<td>0</td>
</tr>
</tbody>
</table>

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 P5.19** 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 $\Omega$, is related to the temperature $T$, in °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 $V_{\text{min}} = 12 \, \text{volts} \leq V_m \leq V_{\text{max}} = 18 \, \text{volts}$. Write a program that accepts a value of $V_m$ and checks that it is 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.

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 $\Omega$, at the temperature $T$ in °K, and $R_0$ is the resistance, in $\Omega$, at the temperature $T_0$ in °K. $\beta$ 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 $\beta = 3,310 \, ^\circ\text{K}$. (Notice that $\beta$ has units of °K. The temperature in °K is obtained by adding 273° to the temperature in °C.) The resistors $R_2$, $R_3$, and $R_4$ have a resistance of 156.3 k$\Omega = 156,300 \, \Omega$.

Write a Java program that prompts the user for a temperature in °F and prints a message indicating whether or not the alarm will sound at that temperature.

A mass $m = 2 \, \text{kilograms}$ is attached to the end of a rope of length $r = 3 \, \text{meters}$. The mass is whirled around at high speed. The rope can withstand a maximum tension of $T = 60 \, \text{Newtons}$. Write a program that accepts a rotation speed $v$ and determines whether such a speed will cause the rope to break. *Hint: $T = m \, v^2/r$. 
**Science P5.22** 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: \(T = m \cdot v^2 / r.\)

**Science P5.23** 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.

*Hint: Escape velocity is \(v_{\text{escape}} = \sqrt{\frac{2GM}{R}}\), where \(G = 6.67 \times 10^{-11}N m^2/kg^2\) is the gravitational constant, \(M = 1.3 \times 10^{22}kg\) is the mass of Halley’s comet, and 
\[R = 1.153 \times 10^6 m\] is its radius.

**Answers to Self-Check Questions**

1. Change the if statement to
   
   ```java
   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)
   
   ```java
   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. (a) 0; (b) 1; (c) An exception occurs.

12. Syntactically incorrect: e, g, h. Logically questionable: a, d, f.

13. if (scoreA > scoreB)
   
   ```java
   System.out.println("A won");
   }
   else if (scoreA < scoreB)
   {
       System.out.println("B won");
   }
   else
   {
       System.out.println("Game tied");
   }
   ```

14. if (x > 0) { s = 1; }

15. You could first set \(s\) to one of the three values:

   ```java
   s = 0;
   if (x > 0) { s = 1; }
   else if (x < 0) { s = -1; }
   else { s = 0; }
   ```
16. The `if (price <= 100)` can be omitted (leaving just `else`), making it clear that the `else` branch is the sole alternative.

17. No destruction of buildings.

18. Add a branch before the final `else`:
```java
else if (richter < 0)
{
    System.out.println("Error: Negative input");
}
```

19. 3200.

20. No. Then the computation is $0.10 \times 32000 + 0.25 \times (32000 - 32000)$.

21. 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.

22. Change `else` in line 22 to
```java
else if (maritalStatus.equals("N"))
```
and add another branch after line 25:
```java
else
{
    System.out.println("Error: Please answer Y or N.");
}
```

23. 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.

24. The “True” arrow from the first decision points into the “True” branch of the second decision, creating spaghetti code.

25. Here is one solution. In Section 5.7, you will see how you can combine the conditions for a more elegant solution.
29. Test Case | Expected Output | Comment
--- | --- | ---
12 | Below 13th floor | 
14 | Above 13th floor | 
13 | ? | The specification is not clear—See Section 5.8 for a version of this program with error handling.

30. 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.

31. 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 18 for a version of this program with error handling. | 

32. Test Case | Expected Output | Comment
--- | --- | ---
(0.5, 0.5) | inside | 
(4, 2) | outside | 
(0, 2) | on the boundary, Close to 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 | 

33. \( x = 0 \land y = 0 \)

34. \( x = 0 \lor y = 0 \)

35. \((x = 0 \land y = 0) \lor (y = 0 \land x = 0)\)

36. The same as the value of \(\text{frozen}\).

37. 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.

38. (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.

39. \( \text{floor} = 13 \lor \text{floor} <= 0 \lor \text{floor} > 20 \)

40. Check for \(\text{in.hasNextDouble()}\), to make sure a researcher didn’t supply an input such as \(\text{oh my}\). Check for \(\text{weight} <= 0\), 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 (\(\text{rattus rattus}\)) 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:

```java
if (in.hasNextDouble())
{
    double weight = in.nextDouble();
    if (weight < 0)
    {
        System.out.println("Error: Weight cannot be negative.");
    }
    else if (weight > 10)
    {
        System.out.println("Error: Weight > 10 kg.");
    }
    else
    {
        Process valid weight.
    }
}
else
{
    System.out.print("Error: Not a number");
}
```

41. The second input fails, and the program terminates without printing anything.
WORKED EXAMPLE 5.1 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

```
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.

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.)

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:

\[
\begin{cases}
    \text{If } \text{str.length()} \% 2 == 1 \\
    \quad \text{position} = \text{str.length()} / 2 \text{ (with remainder discarded)} \\
    \quad \text{length} = 1 \\
    \quad \text{result} = \text{str.substring(position, position + 1)} \\
    \text{Else} \\
    \quad \text{position} = \text{str.length()} / 2 - 1 \\
    \quad \text{length} = 2 \\
    \quad \text{result} = \text{str.substring(position, position + length)}
\end{cases}
\]

The second statement in each branch is almost identical, but the length of the substring differs. Let’s set the length in each branch:

\[
\begin{cases}
    \text{If } \text{str.length()} \% 2 == 1 \\
    \quad \text{position} = \text{str.length()} / 2 \text{ (with remainder discarded)} \\
    \quad \text{length} = 1 \\
    \text{Else} \\
    \quad \text{position} = \text{str.length()} / 2 - 1 \\
    \quad \text{length} = 2 \\
    \quad \text{result} = \text{str.substring(position, position + length)}
\end{cases}
\]

Step 6

Test both branches.

We will use a different set of strings for testing. For an odd-length string, consider "monitor". We get

\[
\begin{align*}
\text{position} &= \text{str.length()} / 2 = 7 / 2 = 3 \text{ (with remainder discarded)} \\
\text{length} &= 1 \\
\text{result} &= \text{str.substring(3, 4)} = "i"
\end{align*}
\]

For the even-length string "monitors", we get

\[
\begin{align*}
\text{position} &= \text{str.length()} / 2 - 1 = 8 / 2 - 1 = 3 \text{ (with remainder discarded)} \\
\text{length} &= 2 \\
\text{result} &= \text{str.substring(3, 5)} = "it"
\end{align*}
\]

Step 7

Assemble the if statement in Java.

Here’s the completed code segment:

```java
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 ch05/worked_example_1 directory of the book’s companion code.
To implement while, for, and do loops
To hand-trace the execution of a program
To learn to use common loop algorithms
To understand nested loops
To implement programs that read and process data sets
To use a computer for simulations
To learn about the debugger

CHAPTER CONTENTS

6.1 THE WHILE LOOP 238
SYN while Statement 239
CE1 Don’t Think “Are We There Yet?” 243
CE2 Infinite Loops 244
CE3 Off-by-One Errors 244

6.2 PROBLEM SOLVING: HAND-TRACING 245
C&S Digital Piracy 249

6.3 THE FOR LOOP 250
SYN for Statement 250
PT1 Use for Loops for Their Intended Purpose Only 255
PT2 Choose Loop Bounds That Match Your Task 256
PT3 Count Iterations 256
ST1 Variables Declared in a for Loop Header 257

6.4 THE DO LOOP 258
PT4 Flowcharts for Loops 259

6.5 APPLICATION: PROCESSING SENTINEL VALUES 259
ST2 Redirection of Input and Output 262
ST3 The “Loop and a Half” Problem 262
ST4 The break and continue Statements 263

6.6 PROBLEM SOLVING: STORYBOARDS 265

6.7 COMMON LOOP ALGORITHMS 268
HT1 Writing a Loop 272
WE1 Credit Card Processing

6.8 NESTED LOOPS 275
WE2 Manipulating the Pixels in an Image

6.9 APPLICATION: RANDOM NUMBERS AND SIMULATIONS 279

6.10 USING A DEBUGGER 282
HT2 Debugging 285
WE3 A Sample Debugging Session
C&S The First Bug 287
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.

6.1 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:

Start with a year value of 0, a column for the interest, and a balance of $10,000.

<table>
<thead>
<tr>
<th>year</th>
<th>interest</th>
<th>balance</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td>$10,000</td>
</tr>
</tbody>
</table>

Repeat the following steps while the balance is less than $20,000.

- Add 1 to the year value.
- Compute the interest as balance \times 0.05 (i.e., 5 percent interest).
- Add the interest to the balance.
- Report the final year value 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 “Repeat steps while the balance is less than $20,000”.

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.
In Java, the `while` statement implements such a repetition (see Syntax 6.1). It has the form

```java
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:

```java
while (balance < targetBalance)
{
    year++;  // Increment year
    double interest = balance * RATE / 100;  // Calculate interest
    balance = balance + interest;  // Add interest to balance
}
```

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).
When you declare a variable \textit{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:

```java
while (balance < targetBalance)
{
    year++;    
    double interest = balance * RATE / 100;
    balance = balance + interest;
}
// interest no longer declared here
```

A new interest variable is created in each iteration.

**Figure 2**
Execution of the Investment Loop

1. Check the loop condition
   ```java
   while (balance < targetBalance)
   {
       year++;    
       double interest = balance * RATE / 100;
       balance = balance + interest;
   }
   The condition is true
   ```

2. Execute the statements in the loop
   ```java
   balance = 10000
   year = 0
   interest = 0
   ```

3. Check the loop condition again
   ```java
   balance = 10500
   year = 1
   ```

4. After 15 iterations
   ```java
   balance = 20789.28
   year = 15
   ```

5. Execute the statement following the loop
   ```java
   System.out.println(year);
   ```
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.

Here is the program that solves the investment problem. Figure 2 illustrates the program’s execution.

```
/**
 * A class to monitor the growth of an investment that
 * accumulates interest at a fixed annual rate.
 */
public class Investment
{
    private double balance;
    private double rate;
    private int year;

    /**
     * Constructs an Investment object from a starting balance and
     * interest rate.
     * @param aBalance the starting balance
     * @param aRate the interest rate in percent
     */
    public Investment(double aBalance, double aRate)
    {
        balance = aBalance;
        rate = aRate;
        year = 0;
    }

    /**
     * Keeps accumulating interest until a target balance has
     * been reached.
     * @param targetBalance the desired balance
     */
    public void waitForBalance(double targetBalance)
    {
        while (balance < targetBalance)
        {
            year++;
            double interest = balance * rate / 100;
            balance = balance + interest;
        }
    }

    /**
     * Gets the current investment balance.
     * @return the current balance
     */
    public double getBalance()
    {
        return balance;
    }

    /**
     * Gets the number of years this investment has accumulated
     * interest.
     * @return the number of years since the start of the investment
     */
```
section_1/InvestmentRunner.java

```java
/**
 * This program computes how long it takes for an investment to double.
 */
public class InvestmentRunner {
    public static void main(String[] args) {
        final double INITIAL_BALANCE = 10000;
        final double RATE = 5;
        Investment invest = new Investment(INITIAL_BALANCE, RATE);
        invest.waitForBalance(2 * INITIAL_BALANCE);
        int years = invest.getYears();
        System.out.println("The investment doubled after "+ years + " years");
    }
}
```

Program Run

The investment doubled after 15 years.

SELF CHECK

1. How many years does it take for the investment to triple? Modify the program and run it.
2. 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.
3. Modify the program so that the balance after each year is printed. How did you do that?
4. Suppose we change the program so that the condition of the `while` loop is `while (balance <= targetBalance)`
   What is the effect on the program? Why?
5. What does the following loop print?
   ```java
   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: R6.4, R6.8, E6.14.
### Table 1 while Loop Examples

<table>
<thead>
<tr>
<th>Loop</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>i = 0; sum = 0; while (sum &lt; 10) { i++; sum = sum + i; Print i and sum; }</code></td>
</tr>
<tr>
<td><strong>Output</strong></td>
</tr>
<tr>
<td>1 1</td>
</tr>
<tr>
<td>2 3</td>
</tr>
<tr>
<td>3 6</td>
</tr>
<tr>
<td>4 10</td>
</tr>
<tr>
<td><strong>Explanation</strong></td>
</tr>
<tr>
<td>When sum is 10, the loop condition is false, and the loop ends.</td>
</tr>
</tbody>
</table>

| `i = 0; sum = 0; while (sum < 10) { i++; sum = sum - i; Print i and sum; }` |
| **Output** |
| 1 -1 |
| 2 -3 |
| 3 -6 |
| 4 -10 |
| **Explanation** |
| Because sum never reaches 10, this is an “infinite loop” (see Common Error 6.2 on page 244). |

| `i = 0; sum = 0; while (sum < 0) { i++; sum = sum - i; Print i and sum; }` |
| **Output** |
| (No output) |
| **Explanation** |
| The statement sum < 0 is false when the condition is first checked, and the loop is never executed. |

| `i = 0; sum = 0; while (sum >= 10) { i++; sum = sum + i; Print i and sum; }` |
| **Output** |
| (No output) |
| **Explanation** |
| 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 6.1 on page 243). |

| `i = 0; sum = 0; while (sum < 10) ; { i++; sum = sum + i; Print i and sum; }` |
| **Output** |
| (No output, program does not terminate) |
| **Explanation** |
| Note the semicolon before the `. This loop has an empty body. It runs forever, checking whether sum < 10 and doing nothing in the body. |

---

**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

```plaintext
balance >= targetBalance
```

But the while loop thinks the opposite: How long am I allowed to keep going? The correct loop condition is

```plaintext
while (balance < targetBalance)
```

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.*
Common Error 6.2

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:

```java
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:

```java
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.)

Common Error 6.3

Off-by-One Errors

Consider our computation of the number of years that are required to double an investment:

```java
int year = 0;
while (balance < targetBalance)
{
    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 < targetBalance` or for `balance <= targetBalance`? 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.

<table>
<thead>
<tr>
<th>year</th>
<th>balance</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>$100</td>
</tr>
<tr>
<td>1</td>
<td>$150</td>
</tr>
<tr>
<td>2</td>
<td>$225</td>
</tr>
</tbody>
</table>

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 * INITIAL_BALANCE. Has the investment doubled after one year? It has. Therefore, the loop should not execute again. If the test condition is balance < targetBalance, the loop stops, as it should. If the test condition had been balance <= targetBalance, the loop would have executed once more.

In other words, you keep adding interest while the balance has not yet doubled.

### 6.2 Problem Solving: Hand-Tracing

In Programming Tip 5.5, 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?

```java
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: \( n \), \( \text{sum} \), and \( \text{digit} \).

The first two variables are initialized with 1729 and 0 before the loop is entered.

```java
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 greater than zero, enter the loop. The variable \( \text{digit} \) is set to 9 (the remainder of dividing 1729 by 10). The variable \( \text{sum} \) is set to 0 + 9 = 9.

Finally in this iteration, \( n \) 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.

Now check the loop condition again.

```java
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 \( \text{digit} \) becomes 2, \( \text{sum} \) is set to \( 9 + 2 = 11 \), and \( n \) is set to 17.

Repeat the loop once again, setting \( \text{digit} \) to 7, \( \text{sum} \) to \( 11 + 7 = 18 \), and \( n \) to 1.

Enter the loop for one last time. Now \( \text{digit} \) is set to 1, \( \text{sum} \) to 19, and \( n \) becomes zero.

The condition \( n > 0 \) is now false. Continue with the statement after the loop.

This statement is an output statement. The value that is output is the value of \( \text{sum} \), 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 \( \text{sum} \).
- We strip the digit off \( n \).

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.)

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.

6. Hand-trace the following code, showing the value of \( n \) and the output.

```java
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?

```java
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 \( n \).

```java
int r = 1;
int i = 1;
while (i <= n)
{
    r = r * a;
    i++;
}
```

9. Trace the following code. What error do you observe?

```java
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 $n$:

```plaintext
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: R6.6, R6.9.

---

**Computing & Society 6.1 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.
6.3 The for Loop

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:

```java
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 6.2).

```java
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.

Syntax 6.2 for Statement

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

These three expressions should be related. See page 255.

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 page 256.

The variable i is defined only in this for loop. See page 257.
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.  
- The condition is checked before each iteration.  
- The update is executed after each iteration.

A for loop can count down instead of up:

```java
for (int counter = 10; counter >= 0; counter--)
    System.out.println(counter);
```

The increment or decrement need not be in steps of 1:

```java
for (int counter = 0; counter <= 10; counter = counter + 2)
    System.out.println(counter);
```

See Table 2 on page 254 for additional variations.

So far, we have always declared the counter variable in the loop initialization:

```java
for (int counter = 1; counter <= 10; counter++)
```

// counter no longer declared here
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:

```java
int counter;
for (counter = 1; counter <= 10; counter++)
{
    ...
}
// counter still declared here
```

A common use of the for loop is to traverse all characters of a string:

```java
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.

Here is another typical use of the for loop. We want to compute the growth of our savings account over a period of years, as shown in this table:

<table>
<thead>
<tr>
<th>Year</th>
<th>Balance</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>10500.00</td>
</tr>
<tr>
<td>2</td>
<td>11025.00</td>
</tr>
<tr>
<td>3</td>
<td>11576.25</td>
</tr>
<tr>
<td>4</td>
<td>12155.06</td>
</tr>
<tr>
<td>5</td>
<td>12762.82</td>
</tr>
</tbody>
</table>

The for loop pattern applies because the variable `year` starts at 1 and then moves in constant increments until it reaches the target:

```java
for (int year = 1; year <= numberOfYears; year++)
{
    Update balance.
}
```

Following is the complete program. Figure 4 shows the corresponding flowchart.

![Flowchart of a for Loop](image-url)
section_3/Investment.java

```java
/**
 * A class to monitor the growth of an investment that
 * accumulates interest at a fixed annual rate.
 */
public class Investment
{
    private double balance;
    private double rate;
    private int year;

    /**
     * Constructs an Investment object from a starting balance and
     * interest rate.
     * @param aBalance the starting balance
     * @param aRate the interest rate in percent
     */
    public Investment(double aBalance, double aRate)
    {
        balance = aBalance;
        rate = aRate;
        year = 0;
    }

    /**
     * Keeps accumulating interest until a target balance has
     * been reached.
     * @param targetBalance the desired balance
     */
    public void waitForBalance(double targetBalance)
    {
        while (balance < targetBalance)
        {
            year++;
            double interest = balance * rate / 100;
            balance = balance + interest;
        }
    }

    /**
     * Keeps accumulating interest for a given number of years.
     * @param numberOfYears the number of years to wait
     */
    public void waitYears(int numberOfYears)
    {
        for (int i = 1; i <= numberOfYears; i++)
        {
            double interest = balance * rate / 100;
            balance = balance + interest;
            year = year + n;
        }
    }

    /**
     * Gets the current investment balance.
     * @return the current balance
     */
    public double getBalance()
    {
    }
```

Chapter 6  Loops

```java
59       return balance;
60     }
61
62 /**
63     * Gets the number of years this investment has accumulated
64     * interest.
65     * @return the number of years since the start of the investment
66     */
67 public int getYears()
68     {
69       year;
70     }
```

section_3/InvestmentRunner.java

```java
/**
  * This program computes how much an investment grows in
  * a given number of years.
  */
public class InvestmentRunner
{
  public static void main(String[] args)
  {
    final double INITIAL_BALANCE = 10000;
    final double RATE = 5;
    final int YEARS = 20;
    Investment invest = new Investment(INITIAL_BALANCE, RATE);
    invest.waitYears(YEARS);
    double balance = invest.getBalance();
    System.out.printf("The balance after %d years is %.2f\n", YEARS, balance);
  }
}
```

**Program Run**

The balance after 20 years is 26532.98

<table>
<thead>
<tr>
<th>Loop</th>
<th>Values of i</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>for (i = 0; i &lt;= 5; i++)</td>
<td>0 1 2 3 4 5</td>
<td>Note that the loop is executed 6 times. (See Programming Tip 6.3 on page 256.)</td>
</tr>
<tr>
<td>for (i = 5; i &gt;= 0; i--)</td>
<td>5 4 3 2 1 0</td>
<td>Use i-- for decreasing values.</td>
</tr>
<tr>
<td>for (i = 0; i &lt; 9; i = i + 2)</td>
<td>0 2 4 6 8</td>
<td>Use i = i + 2 for a step size of 2.</td>
</tr>
<tr>
<td>for (i = 0; i != 9; i = i + 2)</td>
<td>0 2 4 6 8 10 12 14 ... (infinite loop)</td>
<td>You can use &lt; or &lt;= instead of != to avoid this problem.</td>
</tr>
<tr>
<td>for (i = 1; i &lt;= 20; i = i * 2)</td>
<td>1 2 4 8 16</td>
<td>You can specify any rule for modifying i, such as doubling it in every step.</td>
</tr>
<tr>
<td>for (i = 0; i &lt; str.length(); i++)</td>
<td>0 1 2 ... until the last valid index of the string str</td>
<td>In the loop body, use the expression str.charAt(i) to get the i-th character.</td>
</tr>
</tbody>
</table>
11. Write the for loop of the Investment class as a while loop.

12. How many numbers does this loop print?
   ```java
   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 InvestmentRunner.java program to print the balances after 20, 40, ..., 100 years?

Practice It

Now you can try these exercises at the end of the chapter: R6.7, R6.13, E6.9, E6.13.

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
```java
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:

```java
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:

```java
for (int counter = 1; counter <= 100; counter++)
{
  if (counter % 10 != 0) // Skip values that are divisible by 10
  {  
    System.out.println(counter);
  }
}
```
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:

```java
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 bounds**.

When traversing the characters in a string, it is more natural to use the bounds

```java
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 bounds**.

In this case, it is not a good idea to use symmetric bounds:

```java
for (int i = 0; i <= str.length() - 1; i++) // Use < instead
```

The asymmetric form is easier to understand.

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 $\leq b$ 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

```java
for (int i = a; i < b; i++)
```

is executed $b - a$ times. For example, the loop traversing the characters in a string,

```java
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,

```java
for (int i = a; i <= b; i++)
```

is executed $b - a + 1$ times. That “+1” is the source of many programming errors.

For example,

```java
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.
```
6.3 The for Loop

Variables Declared in a for Loop Header

As mentioned, it is legal in Java to declare a variable in the header of a for loop. Here is the most common form of this syntax:

```java
for (int i = 1; i <= n; i++)
{
    ...
}
```

// i no longer defined here

The scope of the variable extends to the end of the for loop. Therefore, i is no longer defined after the loop ends. If you need to use the value of the variable beyond the end of the loop, then you need to declare it outside the loop. In this loop, you don’t need the value of i—you know it is n + 1 when the loop is finished. (Actually, that is not quite true—it is possible to break out of a loop before its end; see Special Topic 6.4 on page 263). When you have two or more exit conditions, though, you may still need the variable. For example, consider the loop

```java
for (i = 1; balance < targetBalance && i <= n; i++)
{
    ...
}
```

You want the balance to reach the target but you are willing to wait only a certain number of years. If the balance doubles sooner, you may want to know the value of i. Therefore, in this case, it is not appropriate to declare the variable in the loop header.

Note that the variables named i in the following pair of for loops are independent:

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

for (int i = 1; i <= 10; i++) // Declares a new variable i
{
    System.out.println(i * i * i);
}
```

In the loop header, you can declare multiple variables, as long as they are of the same type, and you can include multiple update expressions, separated by commas:

```java
for (int i = 0, j = 10; i <= 10; i++, j--)
{
    ...
}
```

However, many people find it confusing if a for loop controls more than one variable. I recommend that you not use this form of the for statement (see Programming Tip 6.1 on page 255). Instead, make the for loop control a single counter, and update the other variable explicitly:

```java
int j = 10;
for (int i = 0; i <= 10; i++)
{
    ...
    j--;
}
```
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):

```
int value;
do
    {
        System.out.print("Enter an integer < 100: ");
        value = in.nextInt();
    }
while (value >= 100);
```

**Figure 5** Flowchart of a do Loop

16. Suppose that we want to check for inputs that are at least 0 and at most 100. Modify the input validation do loop for this test.

17. Rewrite the input validation 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: R6.12, R6.19, R6.20.
In Section 5.5 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 5.5, 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.

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.

*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.*
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.

```java
salary = in.nextDouble();
if (salary != -1)
{
    sum = sum + salary;
    count++;
}
```

We stay in the loop while the sentinel value is not detected.

```java
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:

```java
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:

section_5/SentinelDemo.java

```java
import java.util.Scanner;

/**
 * This program prints the average of salary values that are terminated with a sentinel.
 */

public class SentinelDemo
{
    public static void main(String[] args)
    {
        double sum = 0;
        int count = 0;
        double salary = 0;
        System.out.print("Enter salaries, -1 to finish: ");
        Scanner in = new Scanner(System.in);

        // Process data until the sentinel is entered
        while (salary != -1)
        {
            salary = in.nextDouble();
            if (salary != -1)
            {
                sum = sum + salary;
                count++;
            }
        }

        // Compute and print the average
        if (count > 0)
        {
            double average = sum / count;
        }
    }
}
```
6.5 Application: Processing Sentinel Values

```
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

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 6.4 on page 263 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 5.8, 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
```java
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?

   ```java
   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: R6.16, E6.20, E6.21.

### 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

   ```bash
   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

   ```bash
   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.

### The “Loop and a Half” Problem

Reading input data sometimes requires a loop such as the following, which is somewhat unsightly:

   ```java
   boolean done = false;
   while (!done)
   {
       String input = in.next();
       if (input.equals("Q"))
       {
           done = true;
       }
   }
   ```
The true 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.

Some programmers dislike the introduction of an additional Boolean variable for loop control. Two Java language features can be used to alleviate the "loop and a half" problem. I don’t think either is a superior solution, but both approaches are fairly common, so it is worth knowing about them when reading other people's code.

You can combine an assignment and a test in the loop condition:

```java
while (!(input = in.next()).equals("Q"))
{
    Process data.
}
```

The expression

```
(input = in.next()).equals("Q")
```

means, “First call `in.next()`, then assign the result to `input`, then test whether it equals "Q".”

This is an expression with a side effect. The primary purpose of the expression is to serve as a test for the `while` loop, but it also does some work—namely, reading the input and storing it in the variable `input`. In general, it is a bad idea to use side effects, because they make a program hard to read and maintain. In this case, however, that practice is somewhat seductive, because it eliminates the control variable `done`, which also makes the code hard to read and maintain.

The other solution is to exit the loop from the middle, either by a `return` statement or by a `break` statement (see Special Topic 6.4 on page 263).

```java
public void processInput(Scanner in)
{
    while (true)
    {
        String input = in.next();
        if (input.equals("Q"))
        {
            return;
        }
        Process data.
    }
}
```

### Special Topic 6.4

**The break and continue Statements**

You already encountered the `break` statement in Special Topic 5.2, where it was used to exit a `switch` statement. In addition to breaking out of a `switch` statement, a `break` statement can also be used to exit a `while`, `for`, or `do` loop.

For example, the `break` statement in the following loop terminates the loop when the end of input is reached.

```java
while (true)
{
```
A loop with break statements can be difficult to understand because you have to look closely to find out how to exit the loop. However, when faced with the bother of introducing a separate loop control variable, some programmers find that break statements are beneficial in the “loop and a half” case. This issue is often the topic of heated (and quite unproductive) debate. In this book, we won’t use the break statement, and we leave it to you to decide whether you like to use it in your own programs.

In Java, there is a second form of the break statement that is used to break out of a nested statement. The statement break label; immediately jumps to the end of the statement that is tagged with a label. Any statement (including if and block statements) can be tagged with a label—the syntax is

```
label: statement
```

The labeled break statement was invented to break out of a set of nested loops.

```
outerloop:
while (outer loop condition)
{
    ...
    while (inner loop condition)
    {
        ...
        if (something really bad happened)
        {
            break outerloop;
        }
    }
}
```

Jumps here if something really bad happened.

Naturally, this situation is quite rare. We recommend that you try to introduce additional methods instead of using complicated nested loops.

Finally, there is the continue statement, which jumps to the end of the current iteration of the loop. Here is a possible use for this statement:

```
while (!done)
{
    String input = in.next();
    if (input.equals("Q"))
    {
        done = true;
        continue; // Jump to the end of the loop body
    }
    double x = Double.parseDouble(input);
    data.add(x);
    // continue statement jumps here
}
```

By using the continue statement, you don’t need to place the remainder of the loop code inside an else clause. This is a minor benefit. Few programmers use this statement.
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?

![Figure 6](image_url)

**Figure 6**
Storyboard for the Design of a Web Application

Courtesy of Martin Hardee.
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.)

<table>
<thead>
<tr>
<th>Converting a Sequence of Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>What unit do you want to convert from? cm</td>
</tr>
<tr>
<td>What unit do you want to convert to? in</td>
</tr>
<tr>
<td>Enter values, terminated by zero</td>
</tr>
<tr>
<td>30</td>
</tr>
<tr>
<td>30 cm = 11.81 in</td>
</tr>
<tr>
<td>100</td>
</tr>
<tr>
<td>100 cm = 39.37 in</td>
</tr>
<tr>
<td>0</td>
</tr>
<tr>
<td>What unit do you want to convert from?</td>
</tr>
</tbody>
</table>

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.

<table>
<thead>
<tr>
<th>Handling Unknown Units (needs improvement)</th>
</tr>
</thead>
<tbody>
<tr>
<td>What unit do you want to convert from? cm</td>
</tr>
<tr>
<td>What unit do you want to convert to? inches</td>
</tr>
<tr>
<td>Sorry, unknown unit.</td>
</tr>
<tr>
<td>What unit do you want to convert to? inch</td>
</tr>
<tr>
<td>Sorry, unknown unit.</td>
</tr>
<tr>
<td>What unit do you want to convert to? grrr</td>
</tr>
</tbody>
</table>

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 R6.25 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.
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.

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.

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?

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: R6.24, R6.25, R6.26.
6.7 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.

6.7.1 Sum and Average Value

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.

```java
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.

```java
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;
}
```

6.7.2 Counting Matches

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.

```java
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:

```java
int shortWords = 0;
while (in.hasNext())
{
    String input = in.next();
    if (input.length() <= 3)
    {
        shortWords++;
    }
}
```

### 6.7.3 Finding the First Match

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 space in a string. Because we do not visit all elements in the string, a `while` loop is a better choice than a `for` loop:

```java
boolean found = false;
char ch = ' ';
int position = 0;
while (!found && position < str.length())
{
    ch = str.charAt(position);
    if (ch == ' ') { 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.
6.7.4 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:

```java
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.

6.7.5 Maximum and Minimum

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.

```java
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:

```java
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 height so far, and update it whenever you see a taller one.
6.7.6 Comparing Adjacent Values

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, such as 1 7 2 9 4 9, contains adjacent duplicates.

Now you face a challenge. Consider the typical loop for reading a value:

```java
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, like this:

```java
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:

```java
double input = in.nextDouble();
while (in.hasNextDouble())
{
    double previous = input;
    input = in.nextDouble();
    if (input == previous)
    {
        System.out.println("Duplicate input");
    }
}
```

31. What total is computed when no user input is provided in the algorithm in Section 6.7.1?
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 6.7.3?
34. What is wrong with the following loop for finding the position of the first space in a string?
   ```java
   boolean found = false;
   for (int position = 0; !found && position < str.length(); position++)
   {
   ```
**Chapter 6** Loops

```java
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 6.7.6 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: E6.6, E6.10, E6.11.

### HOW TO 6.1 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 worldclimatic.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.

**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 first value.
Read second value.
If second value is higher than the first, set highest temperature to that value, highest month to 2.
Read next value.
If value is higher than the first and second, set highest temperature to that value, highest month to 3.
Read next value.
If value is higher than the highest temperature seen so far, set highest temperature to that value, 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 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:

\[
\text{If value is higher than the highest temperature, set highest temperature to that value, highest month to current month.}
\]

Altogether our loop is

```
Repeat
    Read next value.
    If value is higher than the highest temperature, 
    set highest temperature to that 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 \textit{count-controlled} loop is executed a definite number of times. In an \textit{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 \textbf{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 \texttt{do} loop. Otherwise, use a \texttt{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. Follow this pattern:

```
boolean done = false;
while (!done)
{
    Do some work.
    If all work has been completed
    {
        done = true;
    }
    else
    {
        Do more work.
    }
}
```

Such a variable is called a \textbf{flag}.

In summary,

- If you know in advance how many times a loop is repeated, use a \texttt{for} loop.
- If the loop body must be executed at least once, use a \texttt{do} loop.
- Otherwise, use a \texttt{while} loop.

In our example, we read 12 temperature values. Therefore, we choose a \texttt{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 first value; store as highest value.
highest month = 1
For current month from 2 to 12
  Read next value.
  If value is higher than the highest value
    Set highest value to that value.
    Set highest month to current month.
```

**Step 6**

Trace the loop with typical examples.

Hand-trace your loop code, as described in Section 6.2. 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:

<table>
<thead>
<tr>
<th>current month</th>
<th>current value</th>
<th>highest month</th>
<th>highest value</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>22.6</td>
<td></td>
<td>Highest value</td>
</tr>
<tr>
<td>2</td>
<td>36.6</td>
<td>2</td>
<td>36.6</td>
</tr>
<tr>
<td>3</td>
<td>44.5</td>
<td>3</td>
<td>44.5</td>
</tr>
<tr>
<td>4</td>
<td>24.2</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The trace demonstrates that highest month and highest value are properly set.

**Step 7**

Implement the loop in Java.

Here’s the loop for our example. Exercise E6.5 asks you to complete the program.

```java
double highestValue;
highestValue = in.nextDouble();
int highestMonth = 1;
```
for (int currentMonth = 2; currentMonth <= 12; currentMonth++)
{
    double nextValue = in.nextDouble();
    if (nextValue > highestValue)
    {
        highestValue = nextValue;
        highestMonth = currentMonth;
    }
}
System.out.println(highestMonth);

WORKED EXAMPLE 6.1

Credit Card Processing

Learn how to use a loop to remove spaces from a credit card number. Go to wiley.com/go/bjeo6examples and download Worked Example 6.1.

6.8 Nested Loops

In Section 5.4, 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 $x^n$.

This loop must be placed inside the preceding loop. We say that the inner loop is nested inside the outer loop.

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.
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 two loops to print the table header. However, those loops are not nested.

**section_8/PowerTable.java**

```java
/**
 * This program prints a table of powers of \( x \).
 */
public class PowerTable {
    public static void main(String[] args) {
        final int NMAX = 4;
        final double XMAX = 10;

        // Print table header
        for (int n = 1; n <= NMAX; n++)
            System.out.printf("%10d", n);
        System.out.println();
    }
}
```
6.8 Nested Loops

```java
for (int n = 1; n <= NMAX; n++)
{
    System.out.printf("%10s", "x ");
} System.out.println();

// Print table body

for (double x = 1; x <= XMAX; x++)
{
    // Print table row
    for (int n = 1; n <= NMAX; n++)
    {
        System.out.printf("%10.0f", Math.pow(x, n));
    }
    System.out.println();
}

Program Run

<table>
<thead>
<tr>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
</tbody>
</table>

| 1  | 1  | 1  | 1  |
| 2  | 4  | 8  | 16 |
| 3  | 9  | 27 | 64 |
| 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?

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?

   ```java
   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:

   ```plaintext
   [[]][[]]
   [[]][[]]
   [[]][[]]
   ```

**Practice It** Now you can try these exercises at the end of the chapter: R6.30, E6.17, E6.19.
Table 3  Nested Loop Examples

<table>
<thead>
<tr>
<th>Nested Loops</th>
<th>Output</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>for (i = 1; i &lt;= 3; i++) {</td>
<td>****</td>
<td>Prints 3 rows of 4 asterisks each.</td>
</tr>
<tr>
<td>for (j = 1; j &lt;= 4; j++) { Print &quot;*&quot; } System.out.println();</td>
<td>****</td>
<td></td>
</tr>
<tr>
<td>}</td>
<td>****</td>
<td></td>
</tr>
<tr>
<td>for (i = 1; i &lt;= 4; i++) {</td>
<td>***</td>
<td>Prints 4 rows of 3 asterisks each.</td>
</tr>
<tr>
<td>for (j = 1; j &lt;= 3; j++) { Print &quot;*&quot; } System.out.println();</td>
<td>***</td>
<td></td>
</tr>
<tr>
<td>}</td>
<td>***</td>
<td></td>
</tr>
<tr>
<td>for (i = 1; i &lt;= 4; i++) {</td>
<td>*</td>
<td>Prints 4 rows of lengths 1, 2, 3, and 4.</td>
</tr>
<tr>
<td>for (j = 1; j &lt;= i; j++) { Print &quot;*&quot; } System.out.println();</td>
<td>**</td>
<td></td>
</tr>
<tr>
<td>}</td>
<td>***</td>
<td></td>
</tr>
<tr>
<td>for (i = 1; i &lt;= 3; i++) {</td>
<td>-<em>-</em>-</td>
<td>Prints asterisks in even columns, dashes in odd</td>
</tr>
<tr>
<td>for (j = 1; j &lt;= 5; j++) {</td>
<td>-<em>-</em>-</td>
<td>columns.</td>
</tr>
<tr>
<td>if (j % 2 == 0) { Print &quot;*&quot; }</td>
<td>-<em>-</em>-</td>
<td></td>
</tr>
<tr>
<td>else { Print &quot;-&quot; }</td>
<td>-<em>-</em>-</td>
<td></td>
</tr>
<tr>
<td>}</td>
<td></td>
<td></td>
</tr>
<tr>
<td>System.out.println();</td>
<td></td>
<td></td>
</tr>
<tr>
<td>}</td>
<td></td>
<td></td>
</tr>
<tr>
<td>for (i = 1; i &lt;= 3; i++) {</td>
<td>* * *</td>
<td>Prints a checkerboard pattern.</td>
</tr>
<tr>
<td>for (j = 1; j &lt;= 5; j++) {</td>
<td>* * *</td>
<td></td>
</tr>
<tr>
<td>if (i % 2 == j % 2) { Print &quot;*&quot; }</td>
<td>* * *</td>
<td></td>
</tr>
<tr>
<td>else { Print &quot; &quot; }</td>
<td>* * *</td>
<td></td>
</tr>
<tr>
<td>}</td>
<td></td>
<td></td>
</tr>
<tr>
<td>System.out.println();</td>
<td></td>
<td></td>
</tr>
<tr>
<td>}</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

WORKED EXAMPLE 6.2  Manipulating the Pixels in an Image

Learn 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/bjeo6examples and download Worked Example 6.2.
6.9 Application: Random Numbers and Simulations

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.

6.9.1 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 Random class of the Java library implements a random number generator that produces numbers that appear to be completely random. To generate random numbers, you construct an object of the Random class, and then apply one of the following methods:

<table>
<thead>
<tr>
<th>Method</th>
<th>Returns</th>
</tr>
</thead>
<tbody>
<tr>
<td>nextInt(n)</td>
<td>A random integer between the integers 0 (inclusive) and n (exclusive)</td>
</tr>
<tr>
<td>nextDouble()</td>
<td>A random floating-point number between 0 (inclusive) and 1 (exclusive)</td>
</tr>
</tbody>
</table>

For example, you can simulate the cast of a die as follows:

```java
Random generator = new Random();
int d = 1 + generator.nextInt(6);
```

The call `generator.nextInt(6)` gives you a random number between 0 and 5 (inclusive). Add 1 to obtain a number between 1 and 6.

To give you a feeling for the random numbers, run the following program a few times.

**section_9_1/Die.java**

```java
import java.util.Random;

/**
 * This class models a die that, when cast, lands on a
 * random face.
 */

public class Die
{
    private Random generator;
    private int sides;
```
Chapter 6  Loops

```java
/**
   Constructs a die with a given number of sides.
   @param s  the number of sides, e.g., 6 for a normal die
*/
public Die(int s)
{
    sides = s;
    generator = new Random();
}

/**
   Simulates a throw of the die.
   @return the face of the die
*/
public int cast()
{
    return 1 + generator.nextInt(sides);
}
```

```java
/**
   This program simulates casting a die ten times.
*/
public class DieSimulator
{
    public static void main(String[] args)
    {
        Die d = new Die(6);
        final int TRIES = 10;
        for (int i = 1; i <= TRIES; i++)
        {
            int n = d.cast();
            System.out.print(n + " ");
        }
        System.out.println();
    }
}
```

Typical Program Run

6 5 6 3 2 6 3 4 4 1

Typical Program Run (Second Run)

3 2 2 1 6 5 3 4 1 2

As you can see, this program produces a different stream of simulated die casts every time it is run.

Actually, the numbers are not completely random. They are drawn from very long sequences of numbers that don’t repeat for a long time. These sequences are computed from fairly simple formulas; they just behave like random numbers. For that reason, they are often called pseudorandom numbers. Generating good sequences of numbers that behave like truly random sequences is an important and well-studied problem in computer science. We won’t investigate this issue further, though; we’ll just use the random numbers produced by the Random class.
6.9.2 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:

```java
double r = generator.nextDouble(); // 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.

Here is the program that carries out the simulation:

```java
import java.util.Random;

/**
 This program computes an estimate of pi by simulating dart throws onto a square.
 */

public class MonteCarlo {
  public static void main(String[] args) {
    final int TRIES = 10000;
    Random generator = new Random();

    int hits = 0;
    for (int i = 1; i <= TRIES; i++) {
      // Generate two random numbers between -1 and 1
      double r = generator.nextDouble();
      double x = -1 + 2 * r; // Between -1 and 1
      r = generator.nextDouble();
      double y = -1 + 2 * r;

```
22  // Check whether the point lies in the unit circle
23  if (x * x + y * y <= 1) { hits++; }
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

Program Run

Estimate for pi: 3.1504

42. How do you simulate a coin toss with the Random class?
43. How do you simulate the picking of a random playing card?
44. How would you modify the DieSimulator program to simulate tossing a pair
    of dice?
45. 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?
46. How do you generate a random floating-point number ≥ 0 and < 100?

Practice It  Now you can try these exercises at the end of the chapter: R6.31, E6.8, E6.22.

6.10 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 print­
out 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 and restart your program and see the contents of variables whenever your pro­
gram is temporarily stopped. At each stop, you have the choice of what variables to
inspect and how many program steps to run until the next stop.

Some people feel that debuggers are just a tool to make programmers lazy. Admit­
tedly some people write sloppy programs and then fix them up with a debugger, but
the majority of programmers make an honest effort to write the best program they
can before trying to run it through a debugger. These programmers realize that a
debugger, while more convenient than print commands, is not cost-free. It does take
time to set up and carry out an effective debugging session.

In actual practice, you cannot avoid using a debugger. The larger your programs get,
the harder it is to debug them simply by inserting print commands. The time invested
in learning about a debugger will be amply repaid in your programming career.

A debugger is a program that you can use to execute another program
and analyze its run-time behavior.
Like compilers, debuggers vary widely from one system to another. Some are quite primitive and require you to memorize a small set of arcane commands; others have an intuitive window interface. Figure 8 shows the debugger in the Eclipse development environment, downloadable for free from the Eclipse Foundation (eclipse.org). Other development environments, such as BlueJ, Netbeans, and IntelliJ IDEA also include debuggers.

You will have to find out how to prepare a program for debugging and how to start a debugger on your system. If you use an integrated development environment (with an editor, compiler, and debugger), this step is usually easy. You build the program in the usual way and pick a command to start debugging. On some systems, you must manually build a debug version of your program and invoke the debugger.

Once you have started the debugger, you can go a long way with just three debugging commands: “set breakpoint”, “single step”, and “inspect variable”. The names and keystrokes or mouse clicks for these commands differ widely, but all debuggers support these basic commands. You can find out how, either from the documentation or a lab manual, or by asking someone who has used the debugger before.

When you start the debugger, it runs at full speed until it reaches a **breakpoint**. Then execution stops, and the breakpoint that causes the stop is displayed (Figure 8). You can now inspect variables and step through the program one line at a time, or continue running the program at full speed until it reaches the next breakpoint. When the program terminates, the debugger stops as well.

![Figure 8](image.png)

*Figure 8*
Stopping at a Breakpoint
Breakpoints stay active until you remove them, so you should periodically clear the breakpoints that you no longer need.

Once the program has stopped, you can look at the current values of variables. Again, the method for selecting the variables differs among debuggers. Some debuggers always show you a window with the current local variables. On other debuggers you issue a command such as “inspect variable” and type in or click on the variable. The debugger then displays the contents of the variable. If all variables contain what you expected, you can run the program until the next point where you want to stop.

When inspecting objects, you often need to give a command to “open up” the object, for example by clicking on a tree node. Once the object is opened up, you see its instance variables (see Figure 9).

Running to a breakpoint gets you there speedily, but you don't know how the program got there. You can also step through the program one line at a time. Then you know how the program flows, but it can take a long time to step through it. The single-step command executes the current line and stops at the next program line. Most debuggers have two single-step commands, one called step into, which steps inside method calls, and one called step over, which skips over method calls.

For example, suppose the current line is

```java
String input = in.next();
Word w = new Word(input);
int syllables = w.countSyllables();
System.out.println("Syllables in " + input + ": " + syllables);
```

When you step over method calls, you get to the next line:

```java
String input = in.next();
Word w = new Word(input);
int syllables = w.countSyllables();
System.out.println("Syllables in " + input + ": " + syllables);
```

However, if you step into method calls, you enter the first line of the countSyllables method.

```java
public int countSyllables()
{
    int count = 0;
```
You should step into a method to check whether it carries out its job correctly. You should step over a method if you know it works correctly.

Finally, when the program has finished running, the debug session is also finished. To debug the program again, you must restart it in the debugger.

A debugger can be an effective tool for finding and removing bugs in your program. However, it is no substitute for good design and careful programming. If the debugger does not find any errors, it does not mean that your program is bug-free. Testing and debugging can only show the presence of bugs, not their absence.

47. In the debugger, you are reaching a call to `System.out.println`. Should you step into the method or step over it?

48. In the debugger, you are reaching the beginning of a method with a couple of loops inside. You want to find out the return value that is computed at the end of the method. Should you set a breakpoint, or should you step through the method?

49. When using the debugger, you find that a variable has an unexpected value. How can you go backwards to see when the variable changed?

50. When using a debugger, should you insert statements to print the values of variables?

51. Instead of using a debugger, could you simply trace a program by hand?

Practice It  Now you can try these exercises at the end of the chapter: R6.33, R6.34, R6.35.

HOW TO 6.2  Debugging

Knowing all about the mechanics of debugging may still leave you helpless when you fire up a debugger to look at a sick program. This How To presents a number of strategies that you can use to recognize bugs and their causes.

Step 1  Reproduce the error.

As you test your program, you notice that it sometimes does something wrong. It gives the wrong output, it seems to print something random, it goes in an infinite loop, or it crashes. Find out exactly how to reproduce that behavior. What numbers did you enter? Where did you click with the mouse?

Run the program again; type in exactly the same numbers, and click with the mouse on the same spots (or as close as you can get). Does the program exhibit the same behavior? If so, then it makes sense to fire up a debugger to study this particular problem. Debuggers are good for analyzing particular failures. They aren’t terribly useful for studying a program in general.

Step 2  Simplify the error.

Before you start up a debugger, it makes sense to spend a few minutes trying to come up with a simpler input that also produces an error. Can you use shorter words or simpler numbers and still have the program misbehave? If so, use those values during your debugging session.
Step 3
Divide and conquer.

Now that you have a particular failure, you want to get as close to the failure as possible. The key point of debugging is to locate the code that produces the failure. Just as with real insect pests, finding the bug can be hard, but once you find it, squashing it is usually the easy part. Suppose your program dies with a division by 0. Because there are many division operations in a typical program, it is often not feasible to set breakpoints to all of them. Instead, use a technique of divide and conquer. Step over the methods in `main`, but don’t step inside them. Eventually, the failure will happen again. Now you know which method contains the bug: It is the last method that was called from `main` before the program died. Restart the debugger and go back to that line in `main`, then step inside that method. Repeat the process.

Eventually, you will have pinpointed the line that contains the bad division. Maybe it is obvious from the code why the denominator is not correct. If not, you need to find the location where it is computed. Unfortunately, you can’t go back in the debugger. You need to restart the program and move to the point where the denominator computation happens.

Step 4
Know what your program should do.

A debugger shows you what the program does. You must know what the program should do, or you will not be able to find bugs. Before you trace through a loop, ask yourself how many iterations you expect the program to make. Before you inspect a variable, ask yourself what you expect to see. If you have no clue, set aside some time and think first. Have a calculator handy to make independent computations. When you know what the value should be, inspect the variable. If the value is what you expected, you must look further for the bug. If the value is different, you may be on to something. Double-check your computation. If you are sure your value is correct, find out why your program comes up with a different value.

In many cases, program bugs are the result of simple errors such as loop termination conditions that are off by one. Quite often, however, programs make computational errors. Maybe they are supposed to add two numbers, but by accident the code was written to subtract them. Programs don’t make a special effort to ensure that everything is a simple integer (and neither do real-world problems). You will need to make some calculations with large integers or nasty floating-point numbers. Sometimes these calculations can be avoided if you just ask yourself, “Should this quantity be positive? Should it be larger than that value?” Then inspect variables to verify those theories.

Step 5
Look at all details.

When you debug a program, you often have a theory about what the problem is. Nevertheless, keep an open mind and look at all details. What strange messages are displayed? Why does the program take another unexpected action? These details count. When you run a debugging session, you really are a detective who needs to look at every clue available.

If you notice another failure on the way to the problem that you are about to pin down, don’t just say, “I’ll come back to it later”. That very failure may be the original cause for your current problem. It is better to make a note of the current problem, fix what you just found, and then return to the original mission.

Step 6
Make sure you understand each bug before you fix it.

Once you find that a loop makes too many iterations, it is very tempting to apply a “Band-Aid” solution and subtract 1 from a variable so that the particular problem doesn’t appear again. Such a quick fix has an overwhelming probability of creating trouble elsewhere. You really need to have a thorough understanding of how the program should be written before you apply a fix.

It does occasionally happen that you find bug after bug and apply fix after fix, and the problem just moves around. That usually is a symptom of a larger problem with the program logic. There is little you can do with the debugger. You must rethink the program design and reorganize it.
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.

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.
Use a debugger to analyze your programs.

- A debugger is a program that you can use to execute another program and analyze its run-time behavior.
- You can make effective use of a debugger by mastering just three concepts: breakpoints, single-stepping, and inspecting variables.
- When a debugger executes a program, the execution is suspended whenever a breakpoint is reached.
- The single-step command executes the program one line at a time.
- Use the divide-and-conquer technique to locate the point of failure of a program.
- During debugging, compare the actual contents of variables against the values you know they should have.

**R6.1** Given the variables

```java
String stars = "*****";
String stripes = "======";
```

what do these loops print?

a. int i = 0;
   while (i < 5) {
      System.out.println(stars.substring(0, i));
      i++;
   }

b. 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); }
   }

**R6.2** 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++; }

**STANDARD LIBRARY ITEMS INTRODUCED IN THIS CHAPTER**

java.util.Random
nextInt
nextDouble

**REVIEW EXERCISES**

- R6.1 Given the variables

```java
String stars = "*****";
String stripes = "======";
```

what do these loops print?

a. int i = 0;
   while (i < 5) {
      System.out.println(stars.substring(0, i));
      i++;
   }

b. 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); }
   }

- R6.2 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++; }
Chapter 6  Loops

**R6.3** 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 = 5; 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);

**R6.4** 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.

**R6.5** 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 $3 + 7 + 7 = 17$.)

**R6.6** 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++; }

**R6.7** 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 + " "); } }

**R6.8** What is an infinite loop? On your computer, how can you terminate a program that executes an infinite loop?

**R6.9** Write a program trace for the pseudocode in Exercise E6.7, assuming the input values are 4 7 -2 -5 0.
Review Exercises

- **R6.10** What is an “off-by-one” error? Give an example from your own programming experience.

- **R6.11** What is a sentinel value? Give a simple rule when it is appropriate to use a numeric sentinel value.

- **R6.12** Which loop statements does Java support? Give simple rules for when to use each loop type.

- **R6.13** How many iterations do the following loops carry out? Assume that \( i \) is not changed in the loop body.
  
  - **a.** \texttt{for (int i = 1; i <= 10; i++)} 
  
  - **b.** \texttt{for (int i = 0; i < 10; i++)} 
  
  - **c.** \texttt{for (int i = 10; i > 0; i--)} 
  
  - **d.** \texttt{for (int i = -10; i <= 10; i++)} 
  
  - **e.** \texttt{for (int i = 10; i >= 0; i++)} 
  
  - **f.** \texttt{for (int i = -10; i <= 10; i = i + 2)} 
  
  - **g.** \texttt{for (int i = -10; i <= 10; i = i + 3)} 

- **R6.14** Write pseudocode for a program that prints a calendar such as the following.

  \[
  \begin{array}{ccccccc}
  \text{Su} & \text{M} & \text{T} & \text{W} & \text{Th} & \text{F} & \text{Sa} \\
  1 & 2 & 3 & 4 & 5 & 6 & 7 \\
  8 & 9 & 10 & 11 & 12 & 13 & 14 \\
  15 & 16 & 17 & 18 & 19 & 20 & 21 \\
  22 & 23 & 24 & 25 & 26 & 27 & 28 \\
  29 & 30 & 31 & & & & 
  \end{array}
  \]

- **R6.15** Write pseudocode for a program that prints a Celsius/Fahrenheit conversion table such as the following.

  \[
  \begin{array}{c|c}
  \text{Celsius} & \text{Fahrenheit} \\
  \hline
  0 & 32 \\
  10 & 50 \\
  20 & 68 \\
  \ldots & \ldots \\
  100 & 212 \\
  \end{array}
  \]

- **R6.16** 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

- **R6.17** 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 \texttt{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.
R6.18  Rewrite the following for loop into a while loop.

```java
int s = 0;
for (int i = 1; i <= 10; i++)
{
    s = s + i;
}
```

R6.19  Rewrite the following do loop into a while loop.

```java
int n = in.nextInt();
double x = 0;
double s;
do {
    s = 1.0 / (1 + n * n);
    n++;
    x = x + s;
}
while (s > 0.01);
```

R6.20  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);

R6.21  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);
R6.22  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);

R6.23  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.

R6.24  Add a storyboard panel for the conversion program in Section 6.6 on page 265 that shows a scenario where a user enters incompatible units.

R6.25  In Section 6.6, 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.

R6.26  Change the storyboards in Section 6.6 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.

R6.27  Draw a flow chart for a program that carries out unit conversions as described in Section 6.6.

R6.28  In Section 6.7.5, 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?

R6.29  What are nested loops? Give an example where a nested loop is typically used.
**R6.30** The nested loops

```java
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.

**R6.31** 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?

**R6.32** 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?

**Testing R6.33** Explain the differences between these debugger operations:
- Stepping into a method
- Stepping over a method

**Testing R6.34** Explain in detail how to inspect the string stored in a String object in your debugger.

**Testing R6.35** Explain in detail how to inspect the information stored in a Rectangle object in your debugger.

**Testing R6.36** Explain in detail how to use your debugger to inspect the balance stored in a BankAccount object.

**Testing R6.37** Explain the divide-and-conquer strategy to get close to a bug in a debugger.

**E6.1** 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.

**E6.2** 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 $3 + 7 + 7 = 17$.)

**E6.3** 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 $3 \ 5 \ 6$.

**E6.4** 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.

**E6.5** Complete the program in How To 6.1 on page 272. Your program should read twelve temperature values and print the month with the highest temperature.

**E6.6** Write a program that reads a set of floating-point values. Ask the user to enter the values (prompting only a single time for 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.

Your program should use a class `DataSet`. That class should have a method

```java
public void add(double value)
```

and methods `getAverage`, `getSmallest`, `getLargest`, and `getRange`.

**E6.7** 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.
While another value has been read successfully
    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.
```

**E6.8** 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:

```
| first | i | middle | j | last |
```
Then replace the string with
   first + word.charAt(j) + middle + word.charAt(i) + last

**E6.9** 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
```

**E6.10** 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
```

**E6.11** 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.

**E6.12** 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
```

**E6.13** Write a program that prints all powers of 2 from $2^0$ up to $2^{20}$.

**E6.14** 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
```

**E6.15** Using the Picture class from Worked Example 6.2, apply a sunset effect to a picture, increasing the red value of each pixel by 30 percent (up to a maximum of 255).

**E6.16** Using the Picture class from Worked Example 6.2, 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.
- **E6.17** 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
```

- **E6.18** 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:

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

- **E6.19** 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:

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

- **Business E6.20** *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 E6.21** 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.

- **E6.22** *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 third door, 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.

**PROGRAMMING PROJECTS**

- **P6.1** Enhance Worked Example 6.1 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.

- **P6.2** *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 \( \{x_1, \ldots, x_n\} \) is \( \bar{x} = \frac{\sum x_i}{n} \), where \( \sum x_i = x_1 + \cdots + x_n \) 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.
  Your program should use a class `DataSet`. That class should have a method
  ```java
  public void add(double value)
  ```
  and methods `getAverage` and `getStandardDeviation`.

- **P6.3** The *Fibonacci numbers* are defined by the sequence
  \[
  f_1 = 1 \\
  f_2 = 1 \\
  f_n = f_{n-1} + f_{n-2}
  \]
  Reformulate that as
  ```java
  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.

**P6.4** *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

Use a class `FactorGenerator` with a constructor `FactorGenerator(int numberToFactor)` and methods `nextFactor` and `hasMoreFactors`. Supply a class `FactorPrinter` whose main method reads a user input, constructs a `FactorGenerator` object, and prints the factors.

**P6.5** *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.

Use a class `PrimeGenerator` with methods `nextPrime` and `isPrime`. Supply a class `PrimePrinter` whose main method reads a user input, constructs a `PrimeGenerator` object, and prints the primes.

**P6.6** *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.

**P6.7** *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 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.

**P6.8** A simple random generator is obtained by the formula

\[ r_{\text{new}} = (a \cdot r_{\text{old}} + b) \mod m \]

and then setting \(r_{\text{old}} = r_{\text{new}}\). 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 \(\text{int}\) type is equivalent to computing the remainder.

Write a program that asks the user to enter a value for \(r_{\text{old}}\) (Such a value is often called a \textit{seed}). Then print the first 100 random integers generated by this formula, using \(a = 32310901\) and \(b = 1729\).

**P6.9** *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 \textit{hit}. (See Figure 10.) Buffon discovered that the quotient \(\text{tries/hits}\) 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 \(\alpha\) 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 11. Stop after 10,000 tries and print the quotient \(\text{tries/hits}\). (This program is not suitable for computing the value of \(\pi\). You need \(\pi\) in the computation of the angle.)

\[ \frac{y_{\text{high}}}{2} \]

\[ y_{\text{low}} \]

\[ 0 \]

**Figure 10**
The Buffon Needle Experiment

**Figure 11**
A Hit in the Buffon Needle Experiment

**P6.10** 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, lose a bet that at least one six would appear when rolling a die four times? And why did he seem to win 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.

**Business P6.11** 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 P6.12** 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 P6.13** 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.

**Science P6.14** In a predator-prey simulation, you compute the populations of predators and prey, using the following equations:

\[
\begin{align*}
prey_{n+1} &= prey_n \times (1 + A - B \times pred_n) \\
pred_{n+1} &= pred_n \times (1 - C + D \times prey_n)
\end{align*}
\]

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.

**Science P6.15** *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 \( s(t) = -\frac{1}{2} gt^2 + v_0 t \), where \( g = 9.81 \text{ m/s}^2 \) is the gravitational force of the earth. No calculus textbook ever states 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

```
const double DELTA_T = 0.01;
```
and update the position by
\[ s = s + v \cdot \text{DELTA}_T; \]
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
\[ v = v - g \cdot \text{DELTA}_T; \]
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 print out 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 P6.16** 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.)

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 \( x \).
Radioactive decay of radioactive materials can be modeled by the equation 

\[ A = A_0 e^{-\frac{t}{(\log 2)/h}} \]

where \( A \) is the amount of the material at time \( t \), \( A_0 \) is the amount at time 0, and \( h \) 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.

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.

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.

Write a graphical application that displays a checkerboard with 64 squares, alternating white and black.

Write a graphical application that draws a spiral, such as this one:

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 \( x_{\text{min}} \) to \( x_{\text{max}} \) and 

\[ d = (x_{\text{max}} - x_{\text{min}})/100. \]

Draw the curve \( f(x) = 0.00005x^3 - 0.03x^2 + 4x + 200 \), where \( x \) ranges from 0 to 400 in this fashion.
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

$$x = r \cdot \cos(\theta), \quad y = r \cdot \sin(\theta)$$

**1.** 23 years.

**2.** 8 years.

**3.** Add a statement

```java
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.**

<table>
<thead>
<tr>
<th>$n$</th>
<th>output</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>4</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>-1</td>
<td>-1</td>
</tr>
</tbody>
</table>

**7.**

<table>
<thead>
<tr>
<th>$n$</th>
<th>output</th>
</tr>
</thead>
<tbody>
<tr>
<td>½</td>
<td>1</td>
</tr>
<tr>
<td>⅔</td>
<td>1, 2</td>
</tr>
<tr>
<td>⅔</td>
<td>1, 2, 3</td>
</tr>
<tr>
<td>1</td>
<td>4</td>
</tr>
</tbody>
</table>

There is a comma after the last value. Usually, commas are between values only.

**8.**

<table>
<thead>
<tr>
<th>$a$</th>
<th>$n$</th>
<th>$r$</th>
<th>$i$</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>4</td>
<td>½</td>
<td>½</td>
</tr>
<tr>
<td></td>
<td>⅔</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>⅔</td>
<td>4</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>§</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td></td>
<td>16</td>
<td>5</td>
<td></td>
</tr>
</tbody>
</table>

The code computes $a^n$.

**9.**

<table>
<thead>
<tr>
<th>$n$</th>
<th>output</th>
</tr>
</thead>
<tbody>
<tr>
<td>½</td>
<td>1</td>
</tr>
<tr>
<td>⅔</td>
<td>11</td>
</tr>
<tr>
<td>⅔</td>
<td>21</td>
</tr>
<tr>
<td>§</td>
<td>31</td>
</tr>
<tr>
<td>§§</td>
<td>41</td>
</tr>
<tr>
<td>§§</td>
<td>51</td>
</tr>
<tr>
<td>§§</td>
<td>61</td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
</tbody>
</table>

This is an infinite loop. $n$ is never equal to 50.

**10.**

<table>
<thead>
<tr>
<th>count</th>
<th>temp</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>123</td>
</tr>
<tr>
<td>2</td>
<td>12.3</td>
</tr>
<tr>
<td>3</td>
<td>1.23</td>
</tr>
</tbody>
</table>

This yields the correct answer. The number 123 has 3 digits.

<table>
<thead>
<tr>
<th>count</th>
<th>temp</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>100</td>
</tr>
<tr>
<td>2</td>
<td>10.0</td>
</tr>
</tbody>
</table>

This yields the wrong answer. The number 100 also has 3 digits. The loop condition should have been while (temp >= 10).

**11.**

```java
int year = 1;
while (year <= numberOfYears)
{
    double interest = balance * RATE / 100;
    balance = balance + interest;
    year++;
}
```

**12.** 11 numbers: 10 9 8 7 6 5 4 3 2 1 0

**13.**

```java
for (int i = 10; i <= 20; i = i + 2)
{
    System.out.println(i);
}
```

**14.**

```java
int sum = 0;
for (int i = 1; i <= n; i++)
{
    sum = sum + i;
}
```

**15.**

```java
final int PERIODS = 5;
for (int i = 1; i <= PERIODS; i++)
{
    invest.waitYears(YEARS);
    System.out.printf("The balance after %d years is %.2f\n", invest.getYears(), invest.getBalance());
}
```

**16.**

```java
do
{
    System.out.print("Enter an integer between 0 and 100: ");
    value = in.nextInt();
} while (value < 0 || value > 100);
```
17. int value = 100;
   while (value >= 100)
   {
      System.out.println("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 nextDouble 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

27. Simple conversion

28. One score is not enough

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

31. The total is zero.

32. double total = 0;
   while (in.hasNextDouble())
   {
306  Chapter 6  Loops

```java
double input = in.nextDouble();
if (input > 0) { total = total + input; }
```

33. position is str.length() and ch is unchanged from its initial value, '?'.
    Note that ch must be initialized with some value—otherwise the
    compiler will complain about a possibly unini-
    tialized 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:
    ```java
    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.
    ```java
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 dis-
    played 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. Compute generator.nextInt(2), and use 0 for
    heads, 1 for tails, or the other way around.

43. Compute generator.nextInt(4) and associate
    the numbers 0 . . . 3 with the four suits. Then
    compute generator.nextInt(13) and associate
    the numbers 0 . . . 12 with Jack, Ace, 2 . . . 10,
    Queen, and King.

44. Construct two Die objects:
    ```java
    Die d1 = new Die(6);
    Die d2 = new Die(6);
    ```
    Then cast and print both of them:
    ```java
    System.out.println(  
        d1.cast() + " " + d2.cast());
    ```

45. The call will produce a value between 2 and
    12, but all values have the same proba-
    bility. When throwing a pair of dice, the number 7 is
    six times as likely as the number 2. The correct
    formula is
    ```java
    int sum = generator.nextInt(6)  
        + generator.nextInt(6) + 2;
    ```

46. generator.nextDouble() * 100.0

47. You should step over it because you are not
    interested in debugging the internals of the
    println method.

48. You should set a breakpoint. Stepping through
    loops can be tedious.

49. Unfortunately, most debuggers do not support
    going backwards. Instead, you must restart the
    program. Try setting breakpoints at the lines in
    which the variable is changed.

50. No, there is no need. You can just inspect the
    variables in the debugger.

51. For short programs, you certainly could. But
    when programs get longer, it would be very
    time-consuming to trace them manually.
WORKED EXAMPLE 6.1  

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".

Step 1  Decide what work must be done inside the loop.

In the loop, we visit each character in turn. You can get the ith 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.

```
Repeat
  Set ch to the ith 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.

```
4 1 2 3 - 5 6 7 8 - 9 0 1 2 - 3 4 5 0
```

```
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

\[
i = 0
\]
\[
\text{While } i < \text{creditCardNumber.length()}
\]
\[
\text{ch} = \text{the ith character of creditCardNumber.}
\]
\[
\text{If ch is a space or dash}
\]
\[
\text{Remove the character from creditCardNumber.}
\]
\[
\text{Else}
\]
\[
\text{Increment } i.
\]

It is a bit tedious to trace a string with 20 characters, so we will use a shorter example:

<table>
<thead>
<tr>
<th>creditCardNumber</th>
<th>i</th>
<th>ch</th>
</tr>
</thead>
<tbody>
<tr>
<td>4-56-7</td>
<td>0</td>
<td>4</td>
</tr>
<tr>
<td>4-56-7</td>
<td>1</td>
<td>-</td>
</tr>
<tr>
<td>456-7</td>
<td>1</td>
<td>5</td>
</tr>
<tr>
<td>456-7</td>
<td>2</td>
<td>6</td>
</tr>
<tr>
<td>456-7</td>
<td>3</td>
<td>-</td>
</tr>
<tr>
<td>4567</td>
<td>3</td>
<td>7</td>
</tr>
</tbody>
</table>

Step 6  Implement the loop in Java.

Here’s the complete program:

```java
/**
 * This program removes spaces and dashes from a credit card number.
 */

public class CCNumber {
    public static void main(String[] args) {
        String creditCardNumber = "4123-5678-9012-3450";
        int i = 0;
        while (i < creditCardNumber.length()) {
            char ch = creditCardNumber.charAt(i);
            if (ch == ' ' || ch == '-') {
                // Remove the character at position i
                String before = creditCardNumber.substring(0, i);
                String after = creditCardNumber.substring(i + 1);
            }
            System.out.println(ch);  // Tracing
            i++;
        }
    }
}
```

Big Java, 6e, Cay Horstmann, Copyright © 2015 John Wiley and Sons, Inc. All rights reserved.
creditCardNumber = before + after;
}

else {
    i++;
}

System.out.println(creditCardNumber);
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.

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:

```java
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. The negative of a Color object is computed like this:

```java
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 7, 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. Because there are pic.getWidth() columns, we use the loop

```java
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

```java
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**

```java
import java.awt.Color;

public class Negative {
  public static void main(String[] args) {
    Picture pic = new Picture();
    pic.load("queen-mary.png");
    for (int x = 0; x < pic.getWidth(); x++)
    {
      for (int y = 0; y < pic.getHeight(); y++)
      {
        Color original = pic.getColorAt(x, y);
        Color negative = new Color(255 - original.getRed(),
                                  255 - original.getGreen(),
                                  255 - original.getBlue());
        pic.setColorAt(x, y, negative);
      }
    }
  }
}
```
Manipulating the Pixels in an Image

```java
}
}
}```
Here is the source code. There are a couple bugs in this class.

```java
/**
   * This class describes words in a document. There are a couple
   * of bugs in this class.
   */
public class Word {
    private String text;

    /**
     * Constructs a word by removing leading and trailing non-
     * letter characters, such as punctuation marks.
     * @param s the input string
     */
    public Word(String s) {
        int i = 0;
        while (i < s.length() && !Character.isLetter(s.charAt(i)))
            i++;
        int j = s.length() - 1;
        while (j > i && !Character.isLetter(s.charAt(j)))
            j--;
        text = s.substring(i, j);
    }

    /**
     * Returns the text of the word, after removal of the
     * leading and trailing non-letter characters.
     * @return the text of the word
     */
    public String getText() {
        return text;
    }
}
```

This Worked Example presents a realistic example for running a debugger by examining a `Word` class whose primary purpose is to count the number of syllables in a word.

**Problem Statement**  The `Word` class uses this rule for counting syllables:

Each group of adjacent vowels (a, e, i, o, u, y) counts as one syllable (for example, the “ea” in “peach” contributes one syllable, but the “e-o” in “yellow” counts as two syllables). However, an “e” at the end of a word doesn’t count as a syllable. Each word has at least one syllable, even if the previous rules give a count of 0.

Also, when you construct a word from a string, any characters at the beginning or end of the string that aren’t letters are stripped off. That is useful when you read the input using the next method of the `Scanner` class. Input strings can still contain quotation marks and punctuation marks, and we don’t want them as part of the word.

Your task is to find and correct the errors in this program.
Counts the syllables in the word.

@return the syllable count

public int countSyllables()
{
    int count = 0;
    int end = text.length() - 1;
    if (end < 0) { return 0; } // The empty string has no syllables
    // An e at the end of the word doesn’t count as a vowel
    char ch = text.charAt(end);
    if (ch == 'e' || ch == 'E') { end--; }
    boolean insideVowelGroup = false;
    for (int i = 0; i <= end; i++)
    {
        ch = text.charAt(i);
        String vowels = "aeiouyAEIOUY";
        if (vowels.indexOf(ch) >= 0)
        {
            // ch is a vowel
            if (!insideVowelGroup)
            {
                // Start of new vowel group
                count++;
                insideVowelGroup = true;
            }
            }
    }
    // Every word has at least one syllable
    if (count == 0) { count = 1; }
    return count;
}

Here is a simple test class. Type in a sentence, and the syllable counts of all words are displayed.

worked_example_3/SyllableCounter.java

import java.util.Scanner;

/**
 * This program counts the syllables of all words in a sentence.
 */
public class SyllableCounter
{
    public static void main(String[] args)
    {
        Scanner in = new Scanner(System.in);
        System.out.println("Enter a sentence ending in a period.");
        String input;
        do
        {
            input = in.next();
            Word w = new Word(input);
            count++;
            insideVowelGroup = true;
        }
    }
}
Supply this input:

```
Hello yellow peach.
```

Then the output is

```
Syllables in Hello: 1
Syllables in yellow: 1
Syllables in peach.: 1
```

That is not very promising.

First, set a breakpoint in the first line of the `countSyllables` method of the `Word` class, in line 43 of `Word.java`. Then start the program. The program will prompt you for the input. The program will stop at the breakpoint you just set.

First, the `countSyllables` method checks the last character of the word to see if it is a letter 'e'. Let's just verify that this works correctly. Run the program to line 51 (see Figure 12).

![Figure 12](image-url) 
**Figure 12**  Debugging the `countSyllables` Method

Now inspect the variable `ch`. This particular debugger has a handy display of all current local and instance variables—see Figure 13. If yours doesn’t, you may need to inspect `ch`
manually. You can see that ch contains the value ‘1’. That is strange. Look at the source code. The end variable was set to text.length() - 1, the last position in the text string, and ch is the character at that position.

Looking further, you will find that end is set to 3, not 4, as you would expect. And text contains the string “He11”, not “He110”. Thus, it is no wonder that countSyllables returns the answer 1. We’ll need to look elsewhere for the culprit. Apparently, the Word constructor contains an error.

Unfortunately, a debugger cannot go back in time. Thus, you must stop the debugger, set a breakpoint in the Word constructor, and restart the debugger. Supply the input once again. The debugger will stop at the beginning of the Word constructor. The constructor sets two variables i and j, skipping past any nonletters at the beginning and the end of the input string. Set a breakpoint past the end of the second loop (see Figure 14) so that you can inspect the values of i and j.

At this point, inspecting i and j shows that i is 0 and j is 4. That makes sense—there were no punctuation marks to skip. So why is text being set to “He11”? Recall that the substring method counts positions up to, but not including, the second parameter. Thus, the correct call should be

```java
  text = s.substring(i, j + 1);
```

This is a very typical off-by-one error.
Fix this error, recompile the program, and try the three test cases again. You will now get the output

- Syllables in Hello: 1
- Syllables in yellow: 1
- Syllables in peach.: 1

As you can see, there still is a problem. Erase all breakpoints and set a breakpoint in the countSyllables method. Start the debugger and supply the input "Hello.". When the debugger stops at the breakpoint, start single stepping through the lines of the method. Here is the code of the loop that counts the syllables:

```java
boolean insideVowelGroup = false;
for (int i = 0; i <= end; i++)
{
    ch = text.charAt(i);
    String vowels = "aeiouyAEIOUY";
    if (vowels.indexOf(ch) >= 0)
    {
        // ch is a vowel
        if (!insideVowelGroup)
        {
            // Start of new vowel group
            count++;
            insideVowelGroup = true;
        }
    }
}
```

In the first iteration through the loop, the debugger skips the if statement. That makes sense, because the first letter, 'H', isn't a vowel. In the second iteration, the debugger enters the if statement, as it should, because the second letter, 'e', is a vowel. The insideVowelGroup variable is set to true, and the vowel counter is incremented. In the third iteration, the if statement is again skipped, because the letter 'l' is not a vowel. But in the fifth iteration, something weird happens. The letter 'o' is a vowel, and the if statement is entered. But the second if statement is skipped, and count is not incremented again.

Why? The insideVowelGroup variable is still true, even though the first vowel group was finished when the consonant 'l' was encountered. Reading a consonant should set insideVowelGroup back to false. This is a more subtle logic error, but not an uncommon one when designing a loop that keeps track of the processing state. To fix it, stop the debugger and add the following clause:

```java
if (vowels.indexOf(ch) >= 0)
{
    ... 
} else insideVowelGroup = false;
```

Now recompile and run the test once again. The output is:

- Syllables in Hello: 2
- Syllables in yellow: 2
- Syllables in peach.: 1

Is the program now free from bugs? That is not a question the debugger can answer. Remember: Testing can show only the presence of bugs, not their absence.
CHAPTER 7
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
To understand the concept of regression testing

CHAPTER CONTENTS

7.1 ARRAYS 308
SYN Arrays 309
CE1 Bounds Errors 314
CE2 Uninitialized and Unfilled Arrays 314
PT1 Use Arrays for Sequences of Related Items 314
PT2 Make Parallel Arrays into Arrays of Objects 314
ST1 Methods with a Variable Number of Arguments 315
C&S Computer Viruses 316

7.2 THE ENHANCED FOR LOOP 317
SYN The Enhanced for Loop 318

7.3 COMMON ARRAY ALGORITHMS 318
CE3 Underestimating the Size of a Data Set 327
ST2 Sorting with the Java Library 327

7.4 PROBLEM SOLVING: ADAPTING ALGORITHMS 327
HT1 Working with Arrays 330
WE1 Rolling the Dice 330

7.5 PROBLEM SOLVING: DISCOVERING ALGORITHMS BY MANIPULATING PHYSICAL OBJECTS 332

7.6 TWO-DIMENSIONAL ARRAYS 336
SYN Two-Dimensional Array Declaration 337
WE2 A World Population Table 337
ST3 Two-Dimensional Arrays with Variable Row Lengths 341
ST4 Multidimensional Arrays 343

7.7 ARRAY LISTS 343
SYN Array Lists 343
CE4 Length and Size 352
ST5 The Diamond Syntax 352

7.8 REGRESSION TESTING 352
PT3 Batch Files and Shell Scripts 354
C&S The Therac-25 Incidents 355
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.

7.1 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.

7.1.1 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:

```java
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):

```java
double[] values;
```

When you declare an array variable, it is not yet initialized. You need to initialize the variable with the array:

```java
double[] values = new double[10];
```
Declare the array variable

Initialize it with an array

Access an array element

**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,

```java
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:

```java
values[4] = 35;
```

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:

```java
System.out.println(values[4]);
```

**Syntax 7.1 Arrays**

**Syntax**

To construct an array:

```
new typeName[length]
```

To access an element:

```
arrayReference[index]
```

**Type of array variable**

`double[]`

**Name of array variable**

`values`

**Element type**

`double`

**Length**

`10`

**Use brackets to access an element.**

```
values[4] = 0;
```

**List of initial values**

```
{ 32, 54, 67.5, 29, 35, 80, 115, 44.5, 100, 65 }
```

The index must be ≥ 0 and < the length of the array.

See page 314.
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

```java
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:

```java
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`.

### Table 1 Declaring Arrays

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>int[] numbers = new int[10];</code></td>
<td>An array of ten integers. All elements are initialized with zero.</td>
</tr>
<tr>
<td><code>final int LENGTH = 10;</code></td>
<td>It is a good idea to use a named constant instead of a “magic number”.</td>
</tr>
<tr>
<td><code>int[] numbers = new int[LENGTH];</code></td>
<td></td>
</tr>
<tr>
<td><code>int length = in.nextInt();</code></td>
<td>The length need not be a constant.</td>
</tr>
<tr>
<td><code>double[] data = new double[length];</code></td>
<td></td>
</tr>
<tr>
<td><code>int[] squares = { 0, 1, 4, 9, 16 };</code></td>
<td>An array of five integers, with initial values.</td>
</tr>
<tr>
<td><code>double[] data = new int[10];</code></td>
<td><strong>Error:</strong> You cannot initialize a <code>double[]</code> variable with an array of type <code>int[]</code>.</td>
</tr>
</tbody>
</table>
The following code ensures that you only access the array when the index variable \( i \) is within the legal bounds:

```java
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 7.3.9.

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:

```java
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]`.

### 7.1.2 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.) You have already seen this behavior with objects in Section 2.8. When you access an object or array, you need not be concerned about the fact that Java uses references. This only becomes important when you copy a reference.

When you copy an array variable into another, both variables refer to the same array (see Figure 2).

```java
int[] scores = { 10, 9, 7, 4, 5 };
int[] values = scores; // Copying array reference
```

You can modify the array through either of the variables:

```java
scores[3] = 10;
System.out.println(values[3]); // Prints 10
```

Section 7.3.9 shows how you can make a copy of the *contents* of the array.
7.1.3 Using Arrays with Methods

Arrays can be method arguments and return values, just like any other values. When you define a method with an array argument, you provide a parameter variable for the array. For example, the following method adds scores to a Student object:

```java
public void addScores(int[] values)
{
    for (int i = 0; i < values.length; i++)
    {
        totalScore = totalScore + values[i];
    }
}
```

To call this method, you have to provide an array:

```java
int[] scores = { 10, 9, 7, 10 };
fred.addScores(scores);
```

Conversely, a method can return an array. For example, a Student class can have a method

```java
public int[] getScores()
```

that returns an array with all of the student's scores.

7.1.4 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:

```java
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:

```java
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.
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?
   ```java
   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?
   ```java
   for (int i = 0; i < 5; i++)
   {
       primes[i]++;
   }
   ```

4. Given the declaration
   ```java
   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 308 without storing the inputs in an array, by using an algorithm similar to the algorithm for finding the maximum in Section 6.7.5?

8. Declare a method of a class `Lottery` that returns a combination of `n` numbers. You don’t need to implement the method.

**Practice It**  Now you can try these exercises at the end of the chapter: R7.5, R7.6, R7.10, E7.1.
**Bounds Errors**

Perhaps the most common error in using arrays is accessing a nonexistent element.

```java
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.

---

**Uninitialized and Unfilled Arrays**

A common error is to allocate an array variable, but not an actual array.

```java
double[] values;
values[0] = 29.95; // Error—values not initialized
```

Array variables work exactly like object variables—they are only references to the actual array. To construct the actual array, you must use the `new` operator:

```java
double[] values = new double[10];
```

Another common error is to allocate an array of objects and expect it to be filled with objects.

```java
BankAccount[] accounts = new BankAccount[10]; // Contains ten null references
```

This array contains null references, not default bank accounts. You need to remember to fill the array, for example:

```java
for (int i = 0; i < 10; i++)
{
    accounts[i] = new BankAccount();
}
```

---

**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:

```java
int[] scores = new int[NUMBER_OF_SCORES];
```

But an array

```java
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.

---

**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.

```java
// Don't do this
int[] accountNumbers;
double[] balances;
```
Arrays such as these are called parallel arrays (see Figure 4). The \( i \)th slice (\( \text{accountNumbers}[i] \) and \( \text{balances}[i] \)) contains data that need to be processed together.

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

```java
BankAccount[] accounts;
```

(See Figure 5.)

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 \( \text{BankAccount} \) 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.

Methods with a Variable Number of Arguments

It is possible to declare methods that receive a variable number of arguments. For example, we can write a method that can add an arbitrary number of scores to a student:

```java
fred.addScores(10, 7); // This method call has two arguments
fred.addScores(1, 7, 2, 9); // Another call to the same method, now with four arguments
```

The method must be declared as

```java
public void addScores(int... values)
```

The \( \text{int...} \) type indicates that the method can receive any number of \( \text{int} \) arguments. The \( \text{values} \) parameter variable is actually an \( \text{int[]} \) array that contains all arguments that were passed to the method.
The method implementation traverses the `values` array and processes the elements:
```java
public void addScores(int... values)
{
    for (int i = 0; i < values.length; i++) // values is an int[]
    {
        totalScore = totalScore + values[i];
    }
}
```

**Computing & Society 7.1  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).

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, all programmers must be very 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. One may well speculate what would possess the virus author to spend weeks designing a program that disabled thousands of computers. It appears that the break-in was fully intended by the author, but the disabling of the computers was a bug caused by continuous reinfection. Morris was sentenced to three years probation, 400 hours of community service, and a $10,000 fine.

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. Keep your operating system and antivirus program up to date, and don’t click on suspicious links on a web page or in your e-mail inbox. 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.
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:

```java
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 with an explicit index variable:

```java
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:

```java
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.

```java
for (int i = 0; i < values.length; i++)
{
    values[i] = 0; // OK
}
```

The enhanced for loop is a convenient mechanism for traversing all elements in a collection.
Chapter 7 Arrays and Array Lists

### Syntax 7.2 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.

**An array**

```
for (double element : values)
{
    sum = sum + element;
}
```

**The variable contains an element, not an index.**

**These statements are executed for each element.**

9. What does this enhanced for loop do?

```java
int counter = 0;
for (double element : values)
{
    if (element == 0) { counter++; }
}
```

10. Write an enhanced for loop that prints all elements in the array `values`.

11. Write an enhanced for loop that multiplies all elements in a `double[]` array named `factors`, accumulating the result in a variable named `product`.

12. Why is the enhanced for loop not an appropriate shortcut for the following basic for loop?

```java
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: R7.11, R7.12, R7.13.

### 7.3 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.

#### 7.3.1 Filling

This loop fills an array with squares (0, 1, 4, 9, 16, ...). Note that the element with index 0 contains 0², the element with index 1 contains 1², and so on.

```java
for (int i = 0; i < values.length; i++)
{
    values[i] = i * i;
}
```
7.3.2 Sum and Average Value

You have already encountered this algorithm in Section 6.7.1. When the values are located in an array, the code looks much simpler:

```java
double total = 0;
for (double element : values)
{
    total = total + element;
}
double average = 0;
if (values.length > 0) { average = total / values.length; }
```

7.3.3 Maximum and Minimum

Use the algorithm from Section 6.7.5 that keeps a variable for the largest element already encountered. Here is the implementation of that algorithm for an array:

```java
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.

7.3.4 Element Separators

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:

```java
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. (You’ll need to import `java.util.Arrays`.) The expression

```
Arrays.toString(values)
```

returns a string describing the contents of the array `values` in the form

```
[32, 54, 67.5, 29, 35]
```
The elements are surrounded by a pair of brackets and separated by commas. This method can be convenient for debugging:

```java
System.out.println("values=" + Arrays.toString(values));
```

### 7.3.5 Linear Search

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:

```java
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. We discuss binary search in Chapter 14.

### 7.3.6 Removing an Element

Suppose you want to remove the element with index `pos` from the array `values`. As explained in Section 7.1.4, 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 6.)

---

**Figure 6**
Removing an Element in an Unordered Array

**Figure 7**
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 7.)

for (int i = pos + 1; i < currentSize; i++)
{
    values[i - 1] = values[i];
}
currentSize--;

7.3.7 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 7.1.4.

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 9).

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;
}
7.3.8 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:

```java
double temp = values[i];
values[i] = values[j];
values[j] = temp;
```

Now we can set $values[j]$ to the saved value.

```
values[j] = temp;
```

Figure 10 shows the process.

![Figure 10: Swapping Array Elements](image)
### 7.3.9 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 11):

```java
double[] values = new double[6];
... // Fill array
double[] prices = values; 1
```

If you want to make a true copy of an array, call the `Arrays.copyOf` method (as shown in Figure 11).

```java
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 > values.length`) into it, and returns the new array.

In order to use the `Arrays` class, you need to add the following statement to the top of your program:

```java
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 12):

```java
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

```java
double[] newValues = Arrays.copyOf(values, n)
```

with
Move elements to a larger array

```java
double[] newValues = new double[n];
for (int i = 0; i < n && i < values.length; i++)
{
    newValues[i] = values[i];
}
```

7.3.10 Reading Input

If you know how many inputs the user will supply, it is simple to place them into an array:

```java
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.

```java
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.

section_3/LargestInArray.java

```java
import java.util.Scanner;

/**
 * This program reads a sequence of values and prints them, marking the largest value.
 */
public class LargestInArray
{
    public static void main(String[] args)
    {
        final int LENGTH = 100;
        double[] values = new double[LENGTH];
        int currentSize = 0;

        // Read inputs
        System.out.println("Please enter values, Q to quit: ");
        Scanner in = new Scanner(System.in);
        while (in.hasNextDouble() && currentSize < values.length)
        {
            values[currentSize] = in.nextDouble();
            currentSize++;
        }

        // Find the largest value
        double largest = values[0];
        for (int i = 1; i < currentSize; i++)
        {
            if (values[i] > largest)
            {
                largest = values[i];
            }
        }
```
34 // Print all values, marking the largest
35 for (int i = 0; i < currentSize; i++)
36 {
37     System.out.print(values[i]);
38     if (values[i] == largest)
39     {
40         System.out.print(" == largest value");
41     }
42     System.out.println();
43 }
44 }
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

13. Given these inputs, what is the output of the LargestInArray program?

20 10 20 Q

14. Write a loop that counts how many elements in an array are equal to zero.

15. 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];
    }
}

16. 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.

17. 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]);
    }

18. 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;
        }
    }
19. 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?

```java
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: R7.16, R7.19, E7.7.

### 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.

### 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

```java
Arrays.sort(values);
```

If the array is partially filled, call

```java
Arrays.sort(values, 0, currentSize);
```

### 7.4 Problem Solving: Adapting Algorithms

In Section 7.3, 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 7.3.2)
- Finding the minimum value (Section 7.3.3)
- Removing an element (Section 7.3.6)

We can formulate a plan of attack that combines these algorithms:

Find the minimum.
Remove it from the array.
Calculate the sum.

Let’s try it out with our example. The minimum of

<p>| | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>7</td>
<td>8.5</td>
<td>9.5</td>
<td>7</td>
<td>4</td>
</tr>
<tr>
<td>10</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

is 4. How do we remove it?

Now we have a problem. The removal algorithm in Section 7.3.6 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 7.3.5)

We need to fix our plan of attack:

Find the minimum value.
Find its position.
Remove that position from the array.
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.

<p>| | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>7</td>
<td>8.5</td>
<td>9.5</td>
<td>7</td>
<td>4</td>
</tr>
<tr>
<td>10</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

We remove it:

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>7</td>
<td>8.5</td>
<td>9.5</td>
<td>7</td>
</tr>
<tr>
<td>10</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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:

```java
double smallest = values[0];
for (int i = 1; i < values.length; i++)
{
    if (values[i] < smallest)
    {
        smallest = values[i];
    }
}
```

You should be familiar with the implementation of fundamental algorithms so that you can adapt them.
When we find the smallest value, we also want to update the position:

```java
if (values[i] < smallest)
{
    smallest = values[i];
    smallestPosition = i;
}
```

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:

```java
int smallestPosition = 0;
for (int i = 1; i < values.length; i++)
{
    if (values[i] < values[smallestPosition])
    {
        smallestPosition = i;
    }
}
```

With this adaptation, our problem is solved with the following strategy:

- Find the position of the minimum.
- Remove it from the array.
- 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.

20. Section 7.3.6 has two algorithms for removing an element. Which of the two should be used to solve the task described in this section?

21. It isn’t actually necessary to remove the minimum in order to compute the total score. Describe an alternative.

22. How can you print the number of positive and negative values in a given array, using one or more of the algorithms in Section 6.7?

23. How can you print all positive values in an array, separated by commas?

24. Consider the following algorithm for collecting all matches in an array:

```java
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 23?

**Practice It** Now you can try these exercises at the end of the chapter: R7.25, R7.26.
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 7.3. 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 scores are

8 7 8.5 9.5 7 5 10

then the final score is 50.

Step 2  Determine which algorithm(s) you need.
Sometimes, a step corresponds to exactly one of the basic array algorithms in Section 7.3. That is the case with calculating the sum (Section 7.3.2) and reading the inputs (Section 7.3.10). At other times, you need to combine several algorithms. To remove the minimum value, you can find the minimum value (Section 7.3.3), find its position (Section 7.3.5), and remove the element at that position (Section 7.3.6).
We have now refined our plan as follows:

Read inputs.
Find the minimum.
Find its position.
Remove the minimum.
Calculate the sum.

This plan will work—see Section 7.4. 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.
Step 3 Use classes and 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 carry out each processing step in a separate method. It is also a good idea to come up with a class that is responsible for collecting and processing the data.

In our example, let's provide a class Student. A student has an array of scores.

```java
public class Student
{
    private double[] scores;
    private double scoresSize;
    ...
    public Student(int capacity) { ... }
    public boolean addScore(double score) { ... }
    public double finalScore() { ... }
}
```

A second class, ScoreAnalyzer, is responsible for reading the user input and displaying the result. Its main method simply calls the Student methods:

```java
Student fred = new Student(100);
System.out.println("Please enter values, Q to quit:");
while (in.hasNextDouble())
{
    if (!fred.addScore(in.nextDouble()))
    {
        System.out.println("Too many scores.");
        return;
    }
}
System.out.println("Final score: " + fred.finalScore());
```

Now the finalScore method must do the heavy lifting. It too should not have to do all the work. Instead, we will supply helper methods

```java
public double sum()
public double minimum()
```

These methods simply implement the algorithms in Sections 7.3.2 and 7.3.3. Then the finalScore method becomes

```java
public double finalScore()
{
    if (scoresSize == 0)
    {
        return 0;
    }
    else if (scoresSize == 1)
    {
        return scores[0];
    }
    else
    {
        return sum() - minimum();
    }
}
```

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:

<table>
<thead>
<tr>
<th>Test Case</th>
<th>Expected Output</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>8 7 8.5 9.5 7 5 10</td>
<td>50</td>
<td>See Step 1.</td>
</tr>
<tr>
<td>8 7 7 9</td>
<td>24</td>
<td>Only one instance of the low score should be removed.</td>
</tr>
<tr>
<td>8</td>
<td>0</td>
<td>After removing the low score, no score remains.</td>
</tr>
<tr>
<td>(no inputs)</td>
<td>Error</td>
<td>That is not a legal input.</td>
</tr>
</tbody>
</table>

The complete program is in the how_to_1 folder of your companion code.

WORKED EXAMPLE 7.1 Rolling the Dice

Learn how to analyze a set of die tosses to see whether the die is “fair”. Go to wiley.com/go/bjeo6examples and download the file for Worked Example 7.1.

7.5 Problem Solving: Discovering Algorithms by Manipulating Physical Objects

In Section 7.4, 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 7.3.7) or swapped (Section 7.3.8), 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 7.3.6):

We can insert a coin (Section 7.3.7):

Or we can swap two coins (Section 7.3.8).

Go ahead—line up some coins and try out these three operations right now so that you get a feel for them.
Chapter 7 Arrays and Array Lists

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:

\[
\begin{align*}
  i &= 0 \\
  j &= \ldots \text{ (we’ll think about that in a minute)} \\
  \text{While (don’t know yet)} \\
  &\quad \text{Swap elements at positions } i \text{ and } j \\
  &\quad i++ \\
  &\quad j++
\end{align*}
\]

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 \(\text{size} / 2\).

And how many iterations do we make? We need to swap all coins in the first half. That is, we need to swap \(\text{size} / 2\) coins.
The pseudocode is

```plaintext
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 6.2). 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 25 asks you to carry out the walkthrough, and Exercise E7.8 asks you to translate the pseudocode to Java. Exercise R7.27 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!

25. 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.

26. Take out some coins and simulate the following pseudocode, using two paper clips to indicate the positions for i and j.

```plaintext
i = 0
j = size - 1
While [i < j]
    Swap elements at positions i and j
    i++
    j--
```

What does the algorithm do?

27. 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
```

Using coins and paper clips, discover an algorithm that solves this task by swapping elements, then describe it in pseudocode.

28. Discover an algorithm for the task of Self Check 27 that uses removal and insertion of elements instead of swapping.

29. Consider the algorithm in Section 6.7.5 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?

Practice It  Now you can try these exercises at the end of the chapter: R7.27, R7.28, E7.8.
7.6 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 13: the medal counts of the figure skating competitions at the 2014 Winter Olympics.

<table>
<thead>
<tr>
<th></th>
<th>Gold</th>
<th>Silver</th>
<th>Bronze</th>
</tr>
</thead>
<tbody>
<tr>
<td>Canada</td>
<td>0</td>
<td>3</td>
<td>0</td>
</tr>
<tr>
<td>Italy</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>Germany</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>Japan</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Kazakhstan</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>Russia</td>
<td>3</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>South Korea</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>United States</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

Figure 13 Figure Skating Medal Counts

7.6.1 Declaring Two-Dimensional Arrays

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:

```java
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:

```java
int[][] counts =
    {
        { 0, 3, 0 },
        { 0, 0, 1 },
        { 0, 0, 1 },
        { 1, 0, 0 },
        { 0, 0, 1 },
        { 3, 1, 1 },
        { 0, 1, 0 },
        { 1, 0, 1 }
    };
```
### Syntax 7.3

**Two-Dimensional Array Declaration**

```java
int[][] data = {
    { 16, 3, 2, 13 },
    { 5, 10, 11, 8 },
    { 9, 6, 7, 12 },
    { 4, 15, 14, 1 },
};
```

- **Name**: `double[][] tableEntries = new double[7][3];`
- **Element type**: `double[][]`
- **Number of rows**: 7
- **Number of columns**: 3

As with one-dimensional arrays, you cannot change the size of a two-dimensional array once it has been declared.

### 7.6.2 Accessing Elements

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 14):

```java
int medalCount = counts[3][1];
```

To access all elements in a two-dimensional array, you use nested loops. For example, the following loop prints all elements of `counts`:

```java
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 14**

Accessing an Element in a Two-Dimensional Array
In these loops, the number of rows and columns were given as constants. Alternatively, you can use the following expressions:

- `counts.length` is the number of rows.
- `counts[0].length` is the number of columns. (See Special Topic 7.3 for an explanation of this expression.)

With these expressions, the nested loops become

```java
for (int i = 0; i < counts.length; i++)
{
    for (int j = 0; j < counts[0].length; j++)
    {
        System.out.printf("%8d", counts[i][j]);
    }
    System.out.println();
}
```

### 7.6.3 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 15 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 `counts[i][j]`. You need to check whether the element is located at the top or bottom of the array:

```java
int total = 0;
if (i > 0) { total = total + counts[i - 1][j]; }
if (i < ROWS - 1) { total = total + counts[i + 1][j]; }
```

![Figure 15]

Neighboring Locations in a Two-Dimensional Array

### 7.6.4 Accessing Rows and Columns

You often need to access all elements in a row or column, for example to compute the sum of the elements or the largest element in a row or column.
In our sample array, the row totals give us the total number of medals won by a particular country.

Finding the correct 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:

![Diagram showing rows and columns]

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:

```java
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 \( \text{COUNTRIES} - 1 \).

```java
int total = 0;
for (int i = 0; i < COUNTRIES; i++)
{
    total = total + counts[i][j];
}
```

Working with two-dimensional arrays is illustrated in the following program. The program prints out the medal counts and the row totals.
This program prints a table of medal winner counts with row totals.

```java
public class Medals {
    public static void main(String[] args) {
        final int COUNTRIES = 8;
        final int MEDALS = 3;

        String[] countries = {
            "Canada",
            "Italy",
            "Germany",
            "Japan",
            "Kazakhstan",
            "Russia",
            "South Korea",
            "United States"
        };

        int[][] counts = {
            {0, 3, 0},
            {0, 0, 1},
            {0, 0, 1},
            {1, 0, 0},
            {0, 0, 1},
            {3, 1, 1},
            {0, 1, 0},
            {1, 0, 1}
        };

        System.out.println("        Country    Gold  Silver  Bronze   Total");

        // Print countries, counts, and row totals
        for (int i = 0; i < COUNTRIES; i++) {
            // Process the ith row
            System.out.printf("%15s", countries[i]);
            int total = 0;

            // Print each row element and update the row total
            for (int j = 0; j < MEDALS; j++) {
                System.out.printf("%8d", counts[i][j]);
                total = total + counts[i][j];
            }

            // Display the row total and print a new line
            System.out.printf("%8d\n", total);
        }
    }
}
```
Program Run

<table>
<thead>
<tr>
<th>Country</th>
<th>Gold</th>
<th>Silver</th>
<th>Bronze</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>Canada</td>
<td>0</td>
<td>3</td>
<td>0</td>
<td>3</td>
</tr>
<tr>
<td>Italy</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Germany</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Japan</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>Kazakhstan</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Russia</td>
<td>3</td>
<td>1</td>
<td>1</td>
<td>5</td>
</tr>
<tr>
<td>South Korea</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>United States</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>2</td>
</tr>
</tbody>
</table>

30. What results do you get if you total the columns in our sample medals data?

31. Consider an $8 \times 8$ array for a board game:

   ```java
   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.
   
32. 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 " ".

33. Write an assignment statement to place an "x" in the upper-right corner of the
   tic-tac-toe board in Self Check 32.

34. Which elements are on the diagonal joining the upper-left and the lower-right
   corners of the tic-tac-toe board in Self Check 32?

Practice It  Now you can try these exercises at the end of the chapter: R7.29, E7.16, E7.17.

WORKED EXAMPLE 7.2  A World Population Table

Learn 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/bjeo6examples and download the file for
Worked Example 7.2.

Two-Dimensional Arrays with Variable Row Lengths

When you declare a two-dimensional array with the command

   ```java
   int[][] a = new int[3][3];
   ```

you get a $3 \times 3$ matrix that can store 9 elements:

   ```java
   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:

\[
\begin{align*}
    & b[0][0] \\
    & b[1][0] \ b[1][1] \\
    & b[2][0] \ b[2][1] \ b[2][2]
\end{align*}
\]

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:

```java
double[][] b = new double[3][];
```

Then allocate each row separately (see Figure 16):

```java
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. 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:

```java
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:

```java
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 \(\text{new int}[3][3]\) automatically allocates an array of three rows, and three arrays for the rows’ contents.

Figure 16  A Triangular Array
### Multidimensional Arrays

You can declare arrays with more than two dimensions. For example, here is a three-dimensional array:

```java
int[][][] rubiksCube = new int[3][3][3];
```

Each array element is specified by three index values:

```java
rubiksCube[i][j][k]
```

### 7.7 Array Lists

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.

### Syntax 7.4 Array Lists

**Syntax**

To construct an array list:

```java
new ArrayList<typeName>()
```

To access an element:

```java
arraylistReference.get(index)
arraylistReference.set(index, value)
```

**Variable type** ArrayList<String>

**Variable name** friends = new ArrayList<String>();

**An array list object of size 0**

Use the get and set methods to access an element.

friends.add("Cindy");
```java
String name = friends.get(i);
friends.set(i, "Harry");
```

The add method appends an element to the array list, increasing its size.

The index must be $0 \leq i < friends.size()$. 

© digital94086/iStockphoto.
7.7.1 Declaring and Using Array Lists

The following statement declares an array list of strings:

```java
ArrayList<String> names = new ArrayList<String>();
```

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 7.7.4 shows how you can collect numbers in an array list.

It is a common error to forget the initialization:

```java
ArrayList<String> names;
names.add("Harry"); // Error—names not initialized
```

Here is the proper initialization:

```java
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.

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.

```java
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"
```

The size increases after each call to `add` (see Figure 17). 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:

```java
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:

```java
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:

```java
names.set(2, "Carolyn");
```

![Figure 17](image-url)  
Adding an Array List Element with `add`
An array list has methods for adding and removing elements in the middle.

Use the add and remove methods to add and remove array list elements.

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 18).

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 18 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:

```java
System.out.println(names); // Prints [Emily, Bob, Carolyn]
```

### 7.7.2 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:

```java
ArrayList<String> names = . . . ;
for (String name : names)
{
    System.out.println(name);
}
```

This loop is equivalent to the following basic for loop:

```java
for (int i = 0; i < names.size(); i++)
{
```
Chapter 7  Arrays and Array Lists

```java
    String name = names.get(i);
    System.out.println(name);
}
```

### 7.7.3 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 19).

```java
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:

```java
ArrayList<String> newNames = new ArrayList<String>(names);
```

---

**Table 2  Working with Array Lists**

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>ArrayList&lt;String&gt; names = new ArrayList&lt;String&gt;();</code></td>
<td>Constructs an empty array list that can hold strings.</td>
</tr>
</tbody>
</table>
| `names.add("Ann");
    names.add("Cindy");` | Adds elements to the end of the array list. |
| `System.out.println(names);` | Prints [Ann, Cindy]. |
| `names.add(1, "Bob");` | Inserts an element at index 1. `names` is now [Ann, Bob, Cindy]. |
| `names.remove(0);` | Removes the element at index 0. `names` is now [Bob, Cindy]. |
| `names.set(0, "Bill");` | Replaces an element with a different value. `names` is now [Bill, Cindy]. |
| `String name = names.get(i);` | Gets an element. |
| `String last = names.get(names.size() - 1);` | Gets the last element. |
| `ArrayList<Integer> squares = new ArrayList<Integer>();
    for (int i = 0; i < 10; i++)
    {
        squares.add(i * i);
    }` | Constructs an array list holding the first ten squares. |
7.7.4 Wrappers and Auto-boxing

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.

<table>
<thead>
<tr>
<th>Primitive Type</th>
<th>Wrapper Class</th>
</tr>
</thead>
<tbody>
<tr>
<td>byte</td>
<td>Byte</td>
</tr>
<tr>
<td>boolean</td>
<td>Boolean</td>
</tr>
<tr>
<td>char</td>
<td>Character</td>
</tr>
<tr>
<td>double</td>
<td>Double</td>
</tr>
<tr>
<td>float</td>
<td>Float</td>
</tr>
<tr>
<td>int</td>
<td>Integer</td>
</tr>
<tr>
<td>long</td>
<td>Long</td>
</tr>
<tr>
<td>short</td>
<td>Short</td>
</tr>
</tbody>
</table>

For example, to collect `double` values in an array list, you use 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 20).

```
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 20](image)
### 7.7.5 Using Array Algorithms with Array Lists

The array algorithms in Section 7.3 can be converted to arrays lists simply by using the array list methods instead of the array syntax (see Table 3 on page 350). For example, this code snippet finds the largest element in an array:

```java
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:

```java
double largest = values.get(0);
for (int i = 1; i < values.size(); i++)
{
    if (values.get(i) > largest)
    {
        largest = values.get(i);
    }
}
```

### 7.7.6 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:

```java
ArrayList<Double> inputs = new ArrayList<Double>();
while (in.hasNextDouble())
{
    inputs.add(in.nextDouble());
}
```

### 7.7.7 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:

```java
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.

We should not increment the index when removing a word. The appropriate pseudocode is

```java
If the element at index i matches the condition
    Remove the element.
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:

```java
int i = 0;
while (i < words.size())
{
    String word = words.get(i);
    if (word.length() < 4)
    {
        words.remove(i);
    }
    else
    {
        i++;
    }
}
```

### 7.7.8 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 stored in an array list. Note how the program is an improvement over the array version on page 325. This program can process input sequences of arbitrary length.
### Table 3 Comparing Array and Array List Operations

<table>
<thead>
<tr>
<th>Operation</th>
<th>Arrays</th>
<th>Array Lists</th>
</tr>
</thead>
<tbody>
<tr>
<td>Get an element.</td>
<td><code>x = values[4];</code></td>
<td><code>x = values.get(4);</code></td>
</tr>
<tr>
<td>Replace an element.</td>
<td><code>values[4] = 35;</code></td>
<td><code>values.set(4, 35);</code></td>
</tr>
<tr>
<td>Number of elements.</td>
<td><code>values.length</code></td>
<td><code>values.size()</code></td>
</tr>
<tr>
<td>Number of filled elements.</td>
<td><code>currentSize</code> (companion variable, see Section 7.1.4)</td>
<td><code>values.size()</code></td>
</tr>
<tr>
<td>Remove an element.</td>
<td>See Section 7.3.6.</td>
<td><code>values.remove(4);</code></td>
</tr>
<tr>
<td>Add an element, growing the collection.</td>
<td>See Section 7.3.7.</td>
<td><code>values.add(35);</code></td>
</tr>
<tr>
<td>Initializing a collection.</td>
<td><code>int[] values = { 1, 4, 9 };</code></td>
<td>No initializer list syntax; call add three times.</td>
</tr>
</tbody>
</table>

**section_7/LargestInArrayList.java**

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

/**
   This program reads a sequence of values and prints them, marking the largest value.
*/
public class LargestInArrayList
{
    public static void main(String[] args)
    {
        ArrayList<Double> values = new ArrayList<Double>();

        // Read inputs
        System.out.println("Please enter values, Q to quit:");
        Scanner in = new Scanner(System.in);
        while (in.hasNextDouble())
        {
            values.add(in.nextDouble());
        }

        // Find the largest value
        double largest = values.get(0);
        for (int i = 1; i < values.size(); i++)
        {
            if (values.get(i) > largest)
            {
                largest = values.get(i);
            }
        }

        // Print all values, marking the largest
        for (double element : values)
        {
```
37. System.out.print(element);
38. if (element == largest)
39. {
40.     System.out.print(" <= largest value");
41. }
42. System.out.println();
43. }
44. }

Program Run

Please enter values, Q to quit:
35 80 115 44.5 Q
35
80
115 <= largest value
44.5

35. Declare an array list of integers called primes that contains the first five prime numbers (2, 3, 5, 7, and 11).

36. Given the array list primes declared in Self Check 35, write a loop to print its elements in reverse order, starting with the last element.

37. 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");

38. What is wrong with this code snippet?
   ArrayList<String> names;
   names.add(Bob);

39. Consider this method that appends the elements of one array list to another:
   public 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);

40. Suppose you want to store the names of the weekdays. Should you use an array list or an array of seven strings?

41. The ch07/section_7 directory of your source code contains an alternate implementation of the problem solution in How To 7.1 on page 330. 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: R7.14, R7.33, E7.17, E7.20.

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. It is a common error to confuse these. You just have to remember the correct syntax for every data type.

<table>
<thead>
<tr>
<th>Data Type</th>
<th>Number of Elements</th>
</tr>
</thead>
<tbody>
<tr>
<td>Array</td>
<td>a.length</td>
</tr>
<tr>
<td>Array list</td>
<td>a.size()</td>
</tr>
<tr>
<td>String</td>
<td>a.length()</td>
</tr>
</tbody>
</table>

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

```java
ArrayList<String> names = new ArrayList<>();
```

instead of

```java
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.

7.8 Regression Testing

It is a common and useful practice to make a new test whenever you find a program bug. You can use that test to verify that your bug fix really works. Don’t throw the test away; feed it to the next version after that and all subsequent versions. Such a collection of test cases is called a test suite.

You will be surprised how often a bug that you fixed will reappear in a future version. This is a phenomenon known as cycling. Sometimes you don’t quite understand the reason for a bug and apply a quick fix that appears to work. Later, you apply a different quick fix that solves a second problem but makes the first problem appear again. Of course, it is always best to think through what really causes a bug and fix the root cause instead of doing a sequence of “Band-Aid” solutions. If you don’t succeed in doing that, however, you at least want to have an honest appraisal of how well the program works. By keeping all old test cases around and testing them against every new version, you get that feedback. The process of checking each version of a program against a test suite is called regression testing.

How do you organize a suite of tests? An easy technique is to produce multiple tester classes, such as ScoreTester1, ScoreTester2, and so on, where each program runs with a separate set of test data. For example, here is a tester for the Student class:

```java
public class ScoreTester1
{
```
Regression testing involves repeating previously run tests to ensure that known failures of prior versions do not appear in new versions of the software.

Another useful approach is to provide a generic tester, and feed it inputs from multiple files, as in the following.

**section_8/ScoreTester.java**

```java
import java.util.Scanner;

public class ScoreTester
{
    public static void main(String[] args)
    {
        Scanner in = new Scanner(System.in);
        double expected = in.nextDouble();
        Student fred = new Student(100);
        while (in.hasNextDouble())
        {
            if (!fred.addScore(in.nextDouble()))
            {
                System.out.println("Too many scores.");
                return;
            }
        }
        System.out.println("Final score: " + fred.finalScore());
        System.out.println("Expected: " + expected);
    }
}
```

The program reads the expected result and the scores. By running the program with different inputs, we can test different scenarios.

Of course, it would be tedious to type in the input values by hand every time the test is executed. It is much better to save the inputs in a file, such as the following:

**section_8/input1.txt**

```
30
10
20
5
```

When running the program from a shell window, one can link the input file to the input of a program, as if all the characters in the file had actually been typed by a user. Type the following command into a shell window:

```
java ScoreTester < input1.txt
```

The program is executed, but it no longer reads input from the keyboard. Instead, the `System.in` object (and the `Scanner` that reads from `System.in`) gets the input from the file `input1.txt`. We discussed this process, called input redirection, in Special Topic 6.2.
The output is still displayed in the console window:

**Program Run**

<table>
<thead>
<tr>
<th>Final score: 30</th>
</tr>
</thead>
<tbody>
<tr>
<td>Expected: 30</td>
</tr>
</tbody>
</table>

You can also redirect output. To capture the program’s output in a file, use the command

```
java ScoreTester < input1.txt > output1.txt
```

This is useful for archiving test cases.

42. Suppose you modified the code for a method. Why do you want to repeat tests that already passed with the previous version of the code?

43. Suppose a customer of your program finds an error. What action should you take beyond fixing the error?

44. Why doesn’t the ScoreTester program contain prompts for the inputs?

Practice It

Now you can try these exercises at the end of the chapter: R7.35, R7.36.

### Batch Files and Shell Scripts

If you need to perform the same tasks repeatedly on the command line, then it is worth learning about the automation features offered by your operating system.

Under Windows, you use batch files to execute a number of commands automatically. For example, suppose you need to test a program by running three testers:

```
java ScoreTester1
java ScoreTester < input1.txt
java ScoreTester < input2.txt
```

Then you find a bug, fix it, and run the tests again. Now you need to type the three commands once more. There has to be a better way. Under Windows, put the commands in a text file and call it `test.bat`:

**File test.bat**

| 1 | java ScoreTester1 |
| 2 | java ScoreTester < input1.txt |
| 3 | java ScoreTester < input2.txt |

Then you just type

```
test.bat
```

and the three commands in the batch file execute automatically.

Batch files are a feature of the operating system, not of Java. On Linux, Mac OS, and UNIX, shell scripts are used for the same purpose. In this simple example, you can execute the commands by typing

```
sh test.bat
```

There are many uses for batch files and shell scripts, and it is well worth it to learn more about their advanced features, such as parameters and loops.
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 \( array[i] \).
- 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.

The Therac-25 is a computerized device to deliver radiation treatment to cancer patients (see the figure). Between June 1985 and January 1987, several of these machines delivered serious overdoses to at least six patients, killing some of them and seriously maiming the others.

The machines were controlled by a computer program. Bugs in the program were directly responsible for the overdoses. According to Leveson and Turner ("An Investigation of the Therac-25 Accidents," IEEE Computer, July 1993, pp. 18–41), the program was written by a single programmer, who had since left the manufacturing company producing the device and could not be located. None of the company employees interviewed could say anything about the educational level or qualifications of the programmer.

The investigation by the federal Food and Drug Administration (FDA) found that the program was poorly documented and that there was neither a specification document nor a formal test plan. (This should make you think. Do you have a formal test plan for your programs?)

The overdoses were caused by the amateurish design of the software that had to control different devices concurrently, namely the keyboard, the display, the printer, and of course the radiation device itself. Synchronization and data sharing between the tasks were done in an ad hoc way, even though safe multitasking techniques were known at the time. Had the programmer enjoyed a formal education that involved these techniques, or taken the effort to study the literature, a safer machine could have been built. Such a machine would have probably involved a commercial multitasking system, which might have required a more expensive computer.

The same flaws were present in the software controlling the predecessor model, the Therac-20, but that machine had hardware interlocks that mechanically prevented overdoses. The hardware safety devices were removed in the Therac-25 and replaced by checks in the software, presumably to save cost.

Frank Houston of the FDA wrote in 1985, “A significant amount of software for life-critical systems comes from small firms, especially in the medical device industry; firms that fit the profile of those resistant to or uninformed of the principles of either system safety or software engineering.”

Who is to blame? The programmer? The manager who not only failed to ensure that the programmer was up to the task but also didn’t insist on comprehensive testing? The hospitals that installed the device, or the FDA, for not reviewing the design process? Unfortunately, even today there are no firm standards for what constitutes a safe software design process.
• A bounds error, which occurs if you supply an invalid array index, can cause your program to terminate.
• Use the expression `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.
• Arrays can occur as method arguments and return values.
• With a partially filled array, keep a companion variable for the current size.
• Avoid parallel arrays by changing them into arrays of objects.

**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.

**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.

Describe the process of regression testing.

• A test suite is a set of tests for repeated testing.
• Regression testing involves repeating previously run tests to ensure that known failures of prior versions do not appear in new versions of the software.

STANDARD LIBRARY ITEMS INTRODUCED IN THIS CHAPTER

- `java.lang.Boolean`
- `java.lang.Double`
- `java.lang.Integer`
- `java.util.Arrays`
- `java.util.ArrayList<E>`
- `add`
- `get`
- `remove`
- `set`
- `size`

REVIEW EXERCISES

**R7.1** Carry out the following tasks with an array:

a. Allocate an array `a` of ten integers.

b. Put the number 17 as the initial element of the array.

c. Put the number 29 as the last element of the array.

d. Fill the remaining elements with –1.

e. Add 1 to each element of the array.

f. Print all elements of the array, one per line.

**R7.2** What is an index of an array? What are the legal index values? What is a bounds error?

**R7.3** Write a program that contains a bounds error. Run the program. What happens on your computer?

**R7.4** Write a loop that reads ten numbers and a second loop that displays them in the opposite order from which they were entered.

**R7.5** Write code that fills an array `values` with each set of numbers below.

<table>
<thead>
<tr>
<th>a</th>
<th>1 2 3 4 5 6 7 8 9 10</th>
</tr>
</thead>
<tbody>
<tr>
<td>b</td>
<td>0 2 4 6 8 10 12 14 16 18 20</td>
</tr>
<tr>
<td>c</td>
<td>1 4 9 16 25 36 49 64 81 100</td>
</tr>
<tr>
<td>d</td>
<td>0 0 0 0 0 0 0 0 0 0</td>
</tr>
<tr>
<td>e</td>
<td>1 4 9 16 9 7 4 9 11</td>
</tr>
<tr>
<td>f</td>
<td>0 1 0 1 0 1 0 1 0 1</td>
</tr>
<tr>
<td>g</td>
<td>0 1 2 3 4 0 1 2 3 4</td>
</tr>
</tbody>
</table>
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; }
```

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]; }
```

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.

Write Java code for a loop that simultaneously computes both the maximum and minimum of an array.

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;
}
```
■ R7.11 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 maximum of all elements in an array.
  c. Counting how many elements in an array are negative.

■ R7.12 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++; }

■ R7.13 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; }

■ R7.14 What is wrong with each of the following code segments?
  a. ArrayList<int> values = new ArrayList<int>();
  b. ArrayList<Integer> values = new ArrayList();
  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); }

■ R7.15 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.

■ R7.16 Trace the flow of the loop in Section 7.3.4 with the given example. Show two columns, one with the value of i and one with the output.

■ R7.17 Consider the following loop for collecting all elements that match a condition; in this case, that the element is larger than 100.

```java
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.

**R7.18** Trace the flow of the loop in Section 7.3.5, where values contains the elements 80 90 100 120 110. Show two columns, for pos and found. Repeat the trace when values contains the elements 80 90 120 70.

**R7.19** Trace the algorithm for removing an element described in Section 7.3.6. Use an array values with elements 110 90 100 120 80, and remove the element at index 2.

**R7.20** 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, as shown at right.

**R7.21** Give pseudocode for an algorithm that removes all negative values from an array, preserving the order of the remaining elements.

**R7.22** Suppose values is a sorted array of integers. Give pseudocode that describes how a new value can be inserted so that the resulting array stays sorted.

**R7.23** 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 3 6 5 5 6 3 1

has length 4.

**R7.24** What is wrong with the following method that aims to fill an array with random numbers?

```java
public void makeCombination(int[] values, int n) {
    Random generator = new Random();
    int[] numbers = new int[values.length];
    for (int i = 0; i < numbers.length; i++) {
        numbers[i] = generator.nextInt(n);
    }
    values = numbers;
}
```

**R7.25** You are given two arrays denoting x- and y-coordinates of a set of points in a 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 7.3?

**R7.26** Solve the quiz score problem described in Section 7.4 by sorting the array first. How do you need to modify the algorithm for computing the total?

**R7.27** Solve the task described in Section 7.5 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 7.5.

**R7.28** 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.

**R7.29** Write Java statements for performing the following tasks with an array declared as

```java
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 in the top and bottom rows with zeroes.
- Compute the sum of all elements.
- Print the array in tabular form.

**R7.30** Write pseudocode for an algorithm that fills the first and last columns as well as the first and last rows of a two-dimensional array of integers with –1.

**R7.31** Section 7.7.7 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.

**R7.32** True or false?

a. All elements of an array are of the same type.

b. Arrays cannot contain strings as elements.

c. Two-dimensional arrays always have the same number of rows and columns.

d. Elements of different columns in a two-dimensional array can have different types.

e. A method cannot return a two-dimensional array.

f. A method cannot change the length of an array argument.

g. A method cannot change the number of columns of an argument that is a two-dimensional array.

**R7.33** How do you perform the following tasks with array lists in Java?

a. Test that two array lists contain the same elements in the same order.

b. Copy one array list to another.

c. Fill an array list with zeroes, overwriting all elements in it.

d. Remove all elements from an array list.

**R7.34** True or false?

a. All elements of an array list are of the same type.

b. Array list index values must be integers.

c. Array lists cannot contain strings as elements.

d. Array lists can change their size, getting larger or smaller.

e. A method cannot return an array list.

f. A method cannot change the size of an array list argument.
**Testing R7.35** Define the terms regression testing and test suite.

**Testing R7.36** What is the debugging phenomenon known as *cycling*? What can you do to avoid it?

---

### PRACTICE EXERCISES

**E7.1** 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.

**E7.2** Modify the `LargestInArray.java` program in Section 7.3 to mark both the smallest and the largest elements.

**E7.3** 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.

**E7.4** Add a method `removeMin` to the `Student` class of Section 7.4 that removes the minimum score without calling other methods.

**E7.5** 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
```

**E7.6** 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
```

**E7.7** 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, 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 element from the second array.**
  - **Remove it and append it to the permutation array.**

**E7.8** Write a method that implements the algorithm developed in Section 7.5.

**E7.9** Write a class `DataSet` that stores a number of values of type `double`. Provide a constructor

```java
public DataSet(int maximumNumberOfValues)
```

and a method

```java
public void add(double value)
```
that adds a value, provided there is still room.
Provide methods to compute the sum, average, maximum, and minimum value.

**E7.10** Write array methods that carry out the following tasks for an array of integers by completing the `ArrayMethods` class below. For each method, provide a test program.

```java
public class ArrayMethods {
    private int[] values;
    public ArrayMethods(int[] initialValues) { values = initialValues; }
    public void swapFirstAndLast() { . . . }
    public void shiftRight() { . . . }
    . . .
}
```

- **a.** Swap the first and last elements in the array.
- **b.** Shift all elements to the right by one 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.
- **c.** Replace all even elements with 0.
- **d.** 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.
- **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).

**E7.11** Consider the following class:

```java
public class Sequence {
    private int[] values;
    public Sequence(int size) { values = new int[size]; }
    public void set(int i, int n) { values[i] = n; }
    public int get(int i) { return values[i]; }
    public int size() { return values.length; }
}
```

Add a method

```java
public boolean equals(Sequence other)
```

that checks whether two sequences have the same values in the same order.

**E7.12** Add a method

```java
public boolean sameValues(Sequence other)
```
to the `Sequence` class of Exercise E7.11 that checks whether two sequences have the same values in some order, ignoring duplicates. For example, the two sequences

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.
E7.13 Add a method

```java
public boolean isPermutationOf(Sequence other)
```

to the Sequence class of Exercise E7.11 that checks whether two sequences have the same values in some order, with the same multiplicities. For example,

```
1 4 9 16 9 7 4 9 11
```

is a permutation of

```
11 1 4 9 16 9 7 4 9
```

but

```
1 4 9 16 9 7 4 9 11
```

is not a permutation of

```
11 11 7 9 16 4 1 4 9
```

You will probably need one or more helper methods.

E7.14 Add a method

```java
public Sequence sum(Sequence other)
```

to the Sequence class of Exercise E7.11 that yields the sum of this sequence and another. If the sequences don’t have the same length, assume that the missing elements are zero. For example, the sum of

```
1 4 9 16 9 7 4 9 11
```

and

```
11 11 7 9 16 4 1
```

is the sequence

```
12 15 16 25 25 11 5 9 11
```

E7.15 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.

E7.16 Add a method to the Table class below that computes the average of the neighbors of a table element in the eight directions shown in Figure 15:

```java
public double neighborAverage(int row, int column)
```

However, if the element is located at the boundary of the array, include only the neighbors that are in the table. For example, if row and column are both 0, there are only three neighbors.

```java
public class Table
{
    private int[][] values;
    public Table(int rows, int columns) { values = new int[rows][columns]; }
    public void set(int i, int j, int n) { values[i][j] = n; }
}
```

E7.17 Given the Table class of Exercise E7.16, add a method that returns the sum of the i-th row (if horizontal is true) or column (if horizontal is false):

```java
public double sum(int i, boolean horizontal)
```

E7.18 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.

**E7.19** Repeat Exercise E7.17, but make the bars vertical, with the tallest bar twenty asterisks high.

```plaintext
* * * * * * **  *** ****************************************************
```

**E7.20** Improve the program of Exercise E7.17 to work correctly when the data set contains negative values.

**E7.21** Improve the program of Exercise E7.17 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 **************

**E7.22** Consider the following class:

```java
public class Sequence {
    private ArrayList<Integer> values;
    public Sequence() { values = new ArrayList<Integer>(); }
    public void add(int n) { values.add(n); }
    public String toString() { return values.toString(); }
}
```

Add a method

```java
public Sequence append(Sequence other)
```

that creates a new sequence, appending this and the other sequence, without modifying either sequence. For example, if a is

```
1 4 9 16
```

and b is the sequence

```
9 7 4 9 11
```

then the call a.append(b) returns the sequence

```
1 4 9 16 9 7 4 9 11
```

without modifying a or b.

**E7.23** Add a method

```java
public Sequence merge(Sequence other)
```

to the Sequence class of Exercise E7.21 that merges two sequences, alternating elements from both sequences. If one sequence is shorter than the other, then alternate
as long as you can and then append the remaining elements from the longer sequence. For example, if \( a \) is
\[
1 \ 4 \ 9 \ 16
\]
and \( b \) is
\[
9 \ 7 \ 4 \ 9 \ 11
\]
then \( a.merge(b) \) returns the sequence
\[
1 \ 9 \ 4 \ 7 \ 9 \ 4 \ 16 \ 9 \ 11
\]
without modifying \( a \) or \( b \).

\[** E7.24\] Add a method
\[
public \ \text{Sequence} \ \text{mergeSorted} (\text{Sequence} \ \text{other})
\]
to the \text{Sequence} class of Exercise E7.21 that merges two sorted sequences, producing a new sorted sequence. Keep an index into each sequence, indicating how much of it has been processed already. Each time, append the smallest unprocessed value from either sequence, then advance the index. For example, if \( a \) is
\[
1 \ 4 \ 9 \ 16
\]
and \( b \) is
\[
4 \ 7 \ 9 \ 9 \ 11
\]
then \( a.mergeSorted(b) \) returns the sequence
\[
1 \ 4 \ 4 \ 7 \ 9 \ 9 \ 9 \ 11 \ 16
\]
If \( a \) or \( b \) is not sorted, merge the longest prefixes of \( a \) and \( b \) that are sorted.

\[** P7.1\] 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:

Set a boolean variable \text{inRun} \ to \text{false}.
For each valid index \text{i} in the array
If \text{inRun}
    If \text{values[i]} \ is different from the preceding value
        Print \text{)}.\n        \text{inRun} = \text{false}.
    If \text{not \text{inRun}}
        If \text{values[i]} \ is the same as the following value
            Print \text{)}.\n            \text{inRun} = \text{true}.
            \text{Print \text{values[i]}}.\n            If \text{inRun}, print \text{)}.\.

\[** P7.2\] 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.
### P7.3
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.

\[
\begin{array}{cccccccccc}
  & & & & & & & & & \\
\end{array}
\]

The first visitor will occupy a middle position:

\[
\begin{array}{cccccccccc}
  & & & & & & & & & X \\
\end{array}
\]

The next visitor will be in the middle of the empty area at the left.

\[
\begin{array}{cccccccccc}
  & & & X & & & & & X & \\
\end{array}
\]

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.

### P7.4
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.

### P7.5
*Magic squares.* An \(n \times n\) matrix that is filled with the numbers \(1, 2, 3, \ldots, n^2\) 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:

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?

### P7.6
Implement the following algorithm to construct magic \(n \times n\) squares; it works only if \(n\) is odd.

\[
\begin{align*}
  \text{Set} & \quad \text{row} = n - 1, \text{column} = n / 2. \\
  \text{For} & \quad k = 1 \ldots n \times n \\
  \text{Place} & \quad k \text{ at } [\text{row}][\text{column}]. \\
  \text{Increment} & \quad \text{row and column.} \\
  \text{If} & \quad \text{the row or column is } n, \text{ replace it with } 0. \\
  \text{If} & \quad \text{the element at } [\text{row}][\text{column}] \text{ has already been filled} \\
  \text{Set} & \quad \text{row and column to their previous values.} \\
  \text{Decrement} & \quad \text{row.}
\end{align*}
\]

Here is the \(5 \times 5\) square that you get if you follow this method:

Write a program whose input is the number \(n\) and whose output is the magic square of order \(n\) if \(n\) is odd.
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
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 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.

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.

In this assignment, you will implement a simulation of a popular casino game usually called video poker. The card deck contains 52 cards, 13 of each suit. At the beginning of the game, the deck is shuffled. You need to devise a fair method for shuffling. (It does not have to be efficient.) The player pays a token for each game. Then the top five cards of the deck are presented to the player. The player can reject none, some, or all of the cards. The rejected cards are replaced from the top of the deck. Now the hand is scored. Your program should pronounce it to be one of the following:

- No pair—The lowest hand, containing five separate cards that do not match up to create any of the hands below.
- One pair—Two cards of the same value, for example two queens. Payout: 1
- Two pairs—Two pairs, for example two queens and two 5’s. Payout: 2
- Three of a kind—Three cards of the same value, for example three queens. Payout: 3
- Straight—Five cards with consecutive values, not necessarily of the same suit, such as 4, 5, 6, 7, and 8. The ace can either precede a 2 or follow a king. Payout: 4
- Flush—Five cards, not necessarily in order, of the same suit. Payout: 5
- Full House—Three of a kind and a pair, for example three queens and two 5’s. Payout: 6
- Four of a Kind—Four cards of the same value, such as four queens. Payout: 25
- Straight Flush—A straight and a flush: Five cards with consecutive values of the same suit. Payout: 50
- Royal Flush—The best possible hand in poker. A 10, jack, queen, king, and ace, all of the same suit. Payout: 250
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 21 shows a cell and its neighbor cells.

Many configurations show interesting behavior when subjected to these rules. Figure 22 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.

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 23).

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.
Business P7.11  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.

Use a class `Item` to describe an item, with any needed methods and a constructor

```
public Item(double price, boolean isPet, int quantity)
```

An invoice holds a collection of `Item` objects; use an array or array list to store them.

In the `Invoice` class, implement methods

```
public void add(Item anItem)
public double getDiscount()
```

Write a program that prompts a cashier to enter each price and quantity, and then a Y for a pet or N for another item. Use a price of –1 as a sentinel. In the loop, call the `add` method; after the loop, call the `getDiscount` method and display the returned value.

Business P7.12  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 store keeps an `ArrayList<Customer>`.

In the `Store` class, implement methods

```
public void addSale(String customerName, double amount)
public String nameOfBestCustomer()
```

to record the sale and return 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 a `Store` object, and displays the best customer’s name. Use a price of 0 as a sentinel.

Business P7.13  Improve the program of Exercise P7.12 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 ArrayList<String> nameOfBestCustomers(int topN)
```

If there were fewer than `topN` customers, include all of them.

Science P7.14  Sounds can be represented by an array of “sample values” that describe the intensity of the sound at a point in time. The program in ch07/sound of your companion code reads a sound file (in WAV format), processes the sample values, and shows the result. Your task is to process the sound 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.

Science P7.15  You are given a two-dimensional array of values that give the height of a terrain at different points in a square. Write a constructor

```
public Terrain(double[][] heights)
```

and a method

```
public void printFloodMap(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:

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

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 P7.16** 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). Given a class Data with instance fields

```java
private double[] values;
private double valuesSize;
```

implement a method

```java
public void smooth()
```

that carries out this operation. You should not create another array in your solution.

---

**Science P7.17** 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, as shown here.

---

**Graphics P7.18** Generate the image of a checkerboard.
Chapter 7  Arrays and Array Lists

- **Graphics P7.19** Generate the image of a sine wave. Draw a line of pixels for every five degrees.

- **Graphics P7.20** Implement a class `Cloud` that contains an array list of `Point2D.Double` objects. Support methods
  
  ```java
  public void add(Point2D.Double aPoint)
  public void draw(Graphics2D g2)
  ```

  Draw each point as a tiny circle. Write a graphical application that draws a cloud of 100 random points.

- **Graphics P7.21** Implement a class `Polygon` that contains an array list of `Point2D.Double` objects. Support methods
  
  ```java
  public void add(Point2D.Double aPoint)
  public void draw(Graphics2D g2)
  ```

  Draw the polygon by joining adjacent points with a line, and then closing it up by joining the end and start points. Write a graphical application that draws a square and a pentagon using two `Polygon` objects.

- **Graphics P7.22** Write a class `Chart` with methods
  
  ```java
  public void add(int value)
  public void draw(Graphics2D g2)
  ```

  that displays a stick chart of the added values, like this: You may assume that the values are pixel positions.

- **Graphics P7.23** Write a class `BarChart` with methods
  
  ```java
  public void add(double value)
  public void draw(Graphics2D g2)
  ```

  that displays a bar chart of the added values. You may assume that all added values are positive. Stretch the bars so that they fill the entire area of the screen. You must figure out the maximum of the values, then scale each bar.

- **Graphics P7.24** Improve the `BarChart` class of Exercise P7.23 to work correctly when the data contains negative values.

- **Graphics P7.25** Write a class `PieChart` with methods
  
  ```java
  public void add(double value)
  public void draw(Graphics2D g2)
  ```

  that displays a pie chart of the added values. Assume that all data values are positive.
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. public class Lottery
   {
      public int[] getCombination(int n) { . . . }
   }
9. It counts how many elements of values are zero.
10. for (double x : values)
    { System.out.println(x); }
11. double product = 1;
    for (double f : factors)
    { product = product * f; }
12. The loop writes a value into values[i]. The enhanced for loop does not have the index variable i.
13. 20 <= largest value
    10
    20 <= largest value
14. int count = 0;
    for (double x : values)
    { if (x == 0) { count++; } }
15. If all elements of values are negative, then the result is incorrectly computed as 0.
16. for (int i = 0; i < values.length; i++)
    { System.out.println(values[i]);
      if (i < values.length - 1)
      { System.out.print(" | ");
    }
17. If the array has no elements, then the program terminates with an exception.
18. If there is a match, then pos is incremented before the loop exits.
19. This loop sets all elements to values[pos].
20. Use the first algorithm. The order of elements does not matter when computing the sum.
21. Find the minimum value.
    Calculate the sum.
    Subtract the minimum value.
22. Use the algorithm for counting matches (Section 6.7.2) twice, once for counting the positive values and once for counting the negative values.
23. You need to modify the algorithm in Section 7.3.4.
    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.
24. Use the algorithm to collect all positive elements in an array, then use the algorithm in Section 7.3.4 to print the array of matches.
25. The paperclip for i assumes positions 0, 1, 2, 3. When i is incremented to 4, the condition i < 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.

(coins) © jamesbenet/iStockphoto; (dollar coins) JordiDelgado/iStockphoto; (paperclip) © Yvan Dube/iStockphoto.
26. It reverses the elements in the array.

27. 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:

\[
i = 0 \\
j = \text{size} - 1 \\
\text{While } (i < j) \\
\quad \text{If } (a[i] \text{ is odd}) \\
\quad \quad \text{Swap elements at positions } i \text{ and } j. \\
\quad \quad j-- \\
\quad \text{Else} \\
\quad \quad i++
\]

28. 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.

\[
i = 0 \\
moved = \text{size} \\
\text{While } (i < moved) \\
\quad \text{If } (a[i] \text{ is odd}) \\
\quad \quad \text{Remove the element at position } i \text{ and add it at the end.} \\
\quad \quad moved--
\]

29. 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.

30. You get the total number of gold, silver, and bronze medals in the competition. In our example, there are four of each.

31. for (int i = 0; i < 8; i++)
{ 
    for (int j = 0; j < 8; j++)
    { 
        board[i][j] = (i + j) % 2;
    }
}

32. String[][] board = new String[3][3];
33. board[0][2] = "x";
34. board[0][0], board[1][1], board[2][2]

35. ArrayList<Integer> primes = 
    new ArrayList<Integer>();
    primes.add(2);
    primes.add(3);
    primes.add(5);
    primes.add(7);
    primes.add(11);

36. for (int i = primes.size() - 1; i >= 0; i--)
{ 
    System.out.println(primes.get(i));
}

37. "Ann", "Cal"

38. The names variable has not been initialized.

39. names1 contains "Emily", "Bob", "Cindy", "Dave"; names2 contains "Dave"

40. 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" };

41. Reading inputs into an array list is much easier.

42. It is possible to introduce errors when modifying code.

43. Add a test case to the test suite that verifies that the error is fixed.

44. There is no human user who would see the prompts because input is provided from a file.
Step 1  Decompose your task into steps.

Our first try at decomposition simply echoes the problem statement:

- Read the die values.
- Count how often the values 1, 2, ..., 6 appear.
- 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
  - Increment the corresponding counter.
  - 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 create the array as

    counters = new int[sides + 1];

Why introduce a sides variable? Suppose you later changed your mind and wanted to investigate 12-sided dice:

Then the program can simply be changed by setting sides to 12.

**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.
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:

```java
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:

```java
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:

```java
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:
- void countInputs()
- void printCounters()

The `main` method calls these methods:

```java
public class DiceAnalyzer
{
    public static void main(String[] args)
    {
        final int SIDES = 6;
        Dice dice = new Dice(SIDES);
        dice.countInputs();
        dice.printCounters();
    }
}
```

The `countInputs` method reads all inputs and increments the matching 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.

<table>
<thead>
<tr>
<th>Test Case</th>
<th>Expected Output</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 2 3 4 5 6</td>
<td>1 1 1 1 1</td>
<td>Each number occurs once.</td>
</tr>
<tr>
<td>1 2 3</td>
<td>1 1 0 0 0</td>
<td>Numbers that don’t appear should have counts of zero.</td>
</tr>
<tr>
<td>1 2 3 1 2 3 4</td>
<td>2 2 2 1 0 0</td>
<td>The counters should reflect how often each input occurs.</td>
</tr>
<tr>
<td>(No input)</td>
<td>0 0 0 0 0</td>
<td>This is a legal input; all counters are zero.</td>
</tr>
<tr>
<td>0 1 2 3 4 5 6 7</td>
<td>Error</td>
<td>Each input should be between 1 and 6.</td>
</tr>
</tbody>
</table>

Here's the complete program:

```java
import java.util.Scanner;

/**
* This program reads a sequence of die toss values and prints how many times each value occurred.
*/
public class Dice {
    private int[] counters;

    public Dice(int sides) {
        counters = new int[SIDES + 1]; // counters[0] is not used
    }

    public void countInputs() {
        System.out.println("Please enter values, Q to quit: ");
        Scanner in = new Scanner(System.in);
        while (in.hasNextInt()) {
            int value = in.nextInt();
            // Increment the counter for the input value
            if (1 <= value && value <= counters.length) {
                counters[value]++;
            } else {
                System.out.println(value + " is not a valid input.");
            }
        }
    }
}
```
public void printCounters()
{
    for (int i = 1; i < counters.length; i++)
    {
        System.out.printf("%2d: %4d\n", i, counters[i]);
    }
}

public class DiceAnalyzer
{
    public static void main(String[] args)
    {
        final int SIDES = 6;
        Dice dice = new Dice(SIDES);
        dice.countInputs();
        dice.printCounters();
    }
}

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
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 population in the given years.

<table>
<thead>
<tr>
<th>Population Per Continent (in millions)</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Year</strong></td>
</tr>
<tr>
<td>----------</td>
</tr>
<tr>
<td>Africa</td>
</tr>
<tr>
<td>Asia</td>
</tr>
<tr>
<td>Australia</td>
</tr>
<tr>
<td>Europe</td>
</tr>
<tr>
<td>North America</td>
</tr>
<tr>
<td>South America</td>
</tr>
</tbody>
</table>

**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:

```java
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.

```java
String[] continents = {
    "Africa",
    "Asia",
    "Australia",
    "Europe",
    "North America",
    "South America"
};
```
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 7.6.4. 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",
```
"North America",
"South America"};

System.out.println("Year 1750 1800 1850 1900 1950 2000 2050");

// Print population data
for (int i = 0; i < ROWS; i++)
{
    // Print the ith row
    System.out.printf("%20s", continents[i]);
    for (int j = 0; j < COLUMNS; j++)
    {
        System.out.printf("%5d", populations[i][j]);
    }
    System.out.println(); // Start a new line at the end of the row
}

// Print column totals
System.out.print("    World");
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);
}
System.out.println();

---

**Program Run**

<table>
<thead>
<tr>
<th></th>
<th>1750</th>
<th>1800</th>
<th>1850</th>
<th>1900</th>
<th>1950</th>
<th>2000</th>
<th>2050</th>
</tr>
</thead>
<tbody>
<tr>
<td>Africa</td>
<td>106</td>
<td>107</td>
<td>111</td>
<td>133</td>
<td>221</td>
<td>767</td>
<td>1766</td>
</tr>
<tr>
<td>Asia</td>
<td>502</td>
<td>635</td>
<td>809</td>
<td>947</td>
<td>1402</td>
<td>3634</td>
<td>5268</td>
</tr>
<tr>
<td>Australia</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>6</td>
<td>13</td>
<td>30</td>
<td>46</td>
</tr>
<tr>
<td>Europe</td>
<td>163</td>
<td>203</td>
<td>276</td>
<td>408</td>
<td>547</td>
<td>729</td>
<td>628</td>
</tr>
<tr>
<td>North America</td>
<td>2</td>
<td>7</td>
<td>26</td>
<td>82</td>
<td>172</td>
<td>307</td>
<td>392</td>
</tr>
<tr>
<td>South America</td>
<td>16</td>
<td>24</td>
<td>38</td>
<td>74</td>
<td>167</td>
<td>511</td>
<td>809</td>
</tr>
<tr>
<td>World</td>
<td>791</td>
<td>978</td>
<td>1262</td>
<td>1650</td>
<td>2522</td>
<td>5978</td>
<td>8909</td>
</tr>
</tbody>
</table>
CHAPTER GOALS

To learn how to choose appropriate classes for a given problem
To understand the concept of cohesion
To minimize dependencies and side effects
To learn how to find a data representation for a class
To understand static methods and variables
To learn about packages
To learn about unit testing frameworks

CHAPTER CONTENTS

8.1 DISCOVERING CLASSES 376
8.2 DESIGNING GOOD METHODS 377
   PT1 Consistency 381
   ST1 Call by Value and Call by Reference 382
8.3 PROBLEM SOLVING: PATTERNS FOR OBJECT DATA 386
8.4 STATIC VARIABLES AND METHODS 391
   PT2 Minimize the Use of Static Methods 393
   CE1 Trying to Access Instance Variables in Static Methods 394
   ST2 Alternative Forms of Instance and Static Variable Initialization 394
   ST3 Static Imports 395
8.5 PROBLEM SOLVING: SOLVE A SIMPLER PROBLEM FIRST 395
8.6 PACKAGES 400
   SYN Package Specification 401
   CE2 Confusing Dots 403
   ST4 Package Access 403
   HT1 Programming with Packages 404
8.7 UNIT TEST FRAMEWORKS 405
   C&S Personal Computing 407
You have used a good number of classes in the preceding chapters and probably designed a few classes yourself as part of your programming assignments. Designing a class can be a challenge—it is not always easy to tell how to start or whether the result is of good quality.

What makes a good class? Most importantly, a class should represent a single concept from a problem domain. Some of the classes that you have seen represent concepts from mathematics:

- Point
- Rectangle
- Ellipse

Other classes are abstractions of real-life entities:

- BankAccount
- CashRegister

For these classes, the properties of a typical object are easy to understand. A Rectangle object has a width and height. Given a BankAccount object, you can deposit and withdraw money. Generally, concepts from a domain related to the program’s purpose, such as science, business, or gaming, make good classes. The name for such a class should be a noun that describes the concept. In fact, a simple rule of thumb for getting started with class design is to look for nouns in the problem description.

One useful category of classes can be described as actors. Objects of an actor class carry out certain tasks for you. Examples of actors are the Scanner class of Chapter 4 and the Random class in Chapter 6. A Scanner object scans a stream for numbers and strings. A Random object generates random numbers. It is a good idea to choose class names for actors that end in “-er” or “-or”. (A better name for the Random class might be RandomNumberGenerator.)

Very occasionally, a class has no objects, but it contains a collection of related static methods and constants. The Math class is an example. Such a class is called a utility class.

Finally, you have seen classes with only a main method. Their sole purpose is to start a program. From a design perspective, these are somewhat degenerate examples of classes.

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 ask you 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 is to turn a single operation 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.

1. What is a simple 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?

Practice It Now you can try these exercises at the end of the chapter: R8.3, R8.4, R8.5.

8.2 Designing Good Methods

In the following sections, you will learn several useful criteria for analyzing and improving the public interface of a class.

8.2.1 Providing a Cohesive Public Interface

A class should represent a single concept. All interface features should be closely related to the single concept that the class represents. Such a public interface is said to be cohesive.

The public interface of a class is cohesive if all of its features are related to the concept that the class represents.

The members of a cohesive team have a common goal.

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 the CashRegister class in Chapter 4:

```java
public class CashRegister
{
    public static final double QUARTER_VALUE = 0.25;
    public static final double DIME_VALUE = 0.1;
    public static final double NICKEL_VALUE = 0.05;
    . . .
    public void receivePayment(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. Exercise E8.3 discusses a more general solution.)

It makes sense to have a separate Coin class and have coins responsible for knowing their values.

```java
public class Coin {
    ...  
    public Coin(double aValue, String aName) { ... }
    public double getValue() { ... }
    ...  
}
```

Then the CashRegister class can be simplified:

```java
public class CashRegister {
    ...  
    public void receivePayment(int coinCount, Coin coinType) { ... }
    {  
        payment = payment + coinCount * coinType.getValue();
    }  
    ...  
}
```

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. The only reason we didn’t follow this approach in Chapter 4 was to keep the CashRegister example simple.

### 8.2.2 Minimizing Dependencies

Many methods need other classes in order to do their jobs. For example, the receivePayment method of the restructured CashRegister class now uses the Coin class. We say that the CashRegister class depends on the Coin class.

To visualize relationships between classes, such as dependence, 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 by Grady Booch, Ivar Jacobson, and James Rumbaugh, three leading researchers in object-oriented software development. (Appendix H has a summary of the UML notation used in this book.) The UML notation distinguishes between object diagrams and class diagrams. In an object diagram the class names are underlined; in a class diagram the class names are not underlined. In a class diagram, you denote dependency by a dashed line with a ▸-shaped open arrow tip that points to the dependent class. Figure 1 shows a class diagram indicating that the CashRegister class depends on the Coin class.

Note that the Coin class does not depend on the CashRegister class. All Coin methods can carry out their work without ever calling any method in the CashRegister class. Conceptually, coins have no idea that they are being collected in cash registers.

Here is an example of minimizing dependencies. Consider how we have always printed a bank balance:

```java
System.out.println("The balance is now $" + momsSavings.getBalance());
```
Why don’t we simply have a `printBalance` method?

```java
public void printBalance() // Not recommended
{
    System.out.println("The balance is now \$" + balance);
}
```

The method depends on `System.out`. Not every computing environment has `System.out`. For example, an automatic teller machine doesn’t display console messages. In other words, this design violates the rule of minimizing dependencies. The `printBalance` method couples the `BankAccount` class with the `System` and `PrintStream` classes.

It is best to place the code for producing output or consuming input in a separate class. That way, you decouple input/output from the actual work of your classes.

### 8.2.3 Separating Accessors and Mutators

A **mutator method** changes the state of an object. Conversely, an **accessor method** asks an object to compute a result, without changing the state.

Some classes have been designed to have only accessor methods and no mutator methods at all. Such classes are called **immutable**. An example is the `String` class. Once a string has been constructed, its content never changes. No method in the `String` class can modify the contents of a string. For example, the `toUpperCase` method does not change characters from the original string. Instead, it constructs a new string that contains the uppercase characters:

```java
String name = "John Q. Public";
String uppercased = name.toUpperCase(); // name is not changed
```

An immutable class has a major advantage: It is safe to give out references to its objects freely. If no method can change the object’s value, then no code can modify the object at an unexpected time.

Not every class should be immutable. Immutability makes most sense for classes that represent values, such as strings, dates, currency amounts, colors, and so on.

In mutable classes, it is still a good idea to cleanly separate accessors and mutators, in order to avoid accidental mutation. As a rule of thumb, a method that returns a value should not be a mutator. For example, one would not expect that calling `getBalance` on a `BankAccount` object would change the balance. (You would be pretty upset if your bank charged you a “balance inquiry fee”.) If you follow this rule, then all mutators of your class have return type `void`.
Sometimes, this rule is bent a bit, and mutator methods return an informational value. For example, the `ArrayList` class has a `remove` method to remove an object.

```java
ArrayList<String> names = ...;
boolean success = names.remove("Romeo");
```

That method returns `true` if the removal was successful; that is, if the list contained the object. Returning this value might be bad design if there was no other way to check whether an object exists in the list. However, there is such a method—the `contains` method. It is acceptable for a mutator to return a value if there is also an accessor that computes it.

The situation is less happy with the `Scanner` class. The `next` method is a mutator that returns a value. (The `next` method really is a mutator. If you call `next` twice in a row, it can return different results, so it must have mutated something inside the `Scanner` object.) Unfortunately, there is no accessor that returns the same value. This sometimes makes it awkward to use a `Scanner`. You must carefully hang on to the value that the `next` method returns because you have no second chance to ask for it. It would have been better if there was another method, say `peek`, that yields the next input without consuming it.

To check the temperature of the water in the bottle, you could take a sip, but that would be the equivalent of a mutator method.

### 8.2.4 Minimizing Side Effects

A side effect of a method is any kind of modification of data that is observable outside the method. Mutator methods have a side effect, namely the modification of the implicit parameter.

There is another kind of side effect that you should avoid. A method should generally not modify its parameter variables. Consider this example:

```java
/**
 * Computes the total balance of the given accounts.
 * @param accounts a list of bank accounts
 */
public double getTotalBalance(ArrayList<String> accounts) {
    double sum = 0;
    while (studentNames.size() > 0) {
        BankAccount account = accounts.remove(0); // Not recommended
        sum = sum + account.getBalance();
    }
    return sum;
}
```

This method removes all names from the `accounts` parameter variable. After a call

```java
double total = getTotalBalance(allAccounts);
```

`allAccounts` is empty! Such a side effect would not be what most programmers expect. It is better if the method visits the elements from the list without removing them.
Another example of a side effect is output. Consider again the `printBalance` method that we discussed in Section 8.2.2:

```java
public void printBalance() // Not recommended
{
    System.out.println("The balance is now "+ balance);
}
```

This method mutates the `System.out` object, which is not a part of the `BankAccount` object. That is a side effect.

To avoid this side effect, keep most of your classes free from input and output operations, and concentrate input and output in one place, such as the `main` method of your program.

*This taxi has an undesirable side effect, spraying bystanders with muddy water.*

3. Why is the `CashRegister` class from Chapter 4 not cohesive?
4. Why does the `Coin` class not depend on the `CashRegister` class?
5. Why is it a good idea to minimize dependencies between classes?
6. Is the `substring` method of the `String` class an accessor or a mutator?
7. Is the `Rectangle` class immutable?
8. If `a` refers to a bank account, then the call `a.deposit(100)` modifies the bank account object. Is that a side effect?
9. Consider the `Student` class of Chapter 7. Suppose we add a method

```java
void read(Scanner in)
{
    while (in.hasNextDouble())
    {
        addScore(in.nextDouble());
    }
}
```

Does this method have a side effect other than mutating the scores?

### Practice It

Now you can try these exercises at the end of the chapter: R8.6, R8.7, R8.11.

### Consistency

In this section you learned of two criteria for analyzing the quality of the public interface of a class. You should maximize cohesion and remove unnecessary dependencies. There is another criterion that we would like you to pay attention to—**consistency**. When you have a set of methods, follow a consistent scheme for their names and parameter variables. This is simply a sign of good craftsmanship.

Sadly, you can find any number of inconsistencies in the standard library. Here is an example: To show an input dialog box, you call

```java
JOptionPane.showInputDialog(promptString)
```
To show a message dialog box, you call

```java
JOptionPane.showMessageDialog(null, messageString)
```

What’s the `null` argument? It turns out that the `showMessageDialog` method needs an argument to specify the parent window, or `null` if no parent window is required. But the `showInputDialog` method requires no parent window. Why the inconsistency? There is no reason. It would have been an easy matter to supply a `showMessageDialog` method that exactly mirrors the `showInputDialog` method.

Inconsistencies such as these are not fatal flaws, but they are an annoyance, particularly because they can be so easily avoided.

---

**Special Topic 8.1**

**Call by Value and Call by Reference**

In Section 8.2.4, we recommended that you don’t invoke a mutator method on a parameter variable. In this Special Topic, we discuss a related issue—what happens when you assign a new value to a parameter variable. Consider this method:

```java
public class BankAccount {
    
    /**
     * Transfers money from this account and tries to add it to a balance.
     * @param amount the amount of money to transfer
     * @param otherBalance balance to add the amount to
     */
    public void transfer(double amount, double otherBalance) {
        balance = balance - amount;
        otherBalance = otherBalance + amount;
        // Won't update the argument
    }
}
```

Now let’s see what happens when we call the `transfer` method:

```java
double savingsBalance = 1000;
harrysChecking.transfer(500, savingsBalance);   // Won't update the argument
System.out.println(savingsBalance);
```

You might expect that after the call, the `savingsBalance` variable has been incremented to 1500. However, that is not the case. As the method starts, the parameter variable `otherBalance` is set to the same value as `savingsBalance` (see Figure 2). Then the `otherBalance` variable is set to a different value. That modification has no effect on `savingsBalance`, because `otherBalance` is a separate variable. When the method terminates, the `otherBalance` variable is removed, and `savingsBalance` isn’t increased.

In Java, parameter variables are initialized with the values of the argument expressions. When the method exits, the parameter variables are removed. Computer scientists refer to this call mechanism as “call by value”.

For that reason, a Java method can never change the contents of a variable that is passed as an argument—the method manipulates a different variable.

Other programming languages such as C++ support a mechanism, called “call by reference”, that can change the arguments of a method call. You will sometimes read in Java books
that “numbers are passed by value, objects are passed by reference”. That is technically not quite correct. In Java, objects themselves are never passed as arguments; instead, both numbers and object references are passed by value.

The confusion arises because a Java method can mutate an object when it receives an object reference as an argument (see Figure 3).

```java
public class BankAccount {
    ...;

    /**
     * Transfers money from this account to another.
     * @param amount the amount of money to transfer
     * @param otherAccount account to add the amount to
    */
```
```java
/*
public void transfer(double amount, BankAccount otherAccount) {
    balance = balance - amount;
    otherAccount.deposit(amount);
}
*/
```

**Figure 3** Methods Can Mutate Any Objects to Which They Hold References
Now we pass an object reference to the `transfer` method:

```java
BankAccount harrysSavings = new BankAccount(1000);
harrysChecking.transfer(500, harrysSavings);  // 1
System.out.println(harrysSavings.getBalance());
```

This example works as expected. The parameter variable `otherAccount` contains a copy of the object reference `harrysSavings`. You saw in Section 2.8 what is means to make a copy of an object reference—you get another reference to the same object. Through that reference, the method is able to modify the object.

However, a method cannot replace an object reference that is passed as an argument. To appreciate this subtle difference, consider this method that tries to set the `otherAccount` parameter variable to a new object:

```
public class BankAccount
{
   ...  
   public void transfer(double amount, BankAccount otherAccount)
   {
      balance = balance - amount;
      double newBalance = otherAccount.balance + amount;
      otherAccount = new BankAccount(newBalance);  // Won’t work
   }
}
```

In this situation, we are not trying to change the state of the object to which the parameter variable `otherAccount` refers; instead, we are trying to replace the object with a different one (see Figure 4). Now the reference stored in parameter variable `otherAccount` is replaced with a reference to a new account. But if you call the method with

```
harrysChecking.transfer(500, savingsAccount);
```

then that change does not affect the `savingsAccount` variable that is supplied in the call. This example demonstrates that objects are not passed by reference.

**Figure 4**
Replacing the Object Reference in a Parameter Variable Has No Effect on the Caller

In Java, a method can change the state of an object reference argument, but it cannot replace the object reference with another.
To summarize:

- A Java method can’t change the contents of any variable passed as an argument.
- A Java method can mutate an object when it receives a reference to it as an argument.

## 8.3 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, 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.

### 8.3.1 Keeping a Total

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:

```java
private double purchase;
```

Locate the methods that affect the total. There is usually a method to increase it by a given amount:

```java
public void recordPurchase(double amount)
{
    purchase = purchase + amount;
}
```

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, one can provide a `clear` method:

```java
public void clear()
{
    purchase = 0;
}
```

There is usually a method that yields the current total. It is easy to implement:

```java
public double getAmountDue()
{
    return purchase;
}
```

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.

### 8.3.2 Counting Events

You often need to count how many times certain events occur in the life of an object. For example:

- In a cash register, you may 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

```java
private int itemCount;
```

Increment the counter in those methods that correspond to the events that you want to count:

```java
public void recordPurchase(double amount)
{
    purchase = purchase + amount;
    itemCount++;  
}
```

You may need to clear the counter, for example at the end of a sale or a statement period:

```java
public void clear()
{
    purchase = 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.

### 8.3.3 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,

```java
public class Question
{
    private ArrayList<String> choices;
    // ...
```

A shopping cart object needs to manage a collection of items.
In the constructor, initialize the instance variable to an empty collection:

```java
public Question()
{
    choices = new ArrayList<String>();
}
```

You need to supply some mechanism for adding values. It is common to provide a method for appending a value to the collection:

```java
public void add(String option)
{
    choices.add(option);
}
```

The user of a `Question` object can call this method multiple times to add the choices.

### 8.3.4 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.

```java
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:

```java
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.

```java
public class Student
{
    private int id;
    ...
    public Student(int anId) { id = anId; }
    public String getId() { return id; }
    // No setId method
    ...
}
```

### 8.3.5 Modeling Objects with Distinct States

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:

```java
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;
    . . .
}
```

(Alternatively, you can use an enumeration—see Special Topic 5.4)

Determine which methods change the state. In this example, a fish that has just eaten won't be hungry. But as the fish moves, it will get hungrier:

```java
public void eat()
{
    hungry = NOT_HUNGRY;
    . . .
}
```

```java
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:

```java
public void move()
{
    if (hungry == VERY_HUNGRY)
    {
        Look for food.
    }
    . . .
}
```

### 8.3.6 Describing the Position of an Object

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:

```java
private double distanceFromTerminus;
```
A bug in a grid needs to store its row, column, and direction.

If the object moves in a grid, remember its current location and direction in the grid:

```java
private int row;
private int column;
private int direction; // 0 = North, 1 = East, 2 = South, 3 = West
```

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:

```java
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:

```java
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.

```java
public void moveOneUnit() {
    if (direction == NORTH) { row--; }
    else if (direction == EAST) { column++; }
    else if (direction == SOUTH) { row++; }
    else if (direction == WEST) { column--; }
}
```

Exercise P8.10 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**

10. 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:

```java
public class BankAccount {
    private int transactionCount;
    .
}
```

In which methods does this counter need to be updated?
11. In How To 3.1, the CashRegister class does not have a getTotalPurchase method. Instead, you have to call receivePayment and then giveChange. Which recommendation of Section 8.2.4 does this design violate? What is a better alternative?

12. In the example in Section 8.3.3, 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?

13. Suppose we want to enhance the CashRegister class in How To 3.1 to track the prices of all purchased items for printing a receipt. Which instance variable should you provide? Which methods should you modify?

14. 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?

15. Suppose the setName method in Section 8.3.4 is changed so that it returns true if the new name is set, false if not. Is this a good idea?

16. Look at the direction instance variable in the bug example in Section 8.3.6. This is an example of which pattern?

Practice It  Now you can try these exercises at the end of the chapter: E8.24, E8.25, E8.26.

### 8.4 Static Variables and Methods

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.

```java
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 all objects share a single copy of the lastAssignedNumber variable (see Figure 5). That variable is stored in a separate location, outside any BankAccount objects.
Each BankAccount object has its own accountNumber instance variable.

There is a single lastAssignedNumber static variable for the BankAccount class.

Figure 5 A Static Variable and Instance Variables

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

```java
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.

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:

```java
public class Financial
{
}
```
8.4 Static Variables and Methods

/**
   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);

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.

17. Name two static variables of the System class.

18. Name a static constant of the Math class.

19. 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?

20. 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: R8.24, E8.6, E8.7.

Minimize the Use of Static Methods

It is possible to solve programming problems by using classes with only static methods. In fact, before object-oriented programming was invented, that approach was quite common. However, it usually leads to a design that is not object-oriented and makes it hard to evolve a program.

Consider the task of How To 7.1. A program reads scores for a student and prints the final score, which is obtained by dropping the lowest one. We solved the problem by implementing a Student class that stores student scores. Of course, we could have simply written a program with a few static methods:

    public class ScoreAnalyzer
    {
        public static double[] readInputs() { ... }
        public static double sum(double[] values) { ... }
        public static double minimum(double[] values) { ... }
        public static double finalScore(double[] values)
        {
            if (values.length == 0) { return 0; }
            else if (values.length == 1) { return values[0]; }
            else { return sum(values) - minimum(values); }
        }
    }

© Alex Slobodkin/iStockphoto.
© Nicholas Homrich/iStockphoto.
© Eric Isselé/iStockphoto.
public static void main(String[] args) {
    System.out.println(finalScore(readInputs()));
}

That solution is fine if one's sole objective is to solve a simple homework problem. But suppose you need to modify the program so that it deals with multiple students. An object-oriented program can evolve the Student class to store grades for many students. In contrast, adding more functionality to static methods gets messy quickly (see Exercise E8.8).

**Trying to Access Instance Variables in Static Methods**

A static method does not operate on an object. In other words, it has no implicit parameter, and you cannot directly access any instance variables. For example, the following code is wrong:

```java
public class SavingsAccount {
    private double balance;
    private double interestRate;

    public static double interest(double amount) {
        return (interestRate / 100) * amount;
        // Error: Static method accesses instance variable
    }
}
```

Because different savings accounts can have different interest rates, the interest method should not be a static method.

**Alternative Forms of Instance and Static Variable Initialization**

As you have seen, instance variables are initialized with a default value (0, false, or null, depending on their type). You can then set them to any desired value in a constructor, and that is the style that we prefer in this book.

However, there are two other mechanisms to specify an initial value. Just as with local variables, you can specify initialization values for instance variables. For example,

```java
public class Coin {
    private double value = 1;
    private String name = "Dollar";
    ...}
```

These default values are used for every object that is being constructed.

There is also another, much less common, syntax. You can place one or more initialization blocks inside the class declaration. All statements in that block are executed whenever an object is being constructed. Here is an example:

```java
public class Coin {
    private double value;
    private String name;
    {
```
For static variables, you use a static initialization block:

```java
public class BankAccount
{
    private static int lastAssignedNumber;
    static
    {
        lastAssignedNumber = 1000;
    }
    ...
}
```

All statements in the static initialization block are executed once when the class is loaded. Initialization blocks are rarely used in practice.

When an object is constructed, the initializers and initialization blocks are executed in the order in which they appear. Then the code in the constructor is executed. Because the rules for the alternative initialization mechanisms are somewhat complex, we recommend that you simply use constructors to do the job of construction.

---

**Static Imports**

There is a variant of the `import` directive that lets you use static methods and variables without class prefixes. For example:

```java
import static java.lang.System.*;
import static java.lang.Math.*;
```

```java
public class RootTester
{
    public static void main(String[] args)
    {
        double r = sqrt(PI); // Instead of Math.sqrt(Math.PI)
        out.println(r); // Instead of System.out
    }
}
```

Static imports can make programs easier to read, particularly if they use many mathematical functions.

---

8.5 Problem Solving: Solve a Simpler Problem First

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.

A Picture class is given to you. It has a constructor

```java
public Picture(String filename)
```

and methods

```java
public void move(int dx, int dy)
public Rectangle getBounds()
```

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:

   ```java
   public class Gallery1
   {
       public static void main(String[] args)
       {
           Picture pic = new Picture("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 rightmost x-coordinate of the preceding picture.

   ```java
   Picicture pic = new Picture("picture1.jpg");
   Picture pic2 = new Picture("picture2.jpg");
   pic2.move(pic.getBounds().getMaxX(), 0);
   
3. The next step is to separate the two by a small gap when the second is moved:

   ```java
   final int GAP = 10;
   
   Picture pic = new Picture("picture1.jpg");
   Picture pic2 = new Picture("picture2.jpg");
   double x = pic.getBounds().getMaxX() + 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 6.7.6).

```java
final int GAP = 10;
final int PICTURES = 20;

Picture pic = new Picture("picture1.jpg");
for (int i = 2; i <= PICTURES; i++)
{
    Picture previous = pic;
    pic = new Picture("picture" + i + ".jpg");
    double x = previous.getBounds().getMaxX() + 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.

```java
double x = previous.getBounds().getMaxX() + GAP;
if (x + pic.getBounds().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:

```java
maxY = Math.max(maxY, pic.getBounds().getMaxY());
```

The following statement places a picture on the next row:

```java
pic.move(0, maxY + GAP);
```
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.

section_5/Gallery6.java

```java
public class Gallery6 {
    public static void main(String[] args) {
        final int MAX_WIDTH = 720;
        final int GAP = 10;
        final int PICTURES = 20;

        Picture pic = new Picture("picture1.jpg");
        double maxY = 0;

        for (int i = 2; i <= 20; i++) {
            maxY = Math.max(maxY, pic.getBounds().getMaxY());
            Picture previous = pic;
            pic = new Picture("picture" + i + ".jpg");
            double x = previous.getBounds().getMaxX() + GAP;
            if (x + pic.getBounds().getWidth() < MAX_WIDTH) {
                pic.move(x, previous.getBounds().getY());
            } else {
                pic.move(0, maxY + GAP);
            }
        }
    }
}
```

21. 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?

22. 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?

23. You want to remove “red eyes” from images and are looking for red circles. What simpler problem can you start with?

24. 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: R8.26, P8.5.
8.6 Packages

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>
<thead>
<tr>
<th>Package</th>
<th>Purpose</th>
<th>Sample Class</th>
</tr>
</thead>
<tbody>
<tr>
<td>java.lang</td>
<td>Language support</td>
<td>Math</td>
</tr>
<tr>
<td>java.util</td>
<td>Utilities</td>
<td>Random</td>
</tr>
<tr>
<td>java.io</td>
<td>Input and output</td>
<td>PrintStream</td>
</tr>
<tr>
<td>java.awt</td>
<td>Abstract Windowing Toolkit</td>
<td>Color</td>
</tr>
<tr>
<td>java.applet</td>
<td>Applets</td>
<td>Applet</td>
</tr>
<tr>
<td>java.net</td>
<td>Networking</td>
<td>Socket</td>
</tr>
<tr>
<td>java.sql</td>
<td>Database access through Structured Query Language</td>
<td>ResultSet</td>
</tr>
<tr>
<td>javax.swing</td>
<td>Swing user interface</td>
<td>JButton</td>
</tr>
<tr>
<td>org.w3c.dom</td>
<td>Document Object Model for XML documents</td>
<td>Document</td>
</tr>
</tbody>
</table>

8.6.1 Organizing Related Classes into Packages

To put one of your classes in a package, you must place a line

```java
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 8.6.3 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`.

The `Financial.java` file must start as follows:

```java
package com.horstmann.bigjava;
public class Financial {
    . . .
}
```

In Java, related classes are grouped into packages.
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.

### 8.6.2 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
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:

```java
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

```java
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`.

### 8.6.3 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.

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.

### 8.6.4 Packages and Source Files

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 6. (Here, we are using UNIX-style file names. Under Windows, you might use c:\Users\Britney\hw8\problem1\com\horstmann\bigjava.)
25. Which of the following are packages?
   a. java
   b. java.lang
   c. java.util
   d. java.lang.Math

26. Is a Java program without import statements limited to using the default and java.lang packages?

27. 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: R8.28, E8.18, E8.19.

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.

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.
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:

```java
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 8.1 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`).

**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 subpackage 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)
- `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
```java
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,
```bash
cd /home/britney
javac homework1/problem2/Tester.java
```
or
```bash
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,
```bash
cd /home/britney
java homework1.problem2.Tester
```
or
```bash
c: cd \Users\Britney
java homework1\problem2\Tester
```

### 8.7 Unit Test Frameworks

Up to now, we have used a very simple approach to testing. We provided tester classes whose *main* method computes values and prints actual and expected values. However, that approach has limitations. The *main* method gets messy if it contains many tests. And if an exception occurs during one of the tests, the remaining tests are not executed.

Unit testing frameworks were designed to quickly execute and evaluate test suites and to make it easy to incrementally add test cases. One of the most popular testing frameworks is JUnit. It is freely available at [http://junit.org](http://junit.org), and it is also built into a number of development environments, including BlueJ and Eclipse. Here we describe JUnit 4, the most current version of the library as this book is written.

When you use JUnit, you design a companion test class for each class that you develop. You provide a method for each test case that you want to have executed. You use “annotations” to mark the test methods. An annotation is an advanced Java feature that places a marker into the code that is interpreted by another tool. In the case of JUnit, the @Test annotation is used to mark test methods.

In each test case, you make some computations and then compare some condition that you believe to be true. You then pass the result to a method that communicates a test result to the framework, most commonly the assertEquals method. The assertEquals method takes as arguments the expected and actual values and, for floating-point numbers, a tolerance value.
It is also customary (but not required) that the name of the test class ends in Test, such as CashRegisterTest. Here is a typical example:

```java
import org.junit.Test;
import org.junit.Assert;

public class CashRegisterTest {
    @Test public void twoPurchases() {
        CashRegister register = new CashRegister();
        register.recordPurchase(0.75);
        register.recordPurchase(1.50);
        register.receivePayment(2, 0, 5, 0, 0);
        double expected = 0.25;
        Assert.assertEquals(expected, register.giveChange(), EPSILON);
    }
    // More test cases...
}
```

If all test cases pass, the JUnit tool shows a green bar (see Figure 7). If any of the test cases fail, the JUnit tool shows a red bar and an error message.

Your test class can also have other methods (whose names should not be annotated with @Test). These methods typically carry out steps that you want to share among test methods.

The JUnit philosophy is simple. Whenever you implement a class, also make a companion test class. You design the tests as you design the program, one test method at a time. The test cases just keep accumulating in the test class. Whenever you have detected an actual failure, add a test case that flushes it out, so that you can be sure that you won’t introduce that particular bug again. Whenever you modify your class, simply run the tests again.

If all tests pass, the user interface shows a green bar and you can relax. Otherwise, there is a red bar, but that’s also good. It is much easier to fix a bug in isolation than inside a complex program.

**28.** Provide a JUnit test class with one test case for the Earthquake class in Chapter 5.

**29.** What is the significance of the EPSILON argument in the assertEquals method?

**Practice It**

Now you can try these exercises at the end of the chapter: R8.30, E8.20, E8.21.
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?
Find classes that are appropriate for solving a programming problem.

- A class should represent a single concept from a problem domain, such as business, science, or mathematics.

Design methods that are cohesive, consistent, and minimize side effects.

- The public interface of a class is cohesive if all of its features are related to the concept that the class represents.
- A class depends on another class if its methods use that class in any way.
- An immutable class has no mutator methods.
- References to objects of an immutable class can be safely shared.
- A side effect of a method is any externally observable data modification.
- When designing methods, minimize side effects.
- In Java, a method can change the state of an object reference argument, but it cannot replace the object reference with another.

Use patterns to design the data representation of an object.

- 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.

Understand the behavior of static variables and static methods.

- A static variable belongs to the class, not to any object of the class.
- A static method is not invoked on an object.

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.
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.

Use JUnit for writing unit tests.

- Unit test frameworks simplify the task of writing classes that contain many test cases.
- The JUnit philosophy is to run all tests whenever you change your code.

R8.1 Consider a car share system in which drivers pick up other riders, enabling them to make money during their commute while reducing traffic congestion. Riders wait at pickup points, are dropped off at their destinations, and pay for the distance traveled. Drivers get a monthly payment. An app lets drivers and riders enter their route and time. It notifies drivers and riders and handles billing. Find classes that would be useful for designing such a system.

R8.2 Suppose you want to design a social network for internship projects at your university. Students can register their skills and availability. Project sponsors describe projects, required skills, expected work effort, and desired completion date. A search facility lets students find matching projects. Find classes that would be useful for designing such a system.

R8.3 Your task is to write a program that simulates a vending machine. Users select a product and provide payment. If the payment is sufficient to cover the purchase price of the product, the product is dispensed and change is given. Otherwise, the payment is returned to the user. Name an appropriate class for implementing this program. Name two classes that would not be appropriate and explain why.

R8.4 Your task is to write a program that reads a customer’s name and address, followed by a sequence of purchased items and their prices, and prints an invoice. Discuss which of the following would be good classes for implementing this program:

- a. Invoice
- b. InvoicePrinter
- c. PrintInvoice
- d. InvoiceProgram
Chapter 8  Designing Classes

R8.5 Your task is to write a program that computes paychecks. Employees are paid an hourly rate for each hour worked; however, if they worked more than 40 hours per week, they are paid at 150 percent of the regular rate for those overtime hours. Name an actor class that would be appropriate for implementing this program. Then name a class that isn’t an actor class that would be an appropriate alternative. How does the choice between these alternatives affect the program structure?

R8.6 Look at the public interface of the java.lang.System class and discuss whether or not it is cohesive.

R8.7 Suppose an Invoice object contains descriptions of the products ordered, and the billing and shipping addresses of the customer. Draw a UML diagram showing the dependencies between the classes Invoice, Address, Customer, and Product.

R8.8 Suppose a vending machine contains products, and users insert coins into the vending machine to purchase products. Draw a UML diagram showing the dependencies between the classes VendingMachine, Coin, and Product.

R8.9 On which classes does the class Integer in the standard library depend?

R8.10 On which classes does the class Rectangle in the standard library depend?

R8.11 Classify the methods of the class Scanner that are used in this book as accessors and mutators.

R8.12 Classify the methods of the class Rectangle as accessors and mutators.

R8.13 Is the Resistor class in Exercise P8.12 a mutable or immutable class? Why?

R8.14 Which of the following classes are immutable?
   a. Rectangle
   b. String
   c. Random

R8.15 Which of the following classes are immutable?
   a. PrintStream
   b. Date
   c. Integer

R8.16 Consider a method
   
   public class DataSet
   {
      /**
         * Reads all numbers from a scanner and adds them to this data set.
         * @param in a Scanner
         */
      public void read(Scanner in) {
         ...
      }
   }

   Describe the side effects of the read method. Which of them are not recommended, according to Section 8.2.4? Which redesign eliminates the unwanted side effect? What is the effect of the redesign on coupling?

R8.17 What side effect, if any, do the following three methods have?
   
   public class Coin
   {
public void print()
{
    System.out.println(name + " " + value);
}

public void print(PrintStream stream)
{
    stream.println(name + " " + value);
}

public String toString()
{
    return name + " " + value;
}
}

**R8.18** Ideally, a method should have no side effects. Can you write a program in which no method has a side effect? Would such a program be useful?

**R8.19** Consider the following method that is intended to swap the values of two integers:

```java
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 method swap the contents of x and y?

**R8.20** How can you write a method that swaps two floating-point numbers?
*Hint: java.awt.Point.*

**R8.21** Draw a memory diagram that shows why the following method can't swap two BankAccount objects:

```java
public static void falseSwap(BankAccount a, BankAccount b)
{
    BankAccount temp = a;
    a = b;
    b = temp;
}
```

**R8.22** Consider an enhancement of the Die class of Chapter 6 with a static variable

```java
public class Die
{
    private int sides;
    private static Random generator = new Random();
    public Die(int s) {...}
    public int cast() {...}
}
Draw a memory diagram that shows three dice:

```java
Die d4 = new Die(4);
Die d6 = new Die(6);
Die d8 = new Die(8);
```

Be sure to indicate the values of the sides and generator variables.

**R8.23** Try compiling the following program. Explain the error message that you get.

```java
public class Print13
{
    public void print(int x)
    {
        System.out.println(x);
    }

    public static void main(String[] args)
    {
        int n = 13;
        print(n);
    }
}
```

**R8.24** Look at the methods in the `Integer` class. Which are static? Why?

**R8.25** Look at the methods in the `String` class (but ignore the ones that take an argument of type `char[]`). Which are static? Why?

**R8.26** Consider the task of fully justifying a paragraph of text to a target length, by putting as many words as possible on each line and evenly distributing extra spaces so that each line has the target length. Devise a plan for writing a program that reads a paragraph of text and prints it fully justified. Describe a sequence of progressively more complex intermediate programs, similar to the approach in Section 8.5.

**R8.27** The `in` and `out` variables of the `System` class are public static variables of the `System` class. Is that good design? If not, how could you improve on it?

**R8.28** Every Java program can be rewritten to avoid `import` statements. Explain how, and rewrite `RectangleComponent.java` from Section 2.9.3 to avoid `import` statements.

**R8.29** What is the default package? Have you used it before this chapter in your programming?

**Testing R8.30** What does JUnit do when a test method throws an exception? Try it out and report your findings.

---

**PRACTICE EXERCISES**

**E8.1** Implement the `Coin` class described in Section 8.2. Modify the `CashRegister` class so that coins can be added to the cash register, by supplying a method

```java
void receivePayment(int coinCount, Coin coinType)
```

The caller needs to invoke this method multiple times, once for each type of coin that is present in the payment.
Exercise E8.2: Modify the `giveChange` method of the `CashRegister` class so that it returns the number of coins of a particular type to return:

```java
int giveChange(Coin coinType)
```

The caller needs to invoke this method for each coin type, in decreasing value.

Exercise E8.3: Real cash registers can handle both bills and coins. Design a single class that expresses the commonality of these concepts. Redesign the `CashRegister` class and provide a method for entering payments that are described by your class. Your primary challenge is to come up with a good name for this class.

Exercise E8.4: Reimplement the `BankAccount` class so that it is immutable. The `deposit` and `withdraw` methods need to return new `BankAccount` objects with the appropriate balance.

Exercise E8.5: Reimplement the `Day` class of Worked Example 2.1 to be immutable. Change mutator methods to return new `Day` objects. Also change the demonstration program.

Exercise E8.6: Write static methods

```java
• public static double cubeVolume(double h)
• public static double cubeSurface(double h)
• 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 cube with height \( h \), sphere with radius \( r \), a cylinder with circular base with radius \( r \) and height \( h \), and a cone with 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.

Exercise E8.7: Solve Exercise E8.6 by implementing classes `Cube`, `Sphere`, `Cylinder`, and `Cone`. Which approach is more object-oriented?

Exercise E8.8: Modify the application of How To 7.1 so that it can deal with multiple students. First, ask the user for all student names. Then read in the scores for all quizzes, prompting for the score of each student. Finally, print the names of all students and their final scores. Use a single class and only static methods.

Exercise E8.9: Repeat Exercise E8.8, using multiple classes. Provide a `GradeBook` class that collects objects of type `Student`.

Exercise E8.10: Write methods

```java
public static double perimeter(Ellipse2D.Double e);
public static double area(Ellipse2D.Double e);
```

that compute the area and the perimeter of the ellipse \( e \). Add these methods to a class `Geometry`. The challenging part of this assignment is to find and implement an accurate formula for the perimeter. Why does it make sense to use a static method in this case?
**E8.11** Write methods

```java
public static double angle(Point2D.Double p, Point2D.Double q)
public static double slope(Point2D.Double p, Point2D.Double q)
```

that compute the angle between the x-axis and the line joining two points, measured in degrees, and the slope of that line. Add the methods to the class Geometry. Supply suitable preconditions. Why does it make sense to use a static method in this case?

**E8.12** Write methods

```java
public static boolean isInside(Point2D.Double p, Ellipse2D.Double e)
public static boolean isOnBoundary(Point2D.Double p, Ellipse2D.Double e)
```

that test whether a point is inside or on the boundary of an ellipse. Add the methods to the class Geometry.

**E8.13** Using the `Picture` class from Worked Example 6.2, write a method

```java
public static Picture superimpose(Picture pic1, Picture pic2)
```

that superimposes two pictures, yielding a picture whose width and height are the larger of the widths and heights of `pic1` and `pic2`. In the area where both pictures have pixels, average the colors.

**E8.14** Using the `Picture` class from Worked Example 6.2, write a method

```java
public static Picture greenScreen(Picture pic1, Picture pic2)
```

that superimposes two pictures, yielding a picture whose width and height are the larger of the widths and heights of `pic1` and `pic2`. In the area where both pictures have pixels, use `pic1`, except when its pixels are green, in which case, you use `pic2`.

**E8.15** Write a method

```java
public static int readInt(
    Scanner in, String prompt, String error, int min, int max)
```

that displays the prompt string, reads an integer, and tests whether it is between the minimum and maximum. If not, print an error message and repeat reading the input. Add the method to a class `Input`.

**E8.16** Consider the following algorithm for computing $x^n$ for an integer $n$. If $n < 0$, $x^n$ is $1/x^{-n}$. If $n$ is positive and even, then $x^n = (x^n/2)^2$. If $n$ is positive and odd, then $x^n = x^{n-1} \times x$. Implement a static method `double intPower(double x, int n)` that uses this algorithm. Add it to a class called `Numeric`.

**E8.17** Improve the `Die` class of Chapter 6. Turn the generator variable into a static variable so that all needles share a single random number generator.

**E8.18** Implement `Coin` and `CashRegister` classes as described in Exercise E8.1. Place the classes into a package called `money`. Keep the `CashRegisterTester` class in the default package.

**E8.19** Place a `BankAccount` class in a package whose name is derived from your e-mail address, as described in Section 8.6. Keep the `BankAccountTester` class in the default package.

**Testing E8.20** Provide a JUnit test class `StudentTest` with three test methods, each of which tests a different method of the `Student` class in How To 7.1.

**Testing E8.21** Provide JUnit test class `TaxReturnTest` with three test methods that test different tax situations for the `TaxReturn` class in Chapter 5.
• Graphics E8.22 Write methods
  
  ```java
  public static void drawH(Graphics2D g2, Point2D.Double p);
  public static void drawE(Graphics2D g2, Point2D.Double p);
  public static void drawL(Graphics2D g2, Point2D.Double p);
  public static void drawO(Graphics2D g2, Point2D.Double p);
  ```

  that show the letters H, E, L, O in the graphics window, where the point \( p \) is the top-left corner of the letter. Then call the methods to draw the words “HELLO” and “HOLE” on the graphics display. Draw lines and ellipses. Do not use the `drawString` method. Do not use `System.out`.

• Graphics E8.23 Repeat Exercise E8.22 by designing classes `LetterH`, `LetterE`, `LetterL`, and `LetterO`, each with a constructor that takes a `Point2D.Double` parameter (the top-left corner) and a method `draw(Graphics2D g2)`. Which solution is more object-oriented?

• E8.24 Add a method `ArrayList<Double> getStatement()` to the `BankAccount` class that returns a list of all deposits and withdrawals as positive or negative values. Also add a method `void clearStatement()` that resets the statement.

• E8.25 Implement a class `LoginForm` that simulates a login form that you find on many web pages. Supply methods
  
  ```java
  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 isLoggedIn 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.

• E8.26 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
  
  ```java
  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".

• P8.1 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.

```java
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() { . . . }
}
```

\[ *** P8.2 \]

Improve the picture gallery program in Section 8.5 to fill the space more efficiently. Instead of lining up all pictures along the top edge, find the first available space where you can insert a picture (still respecting the gaps).

\[ Hint: \] Solve a simpler problem first, lining up the pictures without gaps.

That is still not easy. You need to test whether a new picture fits. Put the bounding rectangles of all placed pictures in an `ArrayList` and implement a method

```java
public static boolean intersects(Rectangle r, ArrayList<Rectangle> rectangles)
```

that checks whether `r` intersects any of the given rectangles. Use the `intersects` method in the `Rectangle` class.

Then you need to figure out where you can try putting the new picture. Try something simple first, and check the corner points of all existing rectangles. Try points with smaller \(y\)-coordinates first.
For a better fit, check all points whose $x$- and $y$-coordinates are the $x$- and $y$-coordinates of corner points, but not necessarily the same point.

Once that works, add the gaps between images.

**P8.3** Simulate a car sharing system in which car commuters pick up and drop off passengers at designated stations. Assume that there are 30 such stations, one at every mile along a route. At each station, randomly generate a number of cars and passengers, each of them with a desired target station.

Each driver picks up three random passengers whose destination is on the way to the car’s destination, drops them off where requested, and picks up more if possible. A driver gets paid per passenger per mile. Run the simulation 1,000 times and report the average revenue per mile.

Use classes Car, Passenger, Station, and Simulation in your solution.

**P8.4** In Exercise P8.3, drivers picked up passengers at random. Try improving that scheme. Are drivers better off picking passengers that want to go as far as possible along their route? Is it worth looking at stations along the route to optimize the loading plan? Come up with a solution that increases average revenue per mile.

**P8.5** Tabular data are often saved in the CSV (comma-separated values) format. Each table row is stored in a line, and column entries are separated by commas. However, if an entry contains a comma or quotation marks, they enclosed in quotation marks, doubling any quotation marks of the entry. For example,

```
John Jacob Astor,1763,1848
"William Backhouse Astor, Jr.",1829,1892
"John Jacob ""Jakey"" Astor VI",1912,1992
```

Provide a class Table with methods

```
public void addLine(String line)
public String getEntry(int row, int column)
public int rows()
public int columns()
```

Solve this problem by producing progressively more complex intermediate versions of your class and a tester, similar to the approach in Section 8.5.

**P8.6** For faster sorting of letters, the U.S. Postal Service encourages companies that send large volumes of mail to use a bar code denoting the ZIP code (see Figure 8).

The encoding scheme for a five-digit ZIP code is shown in Figure 9. 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 sum of the digits in the ZIP code 95014 is 19, so the check digit is 1 to make the sum equal to 20.

Each digit of the ZIP code, and the check digit, is encoded according to the table at right, where 0 denotes a half bar and 1 a full bar. Note that they represent all combinations of two full and three half bars. The digit can be computed easily 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

```
||:|:::|:|:||::::::||:|::|:::|||
```

(Alternatively, write a graphical application that draws real bars.)

Your program should also be able to carry out the opposite conversion: Translate bars into their ZIP code, reporting any errors in the input format or a mismatch of the digits.

**Business P8.7** Implement a program that prints paychecks for a group of student assistants. Deduct federal and Social Security taxes. (You may want to use the tax computation used in Chapter 5. Find out about Social Security taxes on the Internet.) Your program should prompt for the names, hourly wages, and hours worked of each student.

**Business P8.8** 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

```java
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 P8.9** The Downtown Marketing Association wants to promote downtown shopping with a loyalty program similar to the one in Exercise P8.8. 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.
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 deltaSec)` that moves the ball to the next position. First compute the distance traveled in \( \Delta \text{t} \) 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 \(-9.81 \text{ m/s}^2 \); the \( x \)-velocity is unchanged.
- A method `Point getLocation()` that gets the current location of the cannonball, rounded to integer coordinates.
- A method `ArrayList<Point> shoot(double alpha, double v, double deltaSec)` whose arguments are the angle \( \alpha \) 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 the given time interval until the \( y \)-position is 0; return an array list of locations after each call to move.)

Use this class in a program that prompts the user for the starting angle and the initial velocity. Then call `shoot` and print the locations.

Continue Exercise P8.10, and draw the trajectory of the cannonball.

The colored bands on the top-most resistor shown in the photo at right indicate a resistance of 6.2 k\( \Omega \) ± 5 percent. The resistor tolerance of ±5 percent indicates the acceptable variation in the resistance. A 6.2 k\( \Omega \) ± 5 percent resistor could have a resistance as small as 5.89 k\( \Omega \) or as large as 6.51 k\( \Omega \). We say that 6.2 k\( \Omega \) is the *nominal value* of the resistance and that the actual value of the resistance can be any value between 5.89 k\( \Omega \) and 6.51 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 \( \Omega \) ± 10 percent resistors.

In the `Resistor` class from Exercise P8.12, 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.
Chapter 8  Designing Classes

<table>
<thead>
<tr>
<th>Color</th>
<th>Digit</th>
<th>Multiplier</th>
<th>Tolerance</th>
</tr>
</thead>
<tbody>
<tr>
<td>Black</td>
<td>0</td>
<td>x10^0</td>
<td>—</td>
</tr>
<tr>
<td>Brown</td>
<td>1</td>
<td>x10^1</td>
<td>±1%</td>
</tr>
<tr>
<td>Red</td>
<td>2</td>
<td>x10^2</td>
<td>±2%</td>
</tr>
<tr>
<td>Orange</td>
<td>3</td>
<td>x10^3</td>
<td>—</td>
</tr>
<tr>
<td>Yellow</td>
<td>4</td>
<td>x10^4</td>
<td>—</td>
</tr>
<tr>
<td>Green</td>
<td>5</td>
<td>x10^5</td>
<td>±0.5%</td>
</tr>
<tr>
<td>Blue</td>
<td>6</td>
<td>x10^6</td>
<td>±0.25%</td>
</tr>
<tr>
<td>Violet</td>
<td>7</td>
<td>x10^7</td>
<td>±0.1%</td>
</tr>
<tr>
<td>Gray</td>
<td>8</td>
<td>x10^8</td>
<td>±0.05%</td>
</tr>
<tr>
<td>White</td>
<td>9</td>
<td>x10^9</td>
<td>—</td>
</tr>
<tr>
<td>Gold</td>
<td>—</td>
<td>x10^-1</td>
<td>±5%</td>
</tr>
<tr>
<td>Silver</td>
<td>—</td>
<td>x10^-2</td>
<td>±10%</td>
</tr>
<tr>
<td>None</td>
<td>—</td>
<td>—</td>
<td>±20%</td>
</tr>
</tbody>
</table>

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 10⁵, and a tolerance of ±5 percent, for a resistance of 2,700 kΩ, plus or minus 5 percent.

Science P8.14 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 P8.12. 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, \texttt{VoltageDivider} and \texttt{Resistor}. The \texttt{Resistor} class is described in Exercise P8.12. The \texttt{VoltageDivider} class should have two instance variables that are objects of the \texttt{Resistor} class. Provide a single constructor that accepts two \texttt{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 $R_2 = 750 \, \Omega$.

\begin{enumerate}
\item Look for nouns in the problem description.
\item Yes (\texttt{ChessBoard}) and no (\texttt{MovePiece}).
\item Some of its features deal with payments, others with coin values.
\item None of the coin operations require the \texttt{CashRegister} class.
\item If a class doesn’t depend on another, it is not affected by interface changes in the other class.
\item It is an accessor—calling \texttt{substring} doesn’t modify the string on which the method is invoked. In fact, all methods of the \texttt{String} class are accessors.
\item No—\texttt{translate} is a mutator method.
\item It is a side effect; this kind of side effect is common in object-oriented programming.
\item Yes—the method affects the state of the \texttt{Scanner} argument.
\item It needs to be incremented in the \texttt{deposit} and \texttt{withdraw} methods. There also needs to be some method to reset it after the end of a statement period.
\item The \texttt{giveChange} method is a mutator that returns a value that cannot be determined any other way. Here is a better design. The \texttt{receivePayment} method could decrease the \texttt{purchase} instance variable. Then the program user would call \texttt{receivePayment}, determine the change by calling \texttt{getAmountDue}, and call the \texttt{clear} method to reset the cash register for the next sale.
\item The \texttt{ArrayList\textless String\textgt} instance variable is private, and the class users cannot access it.
\item You need to supply an instance variable that can hold the prices for all purchased items. This could be an \texttt{ArrayList\textless Double\textgt} or \texttt{ArrayList\textless String\textgt}, or it could simply be a \texttt{String} to which you append lines. The instance variable needs to be updated in the \texttt{recordPurchase} method. You also need a method that returns the receipt.
\item 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.
\item Section 8.2.3 suggests that a setter should return \texttt{void}, or perhaps a convenience value that the user can also determine in some other way. In this situation, the caller could check whether \texttt{newName} is blank, so the change is fine.
\item It is an example of the “state pattern” described in Section 8.3.5. The direction is a state that changes when the bug turns, and it affects how the bug moves.
\item \texttt{System.in} and \texttt{System.out}.
\item \texttt{Math.PI}
\item The method needs no data of any object. The only required input is the \texttt{values} argument.
\item Yes, it works. Static methods can access static variables of the same class. But it is a terrible idea. As your programming tasks get more complex, you will want to use objects and classes to organize your programs.
\item 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.
\item You could first write a program that prints all substrings of a given string.
\end{enumerate}
23. You can look for a single red pixel, or a block of nine neighboring red pixels.

24. Here is one plan:
   a. Find the position of the first digit in a string.
   b. Find the position of the first non-digit after a given position in a string.
   c. Extract the first integer from a string (using the preceding two steps).
   d. Print all integers from a string. (Use the first three steps, then repeat with the substring that starts after the extracted integer.)

25. (a) No; (b) Yes; (c) Yes; (d) No

26. No—you can use fully qualified names for all other classes, such as java.util.Random and java.awt.Rectangle.

27. /home/me/cs101/hw1/problem1 or, on Windows, c:\Users\Me\cs101\hw1\problem1.

28. Here is one possible answer.
   public class EarthquakeTest
   {
     @Test public void testLevel4()
     {
       Earthquake quake = new Earthquake(4);
       Assert.assertEquals(
         "Felt by many people, no destruction",
         quake.getDescription());
     }
   }

29. It is a tolerance threshold for comparing floating-point numbers. We want the equality test to pass if there is a small roundoff error.
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

CHAPTER CONTENTS

9.1 INHERITANCE HIERARCHIES 424
   PT1 Use a Single Class for Variation in Values, Inheritance for Variation in Behavior 428

9.2 IMPLEMENTING SUBCLASSES 428
   SYN Subclass Declaration 430
   CE1 Replicating Instance Variables from the Superclass 432
   CE2 Confusing Super- and Subclasses 432

9.3 OVERRIDING METHODS 433
   SYN Calling a Superclass Method 433
   CE3 Accidental Overloading 437
   CE4 Forgetting to Use `super` When Invoking a Superclass Method 437
   ST1 Calling the Superclass Constructor 438
   SYN Constructor with Superclass Initializer 438

9.4 POLYMORPHISM 439
   ST2 Dynamic Method Lookup and the Implicit Parameter 442
   ST3 Abstract Classes 443
   ST4 Final Methods and Classes 444
   ST5 Protected Access 444
   HT1 Developing an Inheritance Hierarchy 445
   WE1 Implementing an Employee Hierarchy for Payroll Processing

9.5 OBJECT: THE COSMIC SUPERCLASS 450
   SYN The `instanceof` Operator 453
   CE5 Don't Use Type Tests 454
   ST6 Inheritance and the `toString` Method 455
   ST7 Inheritance and the `equals` Method 456
   C&S Who Controls the Internet? 456
Objects from related classes usually share common behavior. For example, cars, bicycles, and buses all provide transportation. 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

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.

When you use inheritance in your programs, you can reuse code instead of duplicating it. This reuse comes in two forms. First, a subclass inherits the methods of the superclass. For example, if the Vehicle class has a drive method, then a subclass Car automatically inherits the method. It need not be duplicated.

The second form of reuse is more subtle. You can reuse algorithms that manipulate Vehicle objects. 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 a subclass inherits data and behavior from a superclass. You can always use a subclass object in place of a superclass object.

![An Inheritance Hierarchy of Vehicle Classes](image-url)

**Figure 1** An Inheritance Hierarchy of Vehicle Classes
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`:

```java
void processVehicle(Vehicle v)
```

Because `Car` is a subclass of `Vehicle`, you can call that method with a `Car` object:

```java
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 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.

**Figure 2**
An Inheritance Diagram

```
Vehicle
   ↑
  Car
```

We will develop a simple but flexible quiz-taking program to illustrate inheritance.
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.

**section_1/Question.java**

```java
/**
   * A question with a text and an answer.
   */
public class Question {
  private String text;
  private String answer;

  /**
   * Constructs a question with empty question and answer.
   */
  public Question() {
    text = "";
    answer = "";
  }

  /**
   * Sets the question text.
   * @param questionText the text of this question
   */
  public void setText(String questionText) {
    text = questionText;
  }

  /**
   * Sets the answer for this question.
   * @param correctResponse the answer
   */
  public void setAnswer(String correctResponse) {
    answer = correctResponse;
  }

  /**
   * Checks a given response for correctness.
   * @param response the response to check
   * @return true if the response was correct, false otherwise
   */
  public boolean checkAnswer(String response) {
    return response.equals(answer);
  }

  /**
   * Displays this question.
   */
  public void display() {
    System.out.println(text);
  }
}
```
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:

```java
section_1/QuestionDemo1.java
import java.util.Scanner;

/**
 * This program shows a simple quiz with one question.
 */
public class QuestionDemo1 {
    public static void main(String[] args) {
        Scanner in = new Scanner(System.in);
        Question q = new Question();
        q.setText("Who was the inventor of Java?");
        q.setAnswer("James Gosling");
        q.display();
        System.out.print("Your answer: ");
        String response = in.nextLine();
        System.out.println(q.checkAnswer(response));
    }
}
```

**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. What are all the superclasses of the `JFrame` class? Consult the Java API documentation or Appendix D.
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: R9.2, R9.8, R9.10.
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

```java
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.
Implementing Subclasses

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.

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:

```java
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 reserved word extends denotes inheritance. Figure 5 shows how the methods and instance variables are captured in a UML diagram.


Syntax 9.1 Subclass Declaration

```
Syntax  
public class SubclassName extends SuperclassName  
{  
    instance variables  
    methods  
}  
```

- Declare instance variables that are added to the subclass.
- Declare methods that are added to the subclass.
- Declare methods that the subclass overrides.

```
public class ChoiceQuestion extends Question  
{  
    private ArrayList<String> choices;  
    public void addChoice(String choice, boolean correct) { . . . }  
    public void display() { . . . }  
}  
```

The reserved word `extends` denotes inheritance.

Figure 6 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.

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,

```
choiceQuestion.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 question that you are currently modifying—that is, the implicit parameter of the ChoiceQuestion.addChoice method. Remember, if you invoke a method on the implicit parameter, you don’t have to specify the implicit parameter and can write just the method name:

```
setAnswer(choiceString);
```

If you prefer, you can make it clear that the method is executed on the implicit parameter:

```
this.setAnswer(choiceString);
```

### 6. Self Check

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. Employee Class

The `Employee` class 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. Manager Class

Which instance variables does the `Manager` class from Self Check 7 have?

### 9. Method Header

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: R9.4, E9.9, E9.13.

---

**Replicating Instance Variables from the Superclass**

A subclass has no access to the private instance variables of the superclass.

```java
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:

```java
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. The correct solution is to access the instance variable of the superclass through the public interface of the superclass. In our example, the `ChoiceQuestion` constructor should call the `setText` method of the `Question` class.

---

**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

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.

```java
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.

Syntax 9.2 Calling a Superclass Method

```java
Syntax     super.methodName(parameters);
```

Calls the method of the superclass instead of the method of the current class.

Calls the method of the superclass instead of the method of the current class.

If you omit super, this method calls itself. See page 437.
Instead, you can call the `display` method of the superclass, by using the reserved word `super`:

```java
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.

```java
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.

Note that `super`, unlike `this`, is not a reference to an object. There is no separate superclass object—the subclass object contains the instance variables of the superclass. Instead, `super` is simply a reserved word that forces execution of the superclass method.

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.

```java
import java.util.Scanner;

/**
   * This program shows a simple quiz with two choice questions.
   */
public class QuestionDemo2
{
    public static void main(String[] args)
    {
      ChoiceQuestion first = new ChoiceQuestion();
      first.setText("What was the original name of the Java language?");
      first.addChoice("*7", false);
      first.addChoice("Duke", false);
      first.addChoice("Oak", true);
      first.addChoice("Gosling", false);
      
      ChoiceQuestion second = new ChoiceQuestion();
      second.setText("In which country was the inventor of Java born?");
      second.addChoice("Australia", false);
      second.addChoice("Canada", true);
      second.addChoice("Denmark", false);
      second.addChoice("United States", false);
      
      presentQuestion(first);
      presentQuestion(second);
    }
```
```java
/**
 * Presents a question to the user and checks the response.
 * @param q the question
 */
public static void presentQuestion(ChoiceQuestion q) {
    q.display();
    System.out.print("Your answer: ");
    Scanner in = new Scanner(System.in);
    String response = in.nextLine();
    System.out.println(q.checkAnswer(response));
}
```

```java
import java.util.ArrayList;

/**
 * A question with multiple choices.
 */
public class ChoiceQuestion extends Question {
    private ArrayList<String> choices;

    /**
     * Constructs a choice question with no choices.
     */
    public ChoiceQuestion() {
        choices = new ArrayList<String>();
    }

    /**
     * Adds an answer choice to this question.
     * @param choice the choice to add
     * @param correct true if this is the correct choice, false otherwise
     */
    public void addChoice(String choice, boolean correct) {
        choices.add(choice);
        if (correct) {
            // Convert choices.size() to string
            String choiceString = "" + choices.size();
            setAnswer(choiceString);
        }
    }

    public void display() {
        // Display the question text
        super.display();
        // Display the answer choices
        for (int i = 0; i < choices.size(); i++) {
            //
        }
    }
```
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

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: E9.4, E9.5, E9.14.

**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

```java
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,

```java
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.

**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:

```java
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`.

### 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:

```java
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.

### Syntax 9.3 Constructor with Superclass Initializer

```java
Syntax    public ClassName(parameterType parameterName, . . .)
{
    super(arguments);
    . . .
}
```

The superclass constructor is called first. If you omit the superclass constructor call, the superclass constructor with no arguments is invoked.

The constructor body can contain additional statements.
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 displaying and checking. Therefore, we can simply declare the parameter variable of the `presentQuestion` method to have the type `Question`:

```java
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:

```java
ChoiceQuestion second = new ChoiceQuestion();
...
presentQuestion(second); // OK to pass a ChoiceQuestion
```

When the `presentQuestion` method executes, the object references stored in `second` and `q` refer to the same object of type `ChoiceQuestion` (see Figure 7).

However, the variable `q` knows less than the full story about the object to which it refers (see Figure 8).
In the same way that vehicles can differ in their method of locomotion, polymorphic objects carry out tasks in different ways.

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.

---

**section_4/QuestionDemo3.java**

```java
import java.util.Scanner;

/**
  This program shows a simple quiz with two question types.
*/
```
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

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: R9.7, E9.7, P9.17.

---

**Dynamic Method Lookup and the Implicit Parameter**

Suppose we add the presentQuestion method to the Question class itself:

```java
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));
    }
}
```

Now consider the call

```java
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:

```java
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.
**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:

```java
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**:

```java
class Account {
    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.

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`:

```java
class Account {
    abstract void deductFees();
    /*...*/
}
```

```java
class SavingsAccount extends Account {
    /*...*/
    public void deductFees() { /* Provides an implementation */
    }
}
```

If a class extends an abstract class without providing an implementation of all abstract methods, it too is abstract.

```java
class BusinessAccount {
    /* No implementation of deductFees */
}
```

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

When you declare a method as abstract, you force programmers to provide implementations in subclasses. This is better than coming up with a default that might be inherited accidentally.

**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.

**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 (see Special Topic 8.4).

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:
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.

```java
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 will 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.

```java
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() { ... }
}
```

```java
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:

```java
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.

```java
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.

```java
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.

```java
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:

```java
public void monthEnd() {
}
```

In the withdraw method of the SavingsAccount class, the minimum balance is updated. Note the call to the superclass method:

```java
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.

```java
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:

```java
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 five checking accounts and five 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)eposit  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 == "Q")
    {
        done = true;
    }
}
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 9). 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).

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:

```java
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"
```
Figure 9  The Object Class Is the Superclass of Every Java Class

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:

```java
BankAccount momsSavings = new BankAccount(5000);
String s = momsSavings.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.)

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.

```java
public class BankAccount
{

    public String toString()
    {
        return "BankAccount[balance=\" + balance + "]\";
    }
}
```

This works better:

```java
BankAccount momsSavings = new BankAccount(5000);
String s = momsSavings.toString(); // Sets s to "BankAccount[balance=5000]"
```
9.5.2 The equals Method

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:

```java
if (stamp1.equals(stamp2)) . . .
   // Contents are the same—see Figure 10
```

This is different from the test with the == operator, which tests whether two references are identical, referring to the same object:

```java
if (stamp1 == stamp2) . . .
   // Objects are the same—see Figure 11
```

Let's implement the equals method for a Stamp class. You need to override the equals method of the Object class:

```java
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:

```java
Stamp other = (Stamp) otherObject;
```

![Figure 10](image1.png)  Two References to Equal Objects

![Figure 11](image2.png)  Two References to the Same Object
Then you can compare the two stamps:

```java
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.

### 9.5.3 The `instanceof` Operator

As you have seen, it is legal to store a subclass reference in a superclass variable:

```java
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:

```java
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,

```java
   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 9.4 The `instanceof` Operator

```
Syntax     object instanceof TypeName
```

If `anObject` is null, `instanceof` returns `false`.

Returns `true` if `anObject` can be cast to a `Question`.

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:

```java
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:

```java
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();`

### Self Check

21. Why does the call
```java
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?
```java
Object obj = "Hello";
System.out.println(obj.length());
```

23. Will the following code fragment compile? Will it run? If not, what error is reported?
```java
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: E9.10, E9.11, E9.15.

### Don't Use Type Tests

Some programmers use specific type tests in order to implement behavior that varies with each class:

```java
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:

```java
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

```java
q.doTheTask();
```

### 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:

```java
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.

```java
SavingsAccount momsSavings = . . . ;
System.out.println(momsSavings);
// 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.

```java
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.

```java
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:

```java
public boolean equals(Object otherObject) {
    if (getClass() != otherObject.getClass()) { return false; }
    ... // Compare instance variables
    return true;  // It is the same type
}
```

---

**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 computer users 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). 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 Andreesen, 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). Andreesen 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.
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.
- When the virtual machine calls an instance method, it locates the method of the implicit parameter’s class. This is called dynamic method lookup.
- Polymorphism (“having multiple forms”) allows us to manipulate objects that share a set of tasks, even though the tasks are executed in different ways.

Work with the `Object` class and its methods.

- 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.
**Review Exercises**

- **R9.1** In Worked Example 9.1,
  - a. What are the subclasses of Employee?
  - b. What are the superclasses of Manager?
  - c. What are the super- and subclasses of SalariedEmployee?
  - d. Which classes override the weeklyPay method of the Employee class?
  - e. Which classes override the setName method of the Employee class?
  - f. What are the instance variables of an HourlyEmployee object?

- **R9.2** 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

- **R9.3** 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?

- **R9.4** Which methods does the ChoiceQuestion class inherit from its superclass? Which methods does it override? Which methods does it add?

- **R9.5** 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?

- **R9.6** List the instance variables of a CheckingAccount object from How To 9.1.

- **R9.7** Suppose the class Sub extends the class Sandwich. Which of the following assignments are legal?

  ```java
  Sandwich x = new Sandwich();
  Sub y = new Sub();
  a. x = y;
  b. y = x;
  c. y = new Sandwich();
  d. x = new Sub();
  ```

- **R9.8** Draw an inheritance diagram that shows the inheritance relationships between these classes.
  - Person
  - Employee
  - Student
  - Instructor
  - Classroom
  - Object
  - PickupTruck
  - SportUtilityVehicle
  - Minivan
  - Bicycle
  - Motorcycle

- **R9.9** In an object-oriented traffic simulation system, we have the classes listed below. Draw an inheritance diagram that shows the relationships between these classes.
Chapter 9  Inheritance

• R9.10  What inheritance relationships would you establish among the following classes?
  - Student
  - Professor
  - TeachingAssistant
  - Employee
  - Secretary
  - DepartmentChair
  - Janitor
  - SeminarSpeaker
  - Person
  - Course
  - Seminar
  - Lecture
  - ComputerLab

• R9.11  How does a cast such as (BankAccount) x differ from a cast of number values such as (int) x?

•• R9.12  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.
  a. System.out instanceof PrintStream
  b. System.out instanceof OutputStream
  c. System.out instanceof LogStream
  d. System.out instanceof Object
  e. System.out instanceof String
  f. System.out instanceof Writer

PRACTICE EXERCISES

• E9.1  Implement a subclass of BankAccount called BasicAccount whose withdraw method will not withdraw more money than is currently in the account.

• E9.2  Implement a subclass of BankAccount called BasicAccount whose withdraw method charges a penalty of $30 for each withdrawal that results in an overdraft.

• E9.3  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.

• E9.4  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, accept the response as correct.

• E9.5  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 ______

• E9.6  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".

• E9.7  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.

• E9.8  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.

• E9.9  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.

• E9.10  Provide toString methods for the Question and ChoiceQuestion classes.

• 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 for 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 java.awt.Rectangle class of the standard Java library does not supply a method 
to compute the area or 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 Labeled­ 
Point 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.

**Business E9.17** Change the CheckingAccount class in How To 9.1 so that a $1 fee is levied for deposi 
ts 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.

**P9.1** Implement a class Clock whose getHours and getMinutes methods return the current 
time at your location. (Call java.time.Instant.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 over­ 
ride? (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 "⏰") and clear the alarm. What do you need to do to 
make the setAlarm method work for WorldClock objects?

**Business P9.3** 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.

**Business P9.4** Improve the appointment book program of Exercise P9.3. Give the user the option to add new appointments. The user must specify the type of the appointment, the description, and the date.

**Business P9.5** Improve the appointment book program of Exercises P9.3 and P9.4 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 P9.6** 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, \( \omega_0 \), the bandwidth, \( B \), and the gain at the resonant frequency, \( k \).

![Graph showing frequency response of a resonant circuit]

Two simple resonant circuits are shown in the figure below. 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 \( L \).

![Parallel and series resonant circuits]

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 \( \omega_0 \), \( B \), and \( k \). The design equations for the parallel resonant circuit are:

\[
R = k, \quad C = \frac{1}{BR}, \quad \text{and} \quad L = \frac{1}{\omega_0^2C}
\]

Similarly, the design equations for the series resonant circuit are:

\[
R = \frac{1}{k}, \quad L = \frac{R}{B}, \quad \text{and} \quad C = \frac{1}{\omega_0^2L}
\]
Write a Java program that represents ResonantCircuit as a superclass and represents SeriesResonantCircuit and ParallelResonantCircuit as subclasses. Give the superclass three private instance variables representing the parameters $\omega_o$, $B$, 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 $\omega_o$, $B$, 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.

---

**Science P9.7** 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:

![Parallel circuit](image)

Use Ohm’s law to compute the combined resistance.

**Science P9.8** 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.

![Amplifiers](image)

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.

Write a Java program that represents the amplifier as a superclass and represents the inverting, noninverting, and voltage divider amplifiers as subclasses. Give the superclass a getGain method 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.

**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. The classes `Frame`, `Window`, and `Component` in the `java.awt` package, and the class `Object` in the `java.lang` package.

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`
   
   
   ```java
   private double bonus;
   // Constructors and methods omitted
   }
   ```

8. `name`, `baseSalary`, and `bonus`

9. public class `Manager` extends `Employee`
   
   ```java
   . . .
   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()`
   
   ```java
   { return "*" + super.getName(); }
   ```

15. public `double` `getSalary()`
   
   ```java
   { return super.getSalary() + bonus; }
   ```

16. a only

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 superclass 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.
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.

**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:

```
Employee
   |
   v
HourlyEmployee
   |
   v
SalariedEmployee
   |
   v
Manager
```

**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.
```
**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.

```java
/**
 * 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.

```java
public class HourlyEmployee extends Employee
{
    /**
     * Constructs an hourly employee with a given name and weekly wage.
     */
    public HourlyEmployee(String name, double wage) { ... }

    public double weeklyPay(int hoursWorked) { ... }
}

public class SalariedEmployee extends Employee
{
    /**
     * Constructs a salaried employee with a given name and annual salary.
     */
    public SalariedEmployee(String name, double salary) { ... }

    public double weeklyPay(int hoursWorked) { ... }
}

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) { ... }

    public double weeklyPay(int hoursWorked) { ... }
}
```

These constructors need to set the name of the `Employee` object. We will add a method `setName` to the `Employee` class for this purpose:
Implementing an Employee Hierarchy for Payroll Processing

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:

```java
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:

```java
public Manager(String name, double salary, double bonus)
{
   super(name, salary)
   weeklyBonus = bonus;
}
The weekly pay needs to be computed as specified in the problem description:

```java
public class HourlyEmployee extends Employee {
    ...
    public double weeklyPay(int hoursWorked) {
        double pay = hoursWorked * hourlyWage;
        if (hoursWorked > 40) {
            // Add overtime
            pay = pay + ((hoursWorked - 40) * 0.5) * hourlyWage;
        }
        return pay;
    }
}
```

```java
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:

```java
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:

```java
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.
CHAPTER GOALS

To be able to declare and use interface types
To appreciate how interfaces can be used to decouple classes
To learn how to implement helper classes as inner classes
To implement event listeners in graphical applications

CHAPTER CONTENTS

10.1 USING INTERFACES FOR ALGORITHM REUSE 466
SYN Declaring an Interface 468
SYN Implementing an Interface 469
CE1 Forgetting to Declare Implementing Methods as Public 472
CE2 Trying to Instantiate an Interface 472
ST1 Constants in Interfaces 473
J81 Static Methods in Interfaces 473
J82 Default Methods 473
J83 Conflicting Default Methods 474

10.2 WORKING WITH INTERFACE TYPES 475
WE1 Investigating Number Sequences 

10.3 THE COMPARABLE INTERFACE 477
PT1 Comparing Integers and Floating-Point Numbers 478
ST2 The clone Method and the Cloneable Interface 479

10.4 USING INTERFACES FOR CALLBACKS 482
J84 Lambda Expressions 485
ST3 Generic Interface Types 486

10.5 INNER CLASSES 487
ST4 Anonymous Classes 488

10.6 MOCK OBJECTS 489

10.7 EVENT HANDLING 490
CE3 Modifying Parameter Types in the Implementing Method 495
CE4 Trying to Call Listener Methods 495
J85 Lambda Expressions for Event Handling 496

10.8 BUILDING APPLICATIONS WITH BUTTONS 496
CE5 Forgetting to Attach a Listener 498
PT2 Don’t Use a Container as a Listener 499

10.9 PROCESSING TIMER EVENTS 499
CE6 Forgetting to Repaint 502

10.10 MOUSE EVENTS 502
ST5 Keyboard Events 506
ST6 Event Adapters 506
C&S Open Source and Free Software 507
A mixer rotates any tools that will attach to its motor's shaft. In other words, a single motor can be used with multiple tools. We want to be able to reuse software components in the same way. In this chapter, you will learn an important strategy for separating the reusable part of a computation from the parts that vary in each reuse scenario. The reusable part invokes methods of an interface, not caring how the methods are implemented—just as the mixer doesn't care about the shape of the attachment. In a program, the reusable code is combined with a class that implements the interface methods. To produce a different application, you plug in another class that implements the same interface.

10.1 Using Interfaces for Algorithm Reuse

When you provide a service, you want to make it available to the largest possible set of clients. A restaurant serves people, and in Java, one might model this with a method

```
public void serve(Person client)
```

But what if the restaurant is willing to serve other creatures too? Then it makes sense to define a new type with exactly the methods that need to be invoked as the service processes an object. Such a type is called an interface type.

For example, a Customer interface type might have methods `eat` and `pay`. We can then redefine the service as

```
public void serve(Customer client)
```

If the Person and Cat classes conform to the interface, then you can pass objects of those classes to the `serve` method.

As a more practical example, you will study the `Comparable` interface type of the Java library. It has a method `compareTo` that determines which of two objects should come first in sorted order. It is then possible to implement a sorting service that accepts collections of many different classes. All that matters is that the classes conform to the `Comparable` interface. The sorting service doesn't care about anything other than the `compareTo` method, which it uses to arrange the objects in order.

In the following sections, you will learn how to discover when an interface type is useful, which methods it should require, how to define the interface, and how to define classes that conform to it.

10.1.1 Discovering an Interface Type

In this section, we will look at a service that computes averages, and we want to make it as general as possible. Let's start with one implementation of the service that computes the average balance of an array of bank accounts:
public static double average(BankAccount[] objects)

    double sum = 0;
    for (BankAccount obj : objects)
    {
        sum = sum + obj.getBalance();
    }
    if (objects.length > 0) { return sum / objects.length; }
    else { return 0; }

}

Now suppose we want to compute an average of other objects. We have to write that method again. Here it is for Country objects:

public static double average(Country[] objects)

    double sum = 0;
    for (Country obj : objects)
    {
        sum = sum + obj.getArea();
    }
    if (objects.length > 0) { return sum / objects.length; }
    else { return 0; }

}

Clearly, the algorithm for computing the average is the same in all cases, but the details of measurement differ. We would like to provide a single method that provides this service.

But there is a problem. Each class has a different name for the method that returns the value that is being averaged. In the BankAccount class, we call getBalance. In the Country class, we call getArea.

Suppose that the various classes agree on a 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, and so on.

Then we can implement a single method that computes

    sum = sum + obj.getMeasure();

But agreeing on the name of the method is only half the solution. In Java, we also must declare the type of the variable obj. Of course, you can’t write

    BankAccount or Country or . . . obj; // No

We need to invent a new type that describes any class whose objects can be measured. You will see how to do that in the next section.

### 10.1.2 Declaring an Interface Type

In Java, an interface type is used to specify required operations. The declaration is similar to the declaration of a class. You list the methods that the interface requires. However, you need not supply an implementation for the methods. For example, here is the declaration of an interface type that we call Measurable:

    public interface Measurable
    {
        double getMeasure();
    }

The Measurable interface type requires a single method, getMeasure. In general, an interface type can require multiple methods.
Chapter 10 Interfaces

Syntax 10.1 Declaring an Interface

```java
public interface InterfaceName
{
    method headers
}
```

An interface type is similar to a class, but there are several important differences:

- An interface type does not have instance variables.
- Methods in an interface must be `abstract` (that is, without an implementation) or as of Java 8, static, or default methods (see Java 8 Note 10.1 and Java 8 Note 10.2).
- All methods in an interface type are automatically public.
- An interface type has no constructor. Interfaces are not classes, and you cannot construct objects of an interface type.

Now that we have a type that denotes measurability, we can implement a reusable average method:

```java
public static double average(Measurable[] objects)
{
    double sum = 0;
    for (Measurable obj : objects)
    {
        sum = sum + obj.getMeasure();
    }
    if (objects.length > 0) { return sum / objects.length; } 
    else { return 0; }
}
```

This method is useful for objects of any class that conforms to the `Measurable` type. In the next section, you will see what a class must do to make its objects measurable.

Note that the `Measurable` interface is not a type in the standard library—it was created specifically for this book, to provide a very simple example for studying the interface concept.

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.
10.1.3 Implementing an Interface Type

The average method of the preceding section can process objects of any class that implements the Measurable interface. A class implements an interface type if it declares the interface in an implements clause, like this:

```java
public class BankAccount implements Measurable
```

The class should then implement the abstract method or methods that the interface requires:

```java
public class BankAccount implements Measurable
{
    public double getMeasure()
    {
        return balance;
    }
}
```

Note that the class must declare the method as public, whereas the interface need not—all methods in an interface are public.

Once the BankAccount class implements the Measurable interface type, BankAccount objects are instances of the Measurable type:

```java
Measurable obj = new BankAccount(); // OK
```

A variable of type Measurable holds a reference to an object of some class that implements the Measurable interface.

Similarly, it is an easy matter to modify the Country class to implement the Measurable interface:

```java
public class Country implements Measurable
{
    public double getMeasure()
    {
        return area;
    }
}
```

Syntax 10.2 Implementing an Interface

```
Syntax: public class ClassName implements InterfaceName, InterfaceName, ...
{
    instance variables
    methods
}
```

public class BankAccount implements Measurable
{
    public double getMeasure()
    {
        return balance;
    }
}

List all interface types that this class implements.

This method provides the implementation for the abstract method declared in the interface.
The program at the end of this section uses a single `average` method (placed in class `Data`) to compute the average of bank accounts and the average of countries.

This is a typical usage for interface types. By inventing the `Measurable` interface type, we have made the `average` method reusable.

Figure 1 shows the relationships between the `Data` class, the `Measurable` interface, and the classes that implement the interface. Note that the `Data` class depends only on the `Measurable` interface. It is decoupled from the `BankAccount` and `Country` classes.

In the UML notation, interfaces are tagged with an indicator «interface». A dotted arrow with a triangular tip (→→) denotes the `implements` relationship between a class and an interface. You have to look carefully at the arrow tips—a dotted line with an open arrow tip (→) denotes the `uses` relationship or dependency.

```java
public class Data {
    /**
     * Computes the average of the measures of the given objects.
     * @param objects an array of Measurable objects
     * @return the average of the measures
     */
    public static double average(Measurable[] objects) {
        double sum = 0;
        for (Measurable obj : objects) {
            sum = sum + obj.getMeasure();
        }
        if (objects.length > 0) { return sum / objects.length;}
        else { return 0; }
    }
}
```

```java
public class MeasurableTester {
    public static void main(String[] args) {
        
```
10.1.4 Comparing Interfaces and Inheritance

In Chapter 9, you saw how to use inheritance to model hierarchies of related classes, such as different kinds of quiz questions. Multiple-choice questions and fill-in questions are questions with specific characteristics.

Interfaces model a somewhat different relationship. Consider for example the BankAccount and Country classes in the preceding section. Both implement the Measurable interface type, but otherwise they have nothing in common. Being measurable is just one aspect of what it means to be a bank account or country. It is useful to model this common aspect, because it enables other programmers to write tools that exploit the commonality, such as the method for computing averages.

A class can implement more than one interface, for example

```java
public class Country implements Measurable, Named
```

Here, Named is a different interface:

```java
public interface Named
{
    String getName();
}
```

In contrast, a class can only extend (inherit from) a single superclass.

An interface describes the behavior that an implementation should supply. Prior to Java 8, an interface could not provide any implementation. Now, it is possible to supply a default implementation, and the distinction between interfaces and abstract classes (see Special Topic 9.3) has become more subtle. The significant difference that remains is that an interface type has no state (that is, no instance variables).
Generally, you will develop interfaces when you have code that processes objects of different classes in a common way. For example, a drawing program might have different objects that can be drawn, such as lines, images, text, and so on. In this situation, a `Drawable` interface with a `draw` method will be useful. Another example is a traffic simulation that models the movement of people, cars, dogs, balls, and so on. In this example, you might create a `Moveable` interface with methods `move` and `getPosition`.

1. Suppose you want to use the `average` method to find the average salary of an array of `Employee` objects. What condition must the `Employee` class fulfill?

2. Why can’t the `average` method have a parameter variable of type `Object[]`?

3. Why can’t you use the `average` method to find the average length of `String` objects?

4. What is wrong with this code?
   ```java
   Measurable meas = new Measurable();
   System.out.println(meas.getMeasure());
   ```

5. What is wrong with this code?
   ```java
   Measurable meas = new Country("Uruguay", 176220);
   System.out.println(meas.getName());
   ```

**Practice It** Now you can try these exercises at the end of the chapter: E10.1, E10.2, E10.3.

---

### 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—their default access level is “package” access, which we discussed in Chapter 8. It is a common error to forget the `public` reserved word when declaring a method from an interface:

```java
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. The remedy is to declare the method as `public`.

---

### Trying to Instantiate an Interface

You can declare variables whose type is an interface, for example:

```java
Measurable meas;
```

However, you can *never* construct an object of an interface type:

```java
Measurable meas = new Measurable(); // Error
```

Interfaces aren’t classes. There are no objects whose types are interfaces. If a variable of an interface type refers to an object, then the object must belong to some class—a class that implements the interface:

```java
Measurable meas = new BankAccount(); // OK
```
10.1 Using Interfaces for Algorithm Reuse

**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,

```java
public interface Named
{
    String NO_NAME = "(NONE)"
;
}
```

Now the constant `Named.NO_NAME` can be used to denote the absence of a name.

It is not very common to have constants in interface types. In particular, you should avoid multiple related constants (such as `int NORTH = 1`, `int NORTHEAST = 2`, and so on). In that case, use an enumerated type instead (see Special Topic 5.4).

**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 10.1 into the `Measurable` interface:

```java
public interface Measurable
{
    double getMeasure(); // An abstract method
    static double average(Measurable[] objects) // A static method
    {
        . . . // Same implementation as in Section 10.1
    }
}
```

To call this method, provide the name of the interface and the method name:

```java
double meanArea = Measurable.average(countries);
```

**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:

```java
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.

**Conflicting Default Methods**

It is possible (although quite rare) for a class to inherit conflicting default methods from two interfaces, or to inherit a default method that conflicts with one of its methods. There are two rules that deal with this possibility:

1. *Classes win.* When a class extends another class and implements an interface, both of which define the same method, the subclass inherits the superclass method and ignores the default method.
2. *Interfaces clash.* When a class implements two interfaces with the same default method, then the class must override the method and implement it.

To understand these rules, consider this example:

```java
public class Person
{
    public String name() { return firstName() + " " + lastName(); }
    ...
}

public interface Named
{
    default String name() { return "(NONE)"; }
}

public class User extends Person implements Named
{
    // Inherits Person.name()
    ...
}
```

In this situation, the method defined in the superclass wins over the method from the interface. However, the situation is different if `Person` is an interface:

```java
public interface Person
{
    default String name() { return firstName() + " " + lastName(); }
    ...
}
```

Suppose a class implements both interfaces:

```java
public class User implements Person, Named { ... }
```
This class must override the name method, in a way that is appropriate for the context. The details depend on why the designer of the User class chose to implement the Named interface. For example, suppose there is a method that checks a Named[] array for duplicates, and the program wants to call it to ensure that account names are unique. In that case, User.name should return the account name.

10.2 Working with Interface Types

In the preceding section, you saw how to implement a simple service that accepted an interface. As you saw, you were able to pass objects of different classes to the service, and the service was able to invoke a method of the interface. In the following sections, you will learn the rules for working with interface types in Java.

10.2.1 Converting from Classes to Interfaces

Have a close look at the call

```java
double averageBalance = Data.average(accounts);
```

from the program of the preceding section. Here, accounts is an array of BankAccount objects. However, the average method expects an array whose element type is Measurable:

```java
public double average(Measurable[] objects)
```

It is legal to convert from the BankAccount type to the Measurable type. In general, you can convert from a class type to the type of any interface that the class implements.

For example,

```java
BankAccount account = new BankAccount(1000);
Measurable meas = account; // OK
```

Alternatively, a Measurable variable can refer to an object of the Country class of the preceding section because that class also implements the Measurable interface.

```java
Country uruguay = new Country("Uruguay", 176220);
Measurable meas = uruguay; // Also OK
```

However, the Rectangle class from the standard library doesn’t implement the Measurable interface. Therefore, the following assignment is an error:

```java
Measurable meas = new Rectangle(5, 10, 20, 30); // Error
```

You can convert from a class type to an interface type, provided the class implements the interface.

**Figure 2** Variables of Class and Interface Types
10.2.2 Invoking Methods on Interface Variables

Now suppose that the variable `meas` has been initialized with a reference to an object of some class that implements the `Measurable` interface. You don’t know to which class that object belongs. But you do know that the class implements the methods of the interface type, and you can invoke them:

```java
double result = meas.getMeasure();
```

Now let’s think through the call to the `getMeasure` method more carefully. Which `getMeasure` method is called? The `BankAccount` and `Country` classes provide two different implementations of that method. How does the correct method get called if the caller doesn’t even know the exact class to which `meas` belongs?

This is again polymorphism in action. (See Section 9.4 for a discussion of polymorphism.) The Java virtual machine locates the correct method by first looking at the class of the actual object, and then calling the method with the given name in that class. That is, if `meas` refers to a `BankAccount` object, then the `BankAccount.getMeasure` method is called. If `meas` refers to a `Country` object, then the `Country.getMeasure` method is called.

10.2.3 Casting from Interfaces to Classes

Occasionally, it happens that you store an object in an interface reference and you need to convert its type back. Consider this method that returns the object with the larger measure:

```java
public static Measurable larger(Measurable obj1, Measurable obj2) {
    if (obj1.getMeasure() > obj2.getMeasure()) {
        return obj1;
    } else {
        return obj2;
    }
}
```

The `larger` method returns the object with the larger measure, as a `Measurable` reference. It has no choice—it does not know the exact type of the object. Let’s use the method:

```java
Country uruguay = new Country("Uruguay", 176220);
Country thailand = new Country("Thailand", 513120);
Measurable max = larger(uruguay, thailand);
```

Now what can you do with the `max` reference? You know it refers to a `Country` object, but the compiler doesn’t. For example, you cannot call the `getName` method:

```java
String countryName = max.getName(); // Error
```
That call is an error, because the `Measurable` type has no `getName` method. However, as long as you are absolutely sure that `max` refers to a `Country` object, you can use the `cast` notation to convert its type back:

```java
Country maxCountry = (Country) max;
String name = maxCountry.getName();
```

If you are wrong, and the object doesn’t actually refer to a country, a run-time exception will occur.

6. Can you use a cast `(BankAccount) meas` to convert a `Measurable` variable `meas` to a `BankAccount` reference?

7. If both `BankAccount` and `Country` implement the `Measurable` interface, can a `Country` reference be converted to a `BankAccount` reference?

8. Why is it impossible to construct a `Measurable` object?

9. Why can you nevertheless declare a variable whose type is `Measurable`?

10. What does this code fragment print? Why is this an example of polymorphism?

```java
Measurable[] data = { new BankAccount(10000), new Country("Belgium", 30510) };
System.out.println(average(data));
```

**Practice It**
Now you can try these exercises at the end of the chapter: R10.3, R10.4, R10.5.

### 10.3 The Comparable Interface

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

```java
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 if `b` should come before `a`. 

© Janis Dreosti/iStockphoto.
The Comparable interface has a single method:

```java
public interface Comparable
{
    int compareTo(Object otherObject);
}
```

For example, the BankAccount class can implement Comparable like this:

```java
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:

```java
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:

```java
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.

11. How can you sort an array of Country objects by increasing area?
12. Can you use the Arrays.sort method to sort an array of String objects? Check the API documentation for the String class.
13. Can you use the Arrays.sort method to sort an array of Rectangle objects? Check the API documentation for the Rectangle class.
14. Write a method max that finds the larger of any two Comparable objects.
15. Write a call to the method of Self Check 14 that computes the larger of two bank accounts, then prints its balance.

Practice It Now you can try these exercises at the end of the chapter: E10.10, E10.30.

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:

```java
public class Person implements Comparable
{
    ...
}
```
10.3 The Comparable Interface

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 R10.1). However, the `Integer.compare` method always works:

```java
return Integer.compare(id, other.id); // Safe for negative integers
```

You cannot compare floating-point values by subtraction (see Exercise R10.2). Instead, use the `Double.compare` method:

```java
public class BankAccount implements Comparable
{
...
    public int compareTo(Object otherObject)
    {
        BankAccount other = (BankAccount) otherObject;
        return Double.compare(balance, other.balance);
    }
}
```

The `clone` Method and the Cloneable Interface

You know that copying an object reference simply gives you two references to the same object:

```java
BankAccount account = new BankAccount(1000);
BankAccount account2 = account;
account2.deposit(500); // Now both account and account2 refer to a bank account with a balance of 1500
```

What can you do if you actually want to make a copy of an object? That is the purpose of the `clone` method. The `clone` method must return a new object that has an identical state to the existing object (see Figure 4).

Here is how to call it:

```java
BankAccount clonedAccount = (BankAccount) account.clone();
```

The return type of the `clone` method is the class `Object`. When you call the method, you must use a cast to inform the compiler that `account.clone()` really returns a `BankAccount` object.

![Figure 4](https://www.ericisselle/iStockphoto)

Cloning Objects
The `Object.clone` method is the starting point for the `clone` methods in your own classes. It creates a new object of the same type as the original object. It also automatically copies the instance variables from the original object to the cloned object. Here is a first attempt to implement the `clone` method for the `BankAccount` class:

```java
public class BankAccount
{
    ...
    public Object clone()
    {
        // Not complete
        Object clonedAccount = super.clone();
        return clonedAccount;
    }
}
```

However, this `Object.clone` method must be used with care. It only shifts the problem of cloning by one level; it does not completely solve it. Specifically, if an object contains a reference to another object, then the `Object.clone` method makes a copy of that object reference, not a clone of that object. Figure 5 shows how the `Object.clone` method works with a `Customer` object that has references to a `String` object and a `BankAccount` object. As you can see, the `Object.clone` method copies the references to the cloned `Customer` object and does not clone the objects to which they refer. Such a copy is called a shallow copy.

There is a reason why the `Object.clone` method does not systematically clone all sub-objects. In some situations, it is unnecessary. For example, if an object contains a reference to a string, there is no harm in copying the string reference, because Java string objects can never change their contents. The `Object.clone` method does the right thing if an object contains only numbers, Boolean values, and strings. But it must be used with caution when an object contains references to mutable objects.

For that reason, there are two safeguards built into the `Object.clone` method to ensure that it is not used accidentally. First, the method is declared `protected` (see Special Topic 9.5). This prevents you from accidentally calling `x.clone()` if the class to which `x` belongs hasn’t declared `clone` to be public.

As a second precaution, `Object.clone` checks that the object being cloned implements the `Cloneable` interface. If not, it throws an exception. The `Object.clone` method looks like this:

```java
public class Object
{
    protected Object clone() throws CloneNotSupportedException
    {
        if (this instanceof Cloneable)
        {
```
// Copy the instance variables
.
.
else
{
    throw new CloneNotSupportedException();
}
}

Unfortunately, all that safeguarding means that the legitimate callers of `Object.clone()` pay a price—they must catch that exception (see Chapter 11) even if their class implements `Cloneable`.

```java
public class BankAccount implements Cloneable
{
    public Object clone()
    {
        try
        {
            return super.clone();
        }
        catch (CloneNotSupportedException e)
        {
            // Can't happen because we implement Cloneable but we still must catch it.
            return null;
        }
    }
}
```

If an object contains a reference to another mutable object, then you must call `clone` for that reference. For example, suppose the `Customer` class has an instance variable of class `BankAccount`. You can implement `Customer.clone` as follows:

```java
public class Customer implements Cloneable
{
    private String name;
    private BankAccount account;
    
    public Object clone()
    {
        try
        {
            Customer cloned = (Customer) super.clone();
            cloned.account = (BankAccount) account.clone();
            return cloned;
        }
        catch(CloneNotSupportedException e)
        {
            // Can't happen because we implement Cloneable return null;
        }
    }
}
```

In general, implementing the `clone` method requires these steps:

- Make the class implement the `Cloneable` interface type.
- In the `clone` method, call `super.clone()`. Catch the `CloneNotSupportedException` if the superclass is `Object`.
- Clone any mutable instance variables.
In this section, we introduce the notion of a **callback**, show how it leads to a more flexible **average** method, and study how a callback can be implemented in Java by using interface types.

To understand why a further improvement to the **average** method is desirable, consider these limitations of the **Measurable** interface:

- You can add the **Measurable** interface only to classes under your control. If you want to process a set of **Rectangle** objects, you cannot make the **Rectangle** class implement another interface—it is a library class, which you cannot change.
- You can measure an object in only one way. If you want to analyze a set of cars both by speed and price, you are stuck.

Therefore, let’s rethink the **average** method. The method measures objects, requiring them to be of type **Measurable**. The responsibility for measuring lies with the objects themselves. That is the cause for the limitations.

It would be better if we could give the **average** method the data to be averaged, and separately a method for measuring the objects. When collecting rectangles, we might give it a method for computing the area of a rectangle. When collecting cars, we might give it a method for getting the car’s price.

Such a method is called a **callback**. A callback is a mechanism for bundling up a block of code so that it can be invoked at a later time.

In some programming languages, it is possible to specify callbacks directly, as blocks of code or names of methods. But Java is an object-oriented programming language. Therefore, you turn callbacks into objects. This process starts by declaring an interface for the callback:

```java
public interface Measurer
{
    double measure(Object anObject);
}
```

The `measure` method measures an object and returns its measurement. Here we use the fact that all objects can be converted to the type `Object`.

The code that makes the call to the callback receives an object of a class that implements this interface. In our case, the improved **average** method receives a **Measurer** object.

```java
public static double average(Object[] objects, Measurer meas)
{
    double sum = 0;
    for (Object obj : objects)
    {
        sum = sum + meas.measure(obj);
    }
    if (objects.length > 0) { return sum / objects.length; } 
    else { return 0; }
}
```

The **average** method simply makes a callback to the **measure** method whenever it needs to measure any object.

Finally, a specific callback is obtained by implementing the **Measurer** interface. For example, here is how you can measure rectangles by area. Provide a class...
Using Interfaces for Callbacks

Figure 6
UML Diagram of the Data Class and the Measurer Interface

public class AreaMeasurer implements Measurer
{
    public double measure(Object anObject)
    {
        Rectangle aRectangle = (Rectangle) anObject;
        double area = aRectangle.getWidth() * aRectangle.getHeight();
        return area;
    }
}

Note that the measure method has a parameter variable of type Object, even though this particular measurer just wants to measure rectangles. The method parameter types must match those of the measure method in the Measurer interface. Therefore, the anObject parameter variable is cast to the Rectangle type:

    Rectangle aRectangle = (Rectangle) anObject;

What can you do with an AreaMeasurer? You need it to compute the average area of rectangles. Construct an object of the AreaMeasurer class and pass it to the average method:

    Measurer areaMeas = new AreaMeasurer();
    Rectangle[] rects = { new Rectangle(5, 10, 20, 30), new Rectangle(10, 20, 30, 40) };
    double averageArea = average(rects, areaMeas);

The average method will ask the AreaMeasurer object to measure the rectangles.

Figure 6 shows the UML diagram of the classes and interfaces of this solution. As in Figure 1, the Data class (which holds the average method) is decoupled from the class whose objects it processes (Rectangle). However, unlike in Figure 1, the Rectangle class is no longer coupled with another class. Instead, to process rectangles, you provide a small “helper” class AreaMeasurer. This helper class has only one purpose: to tell the average method how to measure its objects.

Here is the complete program:

section_4/Measurer.java

```java
/**
   * Describes any class whose objects can measure other objects.
   */
public interface Measurer
{
    /**
     * Computes the measure of an object.
     */
    public double measure(Object anObject);
}
```
Section 4/MeasurerTester.java

```java
import java.awt.Rectangle;

/**
 * This program demonstrates the use of a Measurer.
 */
public class MeasurerTester
{
    public static void main(String[] args)
    {
        Measurer areaMeas = new AreaMeasurer();
        Rectangle[] rects = new Rectangle[] {
```
16. Suppose you want to use the `average` method of Section 10.1 to find the average length of `String` objects. Why can't this work?

17. How can you use the `average` method of this section to find the average length of `String` objects?

18. Why does the `measure` method of the `Measurer` interface have one more argument than the `getMeasure` method of the `Measurable` interface?

19. Write a method `max` with three arguments that finds the larger of any two objects, using a `Measurer` to compare them.

20. Write a call to the method of Self Check 19 that computes the larger of two rectangles, then prints its width and height.

**Practice It**

Now you can try these exercises at the end of the chapter: R10.9, E10.8, E10.9.

**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:

```java
(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 (λ) 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:

```java
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.

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:

```java
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`:

```java
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:

```java
Measurer areaMeas = (Object obj) -> {
    Rectangle r = (Rectangle) obj;
    return r.getWidth() * r.getHeight();
};
```

Conceptually, lambda expressions are easiest to understand as a convenient notation for callbacks. Consider any method that needs to call some code that varies from one call to the next. 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 (Section 10.5) and comparators (Section 14.8). Chapter 19 uses lambda expressions extensively for processing complex data. In that chapter, we will study lambda expressions in greater depth.

**Generic Interface Types**

In Section 10.3, 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:

```java
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:

```java
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:

```java
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:

```java
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.

### 10.5 Inner Classes

The `AreaMeasurer` class of the preceding section is a very trivial class. We need this class only because the `average` method needs an object of some class that implements the `Measurer` interface. When you have a class that serves a very limited purpose, such as this one, you can declare the class inside the method that needs it:

```java
public class MeasurerTester
{
    public static void main(String[] args)
    {
        class AreaMeasurer implements Measurer
        {
            ... public double measure(Rectangle anObject)
            {
                double area = anObject.getWidth() * anObject.getHeight();
                return area;
            }
        }

        Measurer areaMeas = new AreaMeasurer();
        double averageArea = Data.average(rects, areaMeas);
    }
}
```
A class that is declared inside another class, such as the `AreaMeasurer` class in this example, is called an **inner class**. This arrangement signals to the reader of your program that the `AreaMeasurer` class is not interesting beyond the scope of this method. Because an inner class inside a method is not a publicly accessible feature, you don’t need to document it as thoroughly.

You can also declare an inner class inside an enclosing class, but outside of its methods. Then the inner class is available to all methods of the enclosing class.

```java
public class MeasurerTester
{
    class AreaMeasurer implements Measurer
    {
        . . .
    }

    public static void main(String[] args)
    {
        . . .
        Measurer areaMeas = new AreaMeasurer();
        double averageArea = Data.average(rects, areaMeas);
        . . .
    }
}
```

When you compile the source files for a program that uses inner classes, have a look at the class files in your program directory—you will find that the inner classes are stored in files with curious names, such as `MeasurerTester$1AreaMeasurer.class`. The exact names aren’t important. The point is that the compiler turns an inner class into a regular class file.

21. Why would you use an inner class instead of a regular class?
22. When would you place an inner class inside a class but outside any methods?
23. How many class files are produced when you compile the `MeasurerTester` program from this section?

**Practice It**
Now you can try these exercises at the end of the chapter: E10.11, E10.13.

### Anonymous 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

```java
Country belgium = new Country("Belgium", 30510);
countries.add(belgium);
```

with

```java
countries.add(new Country("Belgium", 30510));
```

if the country is not used elsewhere in the same method. The object `new Country("Belgium", 30510)` 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 `coin`, `dime`, or `aCoin`, you’ll understand this sentiment.

Inner classes often give rise to a similar situation. After a single object of the `AreaMeasurer` has been constructed, 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.
public static void main(String[] args) {
    // Construct an object of an anonymous class
    Measurer m = new Measurer()
    // Class declaration starts here
    {
        public double measure(Object anObject)
        {
            Rectangle aRectangle = (Rectangle) anObject;
            return aRectangle.getWidth() * aRectangle.getHeight();
        }
    }

    double result = Data.average(rectangles, m);
    .
    .
}

This means: Construct an object of a class that implements the Measurer interface by declaring the measure method as specified. This style was popular before lambda expressions were added to Java 8. Nowadays, it is simpler to use a lambda expression. We do not use anonymous classes in this book.

10.6 Mock Objects

When you work on a program that consists of multiple classes, you often want to test some of the classes before the entire program has been completed. A very effective technique for this purpose is the use of mock objects. A mock object provides the same services as another object, but in a simplified manner.

Consider a grade book application that manages quiz scores for students. This calls for a class GradeBook with methods such as

public void addScore(int studentId, double score)
public double getAverageScore(int studentId)
public void save(String filename)

Now consider the class GradingProgram that manipulates a GradeBook object. That class calls the methods of the GradeBook class. We would like to test the GradingProgram class without having a fully functional GradeBook class.

To make this work, declare an interface type with the same methods that the GradeBook class provides. A common convention is to use the letter I as the prefix for such an interface:

public interface IGradeBook
{
    void addScore(int studentId, double score);
    double getAverageScore(int studentId);
    void save(String filename);
    .
    .
}

The GradingProgram class should only use this interface, never the GradeBook class. Of course, the GradeBook class will implement this interface, but as already mentioned, it may not be ready for some time.

In the meantime, provide a mock implementation that makes some simplifying assumptions. Saving is not actually necessary for testing the user interface. We can also temporarily restrict it to the case of a single student.
public class MockGradeBook implements IGradeBook
{
    private ArrayList<Double> scores;

    public MockGradeBook() { scores = new ArrayList<Double>(); }

    public void addScore(int studentId, double score)
    {
        // Ignore studentId
        scores.add(score);
    }

    public double getAverageScore(int studentId)
    {
        double total = 0;
        for (double x : scores) { total = total + x; }
        return total / scores.size();
    }

    public void save(String filename)
    {
        // Do nothing
    }
}

Now construct an instance of MockGradeBook and use it in the GradingProgram class. You can immediately test the GradingProgram class. When you are ready to test the actual class, simply use a GradeBook instance instead. Don’t erase the mock class—it will still come in handy for regression testing.

24. Why is it necessary that the real class and the mock class implement the same interface type?

25. Why is the technique of mock objects particularly effective when the GradeBook and GradingProgram class are developed by two programmers?

Practice It Now you can try these exercises at the end of the chapter: P10.19, P10.20.

10.7 Event Handling

This and the following sections continue the book’s graphics track. You will learn how interfaces are used when programming graphical user interfaces.

In the applications that you have written so far, user input was under control of the program. The program asked 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 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, such as menu selections and mouse clicks. The Java windowing toolkit has a sophisticated mechanism that allows a program to specify the events in which it is interested and which objects to notify when one of these events occurs.

### 10.7.1 Listening to Events

Whenever the user of a graphical program types characters or uses the mouse anywhere inside one of the windows of the program, the Java windowing toolkit sends a notification to the program that an event has occurred. The windowing toolkit generates huge numbers of events. For example, whenever the mouse moves a tiny interval over a window, a “mouse move” event is generated. Whenever the mouse button is clicked, “mouse pressed” and “mouse released” events are generated. In addition, higher-level events are generated when a user selects a menu item or button.

Most programs don’t want to be flooded by irrelevant events. For example, consider what happens when selecting a menu item with the mouse. The mouse moves over the menu item, then the mouse button is pressed, and finally the mouse button is released. Rather than receiving all these mouse events, a program can indicate that it only cares about menu selections, not about the underlying mouse events. However, if the mouse input is used for drawing shapes on a virtual canvas, it is necessary to closely track mouse events.

Every program must indicate which events it needs to receive. It does that by installing event listener objects. An event listener object belongs to a class that you 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 that generates a particular event. You add an event listener object to the appropriate event sources. Whenever the event occurs, the event source notifies all event listeners.

This sounds somewhat abstract, so let’s run through an extremely simple program that prints a message whenever a button is clicked (see Figure 7). Button listeners must belong to a class that implements the ActionListener interface:

```java
public interface ActionListener
{
    void actionPerformed(ActionEvent event);
}
```

![Figure 7 Implementing an Action Listener](image_url)
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:

```java
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;

/**
 * An action listener that prints a message.
 */
public class ClickListener implements ActionListener {
    public void actionPerformed(ActionEvent event) {
        System.out.println("I was clicked.");
    }
}
```

We ignore the values of the `event` parameter variable of the `actionPerformed` method—it contains additional details about the event, such as the time at which it occurred.

Once the listener class has been declared, we need to construct an object of the class and add it to the button:

```java
ActionListener listener = new ClickListener();
button.addActionListener(listener);
```

Whenever the button is clicked, it calls

```java
listener.actionPerformed(event);
```

As a result, the message is printed.

You can think of the `actionPerformed` method as another example of a callback, similar to the `measure` method of the `Measurer` class. The windowing toolkit calls the `actionPerformed` method whenever the button is pressed, whereas the `Data` class calls the `measure` method whenever it needs to measure an object.

The `ButtonViewer` class, shown below, constructs a frame with a button and adds a `ClickListener` to the button. You can test this program out by opening a console window, starting the `ButtonViewer` program from that console window, clicking the button, and watching the messages in the console window.

```java
import java.awt.event.ActionListener;
import javax.swing.JButton;
import javax.swing.JFrame;

/**
 * This program demonstrates how to install an action listener.
 */
public class ButtonViewer {
    private static final int FRAME_WIDTH = 100;
    private static final int FRAME_HEIGHT = 60;

    public static void main(String[] args) {
        JFrame frame = new JFrame("ButtonViewer");
        JButton button = new JButton("Click me!");
        ButtonViewer listener = new ButtonViewer();
        button.addActionListener(listener);
        frame.add(button);
        frame.setSize(FRAME_WIDTH, FRAME_HEIGHT);
        frame.setVisible(true);
    }
}
```
10.7.2 Using Inner Classes for Listeners

In the preceding section, you saw how the code to be executed when a button is clicked is placed into a listener class. It is common to implement listener classes as inner classes like this:

```java
JButton button = new JButton("Click me!");
frame.add(button);

ActionListener listener = new ClickListener();
button.addActionListener(listener);
```

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 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. Suppose we want to add interest to a bank account whenever a button is clicked.

```java
JButton button = new JButton("Add Interest");
final BankAccount account = new BankAccount(INITIAL_BALANCE);

// This inner class is declared in the same method as the account and button variables.
class AddInterestListener implements ActionListener
{
    public void actionPerformed(ActionEvent event)
    {
        // The listener method accesses the account variable from the surrounding block
        double interest = account.getBalance() * INTEREST_RATE / 100;
        account.deposit(interest);
    }
};
```

```java
ActionListener listener = new AddInterestListener();
button.addActionListener(listener);
```
There is a technical wrinkle. In versions of Java before Java 8, an inner class can access a surrounding local variable only if the variable is declared as `final`. As of Java 8, the variable must only 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. In our example, the `account` variable always refers to the same bank account, so it is legal to access it from the inner class. For the benefit of users of Java 7 or earlier, we declare it as `final`.

An inner class can also access `instance` variables of the surrounding class, again with a restriction. The instance variable must belong to the object that constructed the inner class object. If the inner class object was created inside a static method, it can only access static variables.

Here is the source code for the program:

```java
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import javax.swing.JButton;
import javax.swing.JFrame;

/**
 * This program demonstrates how an action listener can access
 * a variable from a surrounding block.
 */

public class InvestmentViewer1
{
    private static final int FRAME_WIDTH = 120;
    private static final int FRAME_HEIGHT = 60;

    private static final double INTEREST_RATE = 10;
    private static final double INITIAL_BALANCE = 1000;

    public static void main(String[] args)
    {
        JFrame frame = new JFrame();

        // The button to trigger the calculation
        JButton button = new JButton("Add Interest");
        frame.add(button);

        // The application adds interest to this bank account
        final BankAccount account = new BankAccount(INITIAL_BALANCE);

        class AddInterestListener implements ActionListener
        {
            public void actionPerformed(ActionEvent event)
            {
                // The listener method accesses the account variable
                // from the surrounding block
                double interest = account.getBalance() * INTEREST_RATE / 100;
                account.deposit(interest);
                System.out.println("balance: " + account.getBalance());
            }
        }

        ActionListener listener = new AddInterestListener();
        button.addActionListener(listener);
    }
}
```
26. Which objects are the event source and the event listener in the ButtonViewer program?

27. Why is it legal to assign a ClickListener object to a variable of type ActionListener?

28. When do you call the actionPerformed method?

29. Why would an inner class method want to access a variable from a surrounding scope?

30. If an inner class accesses a local variable from a surrounding scope, what special rule applies?

Practice It Now you can try these exercises at the end of the chapter: R10.16, R10.22, E10.17.

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 types is a common error. Here is the classic example:

```java
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 and return types to find your error.

Trying to Call Listener Methods

Some students try to call the listener methods themselves:

```java
ActionEvent event = new ActionEvent(); // Don't do this
listener.actionPerformed(event);
```

You should not call the listener. The Java user interface calls it when the program user has clicked a button.
Lambda Expressions for Event Handling

Java 8 Note 10.4 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:

```java
button.addActionListener(
    (ActionEvent event) -> System.out.println("I was clicked.");
```

10.8 Building Applications with Buttons

In this section, you will learn how to structure a graphical application that contains buttons. We will put a button to work in our simple investment viewer program. Whenever the button is clicked, interest is added to a bank account, and the new balance is displayed (see Figure 8).

```java
First, we construct an object of the JButton class, passing the button label to the constructor, like this:

```java
JButton button = new JButton("Add Interest");
```

We also need a user-interface component that displays a message, namely the current bank balance. Such a component is called a label. You pass the initial message string to the JLabel constructor, like this:

```java
JLabel label = new JLabel("balance: " + account.getBalance());
```

The frame of our application contains both the button and the label. However, we cannot simply add both components directly to the frame—they would be placed on top of each other. The solution is to put them into a panel, a container for other user-interface components, and then add the panel to the frame:

```java
JPanel panel = new JPanel();
panel.add(button);
panel.add(label);
frame.add(panel);
```

Now we are ready for the hard part—the event listener that handles button clicks. As in the preceding section, it is necessary to provide a class that implements the ActionListener interface, and to place the button action into the actionPerformed method. Our listener class adds interest to the account and displays the new balance:

```java
class AddInterestListener implements ActionListener
{
```
public void actionPerformed(ActionEvent event)
{
    double interest = account.getBalance() * INTEREST_RATE / 100;
    account.deposit(interest);
    label.setText("balance: " + account.getBalance());
}

There is just a minor technicality. The actionPerformed method manipulates the account
and label variables. These are local variables of the main method of the investment
viewer program, not instance variables of the AddInterestListener class. In versions
prior to Java 8, the account and label variables need to be declared as final so that the
actionPerformed method can access them.

Let’s put the pieces together:

public static void main(String[] args)
{
    JButton button = new JButton("Add Interest");
    final BankAccount account = new BankAccount(INITIAL_BALANCE);
    final JLabel label = new JLabel("balance: " + account.getBalance());

    class AddInterestListener implements ActionListener
    {
        public void actionPerformed(ActionEvent event)
        {
            double interest = account.getBalance() * INTEREST_RATE / 100;
            account.deposit(interest);
            label.setText("balance: " + account.getBalance());
        }
    }
    ActionListener listener = new AddInterestListener();
    button.addActionListener(listener);
}

With a bit of practice, you will learn to glance at this code and translate it into plain
English: “When the button is clicked, add interest and set the label text.”

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.

section_8/InvestmentViewer2.java

```java
/*
 * This program displays the growth of an investment.
 */
public class InvestmentViewer2 {
    private static final int FRAME_WIDTH = 400;
    private static final int FRAME_HEIGHT = 100;
    ...
private static final double INTEREST_RATE = 10;
private static final double INITIAL_BALANCE = 1000;

public static void main(String[] args) {
    JFrame frame = new JFrame();
    // The button to trigger the calculation
    JButton button = new JButton("Add Interest");
    // The application adds interest to this bank account
    final BankAccount account = new BankAccount(INITIAL_BALANCE);
    // The label for displaying the results
    final JLabel label = new JLabel("balance: " + account.getBalance());
    // The panel that holds the user-interface components
    JPanel panel = new JPanel();
    panel.add(button);
    panel.add(label);
    frame.add(panel);

    class AddInterestListener implements ActionListener {
        public void actionPerformed(ActionEvent event) {
            double interest = account.getBalance() * INTEREST_RATE / 100;
            account.deposit(interest);
            label.setText("balance: " + account.getBalance());
        }
    }
    ActionListener listener = new AddInterestListener();
    button.addActionListener(listener);
    frame.setSize(FRAME_WIDTH, FRAME_HEIGHT);
    frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    frame.setVisible(true);
}

31. How do you place the "balance: . . ." message to the left of the "Add Interest" button?

32. Why was it not necessary to declare the button variable as final?

Practice It
Now you can try these exercises at the end of the chapter: E10.18, E10.19, E10.20.

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.
Don't Use a Container 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 instead turn a container (such as a panel or frame) into a listener. Here is a typical example. The actionPerformed method is added to the viewer class. That is, the viewer implements the ActionListener interface.

```java
public class InvestmentViewer
    implements ActionListener // This approach is not recommended
{
    public InvestmentViewer()
    {
        JButton button = new JButton("Add Interest");
        button.addActionListener(this);
        . . .
    }

    public void actionPerformed(ActionEvent event)
    {
        . . .
    }
}
```

Now the actionPerformed method is a part of the InvestmentViewer class rather than part of a separate listener class. The listener is installed as this.

This technique has two major flaws. First, it separates the button declaration from the button action. Also, it doesn’t scale well. 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.

10.9 Processing Timer Events

In this section we will study 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 have an object updated at regular intervals. 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.

```java
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.

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.

section_9/RectangleComponent.java

```java
import java.awt.Graphics;
import java.awt.Graphics2D;
import java.awt.Rectangle;
import javax.swing.JComponent;

/**
 * This component displays a rectangle that can be moved.
 */
public class RectangleComponent extends JComponent {
    private static final int BOX_X = 100;
    private static final int BOX_Y = 100;
    private static final int BOX_WIDTH = 20;
    private static final int BOX_HEIGHT = 30;
    private Rectangle box;

    public RectangleComponent() {
        // The rectangle that the paintComponent method draws
        box = new Rectangle(BOX_X, BOX_Y, BOX_WIDTH, BOX_HEIGHT);
    }

    public void paintComponent(Graphics g) {
        Graphics2D g2 = (Graphics2D) g;
        g2.draw(box);
    }

    /**
     * Moves the rectangle by a given amount.
     * @param dx the amount to move in the x-direction
     * @param dy the amount to move in the y-direction
     */
    public void moveRectangleBy(int dx, int dy) {
        box.translate(dx, dy);
        repaint();
    }
}
```
Note the call to repaint in the moveRectangleBy method. This call is necessary to ensure that the component is repainted after the state of the rectangle object has been changed. Keep in mind that the component object does not contain the pixels that show the drawing. The component merely contains a Rectangle object, which itself contains four coordinate values. Calling translate updates the rectangle coordinate values. 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 simply calls component.moveBy(1, 1). This moves the rectangle one pixel down and to the right. Because the actionPerformed method is called many times per second, the rectangle appears to move smoothly across the frame.

section_9/RectangleFrame.java

```java
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import javax.swing.JFrame;
import javax.swing.Timer;

/**
 * This frame contains a moving rectangle.
 */
public class RectangleFrame extends JFrame
{
    private static final int FRAME_WIDTH = 300;
    private static final int FRAME_HEIGHT = 400;

    private RectangleComponent scene;

    class TimerListener implements ActionListener
    {
        public void actionPerformed(ActionEvent event)
        {
            scene.moveRectangleBy(1, 1);
        }
    }

    public RectangleFrame()
    {
        scene = new RectangleComponent();
        add(scene);
        setSize(FRAME_WIDTH, FRAME_HEIGHT);
        ActionListener listener = new TimerListener();

        final int DELAY = 100; // Milliseconds between timer ticks
        Timer t = new Timer(DELAY, listener);
        t.start();
    }
}
```

section_9/RectangleViewer.java

```java
import javax.swing.JFrame;

/**
 */
```
This program moves the rectangle.

```java
public class RectangleViewer
{
    public static void main(String[] args)
    {
        JFrame frame = new RectangleFrame();
        frame.setTitle("An animated rectangle");
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.setVisible(true);
    }
}
```

### 33. Why does a timer require a listener object?

### 34. What would happen if you omitted the call to `repaint` in the `moveBy` method?

**Practice It**

Now you can try these exercises at the end of the chapter: E10.27, E10.28.

### Forgetting to Repaint

You have to be careful when your event handlers change the data in a painted component. When you make a change to the data, the component is not automatically painted with the new data. You must call the `repaint` method of the component, either in the event handler or in the component's mutator methods. Your component's `paintComponent` method will then be invoked with an appropriate `Graphics` object. Note that you should not call the `paintComponent` method directly.

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.

### 10.10 Mouse Events

If you write programs that show drawings, and you want users to manipulate the drawings with a mouse, then you need to process mouse events. Mouse events are more complex than button clicks or timer ticks.

A mouse listener must implement the `MouseListener` interface, which contains the following five methods:

```java
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 paint a user-interface component in a special way 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:

```java
public class MyMouseListener implements MouseListener
{
   // Implements five methods
}

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 first enhance the `RectangleComponent` class and add a `moveRectangleTo` method to move the rectangle to a new position.

```java
import java.awt.Graphics;
import java.awt.Graphics2D;
import java.awt.Rectangle;
import javax.swing.JComponent;

public class RectangleComponent2 extends JComponent
{
   private static final int BOX_X = 100;
   private static final int BOX_Y = 100;
   private static final int BOX_WIDTH = 20;
   private static final int BOX_HEIGHT = 30;

   private Rectangle box;

   public RectangleComponent2()
   {
      // The rectangle that the paintComponent method draws
      box = new Rectangle(BOX_X, BOX_Y, BOX_WIDTH, BOX_HEIGHT);
   }

   public void paintComponent(Graphics g)
   {
      Graphics2D g2 = (Graphics2D) g;
      g2.draw(box);
   }
}
Moves the rectangle to the given location.
@param x the x-position of the new location
@param y the y-position of the new location
*/
public void moveRectangleTo(int x, int y)
{
    box.setLocation(x, y);
    repaint();
}

Note the call to repaint in the moveRectangleTo method. As explained in the preceding section, 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();
        component.moveRectangleTo(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) {}
}

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 9).
private RectangleComponent2 scene;

class MousePressListener implements MouseListener
{
    public void mousePressed(MouseEvent event)
    {
        int x = event.getX();
        int y = event.getY();
        scene.moveRectangleTo(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) {}
}

public RectangleFrame2()
{
    scene = new RectangleComponent2();
    add(scene);
    MouseListener listener = new MousePressListener();
    scene.addMouseListener(listener);
    setSize(FRAME_WIDTH, FRAME_HEIGHT);
}

section_10/RectangleViewer2.java

import javax.swing.JFrame;

/**
 * This program displays a rectangle that can be moved with the mouse.
 */
public class RectangleViewer2
{
    public static void main(String[] args)
    {
        JFrame frame = new RectangleFrame2();
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.setVisible(true);
    }

35. Why was the moveRectangleBy method in the RectangleComponent replaced with a moveRectangleTo method?

36. Why must the MousePressListener class supply five methods?

Practice It Now you can try these exercises at the end of the chapter: R10.19, E10.29.
Chapter 10  Interfaces

Interfaces

Graphics Track

Keyboard Events

If you program a game, you may want to process key-strokes, 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:

```java
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:

```java
KeyListener listener = new MyKeyListener();
scene.addKeyListener(listener);
```

In order to receive key events, your component must call

```java
scene.setFocusable(true);
scene.requestFocus();
```

Event Adapters

In the preceding section you saw how to install a mouse listener into 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 implements all those methods in which it has no interest as "do-nothing" methods, for example:

```java
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 `MouseAdapter` 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:

```java
class MouseClickListener extends MouseAdapter
{
    public void mouseClicked(MouseEvent event)
    {
        // Mouse click action.
    }
}
```

There is also a `KeyAdapter` class that implements the `KeyListener` interface with three do-nothing methods.

---

**Computing & Society 10.1 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”.

Richard Stallman, a pioneer of the free software movement
Use interfaces for making a service available to multiple classes.

- A Java interface type declares the methods that can be applied to a variable of that type.
- Use the `implements` reserved word to indicate that a class implements an interface type.
- Use interface types to make code more reusable.

Describe how to convert between class and interface types.

- You can convert from a class type to an interface type, provided the class implements the interface.
- Method calls on an interface reference are polymorphic. The appropriate method is determined at run time.
- You need a cast to convert from an interface type to a class type.

Use the `Comparable` interface from the Java library.

- Implement the `Comparable` interface so that objects of your class can be compared, for example, in a sort method.

Describe how to use interface types for providing callbacks.

- A callback is a mechanism for specifying code that is executed at a later time.

Use inner classes to limit the scope of a utility class.

- An inner class is declared inside another class.
- Inner classes are commonly used for utility classes that should not be visible elsewhere in a program.

Use mock objects for supplying test versions of classes.

- A mock object provides the same services as another object, but in a simplified manner.
- Both the mock class and the actual class implement the same interface.

Implement event listeners to react to events in user-interface programming.

- User-interface events include key presses, mouse moves, button clicks, menu selections, and so on.
- An event listener belongs to a class that is provided 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.
- Use `Button` components for buttons. Attach an `ActionListener` to each button.
- Methods of an inner class can access local and instance variables from the surrounding scope.
- Local variables that are accessed by an inner class method must not change after they have been initialized.
Review Exercises

Build graphical applications that use buttons.

• Use a JPanel container to group multiple user-interface components together.
• Specify button click actions through classes that implement the ActionListener interface.

Use a timer for drawing animations.

• A timer generates timer events at fixed intervals.
• The repaint method causes a component to repaint itself. Call repaint whenever you modify the shapes that the paintComponent method draws.

Write programs that process mouse events.

• Use a mouse listener to capture mouse events.

STANDARD LIBRARY ITEMS INTRODUCED IN THIS CHAPTER

- java.awt.Component
- java.awt.event.KeyListener
  - keyPressed
  - keyReleased
  - keyTyped
- javax.swing.AbstractButton
  - addActionListener
- java.awt.Container
- java.awt.Dimension
- java.awt.Rectangle
- setLocation
- java.awt.event.ActionEvent
- actionPerformed
- java.awt.event.ActionListener
- java.awt.event.KeyEvent
  - getKeyStroke
  - getKeyStrokeForEvent
  - keyReleased
  - keyPressed
  - keyTyped
- java.awt.event.MouseEvent
  - getX
  - getY
- javax.swing.JButton
- javax.swing.JLabel
- javax.swing.JPanel
- javax.swing.KeyStroke
  - getKeyStrokeForEvent
- java.lang.Comparable<T>
  - compareTo
- java.lang.Double
- java.lang.Integer
  - compare
- java.awt.event.KeyListener
  - keyPressed
  - keyReleased
  - keyTyped
- javax.swing.AbstractButton
  - addActionListener
  - addMouseListener
  - setFocusable
- java.awt.event.MouseAdapter
  - mouseClicked
  - mouseEntered
  - mouseExited
  - mousePressed
  - mouseReleased
- java.awt.event.MouseListener
  - mouseClicked
  - mouseEntered
  - mouseExited
  - mousePressed
  - mouseReleased
- java.awt.event.KeyListener
  - keyPressed
  - keyReleased
  - keyTyped
- javax.swing.AbstractButton
  - addActionListener
  - addMouseListener
  - setFocusable
  - java.awt.event.MouseAdapter
  - mouseClicked
  - mouseEntered
  - mouseExited
  - mousePressed
  - mouseReleased
  - java.awt.event.MouseListener
  - mouseClicked
  - mouseEntered
  - mouseExited
  - mousePressed
  - mouseReleased
- java.lang.Comparable<T>
  - compareTo
- java.lang.Double
- java.lang.Integer
  - compare
- java.awt.event.KeyListener
  - keyPressed
  - keyReleased
  - keyTyped
- javax.swing.AbstractButton
  - addActionListener
  - addMouseListener
  - setFocusable
  - java.awt.event.MouseAdapter
  - mouseClicked
  - mouseEntered
  - mouseExited
  - mousePressed
  - mouseReleased
  - java.awt.event.MouseListener
  - mouseClicked
  - mouseEntered
  - mouseExited
  - mousePressed
  - mouseReleased
- java.lang.Comparable<T>
  - compareTo
  - java.lang.Double
  - java.lang.Integer
    - compare
  - java.awt.event.KeyListener
    - keyPressed
    - keyReleased
    - keyTyped
  - javax.swing.AbstractButton
    - addActionListener
  - addMouseListener
  - setFocusable
  - java.awt.event.MouseAdapter
    - mouseClicked
    - mouseEntered
    - mouseExited
    - mousePressed
    - mouseReleased
  - java.awt.event.MouseListener
    - mouseClicked
    - mouseEntered
    - mouseExited
    - mousePressed
    - mouseReleased
  - java.lang.Comparable<T>
    - compareTo

REVIEW EXERCISES

- **R10.1** 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)?

- **R10.2** 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)?

- **R10.3** Suppose C is a class that implements the interfaces I and J. Which of the following assignments require a cast?
  
  ```java
  C c = . . .;
  I i = . . .;
  J j = . . .;
  a. c = i;
  b. j = c;
  c. i = j;
  ```
R10.4 Suppose \( C \) is a class that implements the interfaces \( I \) and \( J \), and suppose \( i \) is declared as: \( i = \text{new} \ C(); \)

Which of the following statements will throw an exception?

a. \( C \ c = \text{(C)} \ i; \)

b. \( J \ j = \text{(J)} \ i; \)

c. \( i = \text{(I)} \ \text{null}; \)

R10.5 Suppose the class Sandwich implements the Edible interface, and you are given the variable declarations

\[
\text{Sandwich sub = new Sandwich();}
\]
\[
\text{Rectangle cerealBox = new Rectangle(5, 10, 20, 30);}
\]
\[
\text{Edible e = null;}
\]

Which of the following assignment statements are legal?

a. \( e = \text{sub}; \)

b. \( \text{sub} = e; \)

c. \( \text{sub} = \text{(Sandwich)} \ e; \)

d. \( \text{sub} = \text{(Sandwich)} \ \text{cerealBox}; \)

e. \( e = \text{cerealBox}; \)

f. \( e = \text{(Edible)} \ \text{cerealBox}; \)

g. \( e = \text{(Rectangle)} \ \text{cerealBox}; \)

h. \( e = \text{(Rectangle)} \ \text{null}; \)

R10.6 The classes Rectangle2D.Double, Ellipse2D.Double, and Line2D.Double implement the Shape interface. The Graphics2D class depends on the Shape interface but not on the rectangle, ellipse, and line classes. Draw a UML diagram denoting these facts.

R10.7 Suppose \( r \) contains a reference to a new Rectangle(5, 10, 20, 30). Which of the following assignments is legal? (Look inside the API documentation to check which interfaces the Rectangle class implements.)

a. \( \text{Rectangle a = r;} \)

b. \( \text{Shape b = r;} \)

c. \( \text{String c = r;} \)

d. \( \text{ActionListener d = r;} \)

e. \( \text{Measurable e = r;} \)

f. \( \text{Serializable f = r;} \)

g. \( \text{Object g = r;} \)

R10.8 Classes such as Rectangle2D.Double, Ellipse2D.Double, and Line2D.Double implement the Shape interface. The Shape interface has a method

\[
\text{Rectangle getBounds()}
\]

that returns a rectangle completely enclosing the shape. Consider the method call:

\[
\text{Shape s = \ldots ;}
\]
\[
\text{Rectangle r = s.getBounds();}
\]

Explain why this is an example of polymorphism.

R10.9 Suppose you need to process an array of employees to find the average salary. Discuss what you need to do to use the Data.average method in Section 10.1 (which processes Measurable objects). What do you need to do to use the second implementation (in Section 10.4)? Which is easier?

R10.10 What happens if you try to use an array of String objects with the Data.average method in Section 10.1?

R10.11 How can you use the Data.average method in Section 10.4 if you want to compute the average length of the strings?

R10.12 What happens if you pass an array of strings and an AreaMeasurer to the Data.average method of Section 10.4?
**R10.13** Consider this top-level and inner class. Which variables can the \( f \) method access?

```java
public class T {
    private int t;

    public void m(final int x, int y) {
        int a;
        final int b;

        class C implements I {
            public void f() {
            }
        }

        final int c;
    }
}
```

**R10.14** What happens when an inner class tries to access a local variable that assumes more than one value? Try it out and explain your findings.

**Graphics R10.15** How would you reorganize the `InvestmentViewer1` program if you needed to make `AddInterestListener` into a top-level class (that is, not an inner class)?

**Graphics R10.16** What is an event object? An event source? An event listener?

**Graphics R10.17** From a programmer’s perspective, what is the most important difference between the user interfaces of a console application and a graphical application?

**Graphics R10.18** What is the difference between an `ActionEvent` and a `MouseEvent`?

**Graphics R10.19** Why does the `ActionListener` interface have only one method, whereas the `MouseListener` has five methods?

**Graphics R10.20** Can a class be an event source for multiple event types? If so, give an example.

**Graphics R10.21** What information does an action event object carry? What additional information does a mouse event object carry?

**Graphics R10.22** Why are we using inner classes for event listeners? If Java did not have inner classes, could we still implement event listeners? How?

**Graphics R10.23** What is the difference between the `paintComponent` and `repaint` methods?

**Graphics R10.24** What is the difference between a frame and a panel?

---

**E10.1** Add a method

```java
public static Measurable max(Measurable[] objects)
```

to the `Data` class that returns the object with the largest measure.
E10.2 Implement a class Quiz that implements the Measurable interface. A quiz has a score and a letter grade (such as B+). Use the Data class of Exercise E10.1 to process an array of quizzes. Display the average score and the quiz with the highest score (both letter grade and score).

E10.3 A person has a name and a height in centimeters. Use the Data class of Exercise E10.1 to process an array of Person objects. Display the average height and the name of the tallest person.

E10.4 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.

E10.5 In the Sequence interface of Worked Example 10.1, 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.

E10.6 In Worked Example 10.1, add a default method
  default int[] values(int n)
that yields an array of the first n values of the sequence.

E10.7 In Worked Example 10.1, make the process method a default method of the Sequence interface.

E10.8 Add a method to the Data class that returns the object with the largest measure, as measured by the supplied measurer:
  public static Object max(Object[] objects, Measurer m)

E10.9 Using a different Measurer object, process a set of Rectangle objects to find the rectangle with the largest perimeter.

E10.10 Modify the Coin class from Chapter 8 to have it implement the Comparable interface.

E10.11 Repeat Exercise E10.9, making the Measurer into an inner class inside the main method.

E10.12 Repeat Exercise E10.9, making the Measurer an inner class outside the main method.

E10.13 Implement a class Bag that stores items represented as strings. Items can be repeated. Supply methods for adding an item, and for counting how many times an item has been added:
  public void add(String itemName)
  public int count(String itemName)
Your Bag class should store the data in an ArrayList<Item>, where Item is an inner class with two instance variables: the name of the item and the quantity.

E10.14 Implement a class Grid that stores measurements in a rectangular grid. The grid has a given number of rows and columns, and a description string can be added for any grid location. Supply the following constructor and methods:
  public Grid(int numRows, int numColumns)
  public void add(int row, int column, String description)
  public String getDescription(int row, int column)
  public ArrayList<Location> getDescribedLocations()
Here, `Location` is an inner class that encapsulates the row and the column of a grid location.

**E10.15** Reimplement Exercise E10.14 where the grid is unbounded. The constructor has no arguments, and the row and column parameter variables of the `add` and `getDescription` methods can be arbitrary integers.

**Graphics E10.16** Write a method `randomShape` that randomly generates objects implementing the `Shape` interface in the Java library API: some mixture of rectangles, ellipses, and lines, with random positions. Call it ten times and draw all of them.

**Graphics E10.17** Enhance the `ButtonViewer` program 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.

**Graphics E10.18** Enhance the `ButtonViewer` program 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.

**Graphics E10.19** Enhance the `ButtonViewer` program 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.

**Graphics E10.20** Implement a `ButtonViewer` program as in Exercise E10.19, using only a single listener class.

**Graphics E10.21** Enhance the `ButtonViewer` program so that it prints the time at which the button was clicked.

**Graphics E10.22** Implement the `AddInterestListener` in the `InvestmentViewer1` program as a regular class (that is, not an inner class). *Hint:* Store a reference to the bank account. Add a constructor to the listener class that sets the reference.

**Graphics E10.23** Implement the `AddInterestListener` in the `InvestmentViewer2` program as a regular class (that is, not an inner class). *Hint:* Store references to the bank account and the label in the listener. Add a constructor to the listener class that sets the references.

**E10.24** Reimplement the program in Section 10.7.2, specifying the listener with a lambda expression (see Java 8 Note 10.4).

**E10.25** Reimplement the program in Section 10.8, specifying the listener with a lambda expression (see Java 8 Note 10.4).

**E10.26** Reimplement the program in Section 10.9, specifying the listener with a lambda expression (see Java 8 Note 10.4).

**Graphics E10.27** Write a program that uses a timer to print the current time once a second. *Hint:* The following code prints the current time:

```java
date now = new Date();
System.out.println(now);
```

The `Date` class is in the `java.util` package.

**Graphics E10.28** Change the `RectangleComponent` for the animation program in Section 10.9 so that the rectangle bounces off the edges of the component rather than moving outside.

**Graphics E10.29** Change the `RectangleComponent` for the mouse listener program in Section 10.10 so that a new rectangle is added to the component whenever the mouse is clicked. *Hint:* Keep an `ArrayList<Rectangle>` and draw all rectangles in the `paintComponent` method.
• **E10.30** 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 and print the first and last person among them.

**PROGRAMMING PROJECTS**

• **P10.1** Modify the `display` method of the `LastDigitDistribution` class of Worked Example 10.1 so that it produces a histogram, like this:
  
  0: *************
  1: ******************
  2: *************

  Scale the bars so that widest one has length 40.

• **P10.2** Write a class `PrimeSequence` that implements the `Sequence` interface of Worked Example 10.1, and produces the sequence of prime numbers.

• **P10.3** Add a method `hasNext` to the `Sequence` interface of Worked Example 10.1 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.

• **P10.4** Provide a class `FirstDigitDistribution` that works just like the `LastDigitDistribution` class of Worked Example 10.1, 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.)

• **P10.5** Declare an interface `Filter` as follows:

  ```java
  public interface Filter { boolean accept(Object x); }
  ```

  Modify the implementation of the `Data` class in Section 10.4 to use both a `Measurer` and a `Filter` object. 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.

• **P10.6** Solve Exercise P10.5, using a lambda expression for the filter.

• **P10.7** In Exercise P10.5, add a method to the `Filter` interface that counts how many objects are accepted by the filter:

  ```java
  static int count(Object[] values, Filter condition)
  ```

• **P10.8** In Exercise P10.5, add a method to the `Filter` interface that retains all objects accepted by the filter and removes the others:

  ```java
  static void retainAll(Object[] values, Filter condition)
  ```

• **P10.9** In Exercise P10.5, add a method `default boolean reject(Object x)` to the `Filter` interface that returns `true` for all objects that this filter doesn’t accept.

• **P10.10** In Exercise P10.5, add a method `default Filter invert()` to the `Filter` interface that yields a filter accepting exactly the objects that this filter rejects.
P10.11 Consider an interface

```java
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.

P10.12 Write a method that takes an array of integers and a NumberFormatter object (from Exercise P10.11) and prints each number on a separate line, formatted with the given formatter. The numbers should be right aligned.

P10.13 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 \$\$ format, you can print any class that implements the `Formattable` interface. That interface has a single method:

```java
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:

```java
Appendable a = formatter.out();
```

Appendable is another interface with a method

```java
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.

P10.14 Enhance the `formatTo` method of Exercise P10.13 by taking into account the precision.

Graphics P10.15 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.

Graphics P10.16 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.

Graphics P10.17 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.

**Graphics P10.18** Write a program that lets users design bar charts with a mouse. When the user clicks inside a bar, the next mouse click extends the length of the bar to the $x$-coordinate of the mouse click. (If it is at or near 0, the bar is removed.) When the user clicks below the last bar, a new bar is added whose length is the $x$-coordinate of the mouse click.

**Testing P10.19** Consider the task of writing a program that plays tic-tac-toe against a human opponent. A user interface `TicTacToeUI` reads the user’s moves and displays the computer’s moves and the board. A class `TicTacToeStrategy` determines the next move that the computer makes. A class `TicTacToeBoard` represents the current state of the board. Complete all classes except for the strategy class. Instead, use a mock class that simply picks the first available empty square.

**Testing P10.20** Consider the task of translating a plain-text book from Project Gutenberg (http://gutenberg.org) to HTML. For example, here is the start of the first chapter of Tolstoy’s Anna Karenina:

```
Chapter 1

Happy families are all alike; every unhappy family is unhappy in its own way.

Everything was in confusion in the Oblonskys' house. The wife had discovered that the husband was carrying on an intrigue with a French girl, who had been a governess in their family, and she had announced to her husband that she could not go on living in the same house with him . . .
```

The equivalent HTML is:

```
<h1>Chapter 1</h1>
<p>Happy families are all alike; every unhappy family is unhappy in its own way.</p>
<p>Everything was in confusion in the Oblonskys' house. The wife had discovered that the husband was carrying on an intrigue with a French girl, who had been a governess in their family, and she had announced to her husband that she could not go on living in the same house with him . . .</p>
```

The HTML conversion can be carried out in two steps. First, the plain text is assembled into *segments*, blocks of text of the same kind (heading, paragraph, and so on). Then each segment is converted, by surrounding it with the HTML tags and converting special characters.

Fetching the text from the Internet and breaking it into segments is a challenging task. Provide an interface and a mock implementation. Combine it with a class that uses the mock implementation to finish the formatting task.

**Graphics P10.21** Write a program that demonstrates the growth of a roach population. Start with two roaches and double the number of roaches with each button click.

**Graphics P10.22** Write a program that animates a car so that it moves across a frame.

**Graphics P10.23** Write a program that animates two cars moving across a frame in opposite directions (but at different heights so that they don’t collide.)
• Graphics P10.24 Write a program that prompts the user to enter the x- and y-positions of the center and a radius, using JOptionPane dialogs. When the user clicks a “Draw” button, prompt for the inputs and draw a circle with that center and radius in a component.

• Graphics P10.25 Write a program that allows the user to specify a circle by clicking on the center and then typing the radius in a JOptionPane. Note that you don’t need a “Draw” button.

• Graphics P10.26 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.

• Graphics P10.27 Design an interface MoveableShape that can be used as a generic mechanism for animating a shape. A moveable shape must have two methods: move and draw. Write a generic AnimationPanel that paints and moves any MoveableShape (or array list of MoveableShape objects). Supply moveable rectangle and car shapes.

• P10.28 Your task is to design a general program for managing board games with two players. Your program should be flexible enough to handle games such as tic-tac-toe, chess, or the Game of Nim of Programming Project 6.5.

  Design an interface Game that describes a board game. Think about what your program needs to do. It asks the first player to input a move—a string in a game-specific format, such as Be3 in chess. Your program knows nothing about specific games, so the Game interface must have a method such as

  boolean isValidMove(String move)

  Once the move is found to be valid, it needs to be executed—the interface needs another method executeMove. Next, your program needs to check whether the game is over. If not, the other player’s move is processed. You should also provide some mechanism for displaying the current state of the board.

  Design the Game interface and provide two implementations of your choice—such as Nim and Chess (or TicTacToe if you are less ambitious). Your GamePlayer class should manage a Game reference without knowing which game is played, and process the moves from both players. Supply two programs that differ only in the initialization of the Game reference.

1. It must implement the Measurable interface, and its getMeasure method must return the salary.

2. The Object class doesn’t have a getMeasure method.

3. You cannot modify the String class to implement Measurable—String is a library class. See Section 10.4 for a solution.

4. Measurable is not a class. You cannot construct objects of type Measurable.

5. The variable meas is of type Measurable, and that type has no getName method.

6. Only if meas actually refers to a BankAccount object.

7. No—a Country reference can be converted to a Measurable reference, but if you attempt to cast that reference to a BankAccount, an exception occurs.

8. Measurable is an interface. Interfaces have no instance variables.

9. That variable never refers to a Measurable object. It refers to an object of some class—a class that implements the Measurable interface.
The code fragment prints 20255. The average method calls getMeasure on each object in the array. In the first call, the object is a BankAccount. In the second call, the object is a Country. A different getMeasure method is called in each case. The first call returns the account balance, the second one the area, which are then averaged.

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;
    }
}
```

Yes, you can, because String implements the Comparable interface type.

No. The Rectangle class does not implement the Comparable interface.

```
public static Comparable max(
    Comparable a, Comparable b)
{
    if (a.compareTo(b) > 0) { return a; }
    else { return b; }
}
```

Because the developer of GradingProgram doesn’t have to wait for the GradeBook class to be complete.

The button object is the event source. The listener object is the event listener.

```
The ClickListener class implements the ActionListener interface.
```

You don’t. It is called whenever the button is clicked.

Direct access is simpler than the alternative—passing the variable as an argument to a constructor or method.

The local variable must not change. Prior to Java 8, it must be declared as final.

First add label to the panel, then add button.

The actionPerformed method does not access that variable.

The timer needs to call some method whenever the time interval expires. It calls the actionPerformed method of the listener object.

The moved rectangles won’t be painted, and the rectangle will appear to be stationary until the frame is repainted for an external reason.

Because you know the current mouse position, not the amount by which the mouse has moved.

It implements the MouseListener interface, which has five methods.
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

<table>
<thead>
<tr>
<th>Value</th>
<th>Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>105</td>
</tr>
<tr>
<td>1</td>
<td>94</td>
</tr>
<tr>
<td>2</td>
<td>81</td>
</tr>
<tr>
<td>3</td>
<td>112</td>
</tr>
<tr>
<td>4</td>
<td>89</td>
</tr>
<tr>
<td>5</td>
<td>103</td>
</tr>
<tr>
<td>6</td>
<td>103</td>
</tr>
<tr>
<td>7</td>
<td>100</td>
</tr>
<tr>
<td>8</td>
<td>108</td>
</tr>
<tr>
<td>9</td>
<td>105</td>
</tr>
</tbody>
</table>

In order to produce arbitrary sequences, we declare an interface type with a single method:

```java
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:

```java
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.

```java
public class SquareSequence implements Sequence
{
    private int n;

    public int next()
    {
        n++;
        return n * n;
    }
```

Big Java, 6e, Cay Horstmann, Copyright © 2015 John Wiley and Sons, Inc. All rights reserved.
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 ch10/worked_example_1 directory of the book’s companion code.
CHAPTER 11
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

11.1 READING AND WRITING TEXT FILES 520
   CE1 Backslashes in File Names 523
   CE2 Constructing a Scanner with a String 523
   ST1 Reading Web Pages 523
   ST2 File Dialog Boxes 523
   ST3 Character Encodings 524

11.2 TEXT INPUT AND OUTPUT 525
   ST4 Regular Expressions 532
   ST5 Reading an Entire File 533

11.3 COMMAND LINE ARGUMENTS 533
   HT1 Processing Text Files 536
   WE1 Analyzing Baby Names 🌟
   C&S Encryption Algorithms 539

11.4 EXCEPTION HANDLING 540
   SYN Throwing an Exception 540
   SYN Catching Exceptions 542
   SYN The throws Clause 545
   SYN The try-with-resources Statement 545
   PT1 Throw Early, Catch Late 548
   PT2 Do Not Squelch Exceptions 548
   PT3 Do Throw Specific Exceptions 548
   ST6 Assertions 549
   ST7 The try/finally Statement 549

11.5 APPLICATION: HANDLING INPUT ERRORS 549
   C&S The Ariane Rocket Incident 554
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 at left 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.

11.1 Reading and Writing 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:

```java
File inputFile = new File("input.txt");
```

Then use the File object to construct a Scanner object:

```java
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:

```java
while (in.hasNextDouble())
{
    double value = in.nextDouble();
    Process value.
}
```

To write output to a file, you construct a PrintWriter object with the desired file name, for example

```java
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 `println`, `print`, and `printf` methods with any PrintWriter object:

```java
out.println("Hello, World!");
out.printf("Total: %8.2f\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:

```java
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, lined up in a column and followed by their total, to another file.

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 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:

```java
public static void main(String[] args) throws FileNotFoundException
```

You will see in Section 11.4 how to deal with exceptions in a more professional way.

The File, PrintWriter, and FileNotFoundException classes are contained in the java.io package.

section_1/Total.java

```java
import java.io.File;
import java.io.FileNotFoundException;
import java.io.PrintWriter;
import java.util.Scanner;

/**
 * This program reads a file with numbers, and writes the numbers to another file, lined up in a column and followed by their total.
 */
public class Total {
    public static void main(String[] args) throws FileNotFoundException {
    ```
// Prompt for the input and output file names
Scanner console = new Scanner(System.in);
System.out.print("Input file: ");
String inputFileName = console.next();
System.out.print("Output file: ");
String outputFileName = console.next();

// Construct the Scanner and PrintWriter objects for reading and writing
File inputFile = new File(inputFileName);
Scanner in = new Scanner(inputFile);
PrintWriter out = new PrintWriter(outputFileName);

// Read the input and write the output
double total = 0;

while (in.hasNextDouble())
{
    double value = in.nextDouble();
    out.printf("%15.2f\n", value);
    total = total + value;
}

out.printf("Total: %8.2f\n", total);

in.close();
out.close();

1. 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.

2. What happens when you supply the name of a nonexistent input file to the Total program? Try it out if you are not sure.

3. 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 pseudocode for the solution.

4. How do you modify the Total program so that it shows the average, not the total, of the inputs?

5. 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 |
   | Total:| 622.00 |

Practice It  Now you can try these exercises at the end of the chapter: R11.1, R11.2, E11.1.
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:

```java
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.

Constructing a Scanner with a String

When you construct a PrintWriter with a string, it writes to a file:

```java
PrintWriter out = new PrintWriter("output.txt");
```

However, this does not work for a Scanner. The statement

```java
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 11.2.5.)

You must simply remember to use `File` objects in the `Scanner` constructor:

```java
Scanner in = new Scanner(new File("input.txt")); // OK
```

Reading Web Pages

You can read the contents of a web page with this sequence of commands:

```java
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 11.4.3 for more information on the `throws` clause.)

The `URL` class is contained in the `java.net` package.

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:

```java
JFileChooser chooser = new JFileChooser();
Scanner in = null;
if (chooser.showOpenDialog(null) == JFileChooser.APPROVE_OPTION)
{
    File selectedFile = chooser.getSelectedFile();
in = new Scanner(selectedFile);
...}
```

### Character Encodings

A **character** (such as the letter A, the digit 0, the accented character è, the Greek letter π, the symbol ∫, or the Chinese character 中) is encoded as a sequence of bytes. Each byte is a value between 0 and 255.

Unfortunately, the encoding is not uniform. In 1963, ASCII (the American Standard Code for Information Interchange) defined an encoding for 128 characters, which you can find in Appendix A. ASCII encodes all upper- and lowercase Latin letters and digits, as well as common symbols such as +, %, as values between 0 and 127. For example, the code for the letter A is 65.

As different populations felt the need to encode their own alphabets, they designed their own codes. Many of them built upon ASCII, using the values in the range from 128 to 255 for their own language. For example, in Spain, the letter è was encoded as 233. But in Greece, the code 233 denoted the letter ι (a lowercase iota). As you can imagine, if a Spanish tourist named José sent an e-mail to a Greek hotel, this created a problem.
To resolve this issue, the design of Unicode was begun in 1987. As described in Computing & Society 4.2, each character in the world is given a unique integer value. However, there are still multiple encodings of those integers in binary. The most popular encoding is called UTF-8. It encodes each character as a sequence of one to four bytes. For example, an A is still 65, as in ASCII, but an é is 195 169. The details of the encoding don’t matter, as long as you specify that you want UTF-8 when you read and write a file.

As this book goes to print, the Windows and Macintosh operating systems have not yet made the switch to UTF-8. Java picks up the character encoding from the operating system. Unless you specifically request otherwise, the Scanner and PrintWriter classes will read and write files in that encoding. That’s fine if your files contain only ASCII characters, or if the creator and the recipient use the same encoding. But if you need to process files with accented characters, Chinese characters, or special symbols, you should specifically request the UTF-8 encoding. Construct a scanner with

```
Scanner in = new Scanner(file, "UTF-8");
```

and a print writer with

```
PrintWriter out = new PrintWriter(file, "UTF-8");
```

You may wonder why Java can’t just figure out the character encoding. However, consider the string José. In UTF-8, that’s 74 111 115 195 169. The first three bytes, for Jos, are in the ASCII range and pose no problem. But the next two bytes, 195 169, could be é in UTF-8 or Á in the traditional Spanish encoding. The Scanner object doesn’t understand Spanish, and it can’t decide which encoding to choose.

Therefore, you should always specify the UTF-8 encoding when you exchange files with users from other parts of the world.

11.2 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.

11.2.1 Reading Words

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:

```java
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:

```java
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 11.4.) With this setting, punctuation and numbers are not included in the words returned by the `next` method.

### 11.2.2 Reading Characters

Sometimes, you want to read a file one character at a time. You will see an example in Section 11.3 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:

```java
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:

```java
while (in.hasNext())
{
    char ch = in.next().charAt(0);
    Process ch.
}
```

### 11.2.3 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

```java
Character.isDigit(ch)
```

returns `true` if `ch` is a digit (‘0’ . . . ‘9’ or a digit in another writing system—see Computing & Society 4.2), `false` otherwise.
11.2.4 Reading Lines

When each line of a file is a data record, it is often best to read entire lines with the `nextLine` method:

```java
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](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:

```java
while (in.hasNextLine())
{
    String line = nextLine();
    Process line.
}
```

Use the `isDigit` and `isWhiteSpace` methods in Table 1 to find out where the name ends and the number starts.

Locate the first digit:

```java
int i = 0;
while (!Character.isDigit(line.charAt(i))) { i++; }
```

Then extract the country name and population:

```java
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:

```java
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 11.2.6, you will see how to convert the string to a number.

### 11.2.5 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:

```java
Scanner lineScanner = new Scanner(line);
```

Then you can use `lineScanner` like any other `Scanner` object, reading words and numbers:

```java
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();
```

### 11.2.6 Converting Strings to Numbers

Sometimes you have a string that contains a number, such as the `population` string in Section 11.2.4. For example, suppose that the string is the character sequence "303824646". To get the integer value 303824646, you use the `Integer.parseInt` method:

```java
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".

```java
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:

```java
int populationValue = Integer.parseInt(population.trim());
```

How To 11.1 on page 536 continues this example.

### 11.2.7 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

```java
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

![21st century]

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,

```java
if (in.hasNextInt())
{
    int value = in.nextInt();
    ...
}
```

Similarly, you should call the `hasNextDouble` method before calling `nextDouble`.

### 11.2.8 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:

<table>
<thead>
<tr>
<th>Country</th>
<th>Population</th>
</tr>
</thead>
<tbody>
<tr>
<td>China</td>
<td>1330044605</td>
</tr>
<tr>
<td>India</td>
<td>1147995898</td>
</tr>
<tr>
<td>United States</td>
<td>303824646</td>
</tr>
</tbody>
</table>

Now suppose you read the file with these instructions:

```java
while (in.hasNextLine())
{
    String countryName = in.nextLine();
    int population = in.nextInt();
    Process the country name and population.
}
```
Initially, the input contains

\[
\text{China } \backslash n 133044605 \backslash n \text{India } \backslash n
\]

After the first call to the `nextLine` method, the input contains

\[
133044605 \backslash n \text{India } \backslash n
\]

After the call to `nextInt`, the input contains

\[
\backslash n \text{India } \backslash n
\]

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:

```java
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.

### 11.2.9 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 4.3.2 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

```
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:

```java
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.
### Table 2 Format Flags

<table>
<thead>
<tr>
<th>Flag</th>
<th>Meaning</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>-</td>
<td>Left alignment</td>
<td>1.23 followed by spaces</td>
</tr>
<tr>
<td>0</td>
<td>Show leading zeroes</td>
<td>001.23</td>
</tr>
<tr>
<td>+</td>
<td>Show a plus sign for positive numbers</td>
<td>+1.23</td>
</tr>
<tr>
<td>(</td>
<td>Enclose negative numbers in parentheses</td>
<td>(1.23)</td>
</tr>
<tr>
<td>,</td>
<td>Show decimal separators</td>
<td>12,300</td>
</tr>
<tr>
<td>^</td>
<td>Convert letters to uppercase</td>
<td>1.23E+1</td>
</tr>
</tbody>
</table>

A construct such as `%10s` or `%10.2f` is called a format specifier: it describes how a value should be formatted.

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.

### Table 3 Format Types

<table>
<thead>
<tr>
<th>Code</th>
<th>Type</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>d</td>
<td>Decimal integer</td>
<td>123</td>
</tr>
<tr>
<td>f</td>
<td>Fixed floating-point</td>
<td>12.30</td>
</tr>
<tr>
<td>e</td>
<td>Exponential floating-point</td>
<td>1.23e+1</td>
</tr>
<tr>
<td>g</td>
<td>General floating-point (exponential notation is used for very large or very small values)</td>
<td>12.3</td>
</tr>
<tr>
<td>s</td>
<td>String</td>
<td>Tax:</td>
</tr>
</tbody>
</table>

6. Suppose the input contains the characters `Hello, World!`. What are the values of `word` and `input` after this code fragment?

```java
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?
   ```java
   int number = 0;
   if (in.hasNextInt()) { number = in.nextInt(); }
   String input = in.next();
   ```

8. Suppose the input contains the characters 6E6 $6,995.00. What are the values of x1 and x2 after this code fragment?
   ```java
   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 11.2.4 without using the trim method?

**Practice It**

Now you can try these exercises at the end of the chapter: E11.4, E11.6, E11.7.

---

**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 4.1) in a file.

```java
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 \texttt{x1} will be listed. OK, you want sequences of digits that do **not** immediately follow letters:

```java
grep \[^A-Za-z][0-9]+ \*Homework.java
```

The set \[^A-Za-z\] denotes any characters that are **not** in the ranges \texttt{A} to \texttt{Z} and \texttt{a} to \texttt{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 expressions. For example,

```java
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 11.4**

© Eric Isselé/iStockphoto.
11.3 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:

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

In our example, `args` is an array of length 2, containing the strings

```java
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).
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`.

### section_3/CaesarCipher.java

```java
import java.io.File;
import java.io.FileNotFoundException;
import java.io.PrintWriter;
import java.util.Scanner;

/**
 * This program encrypts a file using the Caesar cipher.
 */
public class CaesarCipher {
    public static void main(String[] args) throws FileNotFoundException {
        final int DEFAULT_KEY = 3;
        int key = DEFAULT_KEY;
        String inFile = "";
        String outFile = "";
        int files = 0; // Number of command line arguments that are files

        for (int i = 0; i < args.length; i++) {
            String arg = args[i];
            if (arg.charAt(0) == '-') {
                // It is a command line option
                char option = arg.charAt(1);
                if (option == 'd') {
                    key = -key;
                } else {
                    usage(); return;
                }
            } else {
                // It is a file name
                files++;
                if (files == 1) { inFile = arg; }
                else if (files == 2) { outFile = arg; }
            }
        }
    }
}
```

© xyno/iStockphoto.
The emperor Julius Caesar used a simple scheme to encrypt messages.

© xyno/iStockphoto.
if (files != 2) { usage(); return; }

Scanner in = new Scanner(new File(inFile));
in.useDelimiter(""; // Process individual characters
PrintWriter out = new PrintWriter(outFile);

while (in.hasNext())
{
    char from = in.next().charAt(0);
    char to = encrypt(from, key);
    out.print(to);
}
in.close();
out.close();
}

/**
 * Encrypts upper- and lowercase characters by shifting them
 * according to a key.
 * @param ch the letter to be encrypted
 * @param key the encryption key
 * @return the encrypted letter
 */
public static char encrypt(char ch, int key)
{
    int base = 0;
    if ('A' <= ch && ch <= 'Z') { base = 'A'; }
    else if ('a' <= ch && ch <= 'z') { base = 'a'; }
    else { return ch; } // Not a letter
    int offset = ch - base + key;
    final int LETTERS = 26; // Number of letters in the Roman alphabet
    if (offset > LETTERS) { offset = offset - LETTERS; }
    else if (offset < 0) { offset = offset + LETTERS; }
    return (char) (base + offset);
}

/**
 * Prints a message describing proper usage.
 */
public static void usage()
{
    System.out.println("Usage: java CaesarCipher [-d] infile outfile");
}

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?


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: R11.5, E11.10, E11.11.
HOW TO 11.1 Processing Text Files

Processing text files that contain real data can be surprisingly challenging. This How To gives you step-by-step guidance using world population data.

**Problem Statement**  Read two country data files, `world-pop.txt` and `world-area.txt` (supplied with your source 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:

<table>
<thead>
<tr>
<th>Country</th>
<th>Density</th>
</tr>
</thead>
<tbody>
<tr>
<td>Afghanistan</td>
<td>50.56</td>
</tr>
<tr>
<td>Akrotiri</td>
<td>127.64</td>
</tr>
<tr>
<td>Albania</td>
<td>125.91</td>
</tr>
<tr>
<td>Algeria</td>
<td>14.18</td>
</tr>
<tr>
<td>American Samoa</td>
<td>288.92</td>
</tr>
<tr>
<td>. . .</td>
<td></td>
</tr>
</tbody>
</table>

**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 consideration 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 one 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.

```plaintext
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:
  ```java
  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 11.2.4.

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 classes and methods to factor out common tasks.

Processing input files usually has repetitive tasks, such as skipping over white space or extract-
ing numbers from strings. It really pays off to isolate these tedious operations from the
remainder of the code.

In our example, we have a task that occurs twice: splitting an input line into the country
name and the value that follows. We implement a simple `CountryValue` class for this purpose,
using the technique described in Section 11.2.4.

Here is the complete source code:

```java
how_to_1/CountryValue.java

/**
 * Describes a value that is associated with a country.
 */
public class CountryValue {
    private String country;
    private double value;

    /**
     * Constructs a CountryValue from an input line.
     * @param line a line containing a country name, followed by a value
     */
    public CountryValue(String line) {
        int i = 0; // Locate the start of the first digit
        while (!Character.isDigit(line.charAt(i))) { i++; }
        int j = i - 1; // Locate the end of the preceding word
        while (Character.isWhitespace(line.charAt(j))) { j--; }
        country = line.substring(0, j + 1); // Extract the country name
        value = Double.parseDouble(line.substring(i).trim()); // Extract the value
    }

    /**
     * Gets the country name.
     * @return the country name
     */
    public String getCountry() { return country; }
```
public double getValue() { return value; }

/**
   * Gets the associated value.
   * @return the value associated with the country
   */

public class PopulationDensity {

    // Open input files
    Scanner in1 = new Scanner(new File("worldpop.txt"));
    Scanner in2 = new Scanner(new File("worldarea.txt"));

    // Open output file
    PrintWriter out = new PrintWriter("world_pop_density.txt");

    // Read lines from each file
    while (in1.hasNextLine() && in2.hasNextLine()) {
        CountryValue population = new CountryValue(in1.nextLine());
        CountryValue area = new CountryValue(in2.nextLine());

        // Compute and print the population density
        double density = 0;
        if (area.getValue() != 0) // Protect against division by zero
            density = population.getValue() / area.getValue();
        out.printf("%40s%15.2f\n", population.getCountry(), density);
    }

    in1.close();
    in2.close();
    out.close();
}

WORKED EXAMPLE 11.1

Analyzing Baby Names

Learn how to use data from the Social Security Administration to analyze the most popular baby names. Go to wiley.com/go/bjeo6examples and download Worked Example 11.1.
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://theory.lcs.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.

![Diagram of Public-Key Encryption](https://example.com/public-key-diagram.png)
Chapter 11
Input/Output and Exception Handling

11.4 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.

11.4.1 Throwing Exceptions

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.

```java
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 inheritance hierarchy, with more specialized classes at the bottom of the tree.)

Look around for an exception type that might describe your situation. How about the ArithmeticException? 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 IllegalArgumentException.

```java
if (amount > balance)
{
    throw new IllegalArgumentException("Amount exceeds balance");
}
```

Syntax 11.1 Throwing an Exception

```
Syntax throw exceptionObject;
```

A new exception object is constructed, then thrown.

```
if (amount > balance)
{
    throw new IllegalArgumentException("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.
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.*
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` statement 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:

```java
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());
}
```

Place the statements that can cause an exception inside a `try` block, and the handler inside a `catch` clause.

### Syntax 11.2 Catching Exceptions

```java
try {
    statement
    statement
    ...
} catch (ExceptionClass exceptionObject) {
    statement
    statement
    ...
}
```

This constructor can throw a `FileNotFoundException`.

When an `IOException` is thrown, execution resumes here.

Additional catch clauses can appear here. Place more specific exceptions before more general ones.

```java
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 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 block.

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 11.5 for a solution.

### 11.4.3 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 as IndexOutOfBoundsException or IllegalArgumentException indicate errors in your code. They are called unchecked exceptions.
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,

```java
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 the method with a `throws` clause:

```java
public void readData(String filename) throws FileNotFoundException {
    File inFile = new File(filename);
    Scanner in = new Scanner(inFile);
    . . .
}
```

The **throws clause** signals to 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.

*Just as trucks with large or hazardous loads carry warning signs, the throws clause warns the caller that an exception may occur.*
11.4 Exception Handling

### Syntax 11.3 The throws Clause

```java
public void readData(String filename)
    throws FileNotFoundException, NumberFormatException
```

You must specify all checked exceptions that this method may throw.

You may also list unchecked exceptions.

### 11.4.4 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:

```java
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:

```java
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.

### Syntax 11.4 The try-with-resources Statement

```java
try (Type1 variable1 = expression1; Type2 variable2 = expression2; . . .)
{
    . . .
}
```

This code may throw exceptions.

```java
try (PrintWriter out = new PrintWriter(filename))
{
    writeData(out);
}
```

At this point, `out.close()` is called, even when an exception occurs.

Implements the AutoCloseable interface.
You can declare multiple variables in a try-with-resources statement, like this:

```java
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. You will find other AutoCloseable classes in Chapters 21 and 24.

11.4.5 Designing Your Own Exception Types

Sometimes none of the standard exception types describe your particular error condition well enough. In that case, you can design your own exception class. Consider a bank account. Let’s report an InsufficientFundsException when an attempt is made to withdraw an amount from a bank account that exceeds the current balance.

```java
if (amount > balance)
{
    throw new InsufficientFundsException("withdrawal of " + amount + " exceeds balance of " + balance);
}
```

Now you need to provide the InsufficientFundsException class. Should it be a checked or an unchecked exception? Is it the fault of some external event, or is it the fault of the programmer? We take the position that the programmer could have prevented the exceptional condition—after all, it would have been an easy matter to check whether amount <= account.getBalance() before calling the withdraw method. Therefore, the exception should be an unchecked exception and extend the RuntimeException class or one of its subclasses.

It is a good idea to extend an appropriate class in the exception hierarchy. For example, we can consider an InsufficientFundsException a special case of an IllegalArgumentException. This enables other programmers to catch the exception as an IllegalArgumentException if they are not interested in the exact nature of the problem.

It is customary to provide two constructors for an exception class: a constructor with no arguments and a constructor that accepts a message string describing the reason for the exception.
Here is the declaration of the exception class:

```java
public class InsufficientFundsException extends IllegalArgumentException {
    public InsufficientFundsException() {}
    public InsufficientFundsException(String message) {
        super(message);
    }
}
```

When the exception is caught, its message string can be retrieved using the `getMessage` method of the `Throwable` class.

16. Suppose `balance` is 100 and `amount` is 200. What is the value of `balance` after these statements?
   ```java
   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
   ```java
   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?
   ```java
   public static void writeAll(String[] lines, String filename) {
      PrintWriter out = new PrintWriter(filename);
      for (String line : lines) {
         out.println(line.toUpperCase());
      }
      out.close();
   }
   ```

22. What is the purpose of the call `super(message)` in the second `InsufficientFundsException` constructor?
23. Suppose you read bank account data from a file. Contrary to your expectation, the next input value is not of type `double`. You decide to implement a `BadDataException`. Which exception class should you extend?

**Practice It** Now you can try these exercises at the end of the chapter: R11.8, R11.9, R11.10.

---

**Programming Tip 11.1**

**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”.

---

**Programming Tip 11.2**

**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:

```java
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 11.3**

**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.
**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:

```java
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.

**The try/finally Statement**

You saw in Section 11.4 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:

```java
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. However, you will see a use of this statement in Chapter 22.

11.5 Application: Handling Input Errors

This section walks through a complete example of a program with exception handling. The program asks a user for the name of a file. The file is expected to contain data values. The first line of the file contains 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
```

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 the file data is in the wrong format, we will throw a BadDataException, a custom checked exception class. We use a checked exception because corruption of a data file is beyond the control of the programmer.

Who can remedy the faults that the exceptions report? Only the main method of the DataAnalyzer program interacts with the user. It catches the exceptions, prints appropriate error messages, and gives the user another chance to enter a correct file.

```
import java.io.FileNotFoundException;
import java.io.IOException;
import java.util.Scanner;

/**
   * This program reads a file containing numbers and analyzes its contents.
   * If the file doesn't exist or contains strings that are not numbers, an
   * error message is displayed.
   */
public class DataAnalyzer {
    public static void main(String[] args) {
        Scanner in = new Scanner(System.in);
        DataSetReader reader = new DataSetReader();
        boolean done = false;
        while (!done) {
            try {
                System.out.println("Please enter the file name: ");
                String filename = in.next();
                double[] data = reader.readFile(filename);
                double sum = 0;
                for (double d : data) { sum = sum + d; }
                System.out.println("The sum is " + sum);
                done = true;
            } catch (FileNotFoundException exception) {
                ...  
            }
        }
    }
}
```
The catch clauses in the main method give a human-readable error report if the file was not found or bad data was encountered.

The following `readFile` method of the `DataSetReader` class constructs the `Scanner` object and calls the `readData` method. It is unconcerned with any exceptions. If there is a problem with the input file, it simply passes the exception to its caller.

```java
public double[] readFile(String filename) throws IOException {
    File inFile = new File(filename);
    try (Scanner in = new Scanner(inFile)) {
        readData(in);
        return data;
    }
}
```

The method throws an `IOException`, the common superclass of `FileNotFoundException` (thrown by the `Scanner` constructor) and `BadDataException` (thrown by the `readData` method).

Next, here is the `readData` method of the `DataSetReader` class. It reads the number of values, constructs an array, and calls `readValue` for each data value.

```java
private void readData(Scanner in) throws BadDataException {
    if (!in.hasNextInt()) {
        throw new BadDataException("Length expected");
    }
    int numberOfValues = in.nextInt();
    data = new double[numberOfValues];
    for (int i = 0; i < numberOfValues; i++) {
        readValue(in, i);
    }
    if (in.hasNext()) {
        throw new BadDataException("End of file expected");
    }
}
```

This method checks for two potential errors. The file might not start with an integer, or it might have additional data after reading all values.
Chapter 11  Input/Output and Exception Handling

However, this method makes no attempt to catch any exceptions. Plus, if the readValue method throws an exception—which it will if there aren’t enough values in the file—the exception is simply passed on to the caller.

Here is the readValue method:

```java
private void readValue(Scanner in, int i) throws BadDataException {
    if (!in.hasNextDouble())
        throw new BadDataException("Data value expected");
    data[i] = in.nextDouble();
}
```

To see the exception handling at work, look at a specific error scenario:

1. DataAnalyzer.main calls DataSetReader.readFile.
2. readFile calls readData.
3. readData calls readValue.
4. readValue doesn’t find the expected value and throws a BadDataException.
5. readValue has no handler for the exception and terminates immediately.
6. readData has no handler for the exception and terminates immediately.
7. readFile has no handler for the exception and terminates immediately after closing the Scanner object.
8. DataAnalyzer.main has a handler for a BadDataException. That handler prints a message to the user. Afterward, the user is given another chance to enter a file name. Note that the statements computing the sum of the values have been skipped.

This example shows the separation between error detection (in the DataSetReader. readValue method) and error handling (in the DataAnalyzer.main method). In between the two are the readData and readFile methods, which just pass exceptions along.

`section_5/DataSetReader.java`

```java
import java.io.File;
import java.io.IOException;
import java.util.Scanner;

/**
 * Reads a data set from a file. The file must have the format
 * numberOfValues
 * value1
 * value2
 * ....
 */
public class DataSetReader {
    private double[] data;

    /**
     * Reads a data set.
     * @param filename the name of the file holding the data
     * @return the data in the file
     */
    public double[] readFile(String filename) throws IOException {
        
```
File inFile = new File(filename);
try (Scanner in = new Scanner(inFile))
{
    readData(in);
    return data;
}

/**
 * Reads all data.
 * @param in the scanner that scans the data
 */
private void readData(Scanner in) throws BadDataException
{
    if (!in.hasNextInt())
    {
        throw new BadDataException("Length expected");
    }
    int numberOfValues = in.nextInt();
    data = new double[numberOfValues];
    for (int i = 0; i < numberOfValues; i++)
    {
        readValue(in, i);
    }
    if (in.hasNext())
    {
        throw new BadDataException("End of file expected");
    }
}

/**
 * Reads one data value.
 * @param in the scanner that scans the data
 * @param i the position of the value to read
 */
private void readValue(Scanner in, int i) throws BadDataException
{
    if (!in.hasNextDouble())
    {
        throw new BadDataException("Data value expected");
    }
    data[i] = in.nextDouble();
}

import java.io.IOException;

/**
 * This class reports bad input data.
 */
public class BadDataException extends IOException
{
    public BadDataException()
    {
    }
    public BadDataException(String message)
    {
        super(message);
24. Why doesn’t the `DataSetReader.readFile` method catch any exceptions?
25. Suppose the user specifies a file that exists and is empty. Trace the flow of execution.
26. If the `readValue` method had to throw a `NoSuchElementException` instead of a `BadDataException` when the next input isn’t a floating-point number, how would the implementation change?
27. What happens to the `Scanner` object if the `readData` method throws an exception?
28. What happens to the `Scanner` object if the `readData` method doesn’t throw an exception?

**Practice It**  Now you can try these exercises at the end of the chapter: R11.16, R11.17, E11.13.

---

**Computing & Society 11.2**  The Ariane Rocket Incident

The European Space Agency (ESA), Europe’s counterpart to NASA, had developed a rocket model named 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, but the chance of them having the same mechanical failure was 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.
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.
- To describe an error condition, provide a subclass of an existing exception class.
- 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.
Chapter 11  Input/Output and Exception Handling

STANDARD LIBRARY ITEMS INTRODUCED IN THIS CHAPTER

```java
java.io.File
java.io.FileNotFoundException
java.io.IOException
java.io.PrintWriter
close
java.lang.AutoCloseable
java.lang.Character
isDigit
isLetter
isLowerCase
isUpperCase
isWhiteSpace
java.lang.Double
parseDouble
java.lang.Error
java.lang.Integer
parseInt
java.lang.IllegalArgumentException
java.lang.NullPointerException
java.lang.NumberFormatException
java.lang.RuntimeException
java.lang.String
replaceAll
split
java.lang.ThrowException
getMessage
printStackTrace
java.net.URL
openStream
java.util.InputMismatchException
java.util.NoSuchElementException
java.util.Scanner
close
hasNextLine
nextLine
useDelimiter
javax.swing.JFileChooser
getSelectedFile
showOpenDialog
showSaveDialog
```

REVIEW EXERCISES

**R11.1** 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?

**R11.2** 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.

**R11.3** What happens when you write to a PrintWriter without closing it? Produce a test program that demonstrates how you can lose data.

**R11.4** How do you open a file whose name contains a backslash, like `c:\temp\output.dat`?

**R11.5** 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?

**R11.6** What is the difference between throwing an exception and catching an exception?

**R11.7** 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?

**R11.8** Why don’t you need to declare that your method might throw an IndexOutOfBoundsException?

**R11.9** When your program executes a throw statement, which statement is executed next?

**R11.10** What happens if an exception does not have a matching catch clause?

**R11.11** What can your program do with the exception object that a catch clause receives?

**R11.12** 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?
• R11.13 What is the purpose of the try-with-resources statement? Give an example of how it can be used.

• R11.14 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.

• R11.15 Which exceptions can the next and nextInt methods of the Scanner class throw? Are they checked exceptions or unchecked exceptions?

• R11.16 Suppose the program in Section 11.5 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?

• R11.17 Can the readFile method in Section 11.5 throw a NullPointerException? If so, how?

• R11.18 The following code tries to close the writer without using a try-with-resources statement:

```java
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?)

• E11.1 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.

• E11.2 Write a program that reads a file, removes any blank lines, and writes the non-blank lines back to the same file.

• E11.3 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.

• E11.4 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

```java
/* 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!
```

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.

- **E11.5** Repeat Exercise E11.4, 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.

- **E11.6** 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.

- **E11.7** Write a program that asks the user for a file name and prints the number of characters, words, and lines in that file.

- **E11.8** 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

```plaintext
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.

- **E11.9** 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 Macintosh and Linux systems in the file `usr/share/dict/words`. (If you don’t have that file on your computer, use the file `ch19/words.txt` in the companion code for this book.) The program should print out all words that it cannot find in the word list.

- **E11.10** 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

```java
retnirPolleH ssalc cilbup

);"!dlroW ,olleH"(nltirp.tuo.metsyS
}
```

Of course, if you run `Reverse` twice on the same file, you get back the original file.
**E11.11** 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
```

**E11.12** 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?

**E11.13** 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.

**E11.14** Modify the BankAccount class to throw an IllegalArgumentException when the account is constructed with a negative balance, when a negative amount is deposited, or when an amount that is not between 0 and the current balance is withdrawn. Write a test program that causes all three exceptions to occur and that catches them all.

**E11.15** Repeat Exercise E11.14, but throw exceptions of three exception types that you provide.

**E11.16** Modify the DataSetReader class so that you do not call hasNextInt or hasNextDouble. Simply have nextInt and nextDouble throw a NoSuchElementException and catch it in the main method.

**P11.1** 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.

**P11.2** 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).

**P11.3** Using the mechanism described in Special Topic 11.1, 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.

**P11.4** 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?""

Implement a class CSVReader that reads a CSV file, and provide methods

```java
int numberOfRows()
int numberOfFields(int row)
String field(int row, int column)
```

**P11.5** Find an interesting data set in CSV format (or in spreadsheet format, sn then use a spreadsheet to save the data as CSV). Using the CSVReader class from the Exercise P11.4, read the data and compute a summary, such as the maximum, minimum, or average of one of the columns.

**P11.6** Using the mechanism described in Special Topic 11.1, write a program that reads all data from a web page and prints all hyperlinks of the form

```html
<a href="link">link 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.)

**P11.7** Write a program that reads in a set of coin descriptions from a file. The input file has the format

```plaintext
coinName1 coinValue1
coinName2 coinValue2
...```

Add a method

```java
void read(Scanner in) throws FileNotFoundException
```

to the Coin class of Section 8.2. Throw an exception if the current line is not properly formatted. Then implement a method

```java
static ArrayList<Coin> readFile(String filename) throws FileNotFoundException
```

In the main method, call readFile. If an exception is thrown, give the user a chance to select another file. If you read all coins successfully, print the total value.

**P11.8** Design a class Bank that contains a number of bank accounts. Each account has an account number and a current balance. Add an accountNumber field to the BankAccount class. Store the bank accounts in an array list. Write a readFile method of the Bank class for reading a file with the format

```plaintext
accountNumber1  balance1
accountNumber2  balance2
...```

Implement read methods for the Bank and BankAccount classes. Write a sample program to read in a file with bank accounts, then print the account with the highest balance. If the file is not properly formatted, give the user a chance to select another file.

**Business P11.9** 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 P11.10** Write a program that reads a text file as described in Exercise P11.9, and that writes a separate file for each service category, containing the entries for that category. Name the output files *Dinner.txt*, *Conference.txt*, and so on.

**Business P11.11** 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 P11.12** 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, \(R = 500\) Ω, and \(C = 0.25\) μF. Consequently

\[
v(t) = 12 \left( 1 - e^{-0.008t} \right)
\]

where \(t\) has units of μs. 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 μs, the twelfth entry in the output file would be:

110  7.02261

**Science P11.13** The figure at right shows a plot of the capacitor voltage from the circuit shown in Exercise P11.12. 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 μs 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 μs. 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 P11.14 Suppose a file contains bond energies and bond lengths for covalent bonds in the following format:

<table>
<thead>
<tr>
<th>Single, double, or triple bond</th>
<th>Bond energy (kJ/mol)</th>
<th>Bond length (nm)</th>
</tr>
</thead>
<tbody>
<tr>
<td>C|C</td>
<td>370</td>
<td>0.154</td>
</tr>
<tr>
<td>C|C</td>
<td>680</td>
<td>0.13</td>
</tr>
<tr>
<td>C||C</td>
<td>890</td>
<td>0.12</td>
</tr>
<tr>
<td>C|H</td>
<td>435</td>
<td>0.11</td>
</tr>
<tr>
<td>C|N</td>
<td>305</td>
<td>0.15</td>
</tr>
<tr>
<td>C|O</td>
<td>360</td>
<td>0.14</td>
</tr>
<tr>
<td>C|F</td>
<td>450</td>
<td>0.14</td>
</tr>
<tr>
<td>C|Cl</td>
<td>340</td>
<td>0.18</td>
</tr>
<tr>
<td>O|H</td>
<td>500</td>
<td>0.10</td>
</tr>
<tr>
<td>O|O</td>
<td>220</td>
<td>0.15</td>
</tr>
<tr>
<td>O|Si</td>
<td>375</td>
<td>0.16</td>
</tr>
<tr>
<td>N|H</td>
<td>430</td>
<td>0.10</td>
</tr>
<tr>
<td>N|O</td>
<td>250</td>
<td>0.12</td>
</tr>
<tr>
<td>F|F</td>
<td>160</td>
<td>0.14</td>
</tr>
<tr>
<td>H|H</td>
<td>435</td>
<td>0.074</td>
</tr>
</tbody>
</table>

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

1. 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.

2. The program throws a FileNotFoundException and terminates.

3. 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.

4. Add a variable count that is incremented whenever a number is read. At the end, print the average, not the total, as

\[
\text{out.printf("Average: %8.2f\n", total / count);}
\]
Because the string "Average" is three characters longer than "Total", change the other output to out.printf("%18.2f\n", value).

5. Add a variable count that is incremented whenever a number is read. Only write a new line when it is even.
   ```java
   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:
   ```java
   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:
   ```java
   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:
    ```java
        int j = i - 1;
        while (Character.isWhiteSpace(line.charAt(j))) {
            j--;
        }
    ```
    Then extract the country name:
    ```java
        String countryName = line.substring(0, j + 1);
    ```

11. args[0] is "-d" and args[1] is "file1.txt"

12. 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
    ```java
    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) {
    ```java
    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 542.

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. To pass the exception message string to the IllegalArgumentException superclass.

23. Because file corruption is beyond the control of the programmer, this should be a checked exception, so it would be wrong to extend RuntimeException or IllegalArgumentException. Because the error is related to input, IOException would be a good choice.

24. It would not be able to do much with them. The DataSetReader class is a reusable class that
may be used for systems with different languages and different user interfaces. Thus, it cannot engage in a dialog with the program user.

25. DataAnalyzer.main calls DataSetReader.readFile, which calls readData. The call in.hasNextInt() returns false, and readData throws a BadDataException. The readFile method doesn’t catch it, so it propagates back to main, where it is caught.

26. It could simply be

```java
private void readValue(Scanner in, int i) {
    data[i] = in.nextDouble();
}
```

The nextDouble method throws a NoSuchElementException or an InputMismatchException (which is a subclass of NoSuchElementException) when the next input isn’t a floating-point number. That exception isn’t a checked exception, so it need not be declared.

27. The close method is called on the Scanner object before the exception is propagated to its handler.

28. The close method is called on the Scanner object before the readFile method returns to its caller.
WORKED EXAMPLE 11.1 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’s 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.

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.
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 classes and 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.

We use a helper class RecordReader for this purpose and construct two objects, one each for processing the boy and girl names. Each RecordReader maintains a separate total, updates it by adding the current percentage, and prints names until the limit has been reached. Our main processing loop then becomes

```java
RecordReader boys = new RecordReader(LIMIT);
RecordReader girls = new RecordReader(LIMIT);
```
while (boys.hasMore() || girls.hasMore())
{
    int rank = in.nextInt();
    System.out.print(rank + " ");
    boys.process(in);
    girls.process(in);
    System.out.println();
}

Here is the code of the process method:

    /**
    * Reads an input record and prints the name if the current total is less than the limit.
    * @param in the input stream
    */
    public void process(Scanner in)
    {
        String name = in.next();
        int count = in.nextInt();
        double percent = in.nextDouble();

        if (total < limit) { System.out.print(name + " "); } 
        total = total + percent;
    }

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 E11.12 asks you to study how this distribution has changed over the years.
worked_example_1/RecordReader.java

```java
import java.util.Scanner;

/**
 * This class processes baby name records.
 */
public class RecordReader {
    private double total;
    private double limit;

    /**
     * Constructs a RecordReader with a zero total.
     */
    public RecordReader(double aLimit) {
        total = 0;
        limit = aLimit;
    }

    /**
     * Reads an input record and prints the name if the current total is less
     * than the limit.
     * @param in the input stream
     */
    public void process(Scanner in) {
        String name = in.next();
        int count = in.nextInt();
        double percent = in.nextDouble();
        if (total < limit) { System.out.print(name + " "); }
        total = total + percent;
    }

    /**
     * Checks whether there are more inputs to process.
     * @return true if the limit has not yet been reached
     */
    public boolean hasMore() {
        return total < limit;
    }
}
```
CHAPTER 12
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 566

12.2 RELATIONSHIPS BETWEEN CLASSES 569
    HT  Using CRC Cards and UML Diagrams in Program Design 572
    ST  Attributes and Methods in UML Diagrams 573
    ST2 Multiplicities 574
    ST3 Aggregation, Association, and Composition 574

12.3 APPLICATION: PRINTING AN INVOICE 575
    C&S Databases and Privacy 586
    WE 1 Simulating an Automatic Teller Machine 587
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

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 Image](https://via.placeholder.com/150)

**Figure 1**
An Invoice

<table>
<thead>
<tr>
<th>Item</th>
<th>Qty</th>
<th>Price</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>Toaster</td>
<td>3</td>
<td>$29.95</td>
<td>$89.85</td>
</tr>
<tr>
<td>Hair Dryer</td>
<td>1</td>
<td>$24.95</td>
<td>$24.95</td>
</tr>
<tr>
<td>Car Vacuum</td>
<td>2</td>
<td>$19.99</td>
<td>$39.98</td>
</tr>
</tbody>
</table>

**AMOUNT DUE:** $154.78
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.

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 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

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 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.

---

**Figure 2**  A CRC Card
### 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

Many classes need other classes in order to do their jobs. For example, in Section 8.2.2, 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 8.2.2 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.

As you saw in Section 8.2, dependency is denoted by a dashed line with a \(\Rightarrow\)-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.

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).

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:

```java
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:

```java
public class Tire {
    private String rating;
    private Circle boundary;
    ...
}
```

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):

```java
public class Car extends Vehicle {
    private Tire[] tires;
    ...
}
```

### Figure 6

**UML Notation for Inheritance and Aggregation**
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>
<thead>
<tr>
<th>Relationship</th>
<th>Symbol</th>
<th>Line Style</th>
<th>Arrow Tip</th>
</tr>
</thead>
<tbody>
<tr>
<td>Inheritance</td>
<td>Solid</td>
<td>Solid</td>
<td>Triangle</td>
</tr>
<tr>
<td>Interface Implementation</td>
<td>Dotted</td>
<td>Dotted</td>
<td>Triangle</td>
</tr>
<tr>
<td>Aggregation</td>
<td>Solid</td>
<td>Solid</td>
<td>Diamond</td>
</tr>
<tr>
<td>Dependency</td>
<td>Dotted</td>
<td>Dotted</td>
<td>Open</td>
</tr>
</tbody>
</table>

6. Consider the `CashRegisterTester` class of Section 8.2. 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.
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.

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.
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](image)

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.

![An Association Relationship](image)

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.

**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 obviously 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 very 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**

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 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.
Use CRC cards
to find classes,
responsibilities,
and collaborators.

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.

12.3.2 CRC Cards

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

(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:

```
Invoice

format the invoice
```

© Scott Cramer/iStockphoto.

An invoice lists the charges for each item and the amount due.

© Scott Cramer/iStockphoto.
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:

<table>
<thead>
<tr>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>format the address</td>
</tr>
</tbody>
</table>

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:

<table>
<thead>
<tr>
<th>Invoice</th>
</tr>
</thead>
<tbody>
<tr>
<td>format the invoice</td>
</tr>
<tr>
<td></td>
</tr>
</tbody>
</table>

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:

<table>
<thead>
<tr>
<th>Product</th>
</tr>
</thead>
<tbody>
<tr>
<td>get description</td>
</tr>
<tr>
<td>get unit price</td>
</tr>
</tbody>
</table>
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.

We now have a set of CRC cards that completes the CRC card process.

### 12.3.3 UML Diagrams

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 7 shows the class relationships that we discovered.
12.3.4 Method Documentation

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.

```java
/**
 * 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()
    {
    }
}
```

Use javadoc comments (with the method bodies left blank) to record the behavior of classes.
Then run the javadoc program to obtain a neatly formatted version of your documentation in HTML format (see Figure 8).

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.
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:

```java
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:

```java
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:

```java
/**
   * 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/bjeo6examples) 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.

section_3/InvoicePrinter.java

```java
/**
   * This program demonstrates the invoice classes by
   * printing a sample invoice.
   */
public class InvoicePrinter
{
    public static void main(String[] args)
    {
        Address samsAddress
            = new Address("Sam's Small Appliances",
                          "100 Main Street", "Anytown", "CA", "98765");

        Invoice samsInvoice = new Invoice(samsAddress);
        samsInvoice.add(new Product("Toaster", 29.95), 3);
        samsInvoice.add(new Product("Hair dryer", 24.95), 1);
        samsInvoice.add(new Product("Car vacuum", 19.99), 2);

        System.out.println(samsInvoice.format());
    }
}
```

section_3/Invoice.java

```java
import java.util.ArrayList;
```
/**
 * Describes an invoice for a set of purchased products.
 */
public class Invoice {
    private Address billingAddress;
    private ArrayList<LineItem> items;

    /**
     * Constructs an invoice.
     * @param anAddress the billing address
     */
    public Invoice(Address anAddress) {
        items = new ArrayList<LineItem>();
        billingAddress = anAddress;
    }

    /**
     * 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) {
        LineItem anItem = new LineItem(aProduct, quantity);
        items.add(anItem);
    }

    /**
     * Formats the invoice.
     * @return the formatted invoice
     */
    public String format() {
        String r = "                     I N V O I C E

" + billingAddress.format() + String.format("%-30s%8s%5s%8s
", "Description", "Price", "Qty", "Total");

        for (LineItem item : items) {
            r = r + item.format() + "\n";
        }

        r = r + String.format("\nAMOUNT DUE: $%8.2f", getAmountDue());

        return r;
    }

    /**
     * Computes the total amount due.
     * @return the amount due
     */
    private double getAmountDue() {
        double amountDue = 0;
        for (LineItem item : items) {
            amountDue = amountDue + item.getTotalPrice();
        }

        return amountDue;
    }
}
section_3/LineItem.java

```java
/**
 * Describes a quantity of an article to purchase.
 */
public class LineItem {
    private int quantity;
    private Product theProduct;

    /**
     * Constructs an item from the product and quantity.
     * @param aProduct the product
     * @param aQuantity the item quantity
     */
    public LineItem(Product aProduct, int aQuantity) {
        theProduct = aProduct;
        quantity = aQuantity;
    }

    /**
     * Computes the total cost of this line item.
     * @return the total price
     */
    public double getTotalPrice() {
        return theProduct.getPrice() * quantity;
    }

    /**
     * Formats this item.
     * @return a formatted string of this line item
     */
    public String format() {
        return String.format("%-30s%8.2f%5d%8.2f", 
                             theProduct.getDescription(), 
                             theProduct.getPrice(), 
                             quantity, getTotalPrice());
    }
}
```

section_3/Product.java

```java
/**
 * Describes a product with a description and a price.
 */
public class Product {
    private String description;
    private double price;

    /**
     * Constructs a product from a description and a price.
     * @param aDescription the product description
     */
    public Product(String aDescription, double aPrice) {
        description = aDescription;
        price = aPrice;
    }

    /**
     * Returns the description.
     * @return the product description
     */
    public String getDescription() {
        return description;
    }

    /**
     * Returns the price.
     * @return the product price
     */
    public double getPrice() {
        return price;
    }

    /**
     * Computes the total cost of the product.
     * @return the total price
     */
    public double getTotalPrice() {
        return price;
    }
}
```
12.3 Application: Printing an Invoice

```java
public Product(String aDescription, double aPrice)
{
    description = aDescription;
    price = aPrice;
}

/**
 * Gets the product description.
 * @return the description
 */
public String getDescription()
{
    return description;
}

/**
 * Gets the product price.
 * @return the unit price
 */
public double getPrice()
{
    return price;
}

section_3/Address.java

```
Chapter 12  Object-Oriented Design

```java
@return the address as a string with three lines
*/
public String format()
{
    return name + "\n" + street + "\n"
    + city + ", " + state + " " + zip;
}
```

13. Which class 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.

If you pay road or bridge tolls with an electronic pass, your records may not be private.
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.

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?
Chapter 12  Object-Oriented Design

**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 8.2.

**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?
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.
**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 2016/10/1 17:30 18:30
CS1 class 2016/10/2 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
   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  or   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 speed 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.

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.

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.

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. Reword 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. 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.
11. There would be no relationship.
12. The Invoice class is responsible for computing the amount due. It collaborates with the LineItem class.
13. 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.
**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 9), 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.
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.

<table>
<thead>
<tr>
<th>Customer</th>
</tr>
</thead>
<tbody>
<tr>
<td>get accounts</td>
</tr>
<tr>
<td>match number and PIN</td>
</tr>
</tbody>
</table>

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.

<table>
<thead>
<tr>
<th>Bank</th>
</tr>
</thead>
<tbody>
<tr>
<td>find customer</td>
</tr>
<tr>
<td>read customers</td>
</tr>
</tbody>
</table>

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.
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 10). 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.

<table>
<thead>
<tr>
<th>ATM</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>manage state</strong></td>
</tr>
<tr>
<td><strong>select customer</strong></td>
</tr>
<tr>
<td><strong>select account</strong></td>
</tr>
<tr>
<td><strong>execute transaction</strong></td>
</tr>
</tbody>
</table>
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 11  Relationships Between the ATM Classes

Figure 11 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:

```java
/**
   * 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)
   */
```
Simulating an Automatic Teller Machine

@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;
   ...
}
Most methods are very straightforward to implement. Consider the `selectCustomer` method. From the design documentation, we have the description

```java
/**
   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:

```java
class BankAccountExample1 {

class BankAccount {

    private double balance;

    public BankAccount() {
        balance = 0;
    }

    public BankAccount(double initialBalance) {
        balance = initialBalance;
    }

    public void deposit(double amount) {
    

    }
}
```

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.

```java
/**
   A bank account has a balance that can be changed by
   deposits and withdrawals.
*/
public class BankAccount {

    private double balance;

    /**
     Constructs a bank account with a zero balance.
     */
    public BankAccount() {
        balance = 0;
    }

    /**
     Constructs a bank account with a given balance.
     @param initialBalance the initial balance
     */
    public BankAccount(double initialBalance) {
        balance = initialBalance;
    }

    /**
     Deposits money into the account.
     @param amount the amount of money to withdraw
     */
    public void deposit(double amount) {
    

    }
```
### Simulating an Automatic Teller Machine

#### bank.Account

```java
public class Account
{
    private double balance;

    public void deposit(double amount)
    {
        balance += amount;
    }

    public void withdraw(double amount)
    {
        balance -= amount;
    }

    public double getBalance()
    {
        return balance;
    }
}
```

### worked_example_1/Customer.java

```java
public class Customer
{
    private int customerNumber;
    private int pin;
    private BankAccount checkingAccount;
    private BankAccount savingsAccount;

    public Customer(int aNumber, int aPin)
    {
        customerNumber = aNumber;
        pin = aPin;
        checkingAccount = new BankAccount();
        savingsAccount = new BankAccount();
    }

    public boolean match(int aNumber, int aPin)
    {
        return customerNumber == aNumber && pin == aPin;
    }
}
```

---

*Big Java, 6e, Cay Horstmann, Copyright © 2015 John Wiley and Sons, Inc. All rights reserved.*
Public BankAccount getCheckingAccount() {
    return checkingAccount;
}

/**
 * Gets the savings account of this customer.
 * @return the checking account
 */
public BankAccount getSavingsAccount() {
    return savingsAccount;
}

import java.io.File;
import java.io.IOException;
import java.util.ArrayList;
import java.util.Scanner;

/**
 * A bank contains customers.
 */
public class Bank {
    private ArrayList<Customer> customers;

    /**
     * Constructs a bank with no customers.
     */
    public Bank() {
        customers = new ArrayList<Customer>();
    }

    /**
     * Reads the customer numbers and pins.
     * @param filename the name of the customer file
     */
    public void readCustomers(String filename) throws IOException {
        Scanner in = new Scanner(new File(filename));
        while (in.hasNext()) {
            int number = in.nextInt();
            int pin = in.nextInt();
            Customer c = new Customer(number, pin);
            addCustomer(c);
        }
        in.close();
    }

}
40 Adds a customer to the bank.
41 @param c the customer to add
42 */
43 public void addCustomer(Customer c) {
44    customers.add(c);
45 }
46
47 /**
48 Finds a customer in the bank.
49 @param aNumber a customer number
50 @param aPin a personal identification number
51 @return the matching customer, or null if no customer matches
52 */
53 public Customer findCustomer(int aNumber, int aPin) {
54    for (Customer c : customers) {
55        if (c.match(aNumber, aPin)) {
56            return c;
57        }
58    }
59    return null;
60 }
61

***/
An ATM that accesses a bank.
*/
public class ATM {
    public static final int CHECKING = 1;
    public static final int SAVINGS = 2;
    private int state;
    private int customerNumber;
    private Customer currentCustomer;
    private BankAccount currentAccount;
    private Bank theBank;

    public static final int START = 1;
    public static final int PIN = 2;
    public static final int ACCOUNT = 3;
    public static final int TRANSACT = 4;

    /**
     * Constructs an ATM for a given bank.
     * @param aBank the bank to which this ATM connects
     */
    public ATM(Bank aBank) {
        theBank = aBank;
        reset();
    }
```java
/**
 * Resets the ATM to the initial state.
 */
public void reset()
{
    customerNumber = -1;
    currentAccount = null;
    state = START;
}

/**
 * Sets the current customer number and sets state to PIN.
 * (Precondition: state is START)
 * @param number the customer number
 */
public void setCustomerNumber(int number)
{
    customerNumber = number;
    state = PIN;
}

/**
 * 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)
{
    currentCustomer
        = theBank.findCustomer(customerNumber, pin);
    if (currentCustomer == null)
    {
        state = START;
    }
    else
    {
        state = ACCOUNT;
    }
}

/**
 * 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)
{
    if (account == CHECKING)
    {
        currentAccount = currentCustomer.getCheckingAccount();
    }
    else
    {
        currentAccount = currentCustomer.getSavingsAccount();
    }
    state = TRANSACT;
}
/**
 * Withdraws amount from current account.
 * (Precondition: state is TRANSACT)
 * @param value the amount to withdraw
 */
 public void withdraw(double value) {
     currentAccount.withdraw(value);
 }

/**
 * Deposits amount to current account.
 * (Precondition: state is TRANSACT)
 * @param value the amount to deposit
 */
 public void deposit(double value) {
     currentAccount.deposit(value);
 }

/**
 * Gets the balance of the current account.
 * (Precondition: state is TRANSACT)
 * @return the balance
 */
 public double getBalance() {
     return currentAccount.getBalance();
 }

/**
 * Moves back to the previous state.
 */
 public void back() {
     if (state == TRANSACT) {
         state = ACCOUNT;
     } else if (state == ACCOUNT) {
         state = PIN;
     } else if (state == PIN) {
         state = START;
     }
 }

/**
 * Gets the current state of this ATM.
 * @return the current state
 */
 public int getState() {
     return state;
 }
The following class implements a console-based user interface for the ATM.

```java
public class ATMSimulator {
    public static void main(String[] args) {
        ATM theATM;
        try {
            Bank theBank = new Bank();
            theBank.readCustomers("customers.txt");
            theATM = new ATM(theBank);
        }
        catch (IOException e) {
            System.out.println("Error opening accounts file.");
            return;
        }
        Scanner in = new Scanner(System.in);

        while (true) {
            int state = theATM.getState();
            if (state == ATM.START) {
                System.out.print("Enter customer number: ");
                int number = in.nextInt();
                theATM.setCustomerNumber(number);
            } else if (state == ATM.PIN) {
                System.out.print("Enter PIN: ");
                int pin = in.nextInt();
                theATM.selectCustomer(pin);
            } else if (state == ATM.ACCOUNT) {
                System.out.print("A=Checking, B=Savings, C=Quit: ");
                String command = in.next();
                if (command.equalsIgnoreCase("A")) {
                    theATM.selectAccount(ATM.CHECKING);
                } else if (command.equalsIgnoreCase("B")) {
                    theATM.selectAccount(ATM.SAVINGS);
                } else if (command.equalsIgnoreCase("C")) {
                    theATM.reset();
                }
            }
        }
    }
}
```
else
{
    System.out.println("Illegal input!");
}

else if (state == ATM.TRANSACT)
{
    System.out.println("Balance=" + theATM.getBalance());
    System.out.print("A=Deposit, B=Withdrawal, C=Cancel: ");
    String command = in.next();
    if (command.equalsIgnoreCase("A"))
    {
        System.out.print("Amount: ");
        double amount = in.nextDouble();
        theATM.deposit(amount);
        theATM.back();
    }
    else if (command.equalsIgnoreCase("B"))
    {
        System.out.print("Amount: ");
        double amount = in.nextDouble();
        theATM.withdraw(amount);
        theATM.back();
    }
    else if (command.equalsIgnoreCase("C"))
    {
        theATM.back();
    }
    else
    {
        System.out.println("Illegal input!");
    }
}
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

import java.awt.BorderLayout;
import java.awt.GridLayout;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import javax.swing.JButton;
import javax.swing.JPanel;
import javax.swing.JTextField;
A component that lets the user enter a number, using a button pad labeled with digits.

```java
public class KeyPad extends JPanel {
    private JPanel buttonPanel;
    private JButton clearButton;
    private JTextField display;

    /**
     * Constructs the keypad panel.
     */
    public KeyPad()
    {
        setLayout(new BorderLayout());

        // Add display field
        display = new JTextField();
        add(display, "North");

        // Make button panel
        buttonPanel = new JPanel();
        buttonPanel.setLayout(new GridLayout(4, 3));

        // Add digit buttons
        addButton("7");
        addButton("8");
        addButton("9");
        addButton("4");
        addButton("5");
        addButton("6");
        addButton("1");
        addButton("2");
        addButton("3");
        addButton("0");
        addButton(".");

        // Add clear entry button
        clearButton = new JButton("CE");
        buttonPanel.add(clearButton);

        class ClearButtonListener implements ActionListener
        {
            public void actionPerformed(ActionEvent event)
            {
                display.setText("");
            }
        }
        ActionListener listener = new ClearButtonListener();
        clearButton.addActionListener(new ClearButtonListener());

        add(buttonPanel, "Center");
    }
}
```
```java
/**
 * Adds a button to the button panel.
 * @param label the button label
 */
private void addButton(final String label)
{
    class DigitButtonListener implements ActionListener
    {
        public void actionPerformed(ActionEvent event)
        {
            // Don't add two decimal points
            if (label.equals("."))
                && display.getText().indexOf(".") != -1)
            {
                return;
            }

            // Append label text to button
            display.setText(display.getText() + label);
        }
    }

    JButton button = new JButton(label);
    buttonPanel.add(button);
    ActionListener listener = new DigitButtonListener();
    button.addActionListener(listener);
}

/**
 * Gets the value that the user entered.
 * @return the value in the text field of the keypad
 */
public double getValue()
{
    return Double.parseDouble(display.getText());
}

/**
 * Clears the display.
 */
public void clear()
{
    display.setText(""d
}
```

**worked_example_1/ATMFrame.java**

```java
import java.awt.FlowLayout;
import java.awt.GridLayout;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import javax.swing.JButton;
import javax.swing.JFrame;
import javax.swing.JPanel;
import javax.swing.JTextArea;
```

```
```
A frame displaying the components of an ATM.

```java
e/*
epublic class ATMFrame extends JFrame
{
  private static final int FRAME_WIDTH = 300;
  private static final int FRAME_HEIGHT = 300;
  private JButton aButton;
  private JButton bButton;
  private JButton cButton;
  private KeyPad pad;
  private JTextArea display;
  private ATM theATM;

  /** Constructs the user interface of the ATM frame. */
  public ATMFrame(ATM anATM)
  {
    theATM = anATM;
    pad = new KeyPad();
    display = new JTextArea(4, 20);
    aButton = new JButton("  A  ");
    aButton.addActionListener(new AButtonListener());
    bButton = new JButton("  B  ");
    bButton.addActionListener(new BButtonListener());
    cButton = new JButton("  C  ");
    cButton.addActionListener(new CButtonListener());
    JPanel buttonPanel = new JPanel();
    buttonPanel.add(aButton);
    buttonPanel.add(bButton);
    buttonPanel.add(cButton);
    setLayout(new FlowLayout());
    add(pad);
    add(display);
    add(buttonPanel);
    showState();
    setSize(FRAME_WIDTH, FRAME_HEIGHT);
  }

  /** Updates display message. */
  public void showState()
  {
    int state = theATM.getState();
    pad.clear();
  }
}
```

if (state == ATM.START) {
    display.setText("Enter customer number\nA = OK");
} else if (state == ATM.PIN) {
    display.setText("Enter PIN\nA = OK");
} else if (state == ATM.ACCOUNT) {
    display.setText("Select Account\n    " + "A = Checking\nB = Savings\nC = Exit");
} else if (state == ATM.TRANSACT) {
    display.setText("Balance = " + theATM.getBalance()
        + "\nEnter amount and select transaction\n    " + "A = Withdraw\nB = Deposit\nC = Cancel");
}
}

class AButtonListener implements ActionListener {
    public void actionPerformed(ActionEvent event) {
        int state = theATM.getState();
        if (state == ATM.START) {
            theATM.setCustomerNumber((int) pad.getValue());
        } else if (state == ATM.PIN) {
            theATM.selectCustomer((int) pad.getValue());
        } else if (state == ATM.ACCOUNT) {
            theATM.selectAccount(ATM.CHECKING);
        } else if (state == ATM.TRANSACT) {
            theATM.withdraw(pad.getValue());
            theATM.back();
        }
        showState();
    }
}

class BButtonListener implements ActionListener {
    public void actionPerformed(ActionEvent event) {
        int state = theATM.getState();
        if (state == ATM.ACCOUNT) {
            theATM.selectAccount(ATM.SAVINGS);
        } else if (state == ATM.TRANSACT) {
            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   }
CHAPTER 13

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

CHAPTER CONTENTS

13.1 TRIANGLE NUMBERS 594
   CE1 Infinite Recursion 598
   CE2 Tracing Through Recursive Methods 598
   HT1 Thinking Recursively 599
   WE1 Finding Files

13.2 RECURSIVE HELPER METHODS 602

13.3 THE EFFICIENCY OF RECURSION 604

13.4 PERMUTATIONS 609
   C&S The Limits of Computation 612

13.5 MUTUAL RECURSION 614

13.6 BACKTRACKING 620
   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

We begin this chapter with a very simple example that demonstrates the power of thinking recursively. In this example, we will look at triangle shapes such as this one:

```
[]
[][
[[]]
```

We’d like to compute the area of a triangle of width \( n \), assuming that each [] square has area 1. The area of the triangle 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.

You may know that there is a very simple formula to compute these numbers, but you should pretend for now that you don’t know about it. The ultimate purpose of this section is not to compute triangle numbers, but to learn about the concept of recursion by working through a simple example.

Here is the outline of the class that we will develop:

```java
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:

```java
public int getArea() {
    if (width == 1) { return 1; }
    ...
}
```
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

```
smallerArea + 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;
    }
}
```

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.
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 is ≤ 0.

Recursion is not really necessary to compute the triangle numbers. The area of a triangle equals the sum

\[
1 + 2 + 3 + \ldots + \text{width}
\]

Of course, we can program a simple loop:

```java
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

\[
1 + 2 + \ldots + n = n \times (n + 1)/2
\]

Thus, the area equals

\[
\text{width} \times (\text{width} + 1) / 2
\]

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.

```java
/**
 * A triangular shape composed of stacked unit squares like this:
 * 
 * ...
 */
public class Triangle
{
    private int width;

    /**
     * Constructs a triangular shape.
     * @param aWidth the width (and height) of the triangle
     */
    public Triangle(int aWidth)
    {
        width = aWidth;
    }

    /**
     * Computes the area of the triangle.
     * @return the area
     */
    public int getArea()
    {
```
1. Why is the statement `else if (width == 1) { return 1; }` in the final version of the `getArea` method unnecessary?

2. How would you modify the program to recursively compute the area of a square?

3. 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?

```java
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
    }
}
```

4. 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.

5. Consider the following recursive method:

```java
public static int mystery(int n) {
    if (n <= 0) { return 0; }
    else {
        // Your code here
    }
}
```
```java
int smaller = n - 1;
return mystery(smaller) + n * n;
}
```

What is `mystery(4)`?

**Practice It**


---

### 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.

---

### 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.

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:

```java
public class Palindromes {
    // Tests whether a text is a palindrome.
    public static boolean isPalindrome(String text) {
        // ... 
    }
}
```

Step 1 Consider various ways to simplify inputs.

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
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

"I’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:

```java
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)
            {
```
Chapter 13
Recursion

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:

```java
/**
 * 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.
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.

```java
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:

```java
public static boolean isPalindrome(String text) {
    return isPalindrome(text, 0, text.length() - 1);
}
```

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

\[
\begin{align*}
  f_1 &= 1 \\
  f_2 &= 1 \\
  f_n &= f_{n-1} + f_{n-2}
\end{align*}
\]

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:

section_3/RecursiveFib.java

```java
import java.util.Scanner;

/**
 * This program computes Fibonacci numbers using a recursive method.
 */

public class RecursiveFib {
    public static void main(String[] args) {
        Scanner in = new Scanner(System.in);
        System.out.print("Enter n: ");
        int n = in.nextInt();
```
for (int i = 1; i <= n; i++)
{
    long f = fib(i);
    System.out.println("fib(" + i + ") = " + f);
}

/**
 * Computes a Fibonacci number.
 * @param n an integer
 * @return the n-th Fibonacci number
 */
public static long fib(int n)
{
    if (n <= 2) { return 1; }
    else { return fib(n - 1) + fib(n - 2); }
}

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:
public static long fib(int n) {
    System.out.println("Entering fib: n = "+ n);
    long f;
    if (n <= 2) { f = 1; } else { f = fib(n - 1) + fib(n - 2); }
    System.out.println("Exiting fib: n = "+ n + " return value = "+ f);
    return f;
}
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:

```java
import java.util.Scanner;

/**
   This program computes Fibonacci numbers using an iterative method.
*/
public class LoopFib {
    public static void main(String[] args) {
        Scanner in = new Scanner(System.in);
        System.out.print("Enter n: ");
        int n = in.nextInt();
        for (int i = 1; i <= n; i++) {
            long f = fib(i);
            System.out.println("fib(" + i + ") = " + f);
        }
    }

    /**
     * Computes a Fibonacci number.
     * @param n an integer
     * @return the \( n \)th Fibonacci number
     */
    public static long fib(int n) {
        if (n <= 2) { return 1; }
        else {
            // Code continues here
        }
    }
}
```
long olderValue = 1;
long oldValue = 1;
long newValue = 1;
for (int i = 3; i <= n; i++)
{
    newValue = oldValue + olderValue;
    olderValue = oldValue;
    oldValue = newValue;
}
return newValue;

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:

```java
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; }  
        }
        else if (!Character.isLetter(last)) { end--; }
        else 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.
This solution keeps two index variables: start and end. The first index starts at the beginning of the string and is advanced whenever a letter has been matched or a non-letter has been ignored. The second index starts at the end of the string and moves toward the beginning. When the two index variables meet, the iteration stops.

Both the iteration and the recursion run at about the same speed. If a palindrome has \( n \) characters, the iteration executes the loop between \( n/2 \) and \( n \) times, depending on how many of the characters are letters, because one or both index variables are moved in each step. Similarly, the recursive solution calls itself between \( n/2 \) and \( n \) times, because one or two characters are removed in each step.

In such a situation, the iterative solution tends to be a bit faster, because each recursive method call takes a certain amount of processor time. In principle, it is possible for a smart compiler to avoid recursive method calls if they follow simple patterns, but most Java compilers don’t do that. From that point of view, an iterative solution is preferable.

However, many problems have recursive solutions that are easier to understand and implement correctly than their iterative counterparts. Sometimes there is no obvious iterative solution at all—see the example in the next section. There is a certain elegance and economy of thought to recursive solutions that makes them more appealing. As the computer scientist (and creator of the GhostScript interpreter for the PostScript graphics description language) L. Peter Deutsch put it: “To iterate is human, to recurse divine.”

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 \( 1 + 2 + 3 + \ldots + \text{width} \)?

11. You can compute the factorial function either with a loop, using the definition that \( n! = 1 \times 2 \times \ldots \times n \), 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

In this section, we will study a more complex example of recursion that would be difficult to program with a simple loop. (As Exercise P13.3 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"  "tea"  "tae"
"eta"  "eat"  "tea"  "tea"
"aet"  "eta"  "tea"  "tae"

The permutations of a string can be obtained more naturally through recursion than with a loop.
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\text{th} letter:

\begin{verbatim}
String shorter = word.substring(0, i) + word.substring(i + 1);
\end{verbatim}

We compute the permutations of the shorter word:

\begin{verbatim}
ArrayList<String> shorterPermutations = permutations(shorter);
\end{verbatim}

Finally, we add the removed letter to the front of all permutations of the shorter word.

\begin{verbatim}
for (String s : shorterPermutations)
{
    result.add(word.charAt(i) + s);
}
\end{verbatim}

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—itsel.
System.out.println(s);
}

/**
 * Gets all permutations of a given word.
 * @param word the string to permute
 * @return a list of all permutations
 */
public static ArrayList<String> permutations(String word)
{
    ArrayList<String> result = new ArrayList<String>();

    // The empty string has a single permutation: itself
    if (word.length() == 0)
    {
        result.add(word);
        return result;
    }
    else
    {
        // Loop through all character positions
        for (int i = 0; i < word.length(); i++)
        {
            // Form a shorter word by removing the ith character
            String shorter = word.substring(0, i) + word.substring(i + 1);

            // Generate all permutations of the simpler word
            ArrayList<String> shorterPermutations = permutations(shorter);

            // Add the removed character to the front of
            // each permutation of the simpler word
            for (String s : shorterPermutations)
            {
                result.add(word.charAt(i) + s);
            }
        }
        // Return all permutations
        return result;
    }
}

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.

![Alan Turing](https://via.placeholder.com/150)
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:

```java
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 \( \text{checker.check}(r, r) \) 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 \( \text{checker.check}(r, r) \) 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.

<table>
<thead>
<tr>
<th>Instruction number</th>
<th>If tape symbol is</th>
<th>Replace with</th>
<th>Then move head</th>
<th>Then go to instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>2</td>
<td>right</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>1</td>
<td>left</td>
<td>4</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>0</td>
<td>right</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>1</td>
<td>right</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>0</td>
<td>left</td>
<td>3</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>1</td>
<td>left</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>2</td>
<td>right</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>2</td>
<td>0</td>
<td>left</td>
<td>4</td>
</tr>
</tbody>
</table>

The Turing Machine

The Turing Machine

Instruction

If tape symbol is

Replace with

Then move head

Then go to instruction

Control unit

Read/write head

Tape
13.5 Mutual Recursion

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

\[
\begin{align*}
3 + 4 \times 5 \\
(3+4) \times 5 \\
1 - (2 - (3 - (4 - 5)))
\end{align*}
\]

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 * or the / that would be required to loop back. So you exit, returning to 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) || ".ELEMENT(2).equals(next))
        {
```
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.

class Calculator {
    public int getExpressionValue() {
        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;
    }

    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;
    }

    public int getExpressionValue() {
        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;
    }

    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 termi-
nates. 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.
package Evaluator;

/**
 * A class that can compute the value of an arithmetic expression.
 */
public class Evaluator {
    private ExpressionTokenizer tokenizer;

    /**
     * Constructs an evaluator.
     * @param anExpression a string containing the expression to be evaluated
     */
    public Evaluator(String anExpression) {
        tokenizer = new ExpressionTokenizer(anExpression);
    }

    /**
     * Evaluates the expression.
     * @return the value of the expression
     */
    public int getExpressionValue() {
        int value = getTermValue();
        boolean done = false;
        while (!done) {
            String next = tokenizer.peekToken();
            if ("+".equals(next) || "-".equals(next)) {
                tokenizer.nextToken(); // Discard "+" or "-"
                int value2 = getTermValue();
                if ("+".equals(next)) { value = value + value2; }
                else { value = value - value2; }
            }
            else {
                done = true;
            }
        }
        return value;
    }

    /**
     * Evaluates the next term found in the expression.
     * @return the value of the term
     */
    public int getTermValue() {
        int value = getFactorValue();
        boolean done = false;
        while (!done) {
            String next = tokenizer.peekToken();
            if ("*".equals(next) || "/".equals(next)) {
                tokenizer.nextToken(); // Discard "*" or "/"
                int value2 = getTermValue();
                if ("*".equals(next)) { value = value * value2; }
                else { value = value / value2; }
            }
            else {
                done = true;
            }
        }
        return value;
    }

    /**
     * Evaluates the next factor found in the expression.
     * @return the value of the factor
     */
    public int getFactorValue() {
        String next = tokenizer.peekToken();
        if ("+".equals(next) || "-".equals(next)) {
            tokenizer.nextToken();
            int value2 = getFactorValue();
            if ("+".equals(next)) { value = value + value2; }
            else { value = value - value2; }
        }
        return value;
    }
}

}
```java
if ("*".equals(next)) { value = value * value2; }
else { value = value / value2; }
else {
    done = true;
}
return value;
}

/**
 Evaluates the next factor found in the expression.
 @return the value of the factor
*/
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;
}
```

```
section_5/ExpressionTokenizer.java

/**
 * This class breaks up a string describing an expression
 * into tokens: numbers, parentheses, and operators.
 */
public class ExpressionTokenizer
{
    private String input;
    private int start; // The start of the current token
    private int end; // The position after the end of the current token

    /**
     * Constructs a tokenizer.
     * @param anInput the string to tokenize
     */
    public ExpressionTokenizer(String anInput)
    {
        input = anInput;
        start = 0;
        end = 0;
        nextToken(); // Find the first token
    }

    /**
     * Peeks at the next token without consuming it.
     */
```
section_5/ExpressionCalculator.java

```java
import java.util.Scanner;

/**
 * This program calculates the value of an expression consisting of numbers, arithmetic operators, and parentheses.
 */
public class ExpressionCalculator {
    public static void main(String[] args) {
        Scanner in = new Scanner(System.in);
        System.out.print("Enter an expression: ");
        String input = in.nextLine();
        Evaluator e = new Evaluator(input);
        int value = e.getExpressionValue();
        System.out.println(input + "=\n" + value);
    }
}
```

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 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.

*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.*
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:

```java
public class PartialSolution {
    private Queen[] queens;

    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; }
    }

    public PartialSolution[] extend() { ... }
}
```

The examine method simply checks whether two queens attack each other:
The extend method takes a given solution and makes eight copies of it. Each copy gets a new queen in a different column.

```java
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:

\[
\frac{\text{row}_2 - \text{row}_1}{\text{column}_2 - \text{column}_1} = \pm 1
\]

\[
\text{row}_2 - \text{row}_1 = \pm (\text{column}_2 - \text{column}_1)
\]

\[
|\text{row}_2 - \text{row}_1| = |\text{column}_2 - \text{column}_1|
\]
Have a close look at the `solve` method in the `EightQueens` class on page 625. 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. When the queen is in column 1, there are four partial solutions with a queen in row 2. Two of them are abandoned immediately. The other two lead to partial solutions with three queens and four, all but one of which are abandoned. One partial solution is extended to four queens, but all of those are abandoned as well. 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.

```java
import java.util.Arrays;

/**
 * A partial solution to the eight queens puzzle.
 */
public class PartialSolution {
    private Queen[] queens;
    private static final int NQUEENS = 8;

    public static final int ACCEPT = 1;
    public static final int ABANDON = 2;
    public static final int CONTINUE = 3;

    /**
     * Constructs a partial solution of a given size.
     * @param size the size
     */
    public PartialSolution(int size) {
        queens = new Queen[size];
    }

    /**
     * Examines a partial solution.
     * @return one of ACCEPT, ABANDON, CONTINUE
     */
    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; }
        }
    }
}
```
```java
/**
   Yields all extensions of this partial solution.
   @return an array of partial solutions that extend this solution.
*/
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;
}

public String toString() { return Arrays.toString(queens); }
```

---

### section_6/Queen.java

```java
/**
   A queen in the eight queens problem.
*/
public class Queen
{
    private int row;
    private int column;

    /**
     Constructs a queen at a given position.
     @param r the row
     @param c the column
     */
    public Queen(int r, int c)
    {
        row = r;
        column = c;
    }

    /**
     Checks whether this queen attacks another.
     @param other the other queen
     @return true if this and the other queen are in the same
     row, column, or diagonal
     */
    public boolean attacks(Queen other)
```
section_6/EightQueens.java

```java
/**
 * This class solves the eight queens problem using backtracking.
 */
public class EightQueens {
    public static void main(String[] args) {
        solve(new PartialSolution(0));
    }

    /**
     * Prints all solutions to the problem that can be extended from
     * a given partial solution.
     * @param sol the partial solution
     */
    public static void solve(PartialSolution sol) {
        int exam = sol.examine();
        if (exam == PartialSolution.ACCEPT) {
            System.out.println(sol);
        } else if (exam == PartialSolution.CONTINUE) {
            for (PartialSolution p : sol.extend()) {
                solve(p);
            }
        }
    }
}

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

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/brjeo6examples 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.

---

**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 $n! = 1 \times 2 \times \ldots \times n$, 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 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 moves($n$) in terms of moves($n-1$). Then show that moves($n$) = $2^n - 1$.

**R13.11** Outline, but do not implement, a recursive solution for generating all subsets of the set $\{1, 2, \ldots, n\}$.

**R13.12** Exercise P13.5 shows an iterative way of generating all permutations of the sequence $(0, 1, \ldots, n-1)$. Explain why the algorithm produces the correct result.

**R13.13** Trace the expression evaluator program from Section 13.5 with inputs $3 - 4 + 5$, $3 - (4 + 5)$, $(3 - 4) \times 5$, and $3 \times 4 + 5 \times 6$. 

---

*REVIEWS 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.

*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

\[
\frac{1}{2} |x_1 y_2 + x_2 y_3 + x_3 y_1 - y_1 x_2 - y_2 x_3 - y_3 x_1|
\]
**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]], [[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

\[
\text{send} + \text{more} = \text{money}
\]

Other examples are \( \text{base} + \text{ball} = \text{games} \) and \( \text{kyoto} + \text{osaka} = \text{tokyo} \).

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 \( \text{fib} \) 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.

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 \texttt{permutations} method of Section 13.4 (which computed all permutations at once) to a \texttt{PermutationIterator} (which computes them one at a time).

```java
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":

```java
PermutationIterator iter = new PermutationIterator("eat");
while (iter.hasMorePermutations())
{
```

PROGRAMMING PROJECTS

---

630  Chapter 13  Recursion
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, …, n – 1, without using recursion.

```java
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:

```java
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 Exercises P13.9 and 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**

1. 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 \(\text{width} \times \text{width}\). The results are identical because
\[
1 + 0 + 2 + 1 + 3 + 2 + \cdots + n + n - 1 = \frac{n(n + 1)}{2} + \frac{(n - 1)n}{2} = n^2
\]
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; } // 2^0 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 \(\text{width} \times (\text{width} + 1) / 2\).
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.
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

```java
boolean isDirectory()
```
tells you which it is. The method

```java
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**  Consider various ways to simplify inputs.

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.

**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:

```java
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:

```java
/**
 * 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);
        }
    }
}
```
Finding Files

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:

```java
/**
 * 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**

```java
import java.io.File;

public class FileFinder2
{
    public static void main(String[] args)
    {
        File startingDirectory = new File("/home/myname");
        find(startingDirectory, ".java");
    }

    /**
     * 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);
            }
        }
    }
}
```
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 }
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

```java
/**
 * This program prints instructions for solving a Towers of Hanoi puzzle.
 */

class TowersOfHanoiInstructions {
    public static void main(String[] args) {
        move(5, 1, 3);
    }

    /**
     * Print instructions for moving a pile of disks from one peg to another.
     * @param disks the number of disks to move
     * @param from the peg from which to move the disks
     * @param to the peg to which to move the disks
     */
```
```java
public static void move(int disks, int from, int to)
{
    if (disks > 0)
    {
        int other = 6 - from - to;
        move(disks - 1, from, other);
        System.out.println("Move disk from peg " + from + " to " + to);
        move(disks - 1, other, to);
    }
}
```

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 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 3 to 1
Move disk from peg 2 to 1
Move disk from peg 3 to 2
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 2 to 3
Move disk from peg 1 to 1
Move disk from peg 1 to 2
Move disk from peg 1 to 3
Move disk from peg 2 to 1
Move disk from peg 1 to 3
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 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].
A TowersOfHanoi puzzle has three towers:

```java
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);
    }
    ...
}
```
Its move method first carries out the move, then prints the contents of the towers:

```java
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 - from - to.

Here is the main method:

```java
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], [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], []]
[[1], [4, 3, 2, 1], [5]]
[[1], [4, 3, 2], [5]]
[[1], [4, 3], [5, 2]]
[[1], [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]]
[[1, 2, 1], [5, 4, 3]]
[[1], [2, 1], [5, 4, 3]]
[[1], [2], [5, 4, 3]]
[[1], [], [5, 4, 3, 2]]
[[1], [], [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.

```java
import java.util.ArrayList;
import java.util.Arrays;

/**
 * This program shows a solution for a Towers of Hanoi puzzle.
 */
public class TowersOfHanoiDemo {
    public static void main(String[] args) {
        final int NDISKS = 5;
        TowersOfHanoi towers = new TowersOfHanoi(NDISKS);
        towers.move(NDISKS, 0, 2);
    }
}
```

```java
import java.util.Arrays;

/**
 * A Towers of Hanoi puzzle with three towers.
 */
public class TowersOfHanoi {
    private Tower[] towers;

    /**
     * Constructs a puzzle in which the first tower has a given number of disks.
     * @param ndisks the number of disks
     */
    public TowersOfHanoi(int ndisks) {
        towers = new Tower[3];
        towers[0] = new Tower(ndisks);
        towers[1] = new Tower(0);
        towers[2] = new Tower(0);
    }

    /**
     * Moves a pile of disks from one peg to another and displays the movement.
     * @param disks the number of disks to move
     * @param from the peg from which to move the disks
     * @param to the peg to which to move the disks
     */
    public void move(int disks, int from, int to) {
        if (disks > 0) {
            int other = 3 - from - to;
            move(disks - 1, from, other);
            move(1, from, to);
            move(disks - 1, to, other);
        }
    }
}
```
34      towers[to].add(towers[from].remove());
35      System.out.println(Arrays.toString(towers));
36      move(disks - 1, other, to);
37    }
38  }
39 }
CHAPTER 14
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

CHAPTER CONTENTS

14.1 SELECTION SORT 636
14.2 PROFILING THE SELECTION SORT ALGORITHM 639
14.3 ANALYZING THE PERFORMANCE OF THE SELECTION SORT ALGORITHM 642
ST1 Oh, Omega, and Theta 644
ST2 Insertion Sort 645
14.4 MERGE SORT 647
14.5 ANALYZING THE MERGE SORT ALGORITHM 650
ST3 The Quicksort Algorithm 652
14.6 SEARCHING 654
C&S The First Programmer 658
14.7 PROBLEM SOLVING: ESTIMATING THE RUNNING TIME OF AN ALGORITHM 659
14.8 SORTING AND SEARCHING IN THE JAVA LIBRARY 664
CE1 The compareTo Method Can Return Any Integer, Not Just –1, 0, and 1 666
ST4 The Comparator Interface 666
J81 Comparators with Lambda Expressions 667
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 \( a \):

\[
\begin{array}{c}
0 & 1 & 2 & 3 & 4 \\
11 & 9 & 17 & 5 & 12
\end{array}
\]

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 \( a[3] \):

\[
\begin{array}{c}
0 & 1 & 2 & 3 & 4 \\
5 & 9 & 17 & 11 & 12
\end{array}
\]

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.
Next we take the minimum of the remaining entries \( a[1] \ldots a[4] \). 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:

\[
\begin{array}{c|c|c|c|c|c}
5 & 9 & 17 & 11 & 12 \\
\end{array}
\]

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:

\[
\begin{array}{c|c|c|c|c|c}
5 & 9 & 11 & 17 & 12 \\
\end{array}
\]

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:

\[
\begin{array}{c|c|c|c|c|c}
5 & 9 & 11 & 12 & 17 \\
\end{array}
\]

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 7.3.4). We also add a method for swapping elements to the ArrayUtil class. (See Section 7.3.8 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.

section_1/SelectionSorter.java

```java
/**
 * The sort method of this class sorts an array, using the selection sort algorithm.
 */
public class SelectionSorter {
    /**
     * Sorts an array, using selection sort.
     * @param a the array to sort
     */
    public static void sort(int[] a) {
        for (int i = 0; i < a.length - 1; i++) {
            int minPos = minimumPosition(a, i);
            ArrayUtil.swap(a, minPos, i);
        }
    }
}
```
/**
 * Finds the smallest element in a tail range of the array.
 * @param a the array to sort
 * @param from the first position in a to compare
 * @return the position of the smallest element in the
 * range a[from] . . . a[a.length - 1]
 */
private static int minimumPosition(int[] a, int from)
{
    int minPos = from;
    for (int i = from + 1; i < a.length; i++)
    {
        if (a[i] < a[minPos]) { minPos = i; }
    }
    return minPos;
}

section_1/SelectionSortDemo.java
import java.util.Arrays;
/**
 * This program demonstrates the selection sort algorithm by
 * sorting an array that is filled with random numbers.
 */
public class SelectionSortDemo
{
    public static void main(String[] args)
    {
        int[] a = ArrayUtil.randomIntArray(20, 100);
        System.out.println(Arrays.toString(a));
        SelectionSorter.sort(a);
        System.out.println(Arrays.toString(a));
    }
}

section_1/ArrayUtil.java
import java.util.Random;
/**
 * This class contains utility methods for array manipulation.
 */
public class ArrayUtil
{
    private static Random generator = new Random();

    /**
     * Creates an array filled with random values.
     * @param length the length of the array
     * @param n the number of possible random values
     * @return an array filled with length numbers between
     * 0 and n - 1
     */
    public static int[] randomIntArray(int length, int n)
    {
    }
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

```java
int[] a = new int[length];
for (int i = 0; i < a.length; i++)
{
    a[i] = generator.nextInt(n);
}
return a;
```

```
/**
 * Swaps two entries of an array.
 * @param a the array
 * @param i the first position to swap
 * @param j the second position to swap
 */
public static void swap(int[] a, int i, int j)
{
    int temp = a[i];
    a[i] = a[j];
    a[j] = temp;
}
```

**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.
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:

```java
public class StopWatch {
    private long elapsedTime;
    private long startTime;
    private boolean isRunning;

    /**
     * Constructs a stopwatch that is in the stopped state and has no time accumulated.
     */
    public StopWatch() {
        reset();
    }

    /**
     * Starts the stopwatch. Time starts accumulating now.
     */
    public void start() {
        if (isRunning) { return; }
        isRunning = true;
        startTime = System.currentTimeMillis();
    }

    /**
     * Stops the stopwatch. Time stops accumulating and is added to the elapsed time.
     */
    public void stop() {
        if (!isRunning) { return; }
        isRunning = false;
        long endTime = System.currentTimeMillis();
        elapsedTime = elapsedTime + endTime - startTime;
    }

    /**
     * Returns the total elapsed time.
     * @return the total elapsed time
     */
    public long getElapsedTime() {
        if (isRunning) {
            return;
        }
    }
```
Here is how to use the stopwatch to measure the sorting algorithm’s performance:

section_2/SelectionSortTimer.java

```java
import java.util.Scanner;

/**
 * This program measures how long it takes to sort an
 * array of a user-specified size with the selection
 * sort algorithm.
 */
public class SelectionSortTimer
{
    public static void main(String[] args)
    {
        Scanner in = new Scanner(System.in);
        System.out.print("Enter array size: ");
        int n = in.nextInt();

        // Construct random array
        int[] a = ArrayUtil.randomIntArray(n, 100);

        // Use stopwatch to time selection sort
        StopWatch timer = new StopWatch();
        timer.start();
        SelectionSorter.sort(a);
        timer.stop();
        System.out.println("Elapsed time: "
                         + timer.getElapsedTime() + " milliseconds");
    }
}
```

Program Run

Enter array size: 50000
Elapsed time: 13321 milliseconds
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?

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 obser-
vation. 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

\[
n + 2 + (n-1) + 2 + \cdots + 2 + 2 = (n + (n-1) + \cdots + 2) + (n-1) \cdot 2
\]

\[
= (2 + \cdots + (n-1) + n) + (n-1) \cdot 2
\]

\[
= \frac{n(n+1)}{2} - 1 + (n-1) \cdot 2
\]

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 exam-
ple, 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:

\[
\left(\frac{\frac{1}{2} \cdot 2000^2}{\frac{1}{2} \cdot 1000^2}\right) = 4
\]

The factor \( \frac{1}{2} \) cancels out in comparisons of this kind. We will simply say, “The num-er of visits is of order \( n^2 \).” That way, we can easily see that the number of compar-
isons increases fourfold when the size of the array doubles: \( 2n \cdot 2n = 4n^2 \).

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 con-
stant 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
(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.

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 7.3.6 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 7.3.6. 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.

### 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 $O(n^2)$ or even $O(n^{10})$. 
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 \( \Omega \) 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 \( \Theta \) symbol is the capital Greek letter theta.)

The \( \Theta \) 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.

### Table 1 Common Big-Oh Growth Rates

<table>
<thead>
<tr>
<th>Big-Oh Expression</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>( O(1) )</td>
<td>Constant</td>
</tr>
<tr>
<td>( O(\log(n)) )</td>
<td>Logarithmic</td>
</tr>
<tr>
<td>( O(n) )</td>
<td>Linear</td>
</tr>
<tr>
<td>( O(n \log(n)) )</td>
<td>Log-linear</td>
</tr>
<tr>
<td>( O(n^2) )</td>
<td>Quadratic</td>
</tr>
<tr>
<td>( O(n^3) )</td>
<td>Cubic</td>
</tr>
<tr>
<td>( O(2^n) )</td>
<td>Exponential</td>
</tr>
<tr>
<td>( O(n!) )</td>
<td>Factorial</td>
</tr>
</tbody>
</table>

**Insertion Sort**

Insertion sort is another simple sorting algorithm. In this algorithm, we assume that the initial sequence

\[ a[0] a[1] \ldots a[k] \]

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.

\[
\begin{array}{cccc}
9 & 11 & 16 & 5 \\
\end{array}
\]

We repeat the process, inserting a[3] or 5 at the very beginning of the initial sequence.

\[
\begin{array}{cccc}
5 & 9 & 11 & 16 \\
\end{array}
\]

Finally, a[4] or 7 is inserted in its correct position, and the sorting is completed.

The following class implements the insertion sort algorithm:

```java
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

\[
2 + 3 + \cdots + n = \frac{n(n+1)}{2} - 1
\]

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.
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
  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:

```java
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);
}
```
The merge method is tedious but quite straightforward. You will find it in the code that follows.

**section_4/MergeSorter.java**

```java
/**
   * The sort method of this class sorts an array, using the merge sort algorithm.
   */
public class MergeSorter {

    /**
     * Sorts an array, using merge sort.
     * @param a the array to sort
     */
    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
        for (int i = 0; i < first.length; i++) {
            first[i] = a[i];
        }
        for (int i = 0; i < second.length; i++) {
            second[i] = a[first.length + i];
        }
        sort(first);
        sort(second);
        merge(first, second, a);
    }

    /**
     * Merges two sorted arrays into an array.
     * @param first the first sorted array
     * @param second the second sorted array
     * @param a the array into which to merge first and second
     */
    private static void merge(int[] first, int[] second, int[] a) {
        int iFirst = 0; // Next element to consider in the first array
        int iSecond = 0; // Next element to consider in the second array
        int j = 0; // Next open position in a
        // As long as neither iFirst nor iSecond past the end, move
        // the smaller element into a
        while (iFirst < first.length && iSecond < second.length) {
            if (first[iFirst] < second[iSecond]) {
                a[j] = first[iFirst];
                iFirst++;
            } else {
                a[j] = second[iSecond];
                iSecond++;
            }
            j++;
        }
    }
}
```
// Note that only one of the two loops below copies entries
// Copy any remaining entries of the first array
while (iFirst < first.length)
    { 
        a[j] = first[iFirst]; 
        iFirst++; j++; 
    }
// Copy any remaining entries of the second half
while (iSecond < second.length)
    { 
        a[j] = second[iSecond]; 
        iSecond++; j++; 
    }

s e c t i o n _ 4 / M e r g e S o r t D e m o . j a v a

import java.util.Arrays;
/**
 * This program demonstrates the merge sort algorithm by sorting an array that is filled with random numbers.
 */
public class MergeSortDemo
{
    public static void main(String[] args)
    {
        int[] a = ArrayUtil.randomIntArray(20, 100);
        System.out.println(Arrays.toString(a));
        MergeSorter.sort(a);
        System.out.println(Arrays.toString(a));
    }
}

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]

S E L F  C H E C K

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.

P r a c t i c e  I t  Now you can try these exercises at the end of the chapter: R14.13, E14.4, E14.14.
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 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:

\[
T\left(\frac{n}{2}\right) = 2T\left(\frac{n}{4}\right) + \frac{5n}{2}
\]

Therefore

\[
T(n) = 2 \times 2T\left(\frac{n}{4}\right) + 5n + 5n
\]

Let us do that again:

\[
T\left(\frac{n}{4}\right) = 2T\left(\frac{n}{8}\right) + \frac{5n}{4}
\]

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 \( k = m \),

\[
T(n) = 2^m T\left(\frac{n}{2^m}\right) + 5nm
\]

\[
= nT(1) + 5nm
\]

\[
= n + 5n \log_2(n)
\]

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,

\[
\log_2(x) = \frac{\log_{10}(x)}{\log_{10}(2)} = \log_{10}(x) \times 3.32193
\]

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 \( 0.75 \times 150 \) 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.
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.

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.

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 $a[from] \ldots a[to]$ of the array $a$, first rearrange the elements in the range so that no element in the range $a[from] \ldots a[p]$ is larger than any element in the range $a[p + 1] \ldots a[to]$. 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:

```java
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 $a[from] \ldots a[i]$, consisting of values at most as large as the pivot and $a[j] \ldots a[to]$, consisting of values at least as large as the pivot. The region $a[i + 1] \ldots a[j - 1]$ 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 $j = to + 1$. 

© Alex Slobodkin/iStockphoto.
Analyzing the Merge Sort Algorithm

14.5

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:

```java
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.

In quicksort, one partitions the elements into two groups, holding the smaller and larger elements. Then one sorts each group.
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 class for executing linear searches in an array.
*/
public class LinearSearcher {

   /**
       Finds a value in an array, using the linear search algorithm.
       @param a the array to search
       @param value the value to find
       @return the index at which the value occurs, or -1 if it does not occur in the array
   */
   public static int search(int[] a, int value) {
      for (int i = 0; i < a.length; i++)
         if (a[i] == value) { return i; }
      return -1;
   }
}
```
**14.6 Searching**

### section_6_1/LinearSearchDemo.java

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

/**
 * This program demonstrates the linear search algorithm.
 */
public class LinearSearchDemo {
    public static void main(String[] args) {
        int[] a = ArrayUtil.randomIntArray(20, 100);
        System.out.println(Arrays.toString(a));
        Scanner in = new Scanner(System.in);

        boolean done = false;
        while (!done) {
            System.out.print("Enter number to search for, -1 to quit: ");
            int n = in.nextInt();
            if (n == -1) {
                done = true;
            } else {
                int pos = LinearSearcher.search(a, n);
                System.out.println("Found in position " + pos);
            }
        }
    }
}
```

**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:

```
[01] [11] [22] [33] [44] [55] [66] [77] [88] [99]
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:

\[
\begin{array}{cccccccccccc}
0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 12 & 17 & 20 & 24 & 32 \\
1 & 4 & 5 & 8 & 9 & 12 & 17 & 20 & 24 & 32
\end{array}
\]

The middle element of this sequence is 20; hence, the value must be located in the sequence:

\[
\begin{array}{cccccccccccc}
0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 12 & 17 & 20 & 24 & 32 \\
1 & 4 & 5 & 8 & 9 & 12 & 17 & 20 & 24 & 32
\end{array}
\]

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:

\[
\begin{array}{cccccccccccc}
0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 12 & 17 & 20 & 24 & 32 \\
1 & 4 & 5 & 8 & 9 & 12 & 17 & 20 & 24 & 32
\end{array}
\]

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.

```java
section_6_2/BinarySearcher.java

/**
 * A class for executing binary searches in an array.
 */
public class BinarySearcher {

    /**
     * Finds a value in a range of a sorted array, using the binary search algorithm.
     * @param a the array in which to search
     * @param low the low index of the range
     * @param high the high index of the range
     * @param value the value to find
     * @return the index at which the value occurs, or -1 if it does not occur in the array
     */
    public int search(int[] a, int low, int high, int value) {
        if (low <= high) {
            int mid = (low + high) / 2;
            if (a[mid] == value) {
                return mid;
            } else if (a[mid] < value) {
                // Search in the second half
            } else { // a[mid] > value
                // Search in the first half
            }
        }
    }
}
```
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 \( \log_2(100) = 6.64386 \), 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.
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.

Computing & Society 14.1  The First Programmer

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 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:

<table>
<thead>
<tr>
<th>x</th>
<th>f(x)</th>
<th>( f(x+1) - f(x) )</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>7</td>
</tr>
<tr>
<td>2</td>
<td>8</td>
<td>19</td>
</tr>
<tr>
<td>3</td>
<td>27</td>
<td>37</td>
</tr>
<tr>
<td>4</td>
<td>64</td>
<td>61</td>
</tr>
<tr>
<td>5</td>
<td>125</td>
<td>91</td>
</tr>
<tr>
<td>6</td>
<td>216</td>
<td></td>
</tr>
</tbody>
</table>

Repeat the process, taking the difference of successive values in the second column, and then repeat once again:

<table>
<thead>
<tr>
<th>x</th>
<th>f(x)</th>
<th>( f(x+2) - f(x+1) )</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>7</td>
</tr>
<tr>
<td>2</td>
<td>8</td>
<td>12</td>
</tr>
<tr>
<td>3</td>
<td>19</td>
<td>13</td>
</tr>
<tr>
<td>4</td>
<td>27</td>
<td>11</td>
</tr>
<tr>
<td>5</td>
<td>64</td>
<td>41</td>
</tr>
<tr>
<td>6</td>
<td>125</td>
<td>33</td>
</tr>
<tr>
<td>7</td>
<td>216</td>
<td></td>
</tr>
</tbody>
</table>

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.
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:

```java
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.

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:

```java
boolean found = false;
for (int i = 0; !found && i < a.length; i++)
    if (a[i] == value) { found = true; }
```
Then the loop can stop in the middle:

```
1 2 3
```

Found the value

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 3 1 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
}
```

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.
14.7 Problem Solving: Estimating the Running Time of an Algorithm

We have just seen that the first phase is \( O(n^2) \). Computing the maximum is \( O(n) \)—look at the algorithm in Section 7.3.3 and note that each step involves a fixed amount of work. Finally, we just saw that finding a value is \( O(n) \).

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 \( a[0] \ldots a[i - 1] \)?

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

![Image of light bulb pattern]

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 \( a[0] \ldots a[i - 1] \). If \( a[i] \) never occurred before, we get an accurate
count by just looking at \( a[i] \) \( \ldots \) \( a[n - 1] \). 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

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, and you will see it again in Chapter 17.

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
```

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,

```java
int count = 0;
for (int i = 0; i < a.length; i++)
{
    count++;
    if (i == a.length - 1 || a[i] != a[i + 1])
    {
```
counts[i] = count;
count = 0;
}
}

That’s a constant amount of work per iteration, even though it visits two elements:

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?
   ```java
   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?
   ```java
   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?
   ```java
   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?

```java
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 contains static `sort` methods to sort arrays of integers and floating-point numbers. For example, you can sort an array of integers simply as

```java
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:

```java
ArrayList<String> names = ...;
Collections.sort(names);
```

## 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,

```java
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

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 10.3. 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 10.1) 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.
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 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

```java
if (a.compareTo(b) == -1) // Error!
```

is generally wrong. Instead, you should test

```java
if (a.compareTo(b) < 0) // OK
```

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

```java
if (a.compareTo(b) == -1) // Error!
```

is generally wrong. Instead, you should test

```java
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:

```java
char c1 = charAt(i);
char c2 = other.charAt(i);
```

If the characters are different, then the method simply returns their difference:

```java
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 10.1).

---

**Special Topic 14.4**

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:

```java
public interface Comparator<Country>
{
    int compare(Country a, Country b);
}
```
The call

```java
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:

```java
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

```java
Arrays.sort(countries, new CountryComparator());
```

### 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

```java
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

```java
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

```java
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/bjeo6examples and download Worked Example 14.1.
**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
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 \(\leq\), \(a.length\) 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

\[
\frac{T(2,000)}{T(1,000)}
\]
\[
\frac{T(4,000)}{T(1,000)}
\]
\[
\frac{T(10,000)}{T(1,000)}
\]
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.

<table>
<thead>
<tr>
<th></th>
<th>( O(n) )</th>
<th>( O(n^2) )</th>
<th>( O(n^3) )</th>
<th>( O(n \log(n)) )</th>
<th>( O(2^n) )</th>
</tr>
</thead>
<tbody>
<tr>
<td>1,000</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>2,000</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3,000</td>
<td></td>
<td>45</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10,000</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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.

\[ O(n) \quad O(\log(n)) \quad O(2^n) \quad O(n\sqrt{n}) \]
\[ O(n^3) \quad O(n^2 \log(n)) \quad O(\sqrt{n}) \quad O(n\log(n)) \]
\[ O(n^n) \quad O(n \log(n)) \]

**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.

```java
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 R7.23). 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:

a. Sort the array, then find the longest run.

b. 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.

c. 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 \(a[0] \ldots a[i - 1]\). If not, count how often it occurs in \(a[i + 1] \ldots a[n - 1]\). 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:

a. 4 7 11 4 9 5 11 7 3 5

b. –7 6 8 7 5 9 0 11 10 5 8

R14.13 Trace a walkthrough of merge sort with these sets:

a. 5 11 7 3 5 4 7 11 4 9

b. 9 0 11 10 5 8 –7 6 8 7 5

R14.14 Trace a walkthrough of:

a. Linear search for 7 in –7 1 3 3 4 7 11 13

b. Binary search for 8 in –7 2 2 3 4 7 8 11 13

c. Binary search for 8 in –7 1 2 3 5 7 10 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.
• 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 10.
• 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:

```java
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.*

**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 ≤ m` for half the elements, and `m ≤ 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 \( a[i \cdot (n - 1) / 8] \), 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

\[
< \quad \geq
\]

(where ≤ denotes the elements that are ≤ the pivot), it is better to partition as

\[
< \quad = \quad >
\]

However, that is tedious to achieve directly. They recommend to partition as

\[
= \quad < \quad >
\]

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

$$2n - 1 + 2(n - 1) - 1 + \cdots + 2 \cdot 3 - 1 + 2 \cdot 2 - 1 = 2(n + (n - 1) + \cdots + 3 + 2 + 1) - (n - 1) = n(n + 1) - 2 - n + 1 = n^2 - 3$$

(because $1 + 2 + 3 + \cdots + (n - 1) + n = n(n + 1)/2$)

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, $iFirst \geq iFirst.length$ or $iSecond \geq iSecond.length$ (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. \[ \frac{2n \log(2n)}{n \log(n)} = 2 \frac{(1 + \log(2))}{\log(n)}. \]

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. It is an $O(n)$ algorithm.

22. It is an $O(n^2)$ algorithm — the number of visits follows a triangle pattern.

23. Sort the array, then make a linear scan to check for adjacent duplicates.

24. It is an $O(n^2)$ algorithm — the outer and inner loops each have $n$ iterations.

25. 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.

26. The Rectangle class does not implement the Comparable interface.

27. The BankAccount class would need to implement the Comparable interface. Its compareTo method must compare the bank balances.

28. Then you know where to insert it so that the array stays sorted, and you can keep using binary search.

29. Otherwise, you would not know whether a value is present when the method returns 0.
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

\[
\begin{array}{ccc}
65 & 46 & 14 \\
38 & 2 & 96 & 39 \\
14 & 33 & 13 & 4 \\
24 & 99 & 89 & 77 \\
73 & 87 & 36 & 81
\end{array}
\]

Now we sort each column:

\[
\begin{array}{ccc}
14 & 2 & 13 & 5 \\
24 & 33 & 14 & 39 \\
38 & 46 & 36 & 52 \\
65 & 87 & 89 & 77 \\
73 & 99 & 96 & 81
\end{array}
\]

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

\[
\begin{align*}
c_1 &= 1 \\
c_2 &= 4 \\
c_3 &= 13 \\
c_4 &= 40 \\
&\vdots \\
c_{i+1} &= 3c_i + 1
\end{align*}
\]

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.
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  36
 52  39  4  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:

```java
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:

```java
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:

```java
while (j >= c && a[j - c] > next)
{
    a[j] = a[j - c];
    j = j - c;
}
```

Putting everything together, we get the following method:

```java
/**
 * 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;
        }
    }
}
```
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.

```java
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:

```java
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.

```java
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.

```
Enter array size: 1000000
Elapsed time with Shell sort: 205 milliseconds
```
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
\[ c = 3 \times c + 1; \]
with
\[ c = 2 \times c; \]
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.
CHAPTER 15
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 678
15.2  LINKED LISTS 681
   C&S  Standardization 686
15.3  SETS 687
   PT1  Use Interface References to Manipulate Data Structures 691
15.4  MAPS 692
   J81  Updating Map Entries 694
   HT1  Choosing a Collection 694
   WE1  Word Frequency 694
   ST1  Hash Functions 696

15.5  STACKS, QUEUES, AND PRIORITY QUEUES 698
15.6  STACK AND QUEUE APPLICATIONS 701
   WE2  Simulating a Queue of Waiting Customers
   ST2  Reverse Polish Notation 709
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 7 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 680 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
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.
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 7.5). For example, when constructing an array list of strings, we will use

```java
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

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>coll = new ArrayList&lt;&gt;();</code></td>
<td>The ArrayList class implements the Collection interface.</td>
</tr>
<tr>
<td><code>coll = new TreeSet&lt;&gt;();</code></td>
<td>The TreeSet class (Section 15.3) also implements the Collection interface.</td>
</tr>
<tr>
<td><code>int n = coll.size();</code></td>
<td>Gets the size of the collection. n is now 0.</td>
</tr>
<tr>
<td><code>coll.add(&quot;Harry&quot;);</code></td>
<td>Adds elements to the collection.</td>
</tr>
<tr>
<td><code>coll.add(&quot;Sally&quot;);</code></td>
<td>Returns a string with all elements in the collection. s is now [Harry, Sally].</td>
</tr>
<tr>
<td><code>String s = coll.toString();</code></td>
<td>Invokes the toString method and prints [Harry, Sally].</td>
</tr>
<tr>
<td><code>coll.remove(&quot;Harry&quot;);</code></td>
<td>Removes an element from the collection, returning false if the element is not present. b is false.</td>
</tr>
<tr>
<td><code>boolean b = coll.remove(&quot;Tom&quot;);</code></td>
<td>Checks whether this collection contains a given element. b is now true.</td>
</tr>
<tr>
<td>`for (String s : coll)</td>
<td>You can use the “for each” loop with any collection. This loop prints the elements on separate lines.</td>
</tr>
<tr>
<td>{ System.out.println(s); }</td>
<td></td>
</tr>
<tr>
<td><code>Iterator&lt;String&gt; iter = coll.iterator();</code></td>
<td>You use an iterator for visiting the elements in the collection (see Section 15.2.3).</td>
</tr>
</tbody>
</table>

---

**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.
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 in the sequence (see Figure 6).
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.

![Figure 8](image)

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>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>LinkedList&lt;String&gt; list = new LinkedList&lt;&gt;();</code></td>
<td>An empty list.</td>
</tr>
<tr>
<td><code>list.addLast(&quot;Harry&quot;);</code></td>
<td>Adds an element to the end of the list. Same as <code>add</code>.</td>
</tr>
<tr>
<td><code>list.addFirst(&quot;Sally&quot;);</code></td>
<td>Adds an element to the beginning of the list. <code>list</code> is now <code>[Sally, Harry]</code>.</td>
</tr>
<tr>
<td><code>list.getFirst();</code></td>
<td>Gets the element stored at the beginning of the list; here &quot;Sally&quot;.</td>
</tr>
<tr>
<td><code>list.getLast();</code></td>
<td>Gets the element stored at the end of the list; here &quot;Harry&quot;.</td>
</tr>
<tr>
<td><code>String removed = list.removeFirst();</code></td>
<td>Removes the first element of the list and returns it. <code>removed</code> is &quot;Sally&quot; and <code>list</code> is <code>[Harry]</code>. Use <code>removeLast</code> to remove the last element.</td>
</tr>
<tr>
<td><code>ListIterator&lt;String&gt; iter = list.listIterator();</code></td>
<td>Provides an iterator for visiting all list elements (see Table 3 on page 684).</td>
</tr>
</tbody>
</table>
15.2.3 List Iterators

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:

```java
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:

```java
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.

```java
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:

```java
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:

```java
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

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>String s = iter.next();</td>
<td>Assume that iter points to the beginning of the list [Sally] before calling next. After the call, s is &quot;Sally&quot; and the iterator points to the end.</td>
</tr>
<tr>
<td>iter.previous();</td>
<td>The set method updates the last element returned by next or previous. The list is now [Juliet].</td>
</tr>
<tr>
<td>iter.set(&quot;Juliet&quot;);</td>
<td>Returns false because the iterator is at the end of the collection.</td>
</tr>
<tr>
<td>iter.hasNext();</td>
<td>hasPrevious returns true because the iterator is not at the beginning of the list. previous and hasPrevious are ListIterator methods.</td>
</tr>
<tr>
<td>if (iter.hasPrevious())</td>
<td>Adds an element before the iterator position (ListIterator only). The list is now [Diana, Juliet].</td>
</tr>
<tr>
<td>s = iter.previous();</td>
<td>remove removes the last element returned by next or previous. The list is now [Diana].</td>
</tr>
<tr>
<td>iter.add(&quot;Diana&quot;);</td>
<td></td>
</tr>
<tr>
<td>iter.next();</td>
<td></td>
</tr>
<tr>
<td>iter.remove();</td>
<td></td>
</tr>
</tbody>
</table>

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 after the iterator, then moves the iterator position past the new element.

`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:

```java
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.
section_2/ListDemo.java

```java
import java.util.LinkedList;
import java.util.ListIterator;
/**
   * This program demonstrates the LinkedList class.
   */
public class ListDemo
{
    public static void main(String[] args)
    {
        LinkedList<String> staff = new LinkedList<>();
        staff.addLast("Diana");
        staff.addLast("Harry");
        staff.addLast("Romeo");
        staff.addLast("Tom");
        // | in the comments indicates the iterator position
        ListIterator<String> iterator = staff.listIterator(); // |DHRT
        iterator.next(); // D|HRT
        iterator.next(); // DH|RT
        // Add more elements after second element
        iterator.add("Juliet"); // DHJ|RT
        iterator.add("Nina"); // DHJN|RT
        iterator.next(); // DHJNR|T
        // Remove last traversed element
        iterator.remove(); // DHJN|T
        // Print all elements
        System.out.println(staff);
        System.out.println("Expected: [Diana, Harry, Juliet, Nina, Tom]"+");
    }
}
```

**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:

```java
ListIterator<String> iter = letters.iterator();
iter.next();
iter.next();
iter.remove();
iter.next();
iter.add("E");
```
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.

### Computing & Society 15.1 Standardization

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.
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 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 hashtable. 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 hashtable 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 hashtable and equals methods of the Object class.
A tree set keeps its elements in sorted order.

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 10.3. (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,

```java
Set<String> names = new HashSet<>();
```
or

```java
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:

```java
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:

```java
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.

```java
Iterator<String> iter = names.iterator();
while (iter.hasNext())
{
```
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. 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

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Set&lt;String&gt; names;</td>
<td>Use the interface type for variable declarations.</td>
</tr>
<tr>
<td>names = new HashSet&lt;&gt;();</td>
<td>Use a TreeSet if you need to visit the elements in sorted order.</td>
</tr>
<tr>
<td>names.add(&quot;Romeo&quot;);</td>
<td>Now names.size() is 1.</td>
</tr>
<tr>
<td>names.add(&quot;Fred&quot;);</td>
<td>Now names.size() is 2.</td>
</tr>
<tr>
<td>names.add(&quot;Romeo&quot;);</td>
<td>names.size() is still 2. You can’t add duplicates.</td>
</tr>
<tr>
<td>if (names.contains(&quot;Fred&quot;))</td>
<td>The contains method checks whether a value is contained in the set. In this case, the method returns true.</td>
</tr>
<tr>
<td>System.out.println(names);</td>
<td>Prints the set in the format [Fred, Romeo]. The elements need not be shown in the order in which they were inserted.</td>
</tr>
<tr>
<td>for (String name : names)</td>
<td>Use this loop to visit all elements of a set.</td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
<tr>
<td>names.remove(&quot;Romeo&quot;);</td>
<td>Now names.size() is 1.</td>
</tr>
<tr>
<td>names.remove(&quot;Juliet&quot;);</td>
<td>It is not an error to remove an element that is not present. The method call has no effect.</td>
</tr>
</tbody>
</table>
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.)

section_3/SpellCheck.java

```java
import java.util.HashSet;
import java.util.Scanner;
import java.util.Set;
import java.io.File;
import java.io.FileNotFoundException;

/**
 * This program checks which words in a file are not present in a dictionary.
 */
public class SpellCheck {

    public static void main(String[] args)
            throws FileNotFoundException {
        // Read the dictionary and the document
        Set<String> dictionaryWords = readWords("words");
        Set<String> documentWords = readWords("alice30.txt");

        // Print all words that are in the document but not the dictionary
        for (String word : documentWords)
        {
            if (!dictionaryWords.contains(word))
            {
                System.out.println(word);
            }
        }
    }

    /**
     * Reads all words from a file.
     * @param filename the name of the file
     * @return a set with all lowercased words in the file. Here, a word is a sequence of upper- and lowercase letters.
     */
    public static Set<String> readWords(String filename)
            throws FileNotFoundException {
        Set<String> words = new HashSet<>();
        Scanner in = new Scanner(new File(filename));
        // Use any characters other than a-z or A-Z as delimiters
        in.useDelimiter("[\da-zA-Z]+"樨;
        while (in.hasNext())
        {
            words.add(in.next().toLowerCase());
        }
        return words;
    }
}
```
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"?

   ```java
   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.

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:

```java
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:

```java
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.
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:

```java
Map<String, Color> favoriteColors = new HashMap<>();
```

Use the put method to add an association:

```java
favoriteColors.put("Juliet", Color.RED);
```

You can change the value of an existing association, simply by calling put again:

```java
favoriteColors.put("Juliet", Color.BLUE);
```

The get method returns the value associated with a key.

```java
Color julietsFavoriteColor = 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:

```java
favoriteColors.remove("Juliet");
```

### Table 5  Working with Maps

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>Map&lt;String, Integer&gt; scores;</code></td>
<td>Keys are strings, values are Integer wrappers. Use the interface type for variable declarations.</td>
</tr>
<tr>
<td><code>scores = new TreeMap&lt;&gt;();</code></td>
<td>Use a HashMap if you don’t need to visit the keys in sorted order.</td>
</tr>
<tr>
<td><code>scores.put(&quot;Harry&quot;, 90); scores.put(&quot;Sally&quot;, 95);</code></td>
<td>Adds keys and values to the map.</td>
</tr>
<tr>
<td><code>scores.put(&quot;Sally&quot;, 100);</code></td>
<td>Modifies the value of an existing key.</td>
</tr>
<tr>
<td><code>int n = scores.get(&quot;Sally&quot;); Integer n2 = scores.get(&quot;Diana&quot;);</code></td>
<td>Gets the value associated with a key, or null if the key is not present. n is 100, n2 is null.</td>
</tr>
<tr>
<td><code>System.out.println(scores);</code></td>
<td>Prints scores.toString(), a string of the form {Harry=90, Sally=100}</td>
</tr>
<tr>
<td><code>for (String key : scores.keySet()) {</code></td>
<td>Iterates through all map keys and values.</td>
</tr>
<tr>
<td><code>    Integer value = scores.get(key);</code></td>
<td></td>
</tr>
<tr>
<td><code>    ...</code></td>
<td></td>
</tr>
<tr>
<td><code>}</code></td>
<td></td>
</tr>
<tr>
<td><code>scores.remove(&quot;Sally&quot;);</code></td>
<td>Removes the key and value.</td>
</tr>
</tbody>
</table>
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`:

```java
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:

```java
import java.awt.Color;
import java.util.HashMap;
import java.util.Map;
import java.util.Set;

/**
 * This program demonstrates a map that maps names to colors.
 */
public class MapDemo {
    public static void main(String[] args) {
        Map<String, Color> favoriteColors = new HashMap<>();
        favoriteColors.put("Juliet", Color.BLUE);
        favoriteColors.put("Romeo", Color.GREEN);
        favoriteColors.put("Adam", Color.RED);
        favoriteColors.put("Eve", Color.BLUE);

        // Print all keys and values in the map
        Set<String> keySet = favoriteColors.keySet();
        for (String key : keySet) {
            Color value = favoriteColors.get(key);
            System.out.println(key + " : " + value);
        }
    }
}
```

**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.

---

### 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:

```java
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 10.4). For example,

```java
frequencies.merge(word, 1, (oldValue, value) -> oldValue + value);
```

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.

**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 10.3), 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/bjeo6examples and download Worked Example 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
```

<table>
<thead>
<tr>
<th>String</th>
<th>Hash Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;eat&quot;</td>
<td>100184</td>
</tr>
<tr>
<td>&quot;tea&quot;</td>
<td>114704</td>
</tr>
<tr>
<td>&quot;Juliet&quot;</td>
<td>-2065036585</td>
</tr>
<tr>
<td>&quot;Ugh&quot;</td>
<td>84982</td>
</tr>
<tr>
<td>&quot;VII&quot;</td>
<td>84982</td>
</tr>
</tbody>
</table>

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.
The hash code of "tea" is quite different, namely

\[ 31 \times (31 \times 't' + 'e') + 'a' = 114704 \]

(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 10.1.

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.

```java
public class Country {
    
    public int hashCode() {
        int h1 = name.hashCode();
        int h2 = new Double(area).hashCode();
        
        final int HASH_MULTIPLIER = 31;
        int h = HASH_MULTIPLIER * h1 + h2;
        return h;
    }
}
```

Then combine the two hash codes:

```java
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.

```java
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.

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.
Chapter 15  The Java Collections Framework

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 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.

```java
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).

<table>
<thead>
<tr>
<th>Table 7  Working with Stacks</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stack&lt;Integer&gt; s = new Stack&lt;&gt;();</td>
</tr>
<tr>
<td>s.push(1); s.push(2); s.push(3);</td>
</tr>
<tr>
<td>int top = s.pop();</td>
</tr>
<tr>
<td>head = s.peek();</td>
</tr>
</tbody>
</table>
15.5.2 Queues

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:

```java
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.

<table>
<thead>
<tr>
<th>Queue&lt;Integer&gt; q = new LinkedList&lt;&gt;();</th>
<th>The LinkedList class implements the Queue interface.</th>
</tr>
</thead>
<tbody>
<tr>
<td>q.add(1); q.add(2); q.add(3);</td>
<td>Adds to the tail of the queue; q is now [1, 2, 3].</td>
</tr>
<tr>
<td>int head = q.remove();</td>
<td>Removes the head of the queue; head is set to 1 and q is [2, 3].</td>
</tr>
<tr>
<td>head = q.peek();</td>
<td>Gets the head of the queue without removing it; head is set to 2.</td>
</tr>
</tbody>
</table>

15.5.3 Priority Queues

A priority queue collects elements, each of which has a priority. A typical example of a priority queue is a collection of work requests, some of which may be more urgent than others. Unlike a regular queue, the priority queue does not maintain a first-in, first-out discipline. Instead, elements are retrieved according to their priority. In other words, new items can be inserted in any order. But whenever an item is removed, it is the item with the most urgent priority.
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.

```java
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 10.3 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

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>PriorityQueue&lt;Integer&gt; q = new PriorityQueue&lt;&gt;();</td>
<td>This priority queue holds Integer objects. In practice, you would use objects that describe tasks.</td>
</tr>
<tr>
<td>q.add(3); q.add(1); q.add(2);</td>
<td>Adds values to the priority queue.</td>
</tr>
<tr>
<td>int first = q.remove();</td>
<td>Each call to remove removes the most urgent item: first is set to 1, second to 2.</td>
</tr>
<tr>
<td>int second = q.remove();</td>
<td>Gets the smallest value in the priority queue without removing it.</td>
</tr>
<tr>
<td>int next = q.peek();</td>
<td></td>
</tr>
</tbody>
</table>

21. Why would you want to declare a variable as

```java
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?

```java
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?

```java
PriorityQueue<String> q = new PriorityQueue<>();
q.add("3 - Shampoo carpets");
q.add("1 - Fix broken sink");
q.add("2 - Order cleaning supplies");
```
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

In Common Error 4.2, you saw a simple trick for detecting unbalanced parentheses in an expression such as

\[-(b \times b - (4 \times a \times c) ) / (2 \times a)\]

Increment a counter when you see an opening parenthesis, and decrement it when you see a closing parenthesis. 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

\[-\{ [b \times b - (4 \times a \times c)] / (2 \times a) \}\]

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.

Here is a walkthrough of the sample expression:

<table>
<thead>
<tr>
<th>Stack</th>
<th>Unread expression</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>Empty</td>
<td>[-{ [b \times b - (4 \times a \times c)] / (2 \times a) }]</td>
<td></td>
</tr>
<tr>
<td>{</td>
<td>{ [b \times b - (4 \times a \times c)] / (2 \times a) }</td>
<td></td>
</tr>
<tr>
<td>[ [</td>
<td>{ [b \times b - (4 \times a \times c)] / (2 \times a) }</td>
<td></td>
</tr>
<tr>
<td>[ [ (</td>
<td>{ } / (2 \times a) }</td>
<td>( matches )</td>
</tr>
<tr>
<td>[</td>
<td>{ / (2 \times a) }</td>
<td>[ matches ]</td>
</tr>
<tr>
<td>[</td>
<td>{ 2 \times a }</td>
<td>[ matches ]</td>
</tr>
<tr>
<td>[</td>
<td>{ }</td>
<td>( matches )</td>
</tr>
<tr>
<td>Empty</td>
<td>No more input</td>
<td>{ matches }</td>
</tr>
<tr>
<td></td>
<td></td>
<td>The parentheses are balanced</td>
</tr>
</tbody>
</table>
15.6.2 Evaluating Reverse Polish Expressions

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 709). To evaluate this expression, apply + to 3 and 4, yielding 7, and then simplify 7 5 \times 35. It gets trickier for complex expressions. For example, 3 4 5 + \times means to compute 4 5 + (that is, 9), and then evaluate 3 9 x. 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 + \times:

<table>
<thead>
<tr>
<th>Stack</th>
<th>Unread expression</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>Empty</td>
<td>3 4 5 + x</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>4 5 + x</td>
<td>Numbers are pushed on the stack</td>
</tr>
<tr>
<td>3 4</td>
<td>5 + x</td>
<td></td>
</tr>
<tr>
<td>3 4 5</td>
<td>+ x</td>
<td></td>
</tr>
<tr>
<td>3 9</td>
<td>x</td>
<td>Pop 4 and 5, push 4 5 +</td>
</tr>
<tr>
<td>27</td>
<td>No more input</td>
<td>Pop 3 and 9, push 3 9 x</td>
</tr>
<tr>
<td>Empty</td>
<td></td>
<td>Pop and display the result, 27</td>
</tr>
</tbody>
</table>

The following program simulates a reverse Polish calculator:

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

/**
 * This calculator uses the reverse Polish notation.
 */
public class Calculator {
    public static void main(String[] args) {
        Scanner in = new Scanner(System.in);
        Stack<Integer> results = new Stack<>();
        System.out.println("Enter one number or operator per line, Q to quit. ");
        boolean done = false;
```
while (!done)
{
    String input = in.nextLine();

    // If the command is an operator, pop the arguments and push the result
    if (input.equals("+"))
    {
        results.push(results.pop() + results.pop());
    }
    else if (input.equals("-"))
    {
        Integer arg2 = results.pop();
        results.push(results.pop() - arg2);
    }
    else if (input.equals("*") || input.equals("x"))
    {
        results.push(results.pop() * results.pop());
    }
    else if (input.equals("/"))
    {
        Integer arg2 = results.pop();
        results.push(results.pop() / arg2);
    }
    else if (input.equals("Q") || input.equals("q"))
    {
        done = true;
    }
    else
    {
        // Not an operator--push the input value
        results.push(Integer.parseInt(input));
    }
    System.out.println(results);
}

15.6.3 Evaluating Algebraic Expressions

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.
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.

<table>
<thead>
<tr>
<th>Number stack</th>
<th>Operator stack</th>
<th>Unprocessed input</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>Empty</td>
<td>Empty</td>
<td>3 + 4</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>3</td>
<td>+ 4</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>+</td>
<td>4</td>
</tr>
<tr>
<td>3</td>
<td>4</td>
<td>+</td>
<td>No more input</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>+</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>7</td>
<td></td>
<td>The result is 7.</td>
</tr>
</tbody>
</table>

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 × 4 + 5. Here are the first processing steps:

<table>
<thead>
<tr>
<th>Number stack</th>
<th>Operator stack</th>
<th>Unprocessed input</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>Empty</td>
<td>Empty</td>
<td>3 × 4 + 5</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>3</td>
<td>×</td>
<td>4 + 5</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>×</td>
<td>4 + 5</td>
</tr>
<tr>
<td>3</td>
<td>4</td>
<td>×</td>
<td>+ 5</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>×</td>
<td></td>
</tr>
</tbody>
</table>

Because × has a higher precedence than +, we are ready to evaluate the top:

<table>
<thead>
<tr>
<th>Number stack</th>
<th>Operator stack</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>12</td>
<td>+ 5</td>
</tr>
<tr>
<td>5</td>
<td>5</td>
<td>+</td>
</tr>
<tr>
<td></td>
<td>12</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>17</td>
<td></td>
</tr>
</tbody>
</table>

With the expression, 3 + 4 × 5, we add × to the operator stack because we must first read the next number; then we can evaluate × and then the +:

<table>
<thead>
<tr>
<th>Number stack</th>
<th>Operator stack</th>
<th>Unprocessed input</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>Empty</td>
<td>Empty</td>
<td>3 + 4 × 5</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>3</td>
<td>+ 4 × 5</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>+</td>
<td>4 + 5</td>
</tr>
</tbody>
</table>
In other words, we keep operators on the stack until they are ready to be evaluated. Here is the remainder of the computation:

<table>
<thead>
<tr>
<th>Number stack</th>
<th>Operator stack</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>4  × 3</td>
<td>No more input</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Evaluate the top.</td>
</tr>
</tbody>
</table>

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:

<table>
<thead>
<tr>
<th>Number stack</th>
<th>Operator stack</th>
<th>Unprocessed input</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>Empty</td>
<td>Empty</td>
<td>$3 \times (4 + 5)$</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>3</td>
<td>$\times (4 + 5)$</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>3  ×</td>
<td>$(4 + 5)$</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>3  (</td>
<td>$4 + 5$</td>
<td>Don’t evaluate × yet.</td>
</tr>
<tr>
<td>4</td>
<td>4  (</td>
<td>$+ 5$</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>4  (</td>
<td>$5$</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>5  +</td>
<td>)</td>
<td>Evaluate the top.</td>
</tr>
<tr>
<td>7</td>
<td>9  (</td>
<td>No more input</td>
<td>Pop ( .</td>
</tr>
<tr>
<td>8</td>
<td>9  ×</td>
<td>Evaluate top again.</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>27</td>
<td>That is the result.</td>
<td></td>
</tr>
</tbody>
</table>
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 ( (\text{and not empty})
    - 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.

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 1. 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 2. Both of them lead to dead ends 3 4.

Now we pop off the path from (3, 4) going east. That too is a dead end 5. Next is the path from (3, 4) going south. At (5, 4), it comes to an intersection. Both choices are pushed on the stack 6. They both lead to dead ends 7 8.

Finally, the path from (3, 4) going west leads to an exit 9.
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.
Chapter 15  The Java Collections Framework

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 cor-
ridors and asterisks for walls, like this:

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

In the example code, a Path object is constructed with a starting position and a direc-
tion (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.

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

```java
results.push(results.pop() - results.pop());
```

28. In the evaluation of the expression $3 - 4 + 5$ with 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 following simple maze. Assuming that we start at the marked point
and push paths in the order West, South, East, North, in which order are the let-
tered 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 cus-
tomers. Go to wiley.com/go/bjeo6examples and download Worked
Example 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.

<table>
<thead>
<tr>
<th>Standard Notation</th>
<th>Reverse Polish Notation</th>
</tr>
</thead>
<tbody>
<tr>
<td>$3 + 4$</td>
<td>$3 4 +$</td>
</tr>
<tr>
<td>$3 + 4 \times 5$</td>
<td>$3 4 5 \times +$</td>
</tr>
<tr>
<td>$3 \times (4 + 5)$</td>
<td>$3 4 5 + \times$</td>
</tr>
<tr>
<td>$(3 + 4) \times (5 + 6)$</td>
<td>$3 4 5 6 + \times$</td>
</tr>
<tr>
<td>$3 + 4 + 5$</td>
<td>$3 4 5 +$</td>
</tr>
</tbody>
</table>

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.
# Chapter 15
The Java Collections Framework

## 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<K, V>
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.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.

```java
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.

```java
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.

```java
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.

```java
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.

```java
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 10.4).

**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 lookups 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 10.4 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.

E15.1 Write a method
   ```java
   public static void downsize(LinkedList<String> employeeNames, int n)
   ```
   that removes every nth employee from a linked list.

E15.2 Write a method
   ```java
   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 will compute 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.

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 \( n \mod 10 \). 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`
Chapter 15  The Java Collections Framework

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 × 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> <ul> <li> </li> </ul> <a> </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 10×10 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.
**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”, \ldots \}. 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.

**P15.2** Reimplement Exercise E15.4 so that the keys of the map are objects of class \texttt{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. \textit{Hint:} Use two maps.

**P15.3** Write a class \texttt{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

```java
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 \(p(x) \times p(x)\).

```java
Polynomial q = p.multiply(p);
q.print();
```

**P15.4** Repeat Exercise P15.3, but use a \texttt{Map<Integer, Double>} for the coefficients.

**P15.5** Try to find two words with the same hash code in a large file. Keep a \texttt{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 \texttt{hashCode} and \texttt{equals} methods to the \texttt{Student} class described in Exercise P15.2. Test the hash code by adding \texttt{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. \textit{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` (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`. 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 719, 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:

```java
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, TreeSet<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.

6. We can simply access each array element with
an integer index.

7. |ABCD
   |BCD
   |CD
   |CD
   |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 test-
ing 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 associa-
tions 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*:

<table>
<thead>
<tr>
<th>Word</th>
<th>Frequency</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>653</td>
</tr>
<tr>
<td>abide</td>
<td>1</td>
</tr>
<tr>
<td>able</td>
<td>1</td>
</tr>
<tr>
<td>about</td>
<td>97</td>
</tr>
<tr>
<td>above</td>
<td>4</td>
</tr>
<tr>
<td>absence</td>
<td>1</td>
</tr>
<tr>
<td>absurd</td>
<td>2</td>
</tr>
</tbody>
</table>

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:

```plaintext
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:

```java
import java.util.Map;
import java.util.Scanner;
import java.util.TreeMap;
import java.io.File;
import java.io.FileNotFoundException;
```

*Big Java, 6e,* Cay Horstmann, Copyright © 2015 John Wiley and Sons, Inc. All rights reserved.
This program prints the frequencies of all words in “Alice in Wonderland”.

```java
public class WordFrequency {
    public static void main(String[] args)
            throws FileNotFoundException {
        Map<String, Integer> frequencies = new TreeMap<>();
        Scanner in = new Scanner(new File("alice30.txt"));
        while (in.hasNext()) {
            String word = clean(in.next());
            // Get the old frequency count
            Integer count = frequencies.get(word);
            // If there was none, put 1; otherwise, increment the count
            if (count == null) { count = 1; }
            else { count = count + 1; }
            frequencies.put(word, count);
        }
        // Print all words and counts
        for (String key : frequencies.keySet()) {
            System.out.printf("%-20s%10d\n", key, frequencies.get(key));
        }
    }

    /**
     * Removes characters from a string that are not letters.
     * @param s a string
     * @return a string with all the letters from s
     */
    public static String clean(String s) {
        String r = "";
        for (int i = 0; i < s.length(); i++) {
            char c = s.charAt(i);
            if (Character.isLetter(c)) {
                r = r + c;
            }
        }
        return r.toLowerCase();
    }
}
```
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.

```java
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.

![Bank diagram]

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:

```java
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(PROCESSING), 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
{
Simulating a Queue of Waiting Customers

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 cus-

tomers 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.<
Due to the random fluctuations of customer arrival and processing, the queue can get quite long:

At other times, the bank is empty again:

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.)

```java
import java.util.LinkedList;
import java.util.Queue;

/**
   * Simulation of customer traffic in a bank.
   */
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

public BankSimulation(int numberOfTellers)
{
  tellers = new Customer[numberOfTellers];
  custQueue = new LinkedList<>();
  totalCustomers = 0;
  totalTime = 0;
}

/**
 * Adds a customer to the bank.
 * @param c the customer
 */
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)));
}

/**
 * Adds a customer to a teller and schedules the departure event.
 * @param i the teller number
 * @param c the customer
 */
private void addToTeller(int i, Customer c)
{
  tellers[i] = c;
  addEvent(new Departure(getCurrentTime() + expdist(PROCESSING), i));
}

/**
 * Removes a customer from a teller.
 * @param i teller position
 */
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());
  }
}
/** Displays tellers and queue. */
public void display()
{
    for (int i = 0; i < tellers.length; i++)
    {
        if (tellers[i] == null)
        {
            System.out.print(".");
        } else
        {
            System.out.print("C");
        }
    }
    System.out.print("<");
    int q = custQueue.size();
    for (int j = 1; j <= q; j++) { System.out.print("C"); }
    System.out.println();
}

/** Displays a summary of the gathered statistics. */
public void displaySummary()
{
    double averageTime = 0;
    if (totalCustomers > 0)
    {
        averageTime = totalTime / totalCustomers;
    }
    System.out.println(totalCustomers + " customers. Average time " +
                        averageTime + " minutes.");
}
CHAPTER 16
BASIC DATA STRUCTURES

CHAPTER GOALS

To understand the implementation of linked lists and array lists
To analyze the efficiency of fundamental operations of lists and arrays
To implement the stack and queue data types
To implement a hash table and understand the efficiency of its operations

CHAPTER CONTENTS

16.1 IMPLEMENTING LINKED LISTS 722
   ST1 Static Classes 736
   WE1 Implementing a Doubly-Linked List

16.2 IMPLEMENTING ARRAY LISTS 737

16.3 IMPLEMENTING STACKS AND QUEUES 741
   ST2 Open Addressing 755

16.4 IMPLEMENTING A HASH TABLE 747
In the preceding chapter, you learned how to use the collection classes in the Java library. In this and the next chapter, we will study how these classes are implemented. This chapter deals with simple data structures in which elements are arranged in a linear sequence. By investigating how these data structures add, remove, and locate elements, you will gain valuable experience in designing algorithms and estimating their efficiency.

16.1 Implementing Linked Lists

In Chapter 15 you saw how to use the linked list class supplied by the Java library. Now we will look at the implementation of a simplified version of this class. This will show you how the list operations manipulate the links as the list is modified.

To keep this sample code simple, we will not implement all methods of the linked list class. We will implement only a singly-linked list, and the list class will supply direct access only to the first list element, not the last one. (A worked example and several exercises explore additional implementation options.) Our list will not use a type parameter. We will simply store raw Object values and insert casts when retrieving them. (You will see how to use type parameters in Chapter 18.) The result will be a fully functional list class that shows how the links are updated when elements are added or removed, and how the iterator traverses the list.

16.1.1 The Node Class

A linked list stores elements in a sequence of nodes. We need a class to represent the nodes. In a singly-linked list, a Node object stores an element and a reference to the next node.

Because the methods of both the linked list class and the iterator class have frequent access to the Node instance variables, we do not make the instance variables of the Node class private. Instead, we make Node a private inner class of the LinkedList class. An inner class is a class that is defined inside another class. The methods of the outer class can access the public features of the inner class. However, because the inner class is private, it cannot be accessed anywhere other than from the outer class.

```java
public class LinkedList
{
    . . .
    class Node
    {
        public Object data;
        public Node next;
    }
}
```

Our LinkedList class holds a reference first to the first node (or null, if the list is completely empty):

```java
public class LinkedList
{
    private Node first;
```
public LinkedList() { first = null; }

public Object getFirst()
{
    if (first == null) { throw new NoSuchElementException(); } return first.data;
}

16.1.2 Adding and Removing the First Element

Figure 1 shows the addFirst method in action. When a new node is added, it becomes the head of the list, and the node that was the old list head becomes its next node:

    public class LinkedList
    {
        ...
        public void addFirst(Object element)
        {
            Node newNode = new Node(); \[1\]
            newNode.data = element;
            newNode.next = first; \[2\]
            first = newNode; \[3\]
        }
        ...
    }

When adding or removing the first element, the reference to the first node must be updated.

**Figure 1**
Adding a Node to the Head of a Linked List
Removing the first element of the list works as follows. The data of the first node are saved and later returned as the method result. The successor of the first node becomes the first node of the shorter list (see Figure 2). Then there are no further references to the old node, and the garbage collector will eventually recycle it.

```java
public class LinkedList {
    ...
    public Object removeFirst() {
        if (first == null) { throw new NoSuchElementException(); }
        Object element = first.data;
        first = first.next;  // 1
        return element;
    }
    ...
}
```

### 16.1.3 The Iterator Class

The `ListIterator` interface in the standard library declares nine methods. Our simplified `ListIterator` interface omits four of them (the methods that move the iterator backward and the methods that report an integer index of the iterator). Our interface requires us to implement list iterator methods `next`, `hasNext`, `remove`, `add`, and `set`.

Our `LinkedList` class declares a private inner class `LinkedListIterator`, which implements our simplified `ListIterator` interface. Because `LinkedListIterator` is an inner class, it has access to the private features of the `LinkedList` class—in particular, the instance variable `first` and the private `Node` class.

Note that clients of the `LinkedList` class don’t actually know the name of the iterator class. They only know it is a class that implements the `ListIterator` interface.

Each iterator object has a reference, `position`, to the currently visited node. We also store a reference to the last node before that, `previous`. We will need that reference to adjust the links properly in the `remove` method. Finally, because calls to `remove` and `set`...
are only valid after a call to next, we use the isAfterNext flag to track when the next method has been called.

```
public class LinkedList
{
    ...
    public ListIterator listIterator()
    {
        return new LinkedListIterator();
    }

    class LinkedListIterator implements ListIterator
    {
        private Node position;
        private Node previous;
        private boolean isAfterNext;

        public LinkedListIterator()
        {
            position = null;
            previous = null;
            isAfterNext = false;
        }
        ...
    }
}
```

### 16.1.4 Advancing an Iterator

When advancing an iterator with the `next` method, the position reference is updated to `position.next`, and the old position is remembered in `previous`. The previous position is used for just one purpose: to remove the element if the `remove` method is called after the `next` method.

There is a special case, however—if the iterator points before the first element of the list, then the old position is `null`, and `position` must be set to `first`:

```
class LinkedListIterator implements ListIterator
{
    ...
    public Object next()
    {
        if (!hasNext()) { throw new NoSuchElementException(); }
        previous = position; // Remember for remove
        isAfterNext = true;

        if (position == null)
        {
            position = first;
        }
        else
        {
            position = position.next;
        }

        return position.data;
    }
    ...
}
```
The `next` method is supposed to be called only when the iterator is not yet at the end of the list, so we declare the `hasNext` method accordingly. The iterator is at the end if the list is empty (that is, `first == null`) or if there is no element after the current position (`position.next == null`):

```java
class LinkedListIterator implements ListIterator {
    ...  
    public boolean hasNext() {
        if (position == null) {
            return first != null;
        } else {
            return position.next != null;
        }
    }  
    ...  
}
```

16.1.5 Removing an Element

Next, we implement the `remove` method of the list iterator. Recall that, in order to remove an element, one must first call `next` and then call `remove` on the iterator.

If the element to be removed is the first element, we just call `removeFirst`. Otherwise, an element in the middle of the list must be removed, and the node preceding it needs to have its `next` reference updated to skip the removed element (see Figure 3).

We also need to update the `position` reference so that a subsequent call to the `next` method skips over the element after the removed one.

![Figure 3: Removing a Node from the Middle of a Linked List](image-url)
According to the specification of the `remove` method, it is illegal to call `remove` twice in a row. Our implementation handles this situation correctly. After completion of the `remove` method, the `isAfterNext` flag is set to `false`. An exception occurs if `remove` is called again without another call to `next`.

```java
class LinkedListIterator implements ListIterator {
    ...
    public void remove() {
        if (!isAfterNext) { throw new IllegalStateException(); }

        if (position == first) {
            removeFirst();
        } else {
            previous.next = position.next;  // 1
            position = previous;            // 2
            isAfterNext = false;            // 3
        }
    }
    ...
}
```

There is a good reason for disallowing `remove` twice in a row. After the first call to `remove`, the current position reverts to the predecessor of the removed element. Its predecessor is no longer known, which makes it impossible to efficiently remove the current element.
16.1.6 Adding an Element

The `add` method of the iterator inserts the new node after the last visited node (see Figure 4).

After adding the new element, we set the `isAfterNext` flag to `false`, in order to disallow a subsequent call to the `remove` or `set` method.

**Figure 4** Adding a Node to the Middle of a Linked List
class LinkedListIterator implements ListIterator {
    ...
    public void add(Object element) {
        if (position == null) {
            addFirst(element);
            position = first;
        } else {
            Node newNode = new Node();
            newNode.data = element;
            newNode.next = position.next;  // 1
            position.next = newNode;     // 2
            position = newNode;         // 3
        }
        isAfterNext = false;          // 4
    }
    ...
}

16.1.7 Setting an Element to a Different Value

The set method changes the data stored in the previously visited element:

public void set(Object element) {
    if (!isAfterNext) { throw new IllegalStateException(); }
    position.data = element;
}

As with the remove method, a call to set is only valid if it was preceded by a call to the next method. We throw an exception if we find that there was a call to add or remove immediately before calling set.

You will find the complete implementation of our LinkedList class after the next section.

16.1.8 Efficiency of Linked List Operations

Now that you have seen how linked list operations are implemented, we can determine their efficiency.

Consider first the cost of accessing an element. To get the \( k \)th element of a linked list, you start at the beginning of the list and advance the iterator \( k \) times. Suppose it takes an amount of time \( T \) to advance the iterator once. This quantity is independent of the iterator position—advancing an iterator does some checking and then it follows the next reference of the current node (see Section 16.1.4).

Therefore, advancing the iterator to the \( k \)th element consumes \( kT \) time. If the linked list has \( n \) elements and \( k \) is chosen at random, then \( k \) will average out to be \( n/2 \), and \( kT \) is on average \( nT/2 \). Because \( T/2 \) is a constant, this is an \( O(n) \) expression. We have determined that accessing an element in a linked list of length \( n \) is an \( O(n) \) operation.

Now consider the cost of adding an element at a given position, assuming that we already have an iterator to the position. Look at the implementation of the add
method in Section 16.1.6. To add an element, one updates a couple of references in the neighboring nodes and the iterator. This operation requires a constant number of steps, independent of the size of the linked list.

Using the big-Oh notation, an operation that requires a bounded amount of time, regardless of the total number of elements in the structure, is denoted as $O(1)$. Adding an element to a linked list takes $O(1)$ time.

Similar reasoning shows that removing an element at a given position is an $O(1)$ operation.

Now consider the task of adding an element at the end of the list. We first need to get to the end, at a cost of $O(n)$. Then it takes $O(1)$ time to add the element. However, we can improve on this performance if we add a reference to the last node to the `LinkedList` class:

```java
public class LinkedList {
    private Node first;
    private Node last;
    . . .
}
```

Of course, this reference must be updated when the last node changes, as elements are added or removed. In order to keep the code as simple as possible, our implementation does not have a reference to the last node. However, we will always assume that a linked list implementation can access the last element in constant time. This is the case for the `LinkedList` class in the standard Java library, and it is an easy enhancement to our implementation. Worked Example 16.1 shows how to add the last reference, update it as necessary, and provide an `addLast` method for adding an element at the end.

To get to the $k$th node of a linked list, one must skip over the preceding nodes.

**Figure 5** Removing the Last Element of a Singly-Linked List
The code for the addLast method is very similar to the addFirst method in Section 16.1.2. It too requires constant time, independent of the length of the list. We conclude that, with an appropriate implementation, adding an element at the end of a linked list is an $O(1)$ operation.

How about removing the last element? We need a reference to the next-to-last element, so that we can set its next reference to null. (See Figure 5.)

We also need to update the last reference and set it to the next-to-last reference. But how can we get that next-to-last reference? It takes $n - 1$ iterations to obtain it, starting at the beginning of the list. Thus, removing an element from the back of a singly-linked list is an $O(n)$ operation.

We can do better in a doubly-linked list, such as the one in the standard Java library. In a doubly-linked list, each node has a reference to the previous node in addition to the next one (see Figure 6).

```
public class LinkedList
{
    . . .
    class Node
    {
        public Object data;
        public Node next;
        public Node previous;
    }
}
```

In that case, removal of the last element takes a constant number of steps:

1. last = last.previous;
2. last.next = null;

![Figure 6](image-url)  
**Figure 6** Removing the Last Element of a Doubly-Linked List
Therefore, removing an element from the end of a doubly-linked list is also an \( O(1) \) operation. Worked Example 16.1 contains a full implementation. Table 1 summarizes the efficiency of linked list operations.

<table>
<thead>
<tr>
<th>Operation</th>
<th>Singly-Linked List</th>
<th>Doubly-Linked List</th>
</tr>
</thead>
<tbody>
<tr>
<td>Access an element</td>
<td>( O(n) )</td>
<td>( O(n) )</td>
</tr>
<tr>
<td>Add/remove at an iterator position</td>
<td>( O(1) )</td>
<td>( O(1) )</td>
</tr>
<tr>
<td>Add/remove first element</td>
<td>( O(1) )</td>
<td>( O(1) )</td>
</tr>
<tr>
<td>Add last element</td>
<td>( O(1) )</td>
<td>( O(1) )</td>
</tr>
<tr>
<td>Remove last element</td>
<td>( O(n) )</td>
<td>( O(1) )</td>
</tr>
</tbody>
</table>

section_1/LinkedList.java

```java
import java.util.NoSuchElementException;

/**
   A linked list is a sequence of nodes with efficient
   element insertion and removal. This class
   contains a subset of the methods of the standard
   java.util.LinkedList class.

   public class LinkedList
   {
      private Node first;

      /**
         Constructs an empty linked list.
         */
      public LinkedList()
      {
         first = null;
      }

      /**
         Returns the first element in the linked list.
         @return the first element in the linked list
         */
      public Object getFirst()
      {
         if (first == null) { throw new NoSuchElementException(); }
         return first.data;
      }

      /**
         Removes the first element in the linked list.
         @return the removed element
         */
      public Object removeFirst()
      {
         if (first == null) { throw new NoSuchElementException(); }
         Object element = first.data;
```
public void addFirst(Object element) {
    Node newNode = new Node();
    newNode.data = element;
    newNode.next = first;
    first = newNode;
}

public ListIterator listIterator() {
    return new LinkedListIterator();
}

class LinkedListIterator implements ListIterator {
    private Node position;
    private Node previous;
    private boolean isAfterNext;
    private Node hasNext() { return position != null; }

    public LinkedListIterator() {
        position = null;
        previous = null;
        isAfterNext = false;
    }

    public Object next() {
        if (!hasNext()) { throw new NoSuchElementException(); }
        previous = position; // Remember for remove
        isAfterNext = true;
        if (position == null) {
            position = first;
        } else {
            position = position.next;
        }
        return position.data;
    }
}

class Node {
    public Object data;
    public Node next;
}

/**
 * Adds an element to the front of the linked list.
 * @param element the element to add
 */
public void addFirst(Object element) {
    Node newNode = new Node();
    newNode.data = element;
    newNode.next = first;
    first = newNode;
}

/**
 * Returns an iterator for iterating through this list.
 * @return an iterator for iterating through this list
 */
public ListIterator listIterator() {
    return new LinkedListIterator();
}

class Node {
    public Object data;
    public Node next;
}

class LinkedListIterator implements ListIterator {
    private Node position;
    private Node previous;
    private boolean isAfterNext;
    private Node hasNext() { return position != null; }

    public LinkedListIterator() {
        position = null;
        previous = null;
        isAfterNext = false;
    }

    public Object next() {
        if (!hasNext()) { throw new NoSuchElementException(); }
        previous = position; // Remember for remove
        isAfterNext = true;
        if (position == null) {
            position = first;
        } else {
            position = position.next;
        }
        return position.data;
    }
}
position = first;
else {
    position = position.next;
}
return position.data;

/**
Tests if there is an element after the iterator position.
@return true if there is an element after the iterator position
*/
public boolean hasNext() {
    if (position == null) {
        return first != null;
    } else {
        return position.next != null;
    }
}

/**
Adds an element before the iterator position and moves the iterator past the inserted element.
@param element the element to add
*/
public void add(Object element) {
    if (position == null) {
        addFirst(element);
        position = first;
    } else {
        Node newNode = new Node();
        newNode.data = element;
        newNode.next = position.next;
        position.next = newNode;
        position = newNode;
    }
    isAfterNext = false;
}

/**
Removes the last traversed element. This method may only be called after a call to the next method.
*/
public void remove() {
    if (!isAfterNext) { throw new IllegalStateException(); }
    if (position == first) {
    }
16.1 Implementing Linked Lists

```java
159     removeFirst();
160     }
161     else
162     {
163         previous.next = position.next;
164     }
165     position = previous;
166     isAfterNext = false;
167 }
168
169     /**
170     * Sets the last traversed element to a different value.
171     * @param element the element to set
172     */
173     public void set(Object element)
174     {
175         if (!isAfterNext) { throw new IllegalStateException(); }
176         position.data = element;
177     }
178 }
```

section_1/ListIterator.java

```java
/**
 * A list iterator allows access to a position in a linked list. This interface contains a subset of the methods of the standard java.util.ListIterator interface. The methods for backward traversal are not included.
 */
public interface ListIterator
{

    /**
     * Moves the iterator past the next element.
     * @return the traversed element
     */
    Object next();

    /**
     * Tests if there is an element after the iterator position.
     * @return true if there is an element after the iterator position
     */
    boolean hasNext();

    /**
     * Adds an element before the iterator position and moves the iterator past the inserted element.
     * @param element the element to add
     */
    void add(Object element);

    /**
     * Removes the last traversed element. This method may only be called after a call to the next method.
     */
    void remove();

    /**
     * Sets the last traversed element to a different value.
     * @param element the element to set
     */
```
SELF CHECK

1. Trace through the addFirst method when adding an element to an empty list.
2. Conceptually, an iterator is located between two elements (see Figure 9 in Chapter 15). Does the position instance variable refer to the element to the left or the element to the right?
3. Why does the add method have two separate cases?
4. Assume that a last reference is added to the LinkedList class, as described in Section 16.1.8. How does the add method of the ListIterator need to change?
5. Provide an implementation of an addLast method for the LinkedList class, assuming that there is no last reference.
6. Expressed in big-Oh notation, what is the efficiency of the addFirst method of the LinkedList class? What is the efficiency of the addLast method of Self Check 5?
7. How much slower is the binary search algorithm for a linked list compared to the linear search algorithm?

Practice It

Static Classes

You first saw the use of inner classes for event handlers in Chapter 10. Inner classes are useful in that context, because their methods have the privilege of accessing private instance variables of outer-class objects. The same is true for the LinkedListIterator inner class in the sample code for this section. The iterator needs to access the first instance variable of its linked list.

However, there is a cost for this feature. Every object of the inner class has a reference to the object of the enclosing class that constructed it. If an inner class has no need to access the enclosing class, you can declare the class as static and eliminate the reference to the enclosing class. This is the case with the Node class.

You can declare it as follows:

```java
public class LinkedList
{
    . . .
    static class Node
    {
        . . .
    }
}
```

However, the LinkedListIterator class cannot be a static class. Its methods must access the first element of the enclosing LinkedList.

WORKED EXAMPLE 16.1

Implementing a Doubly-Linked List

Learn how to modify a singly-linked list to implement a doubly-linked list. Go to wiley.com/go/bjeo6examples and download Worked Example 16.1.
16.2 Implementing Array Lists

Array lists were introduced in Chapter 7. They are conceptually similar to linked lists, allowing you to add and remove elements at any position. In the following sections, we will develop an implementation of an array list, study the efficiency of operations on array lists, and compare them with the equivalent operations on linked lists.

16.2.1 Getting and Setting Elements

An array list maintains a reference to an array of elements. The array is large enough to hold all elements in the collection—in fact, it is usually larger to allow for adding additional elements. When the array gets full, it is replaced by a larger one. We discuss that process in Section 16.2.3.

In addition to the internal array of elements, an array list has an instance field that stores the current number of elements (see Figure 7).

For simplicity, our ArrayList implementation does not work with arbitrary element types, but it simply manages elements of type Object. (Chapter 18 shows how to implement classes with type parameters.)

```java
public class ArrayList {
    private Object[] elements;
    private int currentSize;

    public ArrayList() {
        final int INITIAL_SIZE = 10;
        elements = new Object[INITIAL_SIZE];
        currentSize = 0;
    }

    public int size() { return currentSize; }
    . . .
}
```

To access array list elements, we provide get and set methods. These methods simply check for valid positions and access the internal array at the given position:
private void checkBounds(int n)
{
    if (n < 0 || n >= currentSize)
    {
        throw new IndexOutOfBoundsException();
    }
}

public Object get(int pos)
{
    checkBounds(pos);
    return element[pos];
}

public void set(int pos, Object element)
{
    checkBounds(pos);
    elements[pos] = element;
}

As you can see, getting and setting an element can be carried out with a bounded set of instructions, independent of the size of the array list. These are $O(1)$ operations.

### 16.2.2 Removing or Adding Elements

When removing an element at position $k$, the elements with higher index values need to move (see Figure 8). Here is the implementation, following Section 7.3.6:

public Object remove(int pos)
{
    checkBounds(pos);

    Object removed = elements[pos];

    for (int i = pos + 1; i < currentSize; i++)
    {
        elements[i - 1] = elements[i];
    }

    currentSize--;
    return removed;
}

How many elements are affected? If we assume that removal happens at random locations, then on average, each removal moves $n/2$ elements, where $n$ is the size of the array list.

---

**Figure 8**
Removing and Adding Elements
The same argument holds for inserting an element. On average, \( n / 2 \) elements need to be moved. Therefore, we say that adding and removing elements are \( O(n) \) operations.

There is one situation where adding an element to an array list isn’t so costly: when the insertion happens after the last element. If the current size is less than the length of the array, the size is incremented and the new element is simply stored in the array. This is an \( O(1) \) operation.

```java
public boolean addLast(Object newElement) {
    growIfNecessary();
    currentSize++;
    elements[currentSize - 1] = newElement;
    return true;
}
```

One issue remains: If there is no more room in the internal array, then we need to grow it. That is the topic of the next section.

### 16.2.3 Growing the Internal Array

Before inserting an element into an internal array that is completely full, we must replace the array with a bigger one. This new array is typically twice the size of the current array. (See Figure 9.) The existing elements are then copied into the new array. Reallocation is an \( O(n) \) operation because all elements need to be copied to the new array.

```java
private void growIfNecessary() {
    if (currentSize == elements.length) {
        Object[] newElements = new Object[2 * elements.length];
        for (int i = 0; i < elements.length; i++) {
            newElements[i] = elements[i];
        }
        elements = newElements;
    }
}
```
If we carefully analyze the total cost of a sequence of addLast operations, it turns out that these reallocations are not as expensive as they first appear. The key observation is that array growth does not happen very often. Suppose we start with an array list of capacity 10 and double the size with each reallocation. We must reallocate the array of elements when it reaches sizes 10, 20, 40, 80, 160, 320, 640, 1280, and so on.

Let us assume that one insertion without reallocation takes time \( T_1 \) and that reallocation of \( k \) elements takes time \( kT_2 \). What is the cost of 1280 addLast operations?

Of course, we pay \( 1280 \cdot T_1 \) for the insertions. The reallocation cost is

\[
10T_2 + 20T_2 + 40T_2 + \cdots + 1280T_2 = (1 + 2 + 4 + \cdots + 128) \cdot 10 \cdot T_2
\]

\[
= 255 \cdot 10 \cdot T_2
\]

\[
< 256 \cdot 10 \cdot T_2
\]

\[
= 1280 \cdot 2 \cdot T_2
\]

Therefore, the total cost is a bit less than \( 1280 \cdot (T_1 + 2T_2) \).

In general, the total cost of \( n \) addLast operations is less than \( n \cdot (T_1 + 2T_2) \). Because the second factor is a constant, we conclude that \( n \) addLast operations take \( O(n) \) time.

We know that it isn’t quite true that an individual addLast operation takes \( O(1) \) time. After all, occasionally a call to addLast is unlucky and must reallocate the elements array.

But if the cost of that reallocation is distributed over the preceding addLast operations, then the surcharge for each of them is still a constant amount. We say that addLast takes amortized \( O(1) \) time, which is written as \( O(1)+ \). (Accountants say that a cost is amortized when it is distributed over multiple periods.)

In our implementation, we do not shrink the array when elements are removed. However, it turns out that you can (occasionally) shrink the array and still have \( O(1)+ \) performance for removing the last element (see Exercise E16.10).

### Table 2 Efficiency of Array List and Linked List Operations

<table>
<thead>
<tr>
<th>Operation</th>
<th>Array List</th>
<th>Doubly-Linked List</th>
</tr>
</thead>
<tbody>
<tr>
<td>Add/remove element at end.</td>
<td>( O(1)+ )</td>
<td>( O(1) )</td>
</tr>
<tr>
<td>Add/remove element in the middle.</td>
<td>( O(n) )</td>
<td>( O(1) )</td>
</tr>
<tr>
<td>Get ( k )th element.</td>
<td>( O(1) )</td>
<td>( O(k) )</td>
</tr>
</tbody>
</table>

**SELF CHECK**

8. Why is it much more expensive to get the \( k \)th element in a linked list than in an array list?

9. Why is it much more expensive to insert an element at the beginning of an array list than at the beginning of a linked list?

10. What is the efficiency of adding an element exactly in the middle of a linked list? An array list?

11. Suppose we insert an element at the beginning of an array list, and the internal array must be grown to hold the new element. What is the efficiency of the add operation in this situation?
12. Using big-Oh notation, what is the cost of adding an element to an array list as the second-to-last element?

Practice It Now you can try these exercises at the end of the chapter: R16.9, R16.10, R16.11.

### 16.3 Implementing Stacks and Queues

In Section 15.5, we introduced the stack and queue data types. Stacks and queues are very simple. Elements are added and retrieved, either in last-in, first-out order or in first-in, first-out order.

Stacks and queues are examples of abstract data types. We only specify how the operations must behave, not how they are implemented. In the following sections, we will study several implementations of stacks and queues and determine how efficient they are.

#### 16.3.1 Stacks as Linked Lists

Let us first implement a stack as a sequence of nodes. New elements are added (or “pushed”) to an end of the sequence, and they are removed (or “popped”) from the same end.

Which end? It is up to us to choose, and we will make the least expensive choice: to add and remove elements at the front (see Figure 10).

**Figure 10** Push and Pop for a Stack Implemented as a Linked List
The push and pop operations are identical to the addFirst and removeFirst operations from Section 16.1.2. They are both O(1) operations.

Here is the complete implementation:

section_3_1/LinkedListStack.java

```java
import java.util.NoSuchElementException;

/**
 * An implementation of a stack as a sequence of nodes.
 */
public class LinkedListStack {
    private Node first;

    /**
     * Constructs an empty stack.
     */
    public LinkedListStack()
    {
        first = null;
    }

    /**
     * Adds an element to the top of the stack.
     * @param element the element to add
     */
    public void push(Object element)
    {
        Node newNode = new Node();
        newNode.data = element;
        newNode.next = first;
        first = newNode;
    }

    /**
     * Removes the element from the top of the stack.
     * @return the removed element
     */
    public Object pop()
    {
        if (first == null) { throw new NoSuchElementException(); }
        Object element = first.data;
        first = first.next;
        return element;
    }

    /**
     * Checks whether this stack is empty.
     * @return true if the stack is empty
     */
    public boolean empty()
    {
        return first == null;
    }
}

class Node {
    public Object data;
    public Node next;
}
```
16.3.2 Stacks as Arrays

In the preceding section, you saw how a list was implemented as a sequence of nodes. In this section, we will instead store the values in an array, thus saving the storage of the node references.

Again, it is up to us at which end of the array we place new elements. This time, it is better to add and remove elements at the back of the array (see Figure 11).

Of course, an array may eventually fill up as more elements are pushed on the stack. As with the ArrayList implementation of Section 16.2, the array must grow when it gets full.

The push and pop operations are identical to the addLast and removeLast operations of an array list. They are both $O(1)+$ operations.

![Figure 11](A Stack Implemented as an Array)

16.3.3 Queues as Linked Lists

We now turn to the implementation of a queue. When implementing a queue as a sequence of nodes, we add nodes at one end and remove them at the other. As we discussed in Section 16.1.8, a singly-linked node sequence is not able to remove the last node in $O(1)$ time. Therefore, it is best to remove elements at the front and add them at the back (see Figure 12).

![Figure 12](A Queue Implemented as a Linked List)
The add and remove operations of a queue are $O(1)$ operations because they are the same as the addLast and removeFirst operations of a doubly-linked list. Note that we need a reference to the last node so that we can efficiently add elements.

### 16.3.4 Queues as Circular Arrays

When storing queue elements in an array, we have a problem: elements get added at one end of the array and removed at the other. But adding or removing the first element of an array is an $O(n)$ operation, so it seems that we cannot avoid this expensive operation, no matter which end we choose for adding elements and which for removing them.

However, we can solve this problem with a trick. We add elements at the end, but when we remove them, we don’t actually move the remaining elements. Instead, we increment the index at which the head of the queue is located (see Figure 13).

After adding sufficiently many elements, the last element of the array will be filled. However, if there were also a few calls to remove, then there is additional room in the front of the array. Then we “wrap around” and start storing elements again at index 0—see part 2 of Figure 13. For that reason, the array is called “circular”.

Eventually, of course, the tail reaches the head, and a larger array must be allocated.

As you can see from the source code that follows, adding or removing an element requires a bounded set of operations, independent of the queue size, except for array

### Figure 13  Queue Elements in a Circular Array

**In a circular array, we wrap around to the beginning after the last element.**

In a circular array implementation of a queue, element locations wrap from the end of the array to the beginning.
re allocation. However, as discussed in Section 16.2.3, re allocation happens rarely enough that the total cost is still amortized constant time, \( O(1) + \).

### Table 3  Efficiency of Stack and Queue Operations

<table>
<thead>
<tr>
<th></th>
<th>Stack as Linked List</th>
<th>Stack as Array</th>
<th>Queue as Linked List</th>
<th>Queue as Circular Array</th>
</tr>
</thead>
<tbody>
<tr>
<td>Add an element</td>
<td>( O(1) )</td>
<td>( O(1)+ )</td>
<td>( O(1) )</td>
<td>( O(1)+ )</td>
</tr>
<tr>
<td>Remove an element</td>
<td>( O(1) )</td>
<td>( O(1)+ )</td>
<td>( O(1) )</td>
<td>( O(1)+ )</td>
</tr>
</tbody>
</table>

section_3_4/CircularArrayQueue.java

```java
import java.util.NoSuchElementException;

/**
   An implementation of a queue as a circular array.
*/
public class CircularArrayQueue {
  private Object[] elements;
  private int currentSize;
  private int head;
  private int tail;

  /**
   Constructs an empty queue.
   */
  public CircularArrayQueue() {
    final int INITIAL_SIZE = 10;
    elements = new Object[INITIAL_SIZE];
    currentSize = 0;
    head = 0;
    tail = 0;
  }

  /**
   Checks whether this queue is empty.
   @return true if this queue is empty
   */
  public boolean empty() { return currentSize == 0; }

  /**
   Adds an element to the tail of this queue.
   @param newElement the element to add
   */
  public void add(Object newElement) {
    growIfNecessary();
    currentSize++;
    elements[tail] = newElement;
    tail = (tail + 1) % elements.length;
  }

  /**
   Removes an element from the head of this queue.
   @return the removed element
   */
  public Object remove() { return tail == head ? null : elements[head++]; }
}
```
46 */
47   public Object remove()
48   {
49       if (currentSize == 0) { throw new NoSuchElementException(); }
50       Object removed = elements[head];
51       head = (head + 1) % elements.length;
52       currentSize--;
53       return removed;
54   }
55
56 /**
57   * Grows the element array if the current size equals the capacity.
58   */
59   private void growIfNecessary()
60   {
61       if (currentSize == elements.length)
62           {
63               Object[] newElements = new Object[2 * elements.length];
64               for (int i = 0; i < elements.length; i++)
65                   {
66                       newElements[i] = elements[(head + i) % elements.length];
67                   }
68               elements = newElements;
69               head = 0;
70               tail = currentSize;
71           }
72   }
73
13. Add a method peek to the Stack implementation in Section 16.3.1 that returns the top of the stack without removing it.
14. When implementing a stack as a sequence of nodes, why isn’t it a good idea to push and pop elements at the back end?
15. When implementing a stack as an array, why isn’t it a good idea to push and pop elements at index 0?
16. What is wrong with this implementation of the empty method for the circular array queue?
   public boolean empty()
   {
       return head == 0 && tail == 0;
   }
17. What is wrong with this implementation of the empty method for the circular array queue?
   public boolean empty()
   {
       return head == tail;
   }
18. Have a look at the growIfNecessary method of the CircularArrayQueue class. Why isn’t the loop simply
   for (int i = 0; i < elements.length; i++)
   {
       newElements[i] = elements[i];
   }

Practice It Now you can try these exercises at the end of the chapter: R16.20, R16.23, E16.11, E16.12.
In Section 15.3, you were introduced to the set data structure and its two implementations in the Java collections framework, hash sets and tree sets. In these sections, you will see how hash sets are implemented and how efficient their operations are.

### 16.4.1 Hash Codes

The basic idea behind hashing is to place objects into an array, at a location that can be determined from the object itself. Each object has a hash code, an integer value that is computed from an object in such a way that different objects are likely to yield different hash codes.

Table 4 shows some examples of strings and their hash codes. Special Topic 15.1 shows how these values are computed.

It is possible for two or more distinct objects to have the same hash code; this is called a collision. For example, the strings “VII” and “ugh” happen to have the same hash code.

<table>
<thead>
<tr>
<th>String</th>
<th>Hash Code</th>
<th>String</th>
<th>Hash Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;Adam&quot;</td>
<td>2035631</td>
<td>&quot;Juliet&quot;</td>
<td>-2065036585</td>
</tr>
<tr>
<td>&quot;Eve&quot;</td>
<td>70068</td>
<td>&quot;Katherine&quot;</td>
<td>2079199209</td>
</tr>
<tr>
<td>&quot;Harry&quot;</td>
<td>69496448</td>
<td>&quot;Sue&quot;</td>
<td>83491</td>
</tr>
<tr>
<td>&quot;Jim&quot;</td>
<td>74478</td>
<td>&quot;ugh&quot;</td>
<td>84982</td>
</tr>
<tr>
<td>&quot;Joe&quot;</td>
<td>74656</td>
<td>&quot;VII&quot;</td>
<td>84982</td>
</tr>
</tbody>
</table>

### 16.4.2 Hash Tables

A hash code is used as an array index into a hash table, an array that stores the set elements. In the simplest implementation of a hash table, you could make a very long array and insert each object at the location of its hash code (see Figure 14).

If there are no collisions, it is a very simple matter to find out whether an object is already present in the set or not. Compute its hash code and check whether the array position with that hash code is already occupied. This doesn’t require a search through the entire array!
Elements with the same hash code are placed in the same bucket.

Of course, it is not feasible to allocate an array that is large enough to hold all possible integer index positions. Therefore, we must pick an array of some reasonable size and then “compress” the hash code to become a valid array index. Compression can be easily achieved by using the remainder operation:

```java
int h = x.hashCode();
if (h < 0) { h = -h; }
position = h % arrayLength;
```

See Exercise E16.20 for an alternative compression technique.

After compressing the hash code, it becomes more likely that several objects will collide. There are several techniques for handling collisions. The most common one is called **separate chaining**. All colliding elements are collected in a linked list of elements with the same position value (see Figure 15). Such a list is called a “bucket”. Special Topic 16.2 discusses **open addressing**, in which colliding elements are placed in empty locations of the hash table.

In the following, we will use the first technique. Each entry of the hash table points to a sequence of nodes containing elements with the same (compressed) hash code.

![A hash table can be implemented as an array of buckets—sequences of nodes that hold elements with the same hash code.](image)

![Figure 15  A Hash Table with Buckets to Store Elements with the Same Hash Code](image)
16.4.3 Finding an Element

Let’s assume that our hash table has been filled with a number of elements. Now we want to find out whether a given element is already present.

Here is the algorithm for finding an object obj in a hash table:

1. Compute the hash code and compress it. This gives an index $h$ into the hash table.
2. Iterate through the elements of the bucket at position $h$. For each element of the bucket, check whether it is equal to obj.
3. If a match is found among the elements of that bucket, then obj is in the set. Otherwise, it is not.

How efficient is this operation? It depends on the hash code computation. In the best case, in which there are no collisions, all buckets either are empty or have a single element.

But in practice, some collisions will occur. We need to make some assumptions that are reasonable in practice.

First, we assume that the hash code does a good job scattering the elements into different buckets. In practice, the hash functions described in Special Topic 15.1 work well.

Next, we assume that the table is large enough. This is measured by the load factor $F = n / L$, where $n$ is the number of elements and $L$ the table length. For example, if the table is an array of length 1,000, and it has 700 elements, then the load factor is 0.7.

If the load factor gets too large, the elements should be moved into a larger table. The hash table in the standard Java library reallocates the table when the load factor exceeds 0.75.

Under these assumptions, each bucket can be expected to have, on average, $F$ elements.

Finally, we assume that the hash code, its compression, and the equals method can be computed in bounded time, independent of the size of the set.

Now let us compute the cost of finding an element. Computing the array index takes constant time, due to our last assumption. Now we traverse a chain of buckets, which on average has a bounded length $F$. Finally, we invoke the equals method on each bucket element, which we also assume to be $O(1)$. The entire operation takes constant or $O(1)$ time.

16.4.4 Adding and Removing Elements

Adding an element is an extension of the algorithm for finding an object. First compute the hash code to locate the bucket in which the element should be inserted:

1. Compute the compressed hash code $h$.
2. Iterate through the elements of the bucket at position $h$. For each element of the bucket, check whether it is equal to obj (using the equals method of the element type).
3. If a match is found among the elements of that bucket, then exit.
4. Otherwise, add a node containing obj to the beginning of the node sequence.
5. If the load factor exceeds a fixed threshold, reallocate the table.
As described in the preceding section, the first three steps are $O(1)$. Inserting at the beginning of a node sequence is also $O(1)$. As with array lists, we can choose the new table to be twice the size of the old table, and amortize the cost of reallocation over the preceding insertions. That is, adding an element to a hash table is $O(1)$. Removing an element is equally simple. First compute the hash code to locate the bucket in which the element should be inserted. Try finding the object in that bucket. If it is present, remove it. Otherwise, do nothing. Again, this is a constant time operation. If we shrink a table that becomes too sparse, the cost is $O(1)$. 

**16.4.5 Iterating over a Hash Table**

An iterator for a linked list points to the current node in a list. A hash table has multiple node chains. When we are at the end of one chain, we need to move to the start of the next one. Therefore, the iterator also needs to store the bucket number (see Figure 16).  

When the iterator points into the middle of a node chain, then it is easy to advance it to the next element. However, when the iterator points to the last node in a chain, then we must skip past all empty buckets. When we find a non-empty bucket, we advance the iterator to its first node:

```java
if (current != null && current.next != null) {
    current = current.next; // Move to next element in bucket
} else { // Move to next bucket
    do {
        bucketIndex++;
        if (bucketIndex == buckets.length) {
            throw new NoSuchElementException();
        }
        current = buckets[bucketIndex];
    } while (current == null);
}
```

![Figure 16](image_url)

**Figure 16**  
An Iterator to a Hash Table
As you can see, the cost of iterating over all elements of a hash table is proportional to the table length. Note that the table length could be in excess of $O(n)$ if the table is sparsely filled. This can be avoided if we shrink the table when the load factor gets too small. In that case, iterating over the entire table is $O(n)$, and each iteration step is $O(1)$.

Table 5 summarizes the efficiency of the operations on a hash table.

<table>
<thead>
<tr>
<th>Operation</th>
<th>Hash Table</th>
</tr>
</thead>
<tbody>
<tr>
<td>Find an element.</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>Add/remove an element.</td>
<td>$O(1)+$</td>
</tr>
<tr>
<td>Iterate through all elements.</td>
<td>$O(n)$</td>
</tr>
</tbody>
</table>

Here is an implementation of a hash set. For simplicity, we do not reallocate the table when it grows or shrinks, and we do not support the `remove` operation on iterators. Exercises E16.18 and E16.19 ask you to provide these enhancements.

```java
import java.util.Iterator;
import java.util.NoSuchElementException;

/**
 * This class implements a hash set using separate chaining.
 */
public class HashSet {
    private Node[] buckets;
    private int currentSize;

    /**
     * Constructs a hash table.
     * @param bucketsLength the length of the buckets array
     */
    public HashSet(int bucketsLength) {
        buckets = new Node[bucketsLength];
        currentSize = 0;
    }

    /**
     * Tests for set membership.
     * @param x an object
     * @return true if x is an element of this set
     */
    public boolean contains(Object x) {
        int h = x.hashCode();
        if (h < 0) { h = -h; }
        h = h % buckets.length;
        // Implementation of contains
    }
}
```
Node current = buckets[h];
while (current != null) {
    if (current.data.equals(x)) { return true; }
    current = current.next;
}
return false;

/**
 * Adds an element to this set.
 * @param x an object
 * @return true if x is a new object, false if x was already in the set
 */
public boolean add(Object x) {
    int h = x.hashCode();
    if (h < 0) { h = -h; }
    h = h % buckets.length;

    Node current = buckets[h];
    while (current != null) {
        if (current.data.equals(x)) { return false; }
        // Already in the set
        current = current.next;
    }
    Node newNode = new Node();
    newNode.data = x;
    newNode.next = buckets[h];
    buckets[h] = newNode;
    currentSize++;
    return true;
}

/**
 * Removes an object from this set.
 * @param x an object
 * @return true if x was removed from this set, false if x was not an element of this set
 */
public boolean remove(Object x) {
    int h = x.hashCode();
    if (h < 0) { h = -h; }
    h = h % buckets.length;

    Node current = buckets[h];
    Node previous = null;
    while (current != null) {
        if (current.data.equals(x)) {
            if (previous == null) { buckets[h] = current.next; }
            else { previous.next = current.next; }
            currentSize--;
            return true;
        }
        previous = current;
Implementing a Hash Table

```java
    current = current.next;
    return false;
}

/**
 * Returns an iterator that traverses the elements of this set.
 * @return a hash set iterator
 */
public Iterator iterator()
{
    return new HashSetIterator();
}

/**
 * Gets the number of elements in this set.
 * @return the number of elements
 */
public int size()
{
    return currentSize;
}

class Node
{
    public Object data;
    public Node next;
}

class HashSetIterator implements Iterator
{
    private int bucketIndex;
    private Node current;

    /**
     * Constructs a hash set iterator that points to the
     * first element of the hash set.
     */
    public HashSetIterator()
    {
        current = null;
        bucketIndex = -1;
    }

    public boolean hasNext()
    {
        if (current != null && current.next != null) { return true; }
        for (int b = bucketIndex + 1; b < buckets.length; b++)
        {
            if (buckets[b] != null) { return true; }
        }
        return false;
    }

    public Object next()
    {
        if (current != null && current.next != null)
        {
            current = current.next; // Move to next element in bucket
        }
    }
```
else // Move to next bucket
    {
        do
        {
            bucketIndex++;
            if (bucketIndex == buckets.length)
            {
                throw new NoSuchElementException();
            }
            current = buckets[bucketIndex];
            while (current == null);
        }
        return current.data;
    }

    public void remove()
    {
        throw new UnsupportedOperationException();
    }
}

section_4/HashSetDemo.java

import java.util.Iterator;
/**
 * This program demonstrates the hash set class.
 */
public class HashSetDemo
{
    public static void main(String[] args)
    {
        HashSet names = new HashSet(101);
        names.add("Harry");
        names.add("Sue");
        names.add("Nina");
        names.add("Susannah");
        names.add("Larry");
        names.add("Eve");
        names.add("Sarah");
        names.add("Adam");
        names.add("Tony");
        names.add("Katherine");
        names.add("Juliet");
        names.add("Romeo");
        names.remove("Romeo");
        names.remove("George");
        Iterator iter = names.iterator();
        while (iter.hasNext())
        {
            System.out.println(iter.next());
        }
    }
}
Program Run

| Harry  |
| Sue    |
| Nina   |
| Susannah |
| Larry  |
| Eve    |
| Sarah  |
| Adam   |
| Juliet |
| Katherine |
| Tony   |

19. If a hash function returns 0 for all values, will the hash table work correctly?
20. If a hash table has size 1, will it work correctly?
21. Suppose you have two hash tables, each with n elements. To find the elements that are in both tables, you iterate over the first table, and for each element, check whether it is contained in the second table. What is the big-Oh efficiency of this algorithm?
22. In which order does the iterator visit the elements of the hash table?
23. What does the hasNext method of the HashSetIterator do when it has reached the end of a bucket?
24. Why doesn’t the iterator have an add method?

Practice It  Now you can try these exercises at the end of the chapter: E16.18, E16.20, E16.21.

Open Addressing

In the preceding sections, you studied a hash table implementation that uses separate chaining for collision handling, placing all elements with the same hash code in a bucket. This implementation is fast and easy to understand, but it requires storage for the links to the nodes. If one places the elements directly into the hash table, then one doesn’t need to store any links. This alternative technique is called open addressing. It can be beneficial if one must minimize the memory usage of a hash table.

Of course, open addressing makes collision handling more complicated. If you have two elements with (compressed) hash code h, and the first one is placed at index h, then the second must be placed in another location.

There are different techniques for placing colliding elements. The simplest is linear probing. If possible, place the colliding element at index \( h + 1 \). If that slot is occupied, try \( h + 2 \), \( h + 3 \), and so on, wrapping around to 0, 1, 2, and so on, if necessary. This sequence of index values is called the probing sequence. (You can see other probing sequences in Exercises P16.15 and P16.16.) If the probing sequence contains no empty slots, one must reallocate to a larger table.

How do we find an element in such a hash table? We compute the hash code and traverse the probing sequence until we either find a match or an empty slot. As long as the hash table is not too full, this is still an \( O(1) \) operation, but it may require more comparisons than with separate chaining. With separate chaining, we only compare objects with the same hash code. With open addressing, there may be some objects with different hash codes that happen to lie on the probing sequence.
Adding an element is similar. Try finding the element first. If it is not present, add it in the first empty slot in the probing sequence.

Removing an element is trickier. You cannot simply empty the slot at which you find the element. Instead, you must traverse the probing sequence, look for the last element with the same hash code, and move that element into the slot of the removed element (Exercise P16.14).

Alternatively, you can replace the removed element with a special “inactive” marker that, unlike an empty slot, does not indicate the end of a probing sequence. When adding another element, you can overwrite an inactive slot (Exercise P16.17).

**CHAPTER SUMMARY**

Describe the implementation and efficiency of linked list operations.

- A linked list object holds a reference to the first node object, and each node holds a reference to the next node.
- When adding or removing the first element, the reference to the first node must be updated.
- A list iterator object has a reference to the last visited node.
- To advance an iterator, update the position and remember the old position for the remove method.
- In a doubly-linked list, accessing an element is an \( O(n) \) operation; adding and removing an element is \( O(1) \).

Understand the implementation and efficiency of array list operations.

- Getting or setting an array list element is an \( O(1) \) operation.
- Inserting or removing an array list element is an \( O(n) \) operation.
- Adding or removing the last element in an array list takes amortized \( O(1) \) time.
Compare different implementations of stacks and queues.

- A stack can be implemented as a linked list, adding and removing elements at the front.
- When implementing a stack as an array list, add and remove elements at the back.
- A queue can be implemented as a linked list, adding elements at the back and removing them at the front.
- In a circular array implementation of a queue, element locations wrap from the end of the array to the beginning.

Understand the implementation of hash tables and the efficiencies of its operations.

- A good hash function minimizes collisions—identical hash codes for different objects.
- A hash table uses the hash code to determine where to store each element.
- A hash table can be implemented as an array of buckets—sequences of nodes that hold elements with the same hash code.
- If there are no or only a few collisions, then adding, locating, and removing hash table elements takes constant or $O(1)$ time.

** REVIEW EXERCISES **

- **R16.1** The linked list class in the Java library supports operations addLast and removeLast. To carry out these operations efficiently, the LinkedList class has an added reference last to the last node in the linked list. Draw a “before/after” diagram of the changes to the links in a linked list when the addLast method is executed.

- **R16.2** The linked list class in the Java library supports bidirectional iterators. To go backward efficiently, each Node has an added reference, previous, to the predecessor node in the linked list. Draw a “before/after” diagram of the changes to the links in a linked list when the addFirst and removeFirst methods execute. The diagram should show how the previous references need to be updated.

- **R16.3** What is the big-Oh efficiency of replacing all negative values in a linked list of Integer objects with zeroes? Of removing all negative values?

- **R16.4** What is the big-Oh efficiency of replacing all negative values in an array list of Integer objects with zeroes? Of removing all negative values?

- **R16.5** In the LinkedList implementation of Section 16.1, we use a flag isAfterNext to ensure that calls to the remove and set methods occur only when they are allowed. It is not actually necessary to introduce a new instance variable for this check. Instead, one can set the previous instance variable to a special value at the end of every call to add or remove. With that change, how should the remove and set methods check whether they are allowed?

- **R16.6** What is the big-Oh efficiency of the size method of Exercise E16.4?

- **R16.7** Show that the introduction of the size method in Exercise E16.4 does not affect the big-Oh efficiency of the other list operations.
**R16.8** Given the size method of Exercise E16.6 and the get method of Exercise P16.1, what is the big-Oh efficiency of this loop?

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

**R16.9** Given the size method of Exercise E16.6 and the get method of Exercise P16.3, what is the big-Oh efficiency of this loop?

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

**R16.10** It is not safe to remove the first element of a linked list with the removeFirst method when an iterator has just traversed the first element. Explain the problem by tracing the code and drawing a diagram.

**R16.11** Continue Exercise R16.10 by providing a code example demonstrating the problem.

**R16.12** It is not safe to simultaneously modify a linked list using two iterators. Find a situation where two iterators refer to the same linked list, and when you add an element with one iterator and remove an element with the other, the result is incorrect. Explain the problem by tracing the code and drawing a diagram.

**R16.13** Continue Exercise R16.12 by providing a code example demonstrating the problem.

**R16.14** In the implementation of the LinkedList class of the standard Java library, the problem described in Exercises R16.10 and R16.12 results in a ConcurrentModificationException. Describe how the LinkedList class and the iterator classes can discover that a list was modified through multiple sources. *Hint:* Count mutating operations. Where are the counts stored? Where are they updated? Where are they checked?

**R16.15** Consider the efficiency of locating the $k$th element in a doubly-linked list of length $n$. If $k > n / 2$, it is more efficient to start at the end of the list and move the iterator to the previous element. Why doesn’t this increase in efficiency improve the big-Oh estimate of element access in a doubly-linked list?

**R16.16** A linked list implementor, hoping to improve the speed of accessing elements, provides an array of Node references, pointing to every tenth node. Then the operation get($n$) looks up the reference at position $n - n \% 10$ and follows $n \% 10$ links.

- **a.** With this implementation, what is the efficiency of the get operation?
- **b.** What is the disadvantage of this implementation?

**R16.17** Suppose an array list implementation were to add ten elements at each reallocation instead of doubling the capacity. Show that the addLast operation no longer has amortized constant time.

**R16.18** Consider an array list implementation with a removeLast method that shrinks the internal array to half of its size when it is at most half full. Give a sequence of addLast and removeLast calls that does not have amortized $O(1)$ efficiency.

**R16.19** Suppose the ArrayList implementation of Section 16.2 had a removeLast method that shrinks the internal array by 50 percent when it is less than 25 percent full. Show that any sequence of addLast and removeLast calls has amortized $O(1)$ efficiency.

**R16.20** Given a queue with $O(1)$ methods add, remove, and size, what is the big-Oh efficiency of moving the element at the head of the queue to the tail? Of moving the element at the tail of the queue to the head? (The order of the other queue elements should be unchanged.)
R16.21 A deque (double-ended queue) is a data structure with operations addFirst, removeFirst, addLast, and removeLast. What is the $O(1)$ efficiency of these operations if the deque is implemented as

a. a singly-linked list?

b. a doubly-linked list?

c. a circular array?

R16.22 In our circular array implementation of a queue, can you compute the value of the currentSize from the values of the head and tail fields? Why or why not?

R16.23 Draw the contents of a circular array implementation of a queue $q$, with an initial array size of 10, after each of the following loops:

a. for (int $i = 1; i <= 5; i++$) { $q$.add($i$); }

b. for (int $i = 1; i <= 3; i++$) { $q$.remove(); }

c. for (int $i = 1; i <= 10; i++$) { $q$.add($i$); }

d. for (int $i = 1; i <= 8; i++$) { $q$.remove(); }

R16.24 Suppose you are stranded on a desert island on which stacks are plentiful, but you need a queue. How can you implement a queue using two stacks? What is the big-Oh running time of the queue operations?

R16.25 Suppose you are stranded on a desert island on which queues are plentiful, but you need a stack. How can you implement a stack using two queues? What is the big-Oh running time of the stack operations?

R16.26 Craig Coder doesn’t like the fact that he has to implement a hash function for the objects that he wants to collect in a hash table. “Why not assign a unique ID to each object?” he asks. What is wrong with his idea?

E16.1 Add a method reverse to our LinkedList implementation that reverses the links in a list. Implement this method by directly rerouting the links, not by using an iterator.

E16.2 Consider a version of the LinkedList class of Section 16.1 in which the addFirst method has been replaced with the following faulty version:

```java
public void addFirst(Object element) {
    Node newNode = new Node();
    first = newNode;
    newNode.data = element;
    newNode.next = first;
}
```

Develop a program ListTest with a test case that shows the error. That is, the program should print a failure message with this implementation but not with the correct implementation.

E16.3 Consider a version of the LinkedList class of Section 16.1 in which the iterator’s hasNext method has been replaced with the following faulty version:

```java
public boolean hasNext() { return position != null; }
```
Develop a program `ListTest` with a test case that shows the error. The program should print a failure message with this implementation but not with the correct one.

- **E16.4** Add a method `size` to our implementation of the `LinkedList` class that computes the number of elements in the list by following links and counting the elements until the end of the list is reached.

- **E16.5** Solve Exercise E16.4 recursively by calling a recursive helper method

  ```java
  private static int size(Node start)
  Hint: If start is null, then the size is 0. Otherwise, it is one larger than the size of start.next.
  ```

- **E16.6** Add an instance variable `currentSize` to our implementation of the `LinkedList` class. Modify the `add`, `addLast`, and `remove` methods of both the linked list and the list iterator to update the `currentSize` variable so that it always contains the correct size. Change the `size` method of Exercise E16.4 so that it simply returns the value of `currentSize`.

- **E16.7** Reimplement the `LinkedList` class of Section 16.1 so that the `Node` and `LinkedList-Iterator` classes are not inner classes.

- **E16.8** Reimplement the `LinkedList` class of Section 16.1 so that it implements the `java.util.LinkedList` interface. Hint: Extend the `java.util.AbstractList` class.

- **E16.9** Provide a `listIterator` method for the `ArrayList` implementation in Section 16.2. Your method should return an object of a class implementing `java.util.ListIterator`. Also have the `ArrayList` class implement the `Iterable` interface type and provide a test program that demonstrates that your array list can be used in an enhanced `for` loop.

- **E16.10** Provide a `removeLast` method for the `ArrayList` implementation in Section 16.2 that shrinks the internal array by 50 percent when it is less than 25 percent full.

- **E16.11** Complete the implementation of a stack in Section 16.3.2, using an array for storing the elements.

- **E16.12** Complete the implementation of a queue in Section 16.3.3, using a sequence of nodes for storing the elements.

- **E16.13** Add a method `firstToLast` to the implementation of a queue in Exercise E16.12. The method moves the element at the head of the queue to the tail of the queue. The element that was second in line will now be at the head.

- **E16.14** Add a method `lastToFirst` to the implementation of a queue in Exercise E16.12. The method moves the element at the tail of the queue to the head.

- **E16.15** Add a method `firstToLast`, as described in Exercise E16.13, to the circular array implementation of a queue.

- **E16.16** Add a method `lastToFirst`, as described in Exercise E16.14, to the circular array implementation of a queue.

- **E16.17** The `hasNext` method of the hash set implementation in Section 16.4 finds the location of the next element, but when `next` is called, the same search happens again. Improve the efficiency of these methods so that `next` (or a repeated call to `hasNext`) uses the position located by a preceding call to `hasNext`.

- **E16.18** Reallocate the buckets of the hash set implementation in Section 16.4 when the load factor is greater than 1.0 or less than 0.5, doubling or halving its size. Note that you need to recompute the hash values of all elements.
**E16.19** Implement the \texttt{remove} operation for iterators on the hash set in Section 16.4.

**E16.20** Implement the hash set in Section 16.4, using the “MAD (multiply-add-divide) method” for hash code compression. For that method, you choose a prime number \( p \) larger than the length \( L \) of the hash table and two values \( a \) and \( b \) between 1 and \( p - 1 \). Then reduce \( h \) to \(( (a \cdot h + b) \mod p) \mod L \).

**E16.21** Add methods to count collisions to the hash set in Section 16.4 and the one in Exercise E16.20. Insert all words from a dictionary (in /usr/share/dict/words or in words.txt in your companion code) into both hash set implementations. Does the MAD method reduce collisions? (Use a table size that equals the number of words in the file. Choose \( p \) to be the next prime greater than \( L \), \( a = 3 \), and \( b = 5 \).)

**P16.1** Add methods \texttt{Object get(int n)} and \texttt{void set(int n, Object newElement)} to the \texttt{LinkedList} class. Use a helper method that starts at \texttt{first} and follows \( n \) links:

\[
\texttt{private static Node getNode(int n)}
\]

**P16.2** Solve Exercise P16.1 by using a recursive helper method

\[
\texttt{private static Node getNode(Node start, int distance)}
\]

**P16.3** Improve the efficiency of the \texttt{get} and \texttt{set} methods of Exercise P16.1 by storing (or “caching”) the last known (node, index) pair. If \( n \) is larger than the last known index, start from the corresponding node instead of the front of the list. Be sure to discard the last known pair when it is no longer accurate. (This can happen when another method edits the list).

**P16.4** Add a method \texttt{boolean contains(Object obj)} that checks whether our \texttt{LinkedList} implementation contains a given object. Implement this method by directly traversing the links, not by using an iterator.

Use the \texttt{equals} method to determine whether \( \texttt{obj} \) equals \( \texttt{node.data} \) for a given node.

**P16.5** Solve Exercise P16.4 recursively, by calling a recursive helper method

\[
\texttt{private static boolean contains(Node start, Object obj)}
\]

\textit{Hint:} If \( \texttt{start} \) is \texttt{null}, then it can’t contain the object. Otherwise, check \( \texttt{start.data} \) before recursively moving on to \( \texttt{start.next} \).

**P16.6** A linked list class with an \( O(1) \) \texttt{addLast} method needs an efficient mechanism to get to the end of the list, for example by setting an instance variable to the last element. It is then possible to remove the reference to the first node if one makes the next reference of the last node point to the first node, so that all nodes form a cycle. Such an implementation is called a circular linked list. Turn the linked list implementation of Section 16.1 into a circular singly-linked list.

**P16.7** In a circular doubly-linked list, the \texttt{previous} reference of the first node points to the last node, and the \texttt{next} reference of the last node points to the first node. Change the doubly-linked list implementation of Worked Example 16.1 into a circular list. You should remove the \texttt{last} instance variable because you can reach the last element as \texttt{first.previous}.

**P16.8** Modify the insertion sort algorithm of Special Topic 14.2 to sort a linked list.
The LISP language, created in 1960, implements linked lists in a very elegant way. You will explore a Java analog in this set of exercises. Conceptually, the tail of a list—that is, the list with its head node removed—is also a list. The tail of that list is again a list, and so on, until you reach the empty list. Here is a Java interface for such a list:

```java
class LispList {
    public boolean empty();
    public Object head();
    public LispList tail();
    ...}
```

There are two kinds of lists, empty lists and nonempty lists:

```java
class EmptyList implements LispList {
    ...}
class NonEmptyList implements LispList {
    ...}
```

These classes are quite trivial. The `EmptyList` class has no instance variables. Its head and tail methods simply throw an `UnsupportedOperationException`, and its empty method returns `true`. The `NonEmptyList` class has instance variables for the head and tail.

Here is one way of making a LISP list with three elements:

```java
LispList list = new NonEmptyList("A", new NonEmptyList("B",
    new NonEmptyList("C", new EmptyList())));
```

This is a bit tedious, and it is a good idea to supply a convenience method `cons` that calls the constructor, as well as a static variable `NIL` that is an instance of an empty list. Then our list construction becomes

```java
LispList list = LispList.NIL.cons("C").cons("B").cons("A");
```

Note that you need to build up the list starting from the (empty) tail.

To see the elegance of this approach, consider the implementation of a `toString` method that produces a string containing all list elements. The method must be implemented by both classes:

```java
class EmptyList implements LispList {
    ...
    public String toString() { return "";
    }
}
class NonEmptyList implements LispList {
    ...
    public String toString() { return head() + " " + tail().toString(); }
}
```

Note that no `if` statement is required. A list is either empty or nonempty, and the correct `toString` method is invoked due to polymorphism.

In this exercise, complete the `LispList` interface and the `EmptyList` and `NonEmptyList` classes. Write a test program that constructs a list and prints it.

**P16.10** Add a method `length` to the `LispList` interface of Exercise P16.9 that returns the length of the list. Implement the method in the `EmptyList` and `NonEmptyList` classes.

**P16.11** Add a method

```java
LispList merge(LispList other)
```
to the LispList interface of Exercise P16.9. Implement the method in the EmptyList and NonEmptyList classes. When merging two lists, alternate between the elements, then add the remainder of the longer list. For example, merging the lists with elements 1 2 3 4 and 5 6 yields 1 5 2 6 3 4.

**P16.12** Add a method

```java
boolean contains(Object obj)
```

to the LispList interface of Exercise P16.9 that returns true if the list contains an element that equals obj.

**P16.13** A deque (double-ended queue) is a data structure with operations addFirst, removeFirst, addLast, removeLast, and size. Implement a deque as a circular array, so that these operations have amortized constant time.

**P16.14** Implement a hash table with open addressing. When removing an element that is followed by other elements with the same hash code, replace it with the last such element.

**P16.15** Modify Exercise P16.14 to use quadratic probing. The ith index in the probing sequence is computed as \((b + i^2) \mod L\).

**P16.16** Modify Exercise P16.14 to use double hashing. The ith index in the probing sequence is computed as \((b + i h_2(k)) \mod L\), where \(k\) is the original hash key before compression and \(h_2\) is a function mapping integers to non-zero values. A common choice is \(h_2(k) = 1 + k \mod q\) for a prime \(q\) less than \(L\).

**P16.17** Modify Exercise P16.14 so that you mark removed elements with an “inactive” element. You can’t use null—that is already used for empty elements. Instead, declare a static variable

```java
private static final Object INACTIVE = new Object();
```

Use the test if (table[i] == INACTIVE) to check whether a table entry is inactive.
last.next = new Node();
last.next.data = element;
}

6. \(O(1)\) and \(O(n)\).

7. To locate the middle element takes \(n/2\) steps. To locate the middle of the subinterval to the left or right takes another \(n/4\) steps. The next lookup takes \(n/8\) steps. Thus, we expect almost \(n\) steps to locate an element. At this point, you are better off just making a linear search that, on average, takes \(n/2\) steps.

8. In a linked list, one must follow \(k\) links to get to the \(k\)th elements. In an array list, one can reach the \(k\)th element directly as \(\text{elements}[k]\).

9. In a linked list, one merely updates references to the first and second node—a constant cost that is independent of the number of elements that follow. In an array list of size \(n\), inserting an element at the beginning requires us to move all \(n\) elements.

10. It is \(O(n)\) in both cases. In the case of the linked list, it costs \(O(n)\) steps to move an iterator to the middle.

11. It is still \(O(n)\). Reallocating the array is an \(O(n)\) operation, and moving the array elements also requires \(O(n)\) time.

12. \(O(1)+\). The cost of moving one element is \(O(1)\), but every so often one has to pay for a reallocation.

13. public Object peek()
    {
        if (first == null)
            throw new NoSuchElementException();
        return first.data;
    }

14. Removing an element from a singly-linked list is \(O(n)\).

15. Adding and removing an element at index 0 is \(O(n)\).

16. The queue can be empty when the head and tail are at a position other than zero. For example, after the calls \(q.add(obj)\) and \(q.remove()\), the queue is empty, but head and tail are 1.

17. Indeed, if the queue is empty, then the head and tail are equal. But that situation also occurs when the array is completely full.

18. Then the circular wrapping wouldn’t work. If we simply added new elements without reordering the existing ones, the new array layout would be.

19. Yes, the hash set will work correctly. All elements will be inserted into a single bucket.

20. Yes, but there will be a single bucket containing all elements. Finding, adding, and removing elements is \(O(n)\).

21. The iteration takes \(O(n)\) steps. Each step makes an \(O(1)\) containment check. Therefore, the total cost is \(O(n)\).

22. Elements are visited by increasing (compressed) hash code. This ordering will appear random to users of the hash table.

23. It locates the next bucket in the bucket array and points to its first element.

24. In a set, it doesn’t make sense to add an element at a specific position.
In a doubly-linked list, each node has a reference to the node preceding it, so we will add an instance variable `previous`:

```java
class Node {
    public Object data;
    public Node next;
    public Node previous;
}
```

We will also add a reference to the last node, which speeds up adding and removing elements at the end of the list:

```java
public class LinkedList {
    private Node first;
    private Node last;
    ...
}
```

We need to revisit all methods of the `LinkedList` and `ListIterator` classes to make sure that these instance variables are properly updated. We will also add methods to add, remove, and get the last element.

### Changes in the `LinkedList` Class

In the constructor, we simply add an initialization of the `last` instance variable:

```java
public LinkedList() {
    first = null;
    last = null;
}
```

The `getFirst` method is unchanged. However, in the `removeFirst` method, we need to update the `previous` reference of the node following the one that is being removed. Moreover, we need to take into account the possibility that the list contains a single element before removal. When that element is removed, then the `last` reference needs to be set to `null`:

```java
public Object removeFirst() {
    if (first == null) { throw new NoSuchElementException(); }
    Object element = first.data;
    first = first.next;
    if (first == null) { last = null; } // List is now empty
    else { first.previous = null; }
    return element;
}
```

In the `addFirst` method, we also need to update the `previous` reference of the node following the added node. Moreover, if the list was previously empty, the new node becomes both the first and the last node:

```java
public void addFirst(Object element) {
    ...
}
```
Node newNode = new Node();
newNode.data = element;
newNode.next = first;
newNode.previous = null;
if (first == null) { last = newNode; }
else { first.previous = newNode; }
first = newNode;
}

New Methods for Accessing the Last Element of the List

The getLast, removeLast, and addLast methods are the mirror opposites of the getFirst, removeFirst, and addFirst methods, where the roles of first/last and next/previous are switched.

private Object getLast()
{
    if (last == null) { throw new NoSuchElementException(); }
    return last.data;
}

public Object removeLast()
{
    if (last == null) { throw new NoSuchElementException(); }
    Object element = last.data;
    last = last.previous;
    if (last == null) { first = null; } // List is now empty
    else { last.next = null; }
    return element;
}

public void addLast(Object element)
{
    Node newNode = new Node();
    newNode.data = element;
    newNode.next = null;
    newNode.previous = last;
    if (last == null) { first = newNode; }
    else { last.next = newNode; }
    last = newNode;
}

Compare removeLast/addLast with the removeFirst/addFirst methods given above and pay attention to the first/last and next/previous references!

The Bidirectional Iterator

In the ListIterator class, we no longer need to store the previous reference because we can reach the preceding node as position.previous. We can simply remove it from the constructor and the next method. (Recall that this reference was required to support the iterator’s remove operation.)

In a doubly-linked list, the iterator can move forward and backward. For example,

LinkedList lst = new LinkedList();
lst.addLast("A");
lst.addLast("B");
lst.addLast("C");
ListIterator iter = lst.listIterator(); // The iterator is before the first element |ABC
iter.next(); // Returns “A”; the iterator is after the first element |ABC 1
iter.next(); // Returns “B”; the iterator is after the second element |BC
iter.previous(); // Returns “B”; the iterator is after the first element |AB 2

The `previous` method is similar to the `next` method. However, it returns the value after the iterator position. That is perhaps not so intuitive, and it is best to draw a diagram to verify the point. In the figure below, we show two calls to `next`, followed by a call to `previous`, as in the code example above. Recall that an iterator conceptually points between elements, like the cursor of a word processor, and that the position reference of the iterator points to the element to the left (or to `null` when it is at the beginning of the list).

As you can see, a call to `previous` moves the iterator backward, and the element that is returned is the one to which it pointed before being moved:

```java
public Object previous()
{
    if (!hasPrevious()) { throw new NoSuchElementException(); }
    isAfterNext = false;
    isAfterPrevious = true;

    Object result = position.data;
    position = position.previous;
    return result;
}
```

**Removing and Setting Elements Through an Iterator**

Note the `isAfterNext` and `isAfterPrevious` variables in the `previous` method. They track whether the iterator just carried out a `next` or `previous` call (or neither of the two). This information is needed for implementing the `remove` and `set` methods.

These methods remove or set the element that the iterator just traversed, which is position after a call to `next` or `position.next` after a call to `previous`. (If calling `previous` sets `position` to `null` because we reached the front of the list, then we remove or set `first`.) The following helper method computes this node:

```java
private Node lastPosition()
{
    if (isAfterNext)
    {
```
return position;
} else if (isAfterPrevious)
{
    if (position == null)
    {
        return first;
    } else
    {
        return position.next;
    }
} else { throw new IllegalStateException(); }
}

With this helper method, the set method is simple:

```
public void set(Object element)
{
    Node positionToSet = lastPosition();
    positionToSet.data = element;
}
```

The remove method also uses the lastPosition helper method. To ensure that the first and last references are properly updated, we have separate cases for removing the first or last element. Note that the iterator moves one step back when calling remove after next, and it stays at the same position when calling remove after previous.

```
public void remove()
{
    Node positionToRemove = lastPosition();
    if (positionToRemove == first)
    {
        removeFirst();
    } else if (positionToRemove == last)
    {
        removeLast();
    } else
    {
        positionToRemove.previous.next = positionToRemove.next;  // 1
        positionToRemove.next.previous = positionToRemove.previous;  // 2
    }
    if (isAfterNext)
    {
        position = position.previous;
    }
    isAfterNext = false;
    isAfterPrevious = false;
}
```

The most complex part of this method is the routing of the next and previous references around the removed elements, which is highlighted above. We know that positionToRemove.previous and positionToRemove.next are not null because we don’t remove the first or last element. The following figure shows how the references are updated.
Implementing a Doubly-Linked List

We provide a test method check for this purpose. For example,

```java
LinkedList lst = new LinkedList();
check("", lst, "Constructing empty list");
lst.addLast("A");
check("A", lst, "Adding last to empty list");
lst.addLast("B");
check("AB", lst, "Adding last to non-empty list");
```

The check method has three arguments: the expected contents (as a string—we assume each node contains a string of length 1), the list, and a string describing the test. The strings are used to print messages such as

Passed "Constructing empty list".
Passed "Adding last to empty list".
Passed "Adding last to non-empty list".

When implementing the check method, we use a helper method assertEquals that checks whether an expected value equals an actual one. If it doesn’t, an exception is thrown. For example,

```java
assertEquals(expected.substring(0, 1), actual.getFirst());
```

You can find the implementation of the check and assertEquals methods and the provided test cases in the LinkedListTest class at the end of this example.
public class LinkedList
{
    private Node first;
    private Node last;

    /**
     * Constructs an empty linked list.
     */
    public LinkedList()
    {
        first = null;
        last = null;
    }

    /**
     * Returns the first element in the linked list.
     * @return the first element in the linked list
     */
    public Object getFirst()
    {
        if (first == null) { throw new NoSuchElementException(); }
        return first.data;
    }

    /**
     * Removes the first element in the linked list.
     * @return the removed element
     */
    public Object removeFirst()
    {
        if (first == null) { throw new NoSuchElementException(); }
        Object element = first.data;
        first = first.next;
        if (first == null) { last = null; } // List is now empty
        else { first.previous = null; }
        return element;
    }

    /**
     * Adds an element to the front of the linked list.
     * @param element the element to add
     */
    public void addFirst(Object element)
    {
        Node newNode = new Node();
        newNode.data = element;
        newNode.next = first;
        newNode.previous = null;
        if (first == null) { last = newNode; }
        else { first.previous = newNode; }
        first = newNode;
    }

    /**
     * Returns the last element in the linked list.
     * @return the last element in the linked list
     */
    public Object getLast()
    {
        if (last == null) { throw new NoSuchElementException(); }
    }
```java
// Removes the last element in the linked list.
    @return the removed element
/*
    public Object removeLast()
    {
        if (last == null) { throw new NoSuchElementException(); }
        Object element = last.data;
        last = last.previous;
        if (last == null) { first = null; } // List is now empty
        else { last.next = null; }
        return element;
    }

    /**
     * Adds an element to the back of the linked list.
     * @param element the element to add
     */
    public void addLast(Object element)
    {
        Node newNode = new Node();
        newNode.data = element;
        newNode.next = null;
        newNode.previous = last;
        if (last == null) { first = newNode; }
        else { last.next = newNode; }
        last = newNode;
    }

    /**
     * Returns an iterator for iterating through this list.
     * @return an iterator for iterating through this list
     */
    public ListIterator listIterator()
    {
        return new LinkedListIterator();
    }

    class Node
    {
        public Object data;
        public Node next;
        public Node previous;
    }

    class LinkedListIterator implements ListIterator
    {
        private Node position;
        private boolean isAfterNext;
        private boolean isAfterPrevious;

        /**
         * Constructs an iterator that points to the front
         * of the linked list.
         */
        public LinkedListIterator()
        {
        }
```
126  position = null;
127  isAfterNext = false;
128  isAfterPrevious = false;
129  }
130
131  /**
132   * Moves the iterator past the next element.
133   * @return the traversed element
134  */
135  public Object next()
136  {
137      if (!hasNext()) { throw new NoSuchElementException(); }
138      isAfterNext = true;
139      if (position == null)
140          position = first;
141      else
142          position = position.next;
143      return position.data;
144  }
145
146  /**
147   * Tests if there is an element after the iterator position.
148   * @return true if there is an element after the iterator position
149  */
150  public boolean hasNext()
151  {
152      if (position == null)
153          return first != null;
154      else
155          return position.next != null;
156  }
157
158  /**
159   * Moves the iterator before the previous element.
160   * @return the traversed element
161  */
162  public Object previous()
163  {
164      if (!hasPrevious()) { throw new NoSuchElementException(); }
165      isAfterNext = false;
166      isAfterPrevious = true;
167      Object result = position.data;
168      position = position.previous;
169      return result;
170  }
public boolean hasPrevious()
{
    return position != null;
}

/**
   Tests if there is an element before the iterator position.
   @return true if there is an element before the iterator position
*/

public void add(Object element)
{
    if (position == null)
    {
        addFirst(element);
        position = first;
    }
    else if (position == last)
    {
        addLast(element);
        position = last;
    }
    else
    {
        Node newNode = new Node();
        newNode.data = element;
        newNode.next = position.next;
        newNode.next.previous = newNode;
        position.next = newNode;
        newNode.previous = position;
        position = newNode;
    }

    isAfterNext = false;
    isAfterPrevious = false;
}

/**
   Adds an element before the iterator position and moves the iterator past the inserted element.
   @param element the element to add
*/

public void remove()
{
    Node positionToRemove = lastPosition();

    if (positionToRemove == first)
    {
        removeFirst();
    }
    else if (positionToRemove == last)
    {
        removeLast();
    }
    else
    {
```java
240 } else {
241     positionToRemove.previous.next = positionToRemove.next;
242     positionToRemove.next.previous = positionToRemove.previous;
243 }
244
245 if (isAfterNext) {
246     position = position.previous;
247 }
248
249 isAfterNext = false;
250 isAfterPrevious = false;
251 }
252
253 /**
254  * Sets the last traversed element to a different value.
255  * @param element the element to set
256  */
257 public void set(Object element) {
258     Node positionToSet = lastPosition();
259     positionToSet.data = element;
260 }
261
262 /**
263  * Returns the last node traversed by this iterator, or
264  * throws an IllegalStateException if there wasn’t an immediately
265  * preceding call to next or previous.
266  * @return the last traversed node
267  */
268 private Node lastPosition() {
269     if (isAfterNext) {
270         return position;
271     }
272     else if (isAfterPrevious) {
273         if (position == null) {
274             return first;
275         }
276         else {
277             return position.next;
278         }
279     }
280     else { throw new IllegalStateException(); }
281 }
282 }
283 ```
import java.util.NoSuchElementException;

public class LinkedListTest
{
    public static void main(String[] args)
    {
        LinkedList lst = new LinkedList();
        check("", lst, "Constructing empty list");
        lst.addLast("A");
        check("A", lst, "Adding last to empty list");
        lst.addLast("B");
        check("AB", lst, "Adding last to non-empty list");

        lst = new LinkedList();
        lst.addFirst("A");
        check("A", lst, "Adding first to empty list");
        lst.addFirst("B");
        check("BA", lst, "Adding first to non-empty list");

        assertEquals("B", lst.removeFirst());
        check("A", lst, "Removing first, yielding non-empty list");
        assertEquals("A", lst.removeFirst());
        check("", lst, "Removing first, yielding empty list");

        lst = new LinkedList();
        lst.addLast("A");
        lst.addLast("B");
        check("AB", lst, "");

        assertEquals("B", lst.removeLast());
        check("A", lst, "Removing last, yielding non-empty list");
        assertEquals("A", lst.removeLast());
        check("", lst, "Removing last, yielding empty list");

        lst = new LinkedList();
        lst.addLast("A");
        lst.addLast("B");
        lst.addLast("C");
        check("ABC", lst, "");

        ListIterator iter = lst.listIterator();
        assertEquals("A", iter.next());
        iter.set("D");
        check("DBC", lst, "Set element after next");
        assertEquals("D", iter.previous());
        iter.set("E");
        check("EBC", lst, "Set first element after previous");
        assertEquals("E", iter.next());
        assertEquals("B", iter.next());
        assertEquals("B", iter.previous());
        iter.set("F");
        check("EFC", lst, "Set second element after previous");
        assertEquals("F", iter.next());
        assertEquals("C", iter.next());
        assertEquals("C", iter.previous());
    }
}

/*
 This program tests the doubly-linked list implementation.
 */

public class LinkedListTest
{...
iter.set("C");
check("EFG", lst, "Set last element after previous");

lst = new LinkedList();
lst.addLast("A");
lst.addLast("B");
lst.addLast("C");
lst.addLast("D");
lst.addLast("E");
check("ABCDE", lst, "");
iter = lst.listIterator();
assertEquals("A", iter.next());
iter.remove();
check("BCDE", lst, "Remove first element after next");
assertEquals("B", iter.next());
assertEquals("C", iter.next());
iter.remove();
check("BDE", lst, "Remove middle element after next");
assertEquals("D", iter.next());
assertEquals("E", iter.next());
iter.remove();
check("BD", lst, "Remove last element after next");

lst = new LinkedList();
lst.addLast("A");
lst.addLast("B");
lst.addLast("C");
lst.addLast("D");
lst.addLast("E");
check("ABCDE", lst, "");
iter = lst.listIterator();
assertEquals("A", iter.next());
assertEquals("B", iter.next());
assertEquals("C", iter.next());
assertEquals("D", iter.next());
assertEquals("E", iter.next());
assertEquals("E", iter.previous());
iter.remove();
check("ABCD", lst, "Remove last element after previous");
assertEquals("D", iter.previous());
assertEquals("C", iter.previous());
iter.remove();
check("ABD", lst, "Remove middle element after previous");
assertEquals("B", iter.previous());
assertEquals("A", iter.previous());
iter.remove();
check("BD", lst, "Remove first element after previous");

lst = new LinkedList();
lst.addLast("B");
lst.addLast("C");
check("BC", lst, "");
iter = lst.listIterator();
iter.add("A");
check("ABC", lst, "Add first element");
assertEquals("B", iter.next());
iter.add("D");
check("ABDC", lst, "Add middle element");
assertEquals("C", iter.next());
Implementing a Doubly-Linked List

```java
iter.add("E");
check("ABDCE", lst, "Add last element");

/**
 * Checks whether two objects are equal and throws an exception if not.
 * @param expected the expected value
 * @param actual the actual value
 */
public static void assertEquals(Object expected, Object actual)
{
    if (expected == null && actual != null ||
        !expected.equals(actual))
    {
        throw new AssertionError("Expected " + expected + " but found " + actual);
    }
}

/**
 * Checks whether a linked list has the expected contents, and throws an exception if not.
 * @param expected the letters that are expected in each node
 * @param actual the linked list
 * @param what a string explaining what has been tested. It is included in the message that is displayed when the test passes.
 */
public static void check(String expected, LinkedList actual, String what)
{
    int n = expected.length();
    if (n > 0)
    {
        // Check first and last references
        assertEquals(expected.substring(0, 1), actual.getFirst());
        assertEquals(expected.substring(n - 1), actual.getLast());

        // Check next references
        ListIterator iter = actual.listIterator();
        for (int i = 0; i < n; i++)
        {
            assertEquals(true, iter.hasNext());
            assertEquals(expected.substring(i, i + 1), iter.next());
        }
        assertEquals(false, iter.hasNext());

        // Check previous references
        for (int i = n - 1; i > 0; i--)
        {
            assertEquals(true, iter.hasPrevious());
            assertEquals(expected.substring(i, i + 1), iter.previous());
        }
        assertEquals(false, iter.hasPrevious());
    }
    else
    {
        // Check that first and last are null
        try
        {
            actual.getFirst();
        }
        // ...
    }
}
```
throw new IllegalStateException("first not null");

try {
    actual.getLast();
    throw new IllegalStateException("last not null");
} catch (NoSuchElementException ex) {
}

if (what.length() > 0) {
    System.out.println("Passed \\
" + what + \\
".");
}
To study trees and binary trees
To understand how binary search trees can implement sets
To learn how red-black trees provide performance guarantees for set operations
To choose appropriate methods for tree traversal
To become familiar with the heap data structure
To use heaps for implementing priority queues and for sorting

CHAPTER CONTENTS

17.1 BASIC TREE CONCEPTS 766
17.2 BINARY TREES 770
WE1 Building a Huffman Tree
17.3 BINARY SEARCH TREES 775
17.4 TREE TRAVERSAL 784
17.5 RED-BLACK TREES 790
WE2 Implementing a Red-Black Tree
17.6 HEAPS 797
17.7 THE HEAPSORT ALGORITHM 808
In this chapter, we study data structures that organize elements hierarchically, creating arrangements that resemble trees. These data structures offer better performance for adding, removing, and finding elements than the linear structures you have seen so far. You will learn about commonly used tree-shaped structures and study their implementation and performance.

### 17.1 Basic Tree Concepts

In computer science, a tree is a hierarchical data structure composed of **nodes**. Each node has a sequence of **child nodes**, and one of the nodes is the **root node**.

Like a linked list, a tree is composed of nodes, but with a key difference. In a linked list, a node can have only one child node, so the data structure is a linear chain of nodes. In a tree, a node can have more than one child. The resulting shape resembles an actual tree with branches. However, in computer science, it is customary to draw trees upside-down, with the root on top (see Figure 1).

**Figure 1**  A Family Tree

A family tree shows the descendants of a common ancestor.

A tree is composed of nodes, each of which can have child nodes.

The root is the node with no parent. A leaf is a node with no children.
Trees are commonly used to represent hierarchical relationships. When we talk about nodes in a tree, it is customary to use intuitive words such as roots and leaves, but also parents, children, and siblings—see Table 1 for commonly used terms.

<table>
<thead>
<tr>
<th>Term</th>
<th>Definition</th>
<th>Example (using Figure 1)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Node</td>
<td>The building block of a tree: A tree is composed of linked nodes.</td>
<td>This tree has 26 nodes: George V, Edward VIII, ..., Savannah.</td>
</tr>
<tr>
<td>Child</td>
<td>Each node has, by definition, a sequence of links to other nodes called its child nodes.</td>
<td>The children of Elizabeth II are Charles, Anne, Andrew, and Edward.</td>
</tr>
<tr>
<td>Leaf</td>
<td>A node with no child nodes.</td>
<td>This tree has 16 leaves, including William, Harry, and Savannah.</td>
</tr>
<tr>
<td>Interior node</td>
<td>A node that is not a leaf.</td>
<td>George V or George VI, but not Mary.</td>
</tr>
<tr>
<td>Parent</td>
<td>If the node (c) is a child of the node (p), then (p) is a parent of (c).</td>
<td>Elizabeth II is the parent of Charles.</td>
</tr>
<tr>
<td>Sibling</td>
<td>If the node (p) has children (c) and (d), then these nodes are siblings.</td>
<td>Charles and Anne are siblings.</td>
</tr>
<tr>
<td>Root</td>
<td>The node with no parent. By definition, each tree has one root node.</td>
<td>George V.</td>
</tr>
<tr>
<td>Path</td>
<td>A sequence of nodes (c_1, c_2, ..., c_k), where (c_{i+1}) is a child of (c_i).</td>
<td>Elizabeth II, Anne, Peter, Savannah is a path of length 4.</td>
</tr>
<tr>
<td>Descendant</td>
<td>(d) is a descendant of (c) if there is a path from (c) to (d).</td>
<td>Peter is a descendant of Elizabeth II but not of Henry.</td>
</tr>
<tr>
<td>Ancestor</td>
<td>(c) is an ancestor of (d) if (d) is a descendant of (c).</td>
<td>Elizabeth II is an ancestor of Peter, but Henry is not.</td>
</tr>
<tr>
<td>Subtree</td>
<td>The subtree rooted at node (n) is the tree formed by taking (n) as the root node and including all its descendants.</td>
<td>The subtree with root Anne is</td>
</tr>
<tr>
<td></td>
<td></td>
<td><img src="image" alt="Subtree Diagram" /></td>
</tr>
<tr>
<td>Height</td>
<td>The number of nodes in the longest path from the root to a leaf. (Some authors define the height to be the number of edges in the longest path, which is one less than the height used in this book.)</td>
<td>This tree has height 6. The longest path is George V, George VI, Elizabeth II, Anne, Peter, Savannah.</td>
</tr>
</tbody>
</table>
Trees have many applications in computer science; see for example Figures 2 and 3. There are multiple ways of implementing a tree. Here we present an outline of a simple implementation that is further explored in Exercises P17.1 and P17.2. A node holds a data item and a list of references to the child nodes. A tree holds a reference to the root node.

```java
public class Tree {
    private Node root;

    class Node {
        public Object data;
        public List<Node> children;
    }

    public Tree(Object rootData) {
        root = new Node();
        root.data = rootData;
        root.children = new ArrayList<>();
    }

    public void addSubtree(Tree subtree) {
        root.children.add(subtree.root);
    }
}
```

A tree class uses a node class to represent nodes and has an instance variable for the root node.
Note that, as with linked lists, the Node class is nested inside the Tree class. It is considered an implementation detail. Users of the class only work with Tree objects.

When computing properties of trees, it is often convenient to use recursion. For example, consider the task of computing the tree size, that is, the number of nodes in the tree. Compute the sizes of its subtrees, add them up, and add one for the root.

For example, in Figure 1, the tree with root node Elizabeth II has four subtrees, with node counts 3, 4, 3, and 3, yielding a count of $1 + 3 + 4 + 3 + 3 = 14$ for that tree.

Formally, if $r$ is the root node of a tree, then

$$\text{size}(r) = 1 + \text{size}(c_1) + \cdots + \text{size}(c_k),$$

where $c_1, \ldots, c_k$ are the children of $r$.

To implement this size method, first provide a recursive helper:

```java
class Node {
    // ...

    public int size() {
        int sum = 0;
        for (Node child : children) { sum = sum + child.size(); }
        return 1 + sum;
    }
}
```

Then call this helper method from a method of the Tree class:

```java
class Tree {
    // ...

    public int size() { return root.size(); }
}
```

It is useful to allow an empty tree; a tree whose root node is null. This is analogous to an empty list—a list with no elements. Because we can’t invoke the helper method on a null reference, we need to refine the Tree class’s size method:

```java
public int size() {
    if (root == null) { return 0; }
    else { return root.size(); }
}
```

1. What are the paths starting with Anne in the tree shown in Figure 1?
2. What are the roots of the subtrees consisting of three nodes in the tree shown in Figure 1?
3. What is the height of the subtree with root Anne?
4. What are all possible shapes of trees of height 3 with two leaves?
5. Describe a recursive algorithm for counting all leaves in a tree.

6. Using the public interface of the Tree class in this section, construct a tree that is identical to the subtree with root Anne in Figure 1.

7. Is the size method of the Tree class recursive? Why or why not?

Practice It Now you can try these exercises at the end of the chapter: R17.1, R17.2, E17.1.

17.2 Binary Trees

In the following sections, we discuss binary trees, trees in which each node has at most two children. As you will see throughout this chapter, binary trees have many very important applications.

17.2.1 Binary Tree Examples

In this section, you will see several typical examples of binary trees. Figure 4 shows a decision tree for guessing an animal from one of several choices. Each non-leaf node contains a question. The left subtree corresponds to a “yes” answer, and the right subtree to a “no” answer.

This is a binary tree because every node has either two children (if it is a decision) or no children (if it is a conclusion). Exercises E17.4 and P17.7 show you how you can build decision trees that ask good questions for a particular data set.

Figure 4 A Decision Tree for an Animal Guessing Game
Another example of a binary tree is a Huffman tree. In a Huffman tree, the leaves contain symbols that we want to encode. To encode a particular symbol, walk along the path from the root to the leaf containing the symbol, and produce a zero for every left turn and a one for every right turn. For example, in the Huffman tree of Figure 5, an H is encoded as 0001 and an A as 10. Worked Example 17.1 shows how to build a Huffman tree that gives the shortest codes for the most frequent symbols.

Binary trees are also used to show the evaluation order in arithmetic expressions. For example, Figure 6 shows the trees for the expressions

\[
(3 + 4) \times 5 \\
3 + 4 \times 5
\]

The leaves of the expression trees contain numbers, and the interior nodes contain the operators. Because each operator has two operands, the tree is binary.
17.2.2 Balanced Trees

When we use binary trees to store data, as we will in Section 17.3, we would like to have trees that are balanced. In a balanced tree, all paths from the root to one of the leaf nodes have approximately the same length. Figure 7 shows examples of a balanced and an unbalanced tree.

Recall that the height of a tree is the number of nodes in the longest path from the root to a leaf. The trees in Figure 7 have height 5. As you can see, for a given height, a balanced tree can hold more nodes than an unbalanced tree.

We care about the height of a tree because many tree operations proceed along a path from the root to a leaf, and their efficiency is better expressed by the height of the tree than the number of elements in the tree.

A binary tree of height $h$ can have up to $n = 2^h - 1$ nodes. For example, a completely filled binary tree of height 4 has $1 + 2 + 4 + 8 = 15 = 2^4 - 1$ nodes (see Figure 8).

In other words, $h = \log_2(n + 1)$ for a completely filled binary tree. For a balanced tree, we still have $h \approx \log_2 n$. For example, the height of a balanced binary tree with 1,000 nodes is approximately 10 (because $1000 \approx 1024 = 2^{10}$). A balanced binary tree with 1,000,000 nodes has a height of approximately 20 (because $10^6 \approx 2^{20}$). As you will see in Section 17.3, you can find any element in such a tree in about 20 steps. That is a lot faster than traversing the 1,000,000 elements of a list.

![Balanced and Unbalanced Trees](image)

*Figure 7  Balanced and Unbalanced Trees*
17.2.3 A Binary Tree Implementation

Every node in a binary tree has references to two children, a left child and a right child. Either one may be null. A node in which both children are null is a leaf.

A binary tree can be implemented in Java as follows:

```java
public class BinaryTree {
    private Node root;

    public BinaryTree() { root = null; } // An empty tree

    public BinaryTree(Object rootData, BinaryTree left, BinaryTree right) {
        root = new Node();
        root.data = rootData;
        root.left = left.root;
        root.right = right.root;
    }

    class Node {
        public Object data;
        public Node left;
        public Node right;
    }
    ...
}
```

As with general trees, we often use recursion to define operations on binary trees. Consider computing the height of a tree; that is, the number of nodes in the longest path from the root to a leaf.

To get the height of the tree `t`, take the larger of the heights of the children and add one, to account for the root.

\[
\text{height}(t) = 1 + \max(\text{height}(l), \text{height}(r))
\]

where `l` and `r` are the left and right subtrees.
When we implement this method, we could add a `height` method to the `Node` class. However, nodes can be `null` and you can’t call a method on a `null` reference. It is easier to make the recursive helper method a static method of the `Tree` class, like this:

```java
public class BinaryTree {
    // ... 
    private static int height(Node n) {
        if (n == null) { return 0; }
        else { return 1 + Math.max(height(n.left), height(n.right)); }
    }
    // ...
}
```

To get the height of the tree, we provide this public method:

```java
public class BinaryTree {
    // ... 
    public int height() { return height(root); }
}
```

Note that there are two `height` methods: a public method with no arguments, returning the height of the tree, and a private recursive helper method, returning the height of a subtree with a given node as its root.

8. Encode ALOHA, using the Huffman code in Figure 5.
9. In an expression tree, where is the operator stored that gets executed last?
10. What is the expression tree for the expression $3 - 4 - 5$?
11. How many leaves do the binary trees in Figure 4, Figure 5, and Figure 6 have? How many interior nodes?
12. Show how the recursive `height` helper method can be implemented as an instance method of the `Node` class. What is the disadvantage of that approach?

Practice It Now you can try these exercises at the end of the chapter: R17.4, E17.2, E17.3, E17.4.
17.3 Binary Search Trees

A set implementation is allowed to rearrange its elements in any way it chooses so that it can find elements quickly. Suppose a set implementation sorts its entries. Then it can use **binary search** to locate elements quickly. Binary search takes $O(\log(n))$ steps, where $n$ is the size of the set. For example, binary search in an array of 1,000 elements is able to locate an element in at most 10 steps by cutting the size of the search interval in half in each step.

If we use an array to store the elements of a set, inserting or removing an element is an $O(n)$ operation. In the following sections, you will see how tree-shaped data structures can keep elements in sorted order with more efficient insertion and removal.

### 17.3.1 The Binary Search Property

A **binary search tree** is a binary tree in which all nodes fulfill the following property:
- The data values of all descendants to the left are less than the data value stored in the node, and all descendants to the right have greater data values.

![Figure 9 A Binary Search Tree](image)

The tree in Figure 9 is a binary search tree.

We can verify the binary search property for each node in Figure 9. Consider the node “Juliet”. All descendants to the left have data before “Juliet”. All descendants to the right have data after “Juliet”. Move on to “Eve”. There is a single descendant to the left, with data “Adam” before “Eve”, and a single descendant to the right, with data “Harry” after “Eve”. Check the remaining nodes in the same way.

Figure 10 shows a binary tree that is not a binary search tree. Look carefully — the root node passes the test, but its two children do not.
Chapter 17  Tree Structures

When you implement binary search tree classes, the data variable should have type Comparable, not Object. After all, you must be able to compare the values in a binary search tree in order to place them into the correct position.

```java
public class BinarySearchTree {
    private Node root;

    public BinarySearchTree() { ... }
    public void add(Comparable obj) { ... }
    ...
    class Node {
        public Comparable data;
        public Node left;
        public Node right;

        public void addNode(Node newNode) { ... }
        ...
    }
}
```

17.3.2 Insertion

To insert data into the tree, use the following algorithm:

- If you encounter a non-null node reference, look at its data value. If the data value of that node is larger than the value you want to insert, continue the process with the left child. If the node’s data value is smaller than the one you want to insert, continue the process with the right child. If the node’s data value is the same as the one you want to insert, you are done, because a set does not store duplicate values.
- If you encounter a null node reference, replace it with the new node.

For example, consider the tree in Figure 11. It is the result of the following statements:

```java
BinarySearchTree tree = new BinarySearchTree();
tree.add("Juliet");  // 1
tree.add("Tom");    // 2
tree.add("Diana");  // 3
tree.add("Harry");  // 4
```
We want to insert a new element Romeo into it:

```java
    tree.add("Romeo");  
```

Start with the root node, Juliet. Romeo comes after Juliet, so you move to the right subtree. You encounter the node Tom. Romeo comes before Tom, so you move to the left subtree. But there is no left subtree. Hence, you insert a new Romeo node as the left child of Tom (see Figure 12).

You should convince yourself that the resulting tree is still a binary search tree. When Romeo is inserted, it must end up as a right descendant of Juliet—that is what the binary search tree condition means for the root node Juliet. The root node doesn’t care where in the right subtree the new node ends up. Moving along to Tom, the right child of Juliet, all it cares about is that the new node Romeo ends up somewhere on its left. There is nothing to its left, so Romeo becomes the new left child, and the resulting tree is again a binary search tree.

Here is the code for the add method of the BinarySearchTree class:

```java
public void add(Comparable obj) 
{
    Node newNode = new Node();
    newNode.data = obj;
    newNode.left = null;
    newNode.right = null;
    if (root == null) { root = newNode; }
    else { root.addNode(newNode); }
}
```

**Figure 11** Binary Search Tree After Four Insertions

**Figure 12** Binary Search Tree After Five Insertions
If the tree is empty, simply set its root to the new node. Otherwise, you know that
the new node must be inserted somewhere within the nodes, and you can ask the root
node to perform the insertion. That node object calls the `addNode` method of the `Node`
class, which checks whether the new object is less than the object stored in the node.
If so, the element is inserted in the left subtree; if not, it is inserted in the right subtree:

```java
class Node {
    ...  
    public void addNode(Node newNode) {
        int comp = newNode.data.compareTo(data);
        if (comp < 0) {
            if (left == null) { left = newNode; }
            else { left.addNode(newNode); }
        }
        else if (comp > 0) {
            if (right == null) { right = newNode; }
            else { right.addNode(newNode); }
        }
    }
    ...  
}
```

Let’s trace the calls to `addNode` when inserting Romeo into the tree in Figure 11. The first
call to `addNode` is

```
root.addNode(newNode)
```

Because `root` points to Juliet, you compare Juliet with Romeo and find that you must
call

```
root.right.addNode(newNode)
```

The node `root.right` is Tom. Compare the data values again (Tom vs. Romeo) and find that
you must now move to the left. Because `root.right.left` is null, set `root.right.left` to
`newNode`, and the insertion is complete (see Figure 12).

Unlike a linked list or an array, and like a hash table, a binary tree has no insert posi-
tions. You cannot select the position where you would like to insert an element into a
binary search tree. The data structure is self-organizing; that is, each element finds its
own place.

## 17.3.3 Removal

We will now discuss the removal algorithm. Our task is to remove a node from the
tree. Of course, we must first find the node to be removed. That is a simple matter,
due to the characteristic property of a binary search tree. Compare the data value to
be removed with the data value that is stored in the root node. If it is smaller, keep
looking in the left subtree. Otherwise, keep looking in the right subtree.

Let us now assume that we have located the node that needs to be removed. First,
let us consider the easiest case. If the node to be removed has no children at all, then
the parent link is simply set to `null` (Figure 13).

When the node to be removed has only one child, the situation is still simple (see
Figure 14).
To remove the node, simply modify the parent link that points to the node so that it points to the child instead.

The case in which the node to be removed has two children is more challenging. Rather than removing the node, it is easier to replace its data value with the next larger value in the tree. That replacement preserves the binary search tree property. (Alternatively, you could use the largest element of the left subtree—see Exercise P17.5).

To locate the next larger value, go to the right subtree and find its smallest data value. Keep following the left child links. Once you reach a node that has no left child, you have found the node containing the smallest data value of the subtree. Now remove that node—it is easily removed because it has at most one child to the right. Then store its data value in the original node that was slated for removal. Figure 15 shows the details.
At the end of this section, you will find the source code for the BinarySearchTree class. It contains the add and remove methods that we just described, a find method that tests whether a value is present in a binary search tree, and a print method that we will analyze in Section 17.4.

17.3.4 Efficiency of the Operations

Now that you have seen the implementation of this data structure, you may well wonder whether it is any good. Like nodes in a list, the nodes are allocated one at a time. No existing elements need to be moved when a new element is inserted or removed; that is an advantage. How fast insertion and removal are, however, depends on the shape of the tree. These operations are fast if the tree is balanced.

Because the operations of finding, adding, and removing an element process the nodes along a path from the root to a leaf, their execution time is proportional to the height of the tree, and not to the total number of nodes in the tree.

For a balanced tree, we have $h = O(\log(n))$. Therefore, inserting, finding, or removing an element is an $O(\log(n))$ operation. On the other hand, if the tree happens to be unbalanced, then binary tree operations can be slow—in the worst case, as slow as insertion into a linked list. Table 2 summarizes these observations.

If elements are added in fairly random order, the resulting tree is likely to be well balanced. However, if the incoming elements happen to be in sorted order already, then the resulting tree is completely unbalanced. Each new element is inserted at the end, and the entire tree must be traversed every time to find that end!

Binary search trees work well for random data, but if you suspect that the data in your application might be sorted or have long runs of sorted data, you should not use a binary search tree. There are more sophisticated tree structures whose methods keep trees balanced at all times. In these tree structures, one can guarantee that finding, adding, and removing elements takes $O(\log(n))$ time. The standard Java library uses red-black trees, a special form of balanced binary trees, to implement sets and maps. We discuss these structures in Section 17.5.

<table>
<thead>
<tr>
<th>Table 2 Efficiency of Binary Search Tree Operations</th>
</tr>
</thead>
<tbody>
<tr>
<td>Operation</td>
</tr>
<tr>
<td>-----------</td>
</tr>
<tr>
<td>Find an element.</td>
</tr>
<tr>
<td>Add an element.</td>
</tr>
<tr>
<td>Remove an element.</td>
</tr>
</tbody>
</table>

**section_3/BinarySearchTree.java**

```java
/**
 * This class implements a binary search tree whose nodes hold objects that implement the Comparable interface.
 */
```
public class BinarySearchTree
{
  private Node root;

  /**
   * Constructs an empty tree.
   */
  public BinarySearchTree()
  {
    root = null;
  }

  /**
   * Inserts a new node into the tree.
   * @param obj the object to insert
   */
  public void add(Comparable obj)
  {
    Node newNode = new Node();
    newNode.data = obj;
    newNode.left = null;
    newNode.right = null;
    if (root == null) { root = newNode; }
    else { root.addNode(newNode); }
  }

  /**
   * Tries to find an object in the tree.
   * @param obj the object to find
   * @return true if the object is contained in the tree
   */
  public boolean find(Comparable obj)
  {
    Node current = root;
    while (current != null)
    {
      int d = current.data.compareTo(obj);
      if (d == 0) { return true; }
      else if (d > 0) { current = current.left; }
      else { current = current.right; }
    }
    return false;
  }

  /**
   * Tries to remove an object from the tree. Does nothing
   * if the object is not contained in the tree.
   * @param obj the object to remove
   */
  public void remove(Comparable obj)
  {
    // Find node to be removed

    Node toBeRemoved = root;
    Node parent = null;
    boolean found = false;
    while (!found && toBeRemoved != null)
    {
      int d = toBeRemoved.data.compareTo(obj);
      if (d == 0) { found = true; }
    }
  }
}
else
{
    parent = toBeRemoved;
    if (d > 0) { toBeRemoved = toBeRemoved.left; } 
    else { toBeRemoved = toBeRemoved.right; } 
}

if (!found) { return; }

// toBeRemoved contains obj

// If one of the children is empty, use the other
if (toBeRemoved.left == null || toBeRemoved.right == null)
{
    Node newChild;
    if (toBeRemoved.left == null)
    {
        newChild = toBeRemoved.right;
    }
    else
    {
        newChild = toBeRemoved.left;
    }

    if (parent == null) // Found in root 
    {
        root = newChild;
    }
    else if (parent.left == toBeRemoved)
    {
        parent.left = newChild;
    }
    else
    {
        parent.right = newChild;
    }

    return;
}

// Neither subtree is empty

// Find smallest element of the right subtree
Node smallestParent = toBeRemoved;
Node smallest = toBeRemoved.right;
while (smallest.left != null)
{
    smallestParent = smallest;
    smallest = smallest.left;
}

// smallest contains smallest child in right subtree

// Move contents, unlink child
toBeRemoved.data = smallest.data;
if (smallestParent == toBeRemoved)
{
smallestParent.right = smallest.right;
}
else {
    smallestParent.left = smallest.right;
}
}

/**
 * Prints the contents of the tree in sorted order.
 */
public void print()
{
    print(root);
    System.out.println();
}

/**
 * Prints a node and all of its descendants in sorted order.
 * @param parent the root of the subtree to print
 */
private static void print(Node parent)
{
    if (parent == null) { return; }
    print(parent.left);
    System.out.print(parent.data + " ");
    print(parent.right);
}

/**
 * A node of a tree stores a data item and references
 * to the left and right child nodes.
 */
class Node
{
    public Comparable data;
    public Node left;
    public Node right;

    /**
     * Inserts a new node as a descendant of this node.
     * @param newNode the node to insert
     */
    public void addNode(Node newNode)
    {
        int comp = newNode.data.compareTo(data);
        if (comp < 0)
        {
            if (left == null) { left = newNode; }
            else { left.addNode(newNode); }
        }
        else if (comp > 0)
        {
            if (right == null) { right = newNode; }
            else { right.addNode(newNode); }
        }
    }
}
13. What is the difference between a tree, a binary tree, and a balanced binary tree?

14. Are the left and right children of a binary search tree always binary search trees? Why or why not?

15. Draw all binary search trees containing data values A, B, and C.

16. Give an example of a string that, when inserted into the tree of Figure 12, becomes a right child of Romeo.

17. Trace the removal of the node “Tom” from the tree in Figure 12.

18. Trace the removal of the node “Juliet” from the tree in Figure 12.

**Practice It** Now you can try these exercises at the end of the chapter: R17.7, R17.13, R17.15, E17.6.

## 17.4 Tree Traversal

We often want to visit all elements in a tree. There are many different orderings in which one can visit, or *traverse*, the tree elements. The following sections introduce the most common ones.

### 17.4.1 Inorder Traversal

Suppose you inserted a number of data values into a binary search tree. What can you do with them? It turns out to be surprisingly simple to print all elements in sorted order. You *know* that all data in the left subtree of any node must come before the root node and before all data in the right subtree. That is, the following algorithm will print the elements in sorted order:

1. Print the left subtree.
2. Print the root data.
3. Print the right subtree.

Let’s try this out with the tree in Figure 12 on page 777. The algorithm tells us to

1. Print the left subtree of *Juliet*; that is, *Diana* and descendants.
2. Print *Juliet*.
3. Print the right subtree of *Juliet*; that is, *Tom* and descendants.

How do you print the subtree starting at *Diana*?

1. Print the left subtree of *Diana*. There is nothing to print.
2. Print *Diana*.
3. Print the right subtree of *Diana*, that is, *Harry*.

That is, the left subtree of *Juliet* is printed as

*Diana* *Harry*

The right subtree of *Juliet* is the subtree starting at *Tom*. How is it printed? Again, using the same algorithm:

1. Print the left subtree of *Tom*, that is, *Romeo*.
2. Print *Tom*.
3. Print the right subtree of *Tom*. There is nothing to print.
Thus, the right subtree of Juliet is printed as

Romeo Tom

Now put it all together: the left subtree, Juliet, and the right subtree:

Diana Harry Juliet Romeo Tom

The tree is printed in sorted order.

It is very easy to implement this print method. We start with a recursive helper method:

```java
private static void print(Node parent) {
    if (parent == null) { return; }
    print(parent.left);
    System.out.print(parent.data + " ");
    print(parent.right);
}
```

To print the entire tree, start this recursive printing process at the root:

```java
public void print() {
    print(root);
}
```

This visitation scheme is called inorder traversal (visit the left subtree, the root, the right subtree). There are two related traversal schemes, called preorder traversal and postorder traversal, which we discuss in the next section.

### 17.4.2 Preorder and Postorder Traversals

In Section 17.4.1, we visited a binary tree in order: first the left subtree, then the root, then the right subtree. By modifying the visitation rules, we obtain other traversals.

In preorder traversal, we visit the root before visiting the subtrees, and in postorder traversal, we visit the root after the subtrees.

<table>
<thead>
<tr>
<th>Preorder(n)</th>
<th>Postorder(n)</th>
</tr>
</thead>
</table>
| Visit n.    | For each child c of n
| For each child c of n
| Preorder(c). | Postorder(c). |
| Visit n.    | Visit n.     |

These two visitation schemes will not print a binary search tree in sorted order. However, they are important in other applications. Here is an example.

In Section 17.2, you saw trees for arithmetic expressions. Their leaves store numbers, and their interior nodes store operators. The expression trees describe the order in which the operators are applied.

Let’s apply postorder traversal to the expression trees in Figure 6 on page 771. The first tree yields

\[ 3 \ 4 \ + \ 5 \ * \]

whereas the second tree yields

\[ 3 \ 4 \ 5 \ * \ + \]
You can interpret the traversal result as an expression in “reverse Polish notation” (see Special Topic 15.2), or equivalently, instructions for a stack-based calculator (see Section 15.6.2).

Here is another example of the importance of traversal order. Consider a directory tree such as the following:

Consider the task of removing all directories from such a tree, with the restriction that you can only remove a directory when it contains no other directories. In this case, you use a postorder traversal.

Conversely, if you want to copy the directory tree, you start copying the root, because you need a target directory into which to place the children. This calls for preorder traversal.

Note that pre- and postorder traversal can be defined for any trees, not just binary trees (see the sample code for this section). However, inorder traversal makes sense only for binary trees.

17.4.3 The Visitor Pattern

In the preceding sections, we simply printed each tree node that we visited. Often, we want to process the nodes in some other way. To make visitation more generic, we define an interface type

```java
public interface Visitor {
    void visit(Object data);
}
```
The `preorder` method receives an object of some class that implements this interface type, and calls its `visit` method:

```java
private static void preorder(Node n, Visitor v) {
    if (n == null) { return; }
    v.visit(n.data);
    for (Node c : n.children) { preorder(c, v); }
}
```

```java
public void preorder(Visitor v) { preorder(root, v); }
```

Methods for postorder and, for a binary tree, inorder traversals can be implemented in the same way.

Let's say we want to count short names (with at most five letters). The following visitor will do the job. We’ll make it into an inner class of the method that uses it.

```java
public static void main(String[] args) {
    BinarySearchTree bst = . . .;
    class ShortNameCounter implements Visitor {
        public int counter = 0;
        public void visit(Object data) {
            if (data.toString().length() <= 5) { counter++; }
        }
    }
    ShortNameCounter v = new ShortNameCounter();
    bst.inorder(v);
    System.out.println("Short names: " + v.counter);
}
```

Here, the visitor object accumulates the count. After the visit is complete, we can obtain the result. Because the class is an inner class, we don’t worry about making the counter private.

### 17.4.4 Depth-First and Breadth-First Search

The traversals in the preceding sections are expressed using recursion. If you want to process the nodes of a tree, you supply a visitor, which is applied to all nodes. Sometimes, it is useful to use an iterative approach instead. Then you can stop processing nodes when a goal has been met.

To visit the nodes of a tree iteratively, we replace the recursive calls with a stack that keeps track of the children that need to be visited. Here is the algorithm:

- **Push the root node on a stack.**
- **While the stack is not empty**
  - **Pop the stack; let n be the popped node.**
  - **Process n.**
  - **Push the children of n on the stack, starting with the last one.**
This algorithm is called depth-first search because it goes deeply into the tree and then backtracks when it reaches the leaves (see Figure 16). Note that the tree can be an arbitrary tree—it need not be binary.

We push the children on the stack in right-to-left order so that the visit starts with the leftmost path. In this way, the nodes are visited in preorder. If the leftmost child had been pushed first, we would still have a depth-first search, just in a less intuitive order.

If we replace the stack with a queue, the visitation order changes. Instead of going deeply into the tree, we first visit all nodes at the same level before going on to the next level. This is called breadth-first search (Figure 17).

For this algorithm, we modify the visitor interface of Section 17.4.3. The visit method now returns a flag indicating whether the traversal should continue. For example, if you want to visit the first ten nodes, you should provide an implementation of the visitor interface whose visit method returns false when it has visited the tenth node.

Here is an implementation of the breadth-first algorithm:

```java
public interface Visitor {
    boolean visit(Object data);
}

public void breadthFirst(Visitor v) {
    if (root == null) { return; }
    Queue<Node> q = new LinkedList<>();
    q.add(root);
    boolean more = true;
    while (more && q.size() > 0) {
        Node n = q.remove();
        more = v.visit(n.data);
    }
}
```
if (more)
{
    for (Node c : n.children) { q.add(c); }
}
}

For depth-first search, replace the queue with a stack (Exercise E17.9).

### 17.4.5 Tree Iterators

The Java collection library uses iterators to process elements of a tree, like this:

```java
TreeSet<String> t = ...;
Iterator<String> iter = t.iterator();
String first = iter.next();
String second = iter.next();
```

It is easy to implement such an iterator with depth-first or breadth-first search. Make the stack or queue into an instance variable of the iterator object. The `next` method executes one iteration of the loop that you saw in the last section.

```java
class BreadthFirstIterator
{
    private Queue<Node> q;
    public BreadthFirstIterator(Node root)
    {
        q = new LinkedList<>();
        if (root != null) { q.add(root); }
    }
    public boolean hasNext() { return q.size() > 0; }
    public Object next()
    {
        Node n = q.remove();
        for (Node c : n.children) { q.add(c); }
        return n.data;
    }
}
```

Note that there is no `visit` method. The user of the iterator receives the node data, processes it, and decides whether to call `next` again.

This iterator produces the nodes in breadth-first order. For a binary search tree, one would want the nodes in sorted order instead. Exercise P17.9 shows how to implement such an iterator.

### SELF CHECK

19. What are the inorder traversals of the two trees in Figure 6 on page 771?

20. Are the trees in Figure 6 binary search trees?

21. Why did we have to declare the variable `v` in the sample program in Section 17.4.4 as `ShortNameCounter` and not as `Visitor`?

22. Consider this modification of the recursive inorder traversal. We want traversal to stop as soon as the visit method returns false for a node.

```java
public static void inorder(Node n, Visitor v)
{
    if (n == null) { return; }
    inorder(n.left, v);
    if (v.visit(n.data)) { inorder(n.right, v); }
}
```
23. In what order are the nodes in Figure 17 visited if one pushes children on the stack from left to right instead of right to left?

24. What are the first eight visited nodes in the breadth-first traversal of the tree in Figure 1?

Practice It  Now you can try these exercises at the end of the chapter: R17.11, R17.14, E17.8.

17.5 Red-Black Trees

As you saw in Section 17.3, insertion and removal in a binary search tree are \( O(\log(n)) \) operations \textit{provided that the tree is balanced}. In this section, you will learn about \textbf{red-black trees}, a special kind of binary search tree that rebalances itself after each insertion or removal. With red-black trees, we can guarantee efficiency of these operations. In fact, the Java Collections framework uses red-black trees to implement tree sets and tree maps.

17.5.1 Basic Properties of Red-Black Trees

A red-black tree is a binary search tree with the following additional properties:

- Every node is colored red or black.
- The root is black.
- A red node cannot have a red child (the “no double reds” rule).
- All paths from the root to a null have the same number of black nodes (the “equal exit cost” rule).

Of course, the nodes aren’t actually colored. Each node simply has a flag to indicate whether it is considered red or black. (The choice of these colors is traditional; one could have equally well used some other attributes. Perhaps, in an alternate universe, students learn about chocolate-vanilla trees.)
Instead of thinking of the colors, imagine each node to be a toll booth. As you travel from the root to one of the `null` references (an exit), you have to pay $1 at each black toll booth, but the red toll booths are free. The “equal exit cost” rule says that the cost of the trip is the same, no matter which exit you choose.

Figure 18 shows an example of a red-black tree, and Figures 19 and 20 show examples of trees that violate the “equal exit cost” and “no double reds” rules.

Note that the “equal exit cost” rule does not just apply to paths that end in a leaf, but to any path from the root to a node with one or two empty children. For example, in Figure 19, the path F–B violates the equal exit cost, yet B is not a leaf.
The “equal exit cost” rule eliminates highly unbalanced trees. You can’t have null references high up in the tree. In other words, the nodes that aren’t near the leaves need to have two children.

The “no double reds” rule gives some flexibility to add nodes without having to restructure the tree all the time. Some paths can be a bit longer than others—by alternating red and black nodes—but none can be longer than twice the black height.

The cost of traveling on a path from a given node to a null (that is, the number of black nodes on the path), is called the black height of the node. The cost of traveling from the root to a null is called the black height of the tree.

A tree with given black height $bh$ can’t be too sparse—it must have at least $2^{bh} - 1$ nodes (see Exercise R17.18). Or, if we turn this relationship around,

$$2^{bh} - 1 \leq n$$

$$2^{bh} \leq n + 1$$

$$bh \leq \log(n + 1)$$

The “no double reds” rule says that the total height $h$ of a tree is at most twice the black height:

$$h \leq 2 \cdot bh \leq 2 \cdot \log(n + 1)$$

Therefore, traveling from the root to a null is $O(\log(n))$.

17.5.2 Insertion

To insert a new node into a red-black tree, first insert it as you would into a regular binary search tree (see Section 17.3.2). Note that the new node is a leaf.

If it is the first node of the tree, it must be black. Otherwise, color it red. If its parent is black, we still have a red-black tree, and we are done.

However, if the parent is also red, we have a “double red” and need to fix it. Because the rest of the tree is a proper red-black tree, we know that the grandparent is black.

There are four possible configurations of a “double red”, shown in Figure 21.

Of course, our tree is a binary search tree, and we will now take advantage of that fact. In each tree of Figure 21, we labeled the smallest, middle, and largest of the three nodes as $n_1$, $n_2$, and $n_3$. We also labeled their children in sorted order, starting with $t_1$. To fix the “double red”, rearrange the three nodes as shown in Figure 22, keeping their data values, but updating their left and right references.
17.5 Red-Black Trees

Fixing the "Double Red" Violation

Because the fix preserves the sort order, the result is a binary search tree. The fix does not change the number of black nodes on a path. Therefore, it preserves the "equal exit cost" rule.

If the parent of \( n_2 \) is black, we get a red-black tree, and we are done. If that parent is red, we have another "double red", but it is one level closer to the root. In that case, fix the double-red violation of \( n_2 \) and its parent. You may have to continue fixing double-red violations, moving closer to the root each time. If the red parent is the root, simply turn it black. This increments all path costs, preserving the "equal exit cost" rule.

Worked Example 17.2 has an implementation of this algorithm.

We can determine the efficiency with more precision than we were able to in Section 17.5.1. To find the insertion location requires at most \( h \) steps, where \( h \) is the height of the tree. To fix the "double red" violations takes at most \( h / 2 \) steps. (Look carefully at Figures 21 and 22 to see that each fix pushes the violation up two nodes. If the top node of each subtree in Figure 21 has height \( t \), then the nodes of the double-red violation have heights \( t + 1 \) and \( t + 2 \). In Figure 22, the top node also has height \( t \). If there is a double-red violation, it is between that node and its parent at height \( t - 1 \).)

We know from Section 17.5.1 that \( h = O(\log(n)) \). Therefore, insertion into a red-black tree is guaranteed to be \( O(\log(n)) \).

17.5.3 Removal

To remove a node from a red-black tree, you first use the removal algorithm for binary search trees (Section 17.3.3). Note that in that algorithm, the removed node has at most one child. We never remove a node with two children; instead, we fill it with the value of another node with at most one child and remove that node.

Two cases are easy. First, if the node to be removed is red, there is no problem with the removal—the resulting tree is still a red-black tree.

Next, assume that the node to be removed has a child. Because of the "equal exit cost" rule, the child must be red. Simply remove the parent and color the child black.
The troublesome case is the removal of a black leaf. We can’t just remove it because the exit cost to the null replacing it would be too low. Instead, we’ll first turn it into a red node.

To turn a black node into a red one, we will temporarily “bubble up” the costs, raising the cost of the parent by 1 and lowering the cost of the children by 1.

This process leaves all path costs unchanged, and it turns the black leaf into a red one which we can safely remove.

Now consider a black leaf that is to be removed. Because of the equal-exit rule, it must have a sibling. The sibling and the parent can be black or red, but they can’t both be red. The leaf to be removed can be to the left or to the right. The figure at right shows all possible cases.

In the first column, bubbling up will work perfectly—it simply turns the red node into a black one and the black ones into red ones. One of the red ones is removed. The other may cause a double-red violation with one of its children, which we fix if necessary.

But in the other cases, a new problem arises. Adding 1 to a black parent yields a price of 2, which we call double-black. Subtracting 1 from a red child yields a negative-red node with a price of –1. These are not valid nodes in a red-black tree, and we need to eliminate them.

A negative-red node is always below a double-black one, and the pair can be eliminated by the transformation shown in Figure 23.

Before removing a node in a red-black tree, turn it red and fix any double-black and double-red violations.
Sometimes, the creation of a double-black node also causes a double-red violation below. We can fix the double-red violation as in the preceding section, but now we color the middle node black instead of red—see Figure 24.

To see that this transformation is valid, imagine a trip through one of the node sequences in Figure 24 from the top node to one of the trees below. The price of that portion of the trip is 2 for each tree, both before and after the transformation.

Sometimes, neither of the two transformations applies, and then we need to “bubble up” again, which pushes the double-black node closer to the root. Figure 25 shows the possible cases.

**Figure 24**
Fixing a Double-Red Violation Also Fixes a Double-Black Grandparent

**Figure 25**
Bubbling Up a Double-Black Node
If the double-black node reaches the root, we can replace it with a regular black node. This reduces the cost of all paths by 1 and preserves the “equal exit cost” rule. See Worked Example 17.2 for an implementation of node removal.

Let us now determine the efficiency of this process. Removing a node from a binary search tree requires $O(h)$ steps, where $h$ is the height of the tree. The double-black node may bubble up, perhaps all the way to the root. Bubbling up will happen at most $h$ times, and its cost is constant—it only involves changing the costs of three nodes. If we generate a negative red, we remove it (as shown in Figure 23), and the bubbling stops. We may have to fix one double-red violation, which takes $O(h)$ steps. It is also possible that bubbling creates a double-red violation, but its fix will absorb the double-black node, and bubbling also stops. The entire process takes $O(h)$ steps. Because $h = O(\log(n))$, removal from a red-black tree is also guaranteed to be $O(\log(n))$.

<table>
<thead>
<tr>
<th>Table 3 Efficiency of Red-Black Tree Operations</th>
</tr>
</thead>
<tbody>
<tr>
<td>Find an element.</td>
</tr>
<tr>
<td>Add an element.</td>
</tr>
<tr>
<td>Remove an element.</td>
</tr>
</tbody>
</table>

25. Consider the extreme example of a tree with only right children and at least three nodes. Why can’t this be a red-black tree?

26. What are the shapes and colorings of all possible red-black trees that have four nodes?

27. Why does Figure 21 show all possible configurations of a double-red violation?

28. When inserting an element, can there ever be a triple-red violation in Figure 21? That is, can you have a red node with two red children? (For example, in the first tree, can $t_1$ have a red root?)

29. When removing an element, show that it is possible to have a triple-red violation in Figure 23.

30. What happens to a triple-red violation when the double-red fix is applied?

Practice It Now you can try these exercises at the end of the chapter: R17.18, R17.20, E17.11.

WORKED EXAMPLE 17.2 Implementing a Red-Black Tree

Learn how to implement a red-black tree as described in Section 17.5. Go to wiley.com/go/bjeo6examples and download Worked Example 17.2.
In this section, we discuss a tree structure that is particularly suited for implementing a priority queue from which the smallest element can be removed efficiently. (Priority queues were introduced in Section 15.5.3.)

A heap (or, for greater clarity, min-heap) is a binary tree with two properties:

1. A heap is *almost completely filled*: all nodes are filled in, except the last level which may have some nodes missing toward the right (see Figure 26).
2. All nodes of the tree fulfill the *heap property*: the node value is at most as large as the values of all descendants (see Figure 27 on page 798).

In particular, because the root fulfills the heap property, its value is the minimum of all values in the tree.

A heap is superficially similar to a binary search tree, but there are two important differences:

1. The shape of a heap is very regular. Binary search trees can have arbitrary shapes.
2. In a heap, the left and right subtrees both store elements that are larger than the root element. In contrast, in a binary search tree, smaller elements are stored in the left subtree and larger elements are stored in the right subtree.

**Figure 26** An Almost Completely Filled Tree
Suppose you have a heap and want to insert a new element. After insertion, the heap property should again be fulfilled. The following algorithm carries out the insertion (see Figure 28).

1. First, add a vacant slot to the end of the tree.
2. Next, demote the parent of the empty slot if it is larger than the element to be inserted. That is, move the parent value into the vacant slot, and move the vacant slot up. Repeat this demotion as long as the parent of the vacant slot is larger than the element to be inserted.
3. At this point, either the vacant slot is at the root, or the parent of the vacant slot is smaller than the element to be inserted. Insert the element into the vacant slot.
Figure 28 (continued)  Inserting an Element into a Heap
We will not consider an algorithm for removing an arbitrary node from a heap. The only node that we will remove is the root node, which contains the minimum of all of the values in the heap. Figure 29 shows the algorithm in action.

1. Extract the root node value.
2. Move the value of the last node of the heap into the root node, and remove the last node. Now the heap property may be violated for the root node, because one or both of its children may be smaller.
3. Promote the smaller child of the root node. Now the root node again fulfills the heap property. Repeat this process with the demoted child. That is, promote the smaller of its children. Continue until the demoted child has no smaller children. The heap property is now fulfilled again. This process is called “fixing the heap”.

Figure 29  Removing the Minimum Value from a Heap
Inserting and removing heap elements is very efficient. The reason lies in the balanced shape of a heap. The insertion and removal operations visit at most \( h \) nodes, where \( h \) is the height of the tree. A heap of height \( h \) contains at least \( 2^{h-1} \) elements, but less than \( 2^h \) elements. In other words, if \( n \) is the number of elements, then

\[
2^{h-1} \leq n < 2^h
\]

or

\[
h - 1 \leq \log_2(n) < h
\]

This argument shows that the insertion and removal operations in a heap with \( n \) elements take \( O(\log(n)) \) steps.

Contrast this finding with the situation of a binary search tree. When a binary search tree is unbalanced, it can degenerate into a linked list, so that in the worst case insertion and removal are \( O(n) \) operations.
Heaps have another major advantage. Because of the regular layout of the heap nodes, it is easy to store the node values in an array or array list. First store the first layer, then the second, and so on (see Figure 30). For convenience, we leave the 0 element of the array empty. Then the child nodes of the node with index \( i \) have index \( 2 \cdot i \) and \( 2 \cdot i + 1 \), and the parent node of the node with index \( i \) has index \( \frac{i}{2} \). For example, as you can see in Figure 30, the children of the node with index 4 are the nodes with index values 8 and 9, and the parent is the node with index 2.

Storing the heap values in an array may not be intuitive, but it is very efficient. There is no need to allocate individual nodes or to store the links to the child nodes. Instead, child and parent positions can be determined by very simple computations. The program at the end of this section contains an implementation of a heap. For greater clarity, the computation of the parent and child index positions is carried out in methods `getParentIndex`, `getLeftChildIndex`, and `getRightChildIndex`. For greater efficiency, the method calls could be avoided by using expressions \( \text{index} / 2 \), \( 2 \times \text{index} \), and \( 2 \times \text{index} + 1 \) directly.

In this section, we have organized our heaps such that the smallest element is stored in the root. It is also possible to store the largest element in the root, simply by reversing all comparisons in the heap-building algorithm. If there is a possibility of misunderstanding, it is best to refer to the data structures as min-heap or max-heap.

The test program demonstrates how to use a min-heap as a priority queue.

**section_6/MinHeap.java**

```java
import java.util.*;

/**
 * This class implements a heap.
 */
```
public class MinHeap {
    private ArrayList<Comparable> elements;

    /**
     * Constructs an empty heap.
     */
    public MinHeap() {
        elements = new ArrayList<>();
        elements.add(null);
    }

    /**
     * Adds a new element to this heap.
     * @param newElement the element to add
     */
    public void add(Comparable newElement) {
        // Add a new leaf
        elements.add(null);
        int index = elements.size() - 1;

        // Demote parents that are larger than the new element
        while (index > 1 && getParent(index).compareTo(newElement) > 0) {
            elements.set(index, getParent(index));
            index = getParentIndex(index);
        }

        // Store the new element in the vacant slot
        elements.set(index, newElement);
    }

    /**
     * Gets the minimum element stored in this heap.
     * @return the minimum element
     */
    public Comparable peek() {
        return elements.get(1);
    }

    /**
     * Removes the minimum element from this heap.
     * @return the minimum element
     */
    public Comparable remove() {
        Comparable minimum = elements.get(1);

        // Remove last element
        int lastIndex = elements.size() - 1;
        Comparable last = elements.remove(lastIndex);

        if (lastIndex > 1) {
            elements.set(1, last);
            fixHeap();
        }
    }
}
```java
private void fixHeap()
{
    Comparable root = elements.get(1);
    int lastIndex = elements.size() - 1;
    // Promote children of removed root while they are smaller than root
    int index = 1;
    boolean more = true;
    while (more)
    {
        int childIndex = getLeftChildIndex(index);
        if (childIndex <= lastIndex)
        {
            // Get smaller child
            // Get left child first
            Comparable child = getLeftChild(index);
            // Use right child instead if it is smaller
            if (getRightChildIndex(index) <= lastIndex
                && getRightChild(index).compareTo(child) < 0)
                {
                    childIndex = getRightChildIndex(index);
                    child = getRightChild(index);
                }
            // Check if smaller child is smaller than root
            if (child.compareTo(root) < 0)
                {
                    // Promote child
                    elements.set(index, child);
                    index = childIndex;
                }
            else
                {
                    // Root is smaller than both children
                    more = false;
                }
        }
        else
            {
                // No children
                more = false;
            }
    }
    // Store root element in vacant slot
    elements.set(index, root);
}
```
126    /*
127     * Checks whether this heap is empty.
128     */
129     public boolean empty()
130     {
131         return elements.size() == 1;
132     }
133
134    /*
135     * Returns the index of the left child.
136     * @param index the index of a node in this heap
137     * @return the index of the left child of the given node
138     */
139     private static int getLeftChildIndex(int index)
140     {
141         return 2 * index;
142     }
143
144    /*
145     * Returns the index of the right child.
146     * @param index the index of a node in this heap
147     * @return the index of the right child of the given node
148     */
149     private static int getRightChildIndex(int index)
150     {
151         return 2 * index + 1;
152     }
153
154    /*
155     * Returns the index of the parent.
156     * @param index the index of a node in this heap
157     * @return the index of the parent of the given node
158     */
159     private static int getParentIndex(int index)
160     {
161         return index / 2;
162     }
163
164    /*
165     * Returns the value of the left child.
166     * @param index the index of a node in this heap
167     * @return the value of the left child of the given node
168     */
169     private Comparable getLeftChild(int index)
170     {
171         return elements.get(2 * index);
172     }
173
174    /*
175     * Returns the value of the right child.
176     * @param index the index of a node in this heap
177     * @return the value of the right child of the given node
178     */
179     private Comparable getRightChild(int index)
180     {
181         return elements.get(2 * index + 1);
182     }
183
184    /*
185     * Returns the value of the parent.
186     */
@param index the index of a node in this heap
@return the value of the parent of the given node
*/
private Comparable getParent(int index)
{
    return elements.get(index / 2);
}
}

section_6/WorkOrder.java

/**
   * This class encapsulates a work order with a priority.
   */
public class WorkOrder implements Comparable
{
private int priority;
private String description;

/**
   * Constructs a work order with a given priority and description.
   * @param aPriority the priority of this work order
   * @param aDescription the description of this work order
   */
public WorkOrder(int aPriority, String aDescription)
{
    priority = aPriority;
    description = aDescription;
}

public String toString()
{
    return "priority=\" + priority + \", description=\" + description;
}

public int compareTo(Object otherObject)
{
    WorkOrder other = (WorkOrder) otherObject;
    if (priority < other.priority) { return -1; }
    if (priority > other.priority) { return 1; }
    return 0;
}

section_6/HeapDemo.java

/**
   * This program demonstrates the use of a heap as a priority queue.
   */
public class HeapDemo
{
    public static void main(String[] args)
    {
        MinHeap q = new MinHeap();
        q.add(new WorkOrder(3, "Shampoo carpets");
        q.add(new WorkOrder(7, "Empty trash");
        q.add(new WorkOrder(8, "Water plants");
        q.add(new WorkOrder(10, "Remove pencil sharpener shavings");
        q.add(new WorkOrder(5, "Replace light bulb");
        q.add(new WorkOrder(1, "Fix broken sink");
    }
15    q.add(new WorkOrder(9, "Clean coffee maker"));
16    q.add(new WorkOrder(2, "Order cleaning supplies"));
17
18    while (!q.empty())
19    {
20        System.out.println(q.remove());
21    }
22
23}

Program Run

priority=1, description=Fix broken sink
priority=2, description=Order cleaning supplies
priority=3, description=Shampoo carpets
priority=6, description=Replace light bulb
priority=7, description=Empty trash
priority=8, description=Water plants
priority=9, description=Clean coffee maker
priority=10, description=Remove pencil sharpener shavings

31. The software that controls the events in a user interface keeps the events in a data structure. Whenever an event such as a mouse move or repaint request occurs, the event is added. Events are retrieved according to their importance. What abstract data type is appropriate for this application?

32. In an almost-complete tree with 100 nodes, how many nodes are missing in the lowest level?

33. If you traverse a heap in preorder, will the nodes be in sorted order?

34. What is the heap that results from inserting 1 into the following?

35. What is the result of removing the minimum from the following?

Practice It Now you can try these exercises at the end of the chapter: R17.24, R17.25, E17.12.
The heapsort algorithm is based on inserting elements into a heap and removing them in sorted order.

Heaps are not only useful for implementing priority queues, they also give rise to an efficient sorting algorithm, heapsort. In its simplest form, the heapsort algorithm works as follows. First insert all elements to be sorted into the heap, then keep extracting the minimum.

This algorithm is an $O(n \log(n))$ algorithm: each insertion and removal is $O(\log(n))$, and these steps are repeated $n$ times, once for each element in the sequence that is to be sorted.

The algorithm can be made a bit more efficient. Rather than inserting the elements one at a time, we will start with a sequence of values in an array. Of course, that array does not represent a heap. We will use the procedure of “fixing the heap” that you encountered in the preceding section as part of the element removal algorithm. “Fixing the heap” operates on a binary tree whose child trees are heaps but whose root value may not be smaller than the descendants. The procedure turns the tree into a heap, by repeatedly promoting the smallest child value, moving the root value to its proper location.

Of course, we cannot simply apply this procedure to the initial sequence of unsorted values—the child trees of the root are not likely to be heaps. But we can first fix small subtrees into heaps, then fix larger trees. Because trees of size 1 are automatically heaps, we can begin the fixing procedure with the subtrees whose roots are located in the next-to-last level of the tree.

The sorting algorithm uses a generalized `fixHeap` method that fixes a subtree:

```
public static void fixHeap(int[] a, int rootIndex, int lastIndex)
```

The subtree is specified by the index of its root and of its last node.

The `fixHeap` method needs to be invoked on all subtrees whose roots are in the next-to-last level. Then the subtrees whose roots are in the next level above are fixed, and so on. Finally, the fixup is applied to the root node, and the tree is turned into a heap (see Figure 31).

That repetition can be programmed easily. Start with the last node on the next-to-lowest level and work toward the left. Then go to the next higher level. The node index values then simply run backward from the index of the last node to the index of the root.

```
int n = a.length - 1;
for (int i = (n - 1) / 2; i >= 0; i--)
{
    fixHeap(a, i, n);
}
```

It can be shown that this procedure turns an arbitrary array into a heap in $O(n)$ steps.

Note that the loop ends with index 0. When working with a given array, we don’t have the luxury of skipping the 0 entry. We consider the 0 entry the root and adjust the formulas for computing the child and parent index values.

After the array has been turned into a heap, we repeatedly remove the root element. Recall from the preceding section that removing the root element is achieved by placing the last element of the tree in the root and calling the `fixHeap` method. Because we call the $O(\log(n))$ `fixHeap` method $n$ times, this process requires $O(n \log(n))$ steps.
Figure 31  Turning a Tree into a Heap
Rather than moving the root element into a separate array, we can swap the root element with the last element of the tree and then reduce the tree size. Thus, the removed root ends up in the last position of the array, which is no longer needed by the heap. In this way, we can use the same array both to hold the heap (which gets shorter with each step) and the sorted sequence (which gets longer with each step).

```java
while (n > 0)
{
    ArrayUtil.swap(a, 0, n);
    n--;
    fixHeap(a, 0, n);
}
```

There is just a minor inconvenience. When we use a min-heap, the sorted sequence is accumulated in reverse order, with the smallest element at the end of the array. We could reverse the sequence after sorting is complete. However, it is easier to use a max-heap rather than a min-heap in the heapsort algorithm. With this modification, the largest value is placed at the end of the array after the first step. After the next step, the next-largest value is swapped from the heap root to the second position from the end, and so on (see Figure 32).

![Figure 32 Using Heapsort to Sort an Array](image)

The following class implements the heapsort algorithm:

```java
/*
 * The sort method of this class sorts an array, using the heap sort algorithm.
 */
public class HeapSorter
{
    /**
     * Sorts an array, using selection sort.
     * @param a the array to sort
     */
    public static void sort(int[] a)
    {
        int n = a.length - 1;
        for (int i = (n - 1) / 2; i >= 0; i--)
        {
            fixHeap(a, i, n);
        }
        while (n > 0)
        {
            ArrayUtil.swap(a, 0, n);
            n--;
            fixHeap(a, 0, n);
        }
    }
}
```
The Heapsort Algorithm

```java
ArrayUtil.swap(a, 0, n);
    n--;
    fixHeap(a, 0, n);
}

/**
 * Ensures the heap property for a subtree, provided its
 * children already fulfill the heap property.
 * @param a the array to sort
 * @param rootIndex the index of the subtree to be fixed
 * @param lastIndex the last valid index of the tree that
 * contains the subtree to be fixed
 */
private static void fixHeap(int[] a, int rootIndex, int lastIndex)
{
    // Remove root
    int rootValue = a[rootIndex];

    // Promote children while they are larger than the root
    int index = rootIndex;
    boolean more = true;
    while (more)
    {
        int childIndex = getLeftChildIndex(index);
        if (childIndex <= lastIndex)
        {
            // Use right child instead if it is larger
            int rightChildIndex = getRightChildIndex(index);
            if (rightChildIndex <= lastIndex
                && a[rightChildIndex] > a[childIndex])
            {
                childIndex = rightChildIndex;
            }

            if (a[childIndex] > rootValue)
            {
                // Promote child
                a[index] = a[childIndex];
                index = childIndex;
            }
            else
            {
                // Root value is larger than both children
                more = false;
            }
        }
        else
        {
            // No children
            more = false;
        }
    }

    // Store root value in vacant slot
    a[index] = rootValue;
}
```
812  Chapter 17  Tree Structures

```java
/**
   * Returns the index of the left child.
   * @param index the index of a node in this heap
   * @return the index of the left child of the given node
   */
private static int getLeftChildIndex(int index)
{
    return 2 * index + 1;
}

/**
   * Returns the index of the right child.
   * @param index the index of a node in this heap
   * @return the index of the right child of the given node
   */
private static int getRightChildIndex(int index)
{
    return 2 * index + 2;
}
```

36. Which algorithm requires less storage, heapsort or merge sort?
37. Why are the computations of the left child index and the right child index in the HeapSorter different than in MinHeap?
38. What is the result of calling HeapSorter.fixHeap(a, 0, 4) where a contains 1 4 9 5 3?
39. Suppose after turning the array into a heap, it is 9 4 5 1 3. What happens in the first iteration of the while loop in the sort method?
40. Does heapsort sort an array that is already sorted in $O(n)$ time?

**Practice It** Now you can try these exercises at the end of the chapter: R17.28, E17.13.

**CHAPTER SUMMARY**

Describe and implement general trees.
- A tree is composed of nodes, each of which can have child nodes.
- The root is the node with no parent. A leaf is a node with no children.
- A tree class uses a node class to represent nodes and has an instance variable for the root node.
- Many tree properties are computed with recursive methods.

Describe binary trees and their applications.
- A binary tree consists of nodes, each of which has at most two child nodes.
- In a Huffman tree, the left and right turns on the paths to the leaves describe binary encodings.
- An expression tree shows the order of evaluation in an arithmetic expression.
- In a balanced tree, all paths from the root to the leaves have approximately the same length.
Explain the implementation of a binary search tree and its performance characteristics.

- All nodes in a binary search tree fulfill the property that the descendants to the left have smaller data values than the node data value, and the descendants to the right have larger data values.
- To insert a value into a binary search tree, keep comparing the value with the node data and follow the nodes to the left or right, until reaching a null node.
- When removing a node with only one child from a binary search tree, the child replaces the node to be removed.
- When removing a node with two children from a binary search tree, replace it with the smallest node of the right subtree.
- In a balanced tree, all paths from the root to the leaves have about the same length.
- If a binary search tree is balanced, then adding, locating, or removing an element takes $O(\log(n))$ time.

Describe preorder, inorder, and postorder tree traversal.

- To visit all elements in a tree, visit the root and recursively visit the subtrees.
- We distinguish between preorder, inorder, and postorder traversal.
- Postorder traversal of an expression tree yields the instructions for evaluating the expression on a stack-based calculator.
- Depth-first search uses a stack to track the nodes that it still needs to visit.
- Breadth-first search first visits all nodes on the same level before visiting the children.

Describe how red-black trees provide guaranteed $O(\log(n))$ operations.

- In a red-black tree, node coloring rules ensure that the tree is balanced.
- To rebalance a red-black tree after inserting an element, fix all double-red violations.
- Before removing a node in a red-black tree, turn it red and fix any double-black and double-red violations.
- Adding or removing an element in a red-black tree is an $O(\log(n))$ operation.

Describe the heap data structure and the efficiency of its operations.

- A heap is an almost completely filled tree in which the value of any node is less than or equal to the values of its descendants.
- Inserting or removing a heap element is an $O(\log(n))$ operation.
- The regular layout of a heap makes it possible to store heap nodes efficiently in an array.

Describe the heapsort algorithm and its run-time performance.

- The heapsort algorithm is based on inserting elements into a heap and removing them in sorted order.
- Heapsort is an $O(n \log(n))$ algorithm.
R17.1 What are all possible shapes of trees of height $h$ with one leaf? Of height 2 with $k$ leaves?

R17.2 Describe a recursive algorithm for finding the maximum number of siblings in a tree.

R17.3 Describe a recursive algorithm for finding the total path length of a tree. The total path length is the sum of the lengths of all paths from the root to the leaves. (The length of a path is the number of nodes on the path.) What is the efficiency of your algorithm?

R17.4 Show that a binary tree with $l$ leaves has at least $l - 1$ interior nodes, and exactly $l - 1$ interior nodes if all of them have two children.

R17.5 What is the difference between a binary tree and a binary search tree? Give examples of each.

R17.6 What is the difference between a balanced tree and an unbalanced tree? Give examples of each.

R17.7 The following elements are inserted into a binary search tree. Make a drawing that shows the resulting tree after each insertion.

Adam
Eve
Romeo
Juliet
Tom
Diana
Harry

R17.8 Insert the elements of Exercise R17.7 in opposite order. Then determine how the BinarySearchTree.print method from Section 17.4 prints out both the tree from Exercise R17.7 and this tree. Explain how the printouts are related.

R17.9 Consider the following tree. In which order are the nodes printed by the BinarySearchTree.print method? The numbers identify the nodes. The data stored in the nodes is not shown.

R17.10 Design an algorithm for finding the $k$th element (in sort order) of a binary search tree. How efficient is your algorithm?

R17.11 Design an $O(\log(n))$ algorithm for finding the $k$th element in a binary search tree, provided that each node has an instance variable containing the size of the subtree. Also describe how these instance variables can be maintained by the insertion and removal operations without affecting their big-Oh efficiency.
**R17.12** Design an algorithm for deciding whether two binary trees have the same shape. What is the running time of your algorithm?

**R17.13** Insert the following eleven words into a binary search tree:

Mary had a little lamb. Its fleece was white as snow.

Draw the resulting tree.

**R17.14** What is the result of printing the tree from Exercise R17.13 using preorder, inorder, and postorder traversal?

**R17.15** Locate nodes with no children, one child, and two children in the tree of Exercise R17.13. For each of them, show the tree of size 10 that is obtained after removing the node.

**R17.16** Repeat Exercise R17.13 for a red-black tree.

**R17.17** Repeat Exercise R17.15 for a red-black tree.

**R17.18** Show that a red-black tree with black height $bh$ has at least $2^{bh} - 1$ nodes. *Hint:* Look at the root. A black child has black height $bh - 1$. A red child must have two black children of black height $bh - 1$.

**R17.19** Let $rbts(bh)$ be the number of red-black trees with black height $bh$. Give a recursive formula for $rbts(bh)$ in terms of $rbts(bh - 1)$. How many red-black trees have heights 1, 2, and 3? *Hint:* Look at the hint for Exercise R17.18.

**R17.20** What is the maximum number of nodes in a red-black tree with black height $bh$?

**R17.21** Show that any red-black tree must have fewer interior red nodes than it has black nodes.

**R17.22** Show that the “black root” rule for red-black trees is not essential. That is, if one allows trees with a red root, insertion and deletion still occur in $O(\log(n))$ time.

**R17.23** Many textbooks use “dummy nodes”—black nodes with two null children—instead of regular null references in red-black trees. In this representation, all non-dummy nodes of a red-black tree have two children. How does this simplify the description of the removal algorithm?

**R17.24** Could a priority queue be implemented efficiently as a binary search tree? Give a detailed argument for your answer.

**R17.25** Will preorder, inorder, or postorder traversal print a heap in sorted order? Why or why not?

**R17.26** Prove that a heap of height $h$ contains at least $2^{h-1}$ elements but less than $2^h$ elements.
Suppose the heap nodes are stored in an array, starting with index 1. Prove that the child nodes of the heap node with index \( i \) have index \( 2 \cdot i \) and \( 2 \cdot i + 1 \), and the parent node of the heap node with index \( i \) has index \( i/2 \).

Simulate the heapsort algorithm manually to sort the array

\[ 11 \ 27 \ 8 \ 14 \ 45 \ 6 \ 24 \ 81 \ 29 \ 33 \]

Show all steps.

**Practice Exercises**

**E17.1** Write a method that counts the number of all leaves in a tree.

**E17.2** Add a method `countNodesWithOneChild` to the `BinaryTree` class.

**E17.3** Add a method `swapChildren` that swaps all left and right children to the `BinaryTree` class.

**E17.4** Implement the animal guessing game described in Section 17.2.1. Start with the tree in Figure 4, but present the leaves as “Is it a(n) X?” If it wasn’t, ask the user what the animal was, and ask for a question that is true for that animal but false for X. For example,

- Is it a mammal? \( Y \)
- Does it have stripes? \( N \)
- Is it a pig? \( N \)
- I give up. What is it? A hamster
  Please give me a question that is true for a hamster and false for a pig.
  Is it small and cuddly?

In this way, the program learns additional facts.

**E17.5** Reimplement the `addNode` method of the `Node` class in `BinarySearchTree` as a static method of the `BinarySearchTree` class:

```java
private static Node addNode(Node parent, Node newNode)
If parent is null, return newNode. Otherwise, recursively add newNode to parent and return parent. Your implementation should replace the three null checks in the add and original `addNode` methods with just one null check.
```

**E17.6** Write a method of the `BinarySearchTree` class

```java
Comparable smallest()
```

that returns the smallest element of a tree. You will also need to add a method to the `Node` class.

**E17.7** Add methods

```java
void preorder(Visitor v)
void inorder(Visitor v)
void postorder(Visitor v)
```

to the `BinaryTree` class of Section 17.2.

**E17.8** Using a visitor, compute the average value of the elements in a binary tree filled with `Integer` objects.
- **E17.9** Add a method `void depthFirst(Visitor v)` to the Tree class of Section 17.4. Keep visiting until the `visit` method returns `false`.

- **E17.10** Implement an `inorder` method for the `BinaryTree` class of Section 17.2 so that it stops visiting when the `visit` method returns `false`. *(Hint: Have `inorder` return `false` when `visit` returns `false`.)*

- **E17.11** Write a method for the `RedBlackTree` class of Worked Example 17.2 that checks that the tree fulfills the rules for a red-black tree.

- **E17.12** Modify the implementation of the `MinHeap` class so that the parent and child index positions and elements are computed directly, without calling helper methods.

- **E17.13** Time the results of heapsort and merge sort. Which algorithm behaves better in practice?

---

**PROGRAMMING PROJECTS**

- **P17.1** A general tree (in which each node can have arbitrarily many children) can be implemented as a binary tree in this way: For each node with \( n \) children, use a chain of \( n \) binary nodes. Each left reference points to a child and each right reference points to the next node in the chain. Using the binary tree implementation of Section 17.2, implement a tree class with the same interface as the one in Section 17.1.

- **P17.2** A general tree in which all non-leaf nodes have `null` data can be implemented as a list of lists. For example, the tree

  ```
  B
  / \ /
 A C D
  ```

  is the list `[[A, B], C, [D]]`.

  Using the list implementation from Section 16.1, implement a tree class with the same interface as the one in Section 17.1. *(Hint: Use `instanceof` `List` to check whether a list element `n` is a subtree or a leaf.)*

- **P17.3** Continue Exercise E17.4 and write the tree to a file when the program exits. Load the file when the program starts again.

- **P17.4** Change the `BinarySearchTree.print` method to print the tree as a tree shape. You can print the tree sideways. Extra credit if you instead display the tree with the root node centered on the top.

- **P17.5** In the `BinarySearchTree` class, modify the `remove` method so that a node with two children is replaced by the largest child of the left subtree.

- **P17.6** Reimplement the `remove` method in the `RedBlackTree` class of Worked Example 17.2 so that the node is first removed using the binary search tree removal algorithm, and the tree is rebalanced after removal.
The ID3 algorithm describes how to build a decision tree for a given set of sample facts. The tree asks the most important questions first. We have a set of criteria (such as “Is it a mammal?”) and an objective that we want to decide (such as “Can it swim?”). Each fact has a value for each criterion and the objective. Here is a set of five facts about animals. (Each row is a fact.) There are four criteria and one objective (the columns of the table). For simplicity, we assume that the values of the criteria and objective are binary (Y or N).

<table>
<thead>
<tr>
<th>Is it a mammal?</th>
<th>Does it have fur?</th>
<th>Does it have a tail?</th>
<th>Does it lay eggs?</th>
<th>Can it swim?</th>
</tr>
</thead>
<tbody>
<tr>
<td>N</td>
<td>N</td>
<td>Y</td>
<td>Y</td>
<td>N</td>
</tr>
<tr>
<td>N</td>
<td>N</td>
<td>N</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>N</td>
<td>N</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>Y</td>
<td>N</td>
<td>Y</td>
<td>N</td>
<td>Y</td>
</tr>
<tr>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>N</td>
<td>N</td>
</tr>
</tbody>
</table>

We now need several definitions. Given any probability value $p$ between 0 and 1, its uncertainty is

$$U(p) = -p \log_2(p) - (1 - p) \log_2(1 - p)$$

If $p$ is 0 or 1, the outcome is certain, and the uncertainty $U(p)$ is 0. If $p = 1/2$, then the outcome is completely uncertain and $U(p) = 1$.

Let $n$ be the number of facts and $n(c = Y)$ be the number of facts for which the criterion $c$ has the value Y. Then the uncertainty $U(c, o)$ that $c$ contributes to the outcome $o$ is the weighted average of two uncertainties:

$$U(c, o) = \frac{n(c = Y)}{n} \cdot U\left(\frac{n(c = Y, o = Y)}{n(c = Y)}\right) + \frac{n(c = N)}{n} \cdot U\left(\frac{n(c = N, o = Y)}{n(c = N)}\right)$$

Find the criterion $c$ that minimizes the uncertainty $U(c, o)$. That question becomes the root of your tree. Recursively, repeat for the subsets of the facts for which $c$ is Y (in the left subtree) and N (in the right subtree). If it happens that the objective is constant, then you have a leaf with an answer, and the recursion stops.
In our example, we have

Two out of five are mammals.

Is it a mammal? \( \frac{2}{5} \cdot U\left(\frac{1}{2}\right) + \frac{3}{5} \cdot U\left(\frac{2}{3}\right) = 0.95 \)

Does it have fur? \( \frac{1}{5} \cdot U\left(\frac{0}{1}\right) + \frac{4}{5} \cdot U\left(\frac{3}{4}\right) = 0.65 \)

Does it have a tail? \( \frac{4}{5} \cdot U\left(\frac{2}{4}\right) + \frac{1}{5} \cdot U\left(\frac{1}{1}\right) = 0.8 \)

Does it lay eggs? \( \frac{3}{5} \cdot U\left(\frac{2}{3}\right) + \frac{2}{5} \cdot U\left(\frac{1}{2}\right) = 0.95 \)

Therefore, we choose “Does it have fur?” as our first criterion.

In the left subtree, look at the animals with fur. There is only one, a non-swimmer, so you can declare “It doesn’t swim.” For the right subtree, you now have four facts (the animals without fur) and three criteria. Repeat the process.

*** P17.8 Modify the expression evaluator from Section 13.5 to produce an expression tree. (Note that the resulting tree is a binary tree but not a binary search tree.) Then use postorder traversal to evaluate the expression, using a stack for the intermediate results.

*** P17.9 Implement an iterator for the BinarySearchTree class that visits the nodes in sorted order. Hint: In the constructor, keep pushing left nodes on a stack until you reach null. In each call to next, deliver the top of the stack as the visited node, but first push the left nodes in its right subtree.

*** P17.10 Implement an iterator for the RedBlackTree class in Worked Example 17.2 that visits the nodes in sorted order. Hint: Take advantage of the parent links.

*** P17.11 Modify the implementation of the MinHeap class in Section 17.6 so that the 0 element of the array is not wasted.
1. There are four paths:
   Anne
   Anne, Peter
   Anne, Zara
   Anne, Peter, Savannah

2. There are three subtrees with three nodes—they have roots Charles, Andrew, and Edward.

3. 3.

4. 

5. If \( n \) is a leaf, the leaf count is 1. Otherwise
   
   Let \( c_1 \ldots c_n \) be the children of \( n \).
   The leaf count is \( \text{leafCount}(c_1) + \ldots + \text{leafCount}(c_n) \).

6. Tree \( t_1 = \text{new Tree}("Anne") \);
   Tree \( t_2 = \text{new Tree}("Peter") \);
   \( t_1 \text{.addSubtree}(t_2) \);
   Tree \( t_3 = \text{new Tree}("Zara") \);
   \( t_1 \text{.addSubtree}(t_3) \);
   Tree \( t_4 = \text{new Tree}("Savannah") \);
   \( t_2 \text{.addSubtree}(t_4) \);

7. It is not. However, it calls a recursive method—the size method of the Node class.

8. \( A=10, L=0000, O=001, H=0001 \), therefore ALOHA = 100000001000110.

9. In the root.

10. 

11. Figure 4: 6 leaves, 5 interior nodes.
    Figure 5: 13 leaves, 12 interior nodes.
    Figure 6: 3 leaves, 2 interior nodes.
    You might guess from these data that the number of leaves always equals the number of interior nodes + 1. That is true if all interior nodes have two children, but it is false otherwise—consider this tree whose root only has one child.

12. 

13. In a tree, each node can have any number of children. In a binary tree, a node has at most two children. In a balanced binary tree, all nodes have approximately as many descendants to the left as to the right.

14. Yes—because the binary search condition holds for all nodes of the tree, it holds for all nodes of the subtrees.

15. 


16. For example, Sarah. Any string between Romeo and Tom will do.

17. “Tom” has a single child. That child replaces “Tom” in the parent “Juliet”.

18. “Juliet” has two children. We look for the smallest child in the right subtree, “Romeo”. The data replaces “Juliet”, and the node is removed from its parent “Tom”.

19. For both trees, the inorder traversal is \(3 + 4 \times 5\).

20. No—for example, consider the children of \(+\). Even without looking up the Unicode values for 3, 4, and \(+\), it is obvious that \(+\) isn’t between 3 and 4.

21. Because we need to call \(v\).counter in order to retrieve the result.

22. When the method returns to its caller, the caller can continue traversing the tree. For example, suppose the tree is

23. AGIHFBEDC

24. That’s the royal family tree, the first tree in the chapter: George V, Edward VIII, George VI, Mary, Henry, George, John, Elizabeth II.

25. The root must be black, and the second or third node must also be black, because of the “no double reds” rule. The left null of the root has black height 1, but the null child of the next black node has black height 2.

26. The top red node can be the left or right child of the black parent, and the bottom red node can be the left or right child of its (red) parent, yielding four configurations.

27. No. Look at the first tree. At the beginning, \(n_2\) must have been the inserted node. Because the tree was a valid red-black tree before insertion, \(t_1\) couldn’t have had a red root. Now consider the step after one double-red removal. The parent of \(n_2\) in Figure 22 may be red, but then \(n_2\) can’t have a red sibling—otherwise the tree would not have been a red-black tree.
29. Consider this scenario, where X is the black leaf to be removed.

![Diagram of a tree structure with nodes labeled n1, n2, n3, and X, showing the process of bubble up and fix the negative-red operations.]

30. It goes away. Suppose the sibling of the red grandchild in Figure 21 is also red. That means that one of the $t_i$ has a red root. However, all of them become children of the black $n_1$ and $n_3$ in Figure 22.

31. A priority queue is appropriate because we want to get the important events first, even if they have been inserted later.

32. 27. The next power of 2 greater than 100 is 128, and a completely filled tree has 127 nodes.

33. Generally not. For example, the heap in Figure 30 in preorder is 20 75 84 90 96 91 93 43 57 71.

34. ![Diagram of a binary tree illustrating the process of fixing a heap after removing a node.]

35. ![Diagram of another binary tree showing the same process for a different structure.]

36. Heapsort requires less storage because it doesn’t need an auxiliary array.

37. The $\text{MinHeap}$ wastes the 0 entry to make the formulas more intuitive. When sorting an array, we don’t want to waste the 0 entry, so we adjust the formulas instead.

38. In tree form, that is

![Tree diagram with numbers 1, 4, 9, 5, 3, 2, 5, 4, 9, 3, 9, 4, 9, 5, 5 showing the heap structure after adjustments.]

39. The 9 is swapped with 3, and the heap is fixed up again, yielding 5 4 3 1 | 9.

40. Unfortunately not. The largest element is removed first, and it must be moved to the root, requiring $O(\log(n))$ steps. The second-largest element is still toward the end of the array, again requiring $O(\log(n))$ steps, and so on.
A Huffman code encodes symbols into sequences of zeroes and ones, so that the most frequently occurring symbols have the shortest encodings. The symbols can be characters of the alphabet, but they can also be something else. For example, when images are compressed using a Huffman encoding, the symbols are the colors that occur in the image.

**Problem Statement** Encode a child’s painting like the one below by building a Huffman tree with an optimal encoding. Most of the pixels are white (50%), there are lots of orange (20%) and pink (20%) pixels, and small amounts of yellow (5%), blue (3%), and green (2%).

We want a short code (perhaps 0) for white and a long one (perhaps 1110) for green. Such a variable-length encoding minimizes the overall length of the encoded data.

The challenge is to build a tree that yields an optimal encoding. The following algorithm, developed by David Huffman when he was a graduate student, achieves this task.

- Make a tree node for each symbol to be encoded. Each node has an instance variable for the frequency.
- Add all nodes to a priority queue.
- While there are two nodes left
  - Remove the two nodes with the smallest frequencies.
  - Make them children of a parent whose frequency is the sum of the child frequencies.
  - Add the parent to the priority queue.
- The remaining node is the root of the Huffman tree.
The following figure shows the algorithm applied to our sample data.

After the tree has been constructed, the frequencies are no longer needed. The resulting code is

White  0  
Pink   100
Yellow 1010
Blue   10110
Green  10111
Orange 11

Note that this is not a code for encrypting information. The code is known to all; its purpose is to compress data by using the shortest codes for the most common symbols. Also note that the code has the property that no codeword is the prefix of another codeword. For example, because white is encoded as 0, no other codeword starts with 0, and because orange is 11, no other codeword starts with 11.
Building a Huffman Tree

The implementation is very straightforward. The Node class needs instance variables for holding the symbol to be encoded (which we assume to be a character) and its frequency. It must also implement the Comparable interface so that we can put nodes into a priority queue:

```java
class Node implements Comparable<Node> {
    public char character;
    public int frequency;
    public Node left;
    public Node right;
    public int compareTo(Node other) { return frequency - other.frequency; }
}
```

When constructing a tree, we need the frequencies for all characters. The tree constructor receives them in a Map<Character, Integer>. The frequencies need not be percentages. They can be counts from a sample text.

First, we make a node for each character to be encoded, and add each node to a priority queue:

```java
PriorityQueue<Node> nodes = new PriorityQueue<>();
for (char ch : frequencies.keySet()) {
    Node newNode = new Node();
    newNode.character = ch;
    newNode.frequency = frequencies.get(ch);
    nodes.add(newNode);
}
```

Then, following the algorithm, we keep combining the two nodes with the lowest frequencies:

```java
while (nodes.size() > 1) {
    Node smallest = nodes.remove();
    Node nextSmallest = nodes.remove();
    Node newNode = new Node();
    newNode.frequency = smallest.frequency + nextSmallest.frequency;
    newNode.left = smallest;
    newNode.right = nextSmallest;
    nodes.add(newNode);
}
root = nodes.remove();
```

Decoding a sequence of zeroes and ones is very simple: just follow the links to the left or right until a leaf is reached. Note that each node has either two or no children, so we only need to check whether one of the children is null to detect a leaf. Here we use strings of 0 or 1 characters, not actual bits, to keep the demonstration simple.

```java
public String decode(String input) {
    String result = "";
    Node n = root;
    for (int i = 0; i < input.length(); i++) {
        char ch = input.charAt(i);
        if (ch == '0') {
            n = n.left;
        } else {
            n = n.right;
        }
    }
    return result;
}
```
The tree is not useful for efficient encoding because we don’t want to search through the leaves each time we encode a character. Instead, we will just compute a map that maps each character to its encoding. This can be done by recursively visiting the subtrees and remembering the current prefix, that is, the path to the root of the subtree. Follow the left or right children, adding a 0 or 1 to the end of that prefix, or, if the subtree is a leaf, simply add the character and the prefix to the map:

```java
class Node implements Comparable<Node> {
    ...
    public void fillEncodingMap(Map<Character, String> map, String prefix) {
        if (left == null) // It's a leaf
        {
            map.put(character, prefix);
        }
        else
        {
            left.fillEncodingMap(map, prefix + "0");
            right.fillEncodingMap(map, prefix + "1");
        }
    }
}
```

This recursive helper method is called from the HuffmanTree class:

```java
public class HuffmanTree {
    ...
    public Map<Character, String> getEncodingMap() {
        Map<Character, String> map = new HashMap<>();
        if (root != null) { root.fillEncodingMap(map, ""); }
        return map;
    }
}
```

The demonstration program (in your ch17/worked_example_1 code folder) computes the Huffman encoding for the Hawaiian language, which was chosen because it uses fewer letters than most other languages. The frequencies were obtained from a text sample on the Internet.
The Node Implementation

The nodes of the red-black tree need to store the “color”, which we represent as the cost of traversing the node:

```java
static final int BLACK = 1;
static final int RED = 0;
private static final int NEGATIVE_RED = -1;
private static final int DOUBLE_BLACK = 2;
```

```java
static class Node
{
    public Comparable data;
    public Node left;
    public Node right;
    public Node parent;
    public int color;
    ...
}
```

The first two color constants and the Node class have package visibility. We will add a test class to the same package, which is discussed later in this worked example.

Nodes in a red-black tree also have a link to the parent. When adding or moving a node, it is important that the parent and child links are synchronized. Because this synchronization is tedious and error-prone, we provide several helper methods:

```java
public class RedBlackTree
{
    ...
    static class Node
    {
        ...

        /**
         * Sets the left child and updates its parent reference.
         * @param child the new left child
         */
        public void setLeftChild(Node child)
        {
            left = child;
            if (child != null) { child.parent = this; }
        }

        /**
         * Sets the right child and updates its parent reference.
         * @param child the new right child
         */
        public void setRightChild(Node child)
        {
            right = child;
            if (child != null) { child.parent = this; }
        }
    }
```
Updates the parent’s and replacement node’s links when a node is replaced. Also updates the root reference if the root is replaced.

@param toBeReplaced the node that is to be replaced
@param replacement the node that replaces that node

private void replaceWith(Node toBeReplaced, Node replacement) {
    if (toBeReplaced.parent == null) {
        replacement.parent = null;
        root = replacement;
    } else if (toBeReplaced == toBeReplaced.parent.left) {
        toBeReplaced.parent.setLeftChild(replacement);
    } else {
        toBeReplaced.parent.setRightChild(replacement);
    }
}

Insertion

Insertion is handled as it is in a binary search tree. We insert a red node. Afterward, we call a method that fixes up the tree so it is a red-black tree again:

public void add(Comparable obj) {
    Node newNode = new Node();
    newNode.data = obj;
    newNode.left = null;
    newNode.right = null;
    if (root == null) { root = newNode; } else { root.addNode(newNode); } fixAfterAdd(newNode);
}

If the inserted node is the root, it is turned black. Otherwise, we fix up any double-red violations:

private void fixAfterAdd(Node newNode) {
    if (newNode.parent == null) {
        newNode.color = BLACK;
    } else {
        newNode.color = RED;
        if (newNode.parent.color == RED) { fixDoubleRed(newNode); }
    }
}
The code for fixing up a double-red violation is quite long. Recall that there are four possible arrangements of the double red nodes:

In each case, we must sort the nodes and their children. Once we have the seven references $n_1, n_2, n_3, t_1, t_2, t_3,$ and $t_4,$ the remainder of the procedure is straightforward. We build the replacement tree, change the reds to black, and subtract one from the color of the grandparent (which might be a double-black node when this method is called during node removal).

If we find that we introduced another double-red violation, we continue fixing it. Eventually, the violation is removed, or we reach the root, in which case the root is simply colored black:

```java
/**
 * Fixes a “double red” violation.
 * @param child the child with a red parent
 */
```
private void fixDoubleRed(Node child) {
    Node parent = child.parent;
    Node grandParent = parent.parent;
    if (grandParent == null) { parent.color = BLACK; return; }
    Node n1, n2, n3, t1, t2, t3, t4;
    if (parent == grandParent.left) {
        n3 = grandParent; t4 = grandParent.right;
        if (child == parent.left) {
            n1 = child; n2 = parent;
            t1 = child.left; t2 = child.right; t3 = parent.right;
        } else {
            n1 = parent; n2 = child;
            t1 = parent.left; t2 = child.left; t3 = child.right;
        }
    } else {
        n1 = grandParent; t1 = grandParent.left;
        if (child == parent.left) {
            n2 = child; n3 = parent;
            t2 = child.left; t3 = child.right; t4 = parent.right;
        } else {
            n2 = parent; n3 = child;
            t2 = parent.left; t3 = child.left; t4 = child.right;
        }
    }
    replaceWith(grandParent, n2);
    n1.setLeftChild(t1);
    n1.setRightChild(t2);
    n2.setLeftChild(n1);
    n2.setRightChild(n3);
    n3.setLeftChild(t3);
    n3.setRightChild(t4);
    n2.color = grandParent.color - 1;
    n1.color = BLACK;
    n3.color = BLACK;
    if (n2 == root) {
        root.color = BLACK;
    } else if (n2.color == RED && n2.parent.color == RED) {
        fixDoubleRed(n2);
    }
}

Big Java, 6e, Cay Horstmann, Copyright © 2015 John Wiley and Sons, Inc. All rights reserved.
Removal

We remove a node in the same way as in a binary search tree. However, before removing it, we want to make sure that it is colored red. There are two cases for removal, removing an element with one child and removing the successor of an element with two children. Both branches must be modified:

```java
public void remove(Comparable obj)
{
    // Find node to be removed
    Node toBeRemoved = root;
    boolean found = false;
    while (!found && toBeRemoved != null)
    {
        int d = toBeRemoved.data.compareTo(obj);
        if (d == 0) { found = true; }
        else
        {
            if (d > 0) { toBeRemoved = toBeRemoved.left; }
            else { toBeRemoved = toBeRemoved.right; }
        }
    }
    if (!found) { return; }
    // toBeRemoved contains obj
    // If one of the children is empty, use the other
    if (toBeRemoved.left == null || toBeRemoved.right == null)
    {
        Node newChild;
        if (toBeRemoved.left == null) { newChild = toBeRemoved.right; }
        else { newChild = toBeRemoved.left; }
        fixBeforeRemove(toBeRemoved);
        replaceWith(toBeRemoved, newChild);
        return;
    }
    // Neither subtree is empty
    // Find smallest element of the right subtree
    Node smallest = toBeRemoved.right;
    while (smallest.left != null)
    {
        smallest = smallest.left;
    }
    // smallest contains smallest child in right subtree
    // Move contents, unlink child
    toBeRemoved.data = smallest.data;
    fixBeforeRemove(smallest);
    replaceWith(smallest, smallest.right);
}
```
The replaceWith helper method, which was shown earlier, takes care of updating the parent, child, and root links. The fixBeforeRemove method has three cases. Removing a red leaf is safe. If a black node has a single child, that child must be red, and we can safely swap the colors. (We don’t actually bother to color the node that is to be removed.) The case with a black leaf is the hardest. We need to initiate the “bubbling up” process:

```java
/**
 * Fixes the tree so that it is a red-black tree after a node has been removed.
 * @param toBeRemoved the node that is to be removed
 */
private void fixBeforeRemove(Node toBeRemoved)
{
    if (toBeRemoved.color == RED) { return; }
    if (toBeRemoved.left != null || toBeRemoved.right != null) // It is not a leaf
    {
        // Color the child black
        if (toBeRemoved.left == null) { toBeRemoved.right.color = BLACK; }
        else { toBeRemoved.left.color = BLACK; }
    }
    else { bubbleUp(toBeRemoved.parent); }
}
```

To bubble up, we move a “toll charge” from the children to the parent. This may result in a negative-red or double-red child, which we fix. If neither fix was successful, and the parent node is still double-black, we bubble up again until we reach the root. The root color can be safely changed to black.

```java
/**
 * Move a charge from two children of a parent.
 * @param parent a node with two children, or null (in which case nothing is done)
 */
private void bubbleUp(Node parent)
{
    if (parent == null) { return; }
    parent.color++;
    parent.left.color--;
    parent.right.color--;
    if (bubbleUpFix(parent.left)) { return; }
    if (bubbleUpFix(parent.right)) { return; }
    if (parent.color == DOUBLE_BLACK)
    {
        if (parent.parent == null) { parent.color = BLACK; }
        else { bubbleUp(parent.parent); }
    }
}
```

```java
/**
 * Fixes a negative-red or double-red violation introduced by bubbling up.
 * @param child the child to check for negative-red or double-red violations
 * @return true if the tree was fixed
 */
private boolean bubbleUpFix(Node child)
{
    if (child.color == NEGATIVE_RED) { fixNegativeRed(child); return true; }
    else if (child.color == RED)
    {
```
Implementing a Red-Black Tree

if (child.left != null && child.left.color == RED) {
    fixDoubleRed(child.left); return true;
} 
if (child.right != null && child.right.color == RED) {
    fixDoubleRed(child.right); return true;
}
return false;

We are left with the negative red removal. In the diagram in the book, we show only one of the two possible situations. In the code, we also need to handle the mirror image.

The implementation is not difficult, just long.

/**
 * Fixes a "negative red" violation.
 * @param negRed the negative red node
 */
private void fixNegativeRed(Node negRed) {
    Node parent = negRed.parent;
    Node child;
    if (parent.left == negRed) {
        Node n1 = negRed.left;
        Node n2 = negRed;
        if (n1.left != null && n1.left.color == RED) {
            fixDoubleRed(n1.left); return true;
        }
        if (n1.right != null && n1.right.color == RED) {
            fixDoubleRed(n1.right); return true;
        }
        return false;
    } 
    if (parent.right == negRed) {
        Node n3 = negRed.right;
        Node n4 = negRed;
        if (n3.left != null && n3.left.color == RED) {
            fixDoubleRed(n3.left); return true;
        }
        if (n3.right != null && n3.right.color == RED) {
            fixDoubleRed(n3.right); return true;
        }
        return false;
    }
    return true;
}
Node n3 = negRed.right;
Node n4 = parent;
Node t1 = n3.left;
Node t2 = n3.right;
Node t3 = n4.right;
n1.color = RED;
n2.color = BLACK;
n4.color = BLACK;

replaceWith(n4, n3);
n3.setLeftChild(n2);
n3.setRightChild(n4);
n2.setLeftChild(n1);
n2.setRightChild(t1);
n4.setLeftChild(t2);
n4.setRightChild(t3);

child = n1;
}
else // Mirror image
{
Node n4 = negRed.right;
Node n3 = negRed;
Node n2 = negRed.left;
Node n1 = parent;
Node t3 = n2.right;
Node t2 = n2.left;
Node t1 = n1.left;
n4.color = RED;
n3.color = BLACK;
n1.color = BLACK;

replaceWith(n1, n2);
n2.setRightChild(n3);
n2.setLeftChild(n1);
n3.setRightChild(n4);
n3.setLeftChild(t3);
n1.setRightChild(t2);
n1.setLeftChild(t1);

child = n4;
}

if (child.left != null && child.left.color == RED)
{
fixDoubleRed(child.left);
}
else if (child.right != null && child.right.color == RED)
{
fixDoubleRed(child.right);
}
Implementing a Red-Black Tree

Simple Tests

With such a complex implementation, it is extremely likely that some errors slipped in somewhere, and it is important to carry out thorough testing.

We can start with the test case used for the binary search tree from the book:

```java
public static void testFromBook()
{
    RedBlackTree t = new RedBlackTree();
    t.add("D");
    t.add("B");
    t.add("A");
    t.add("C");
    t.add("F");
    t.add("E");
    t.add("I");
    t.add("G");
    t.add("H");
    t.add("J");
    t.remove("A"); // Removing leaf
    t.remove("B"); // Removing element with one child
    t.remove("F"); // Removing element with two children
    t.remove("D"); // Removing root
    assertEquals("C E G H I J ", t.toString());
}
```

The `toString` method is just like the `print` method of the binary search tree, but it returns the string instead of printing it.

If this test fails (which it did for the author at the first attempt), it is fairly easy to debug. If it passes, it gives some confidence. But there are so many different configurations that more thorough tests are required.

For a more exhaustive test, we can insert all permutations of the ten letters A – J and check that the resulting tree has the desired contents. Here, we use the permutation generator from Section 13.4.

```java
/**
 * Inserts all permutations of a string into a red-black tree and checks that it contains the strings afterwards.
 * @param letters a string of letters without repetition
 */
public static void insertionTest(String letters)
{
    PermutationGenerator gen = new PermutationGenerator(letters);
    for (String perm : gen.getPermutations())
    {
        RedBlackTree t = new RedBlackTree();
        for (int i = 0; i < perm.length(); i++)
        {
            String s = perm.substring(i, i + 1);
            t.add(s);
        }
        assertEquals(letters, t.toString().remove(" ", ");
    }
}
```

This test runs through $10! = 3,628,800$ permutations, which seems pretty exhaustive. But how do we really know that all possible configurations of red and black nodes have been covered? For example, it seems plausible that all four possible configurations of Figure 21 occur somewhere in these test cases, but how do we know for sure? We take up that question in the next section.
An Advanced Test

In the previous section, we reached the limits of what one can achieve with “black box” testing. For more exhaustive coverage, we need to manufacture red-black trees with all possible patterns of red and black nodes. At first, that seems hopeless. According to Exercise R17.19, there are 435,974,400 red-black trees with black height 2, far too many to generate and test.

Fortunately, we don’t have to test them all. The algorithms for insertion and removal fix up nodes that form a direct path to the root. It is enough to fill this path, and its neighboring elements with all possible color combinations. Let us test the most complex case: removing a black leaf. We allow for two nodes between the leaf and the root.

Along the path to the root, we add siblings that can be red or black. We also add a couple of nodes to allow double-red violations. Each of the seven white nodes will be filled in with red or black, yielding 128 test cases. We also test all mirror images, for a total of 256 test cases.

Of course, if we fill in arbitrary combinations of red and black, the result may not be a red-black tree. First off, we add completely black subtrees to each leaf so that the black height is constant (and equal to the black height of the node to be deleted). Then we remove trees with double-red violations. For the remaining trees, we fill in data values 1, 2, 3, so that we have a binary search tree. Then we remove the target node and check that the tree is still a proper red-black tree and that it contains the required values.

This seems like an ambitious undertaking, but it is better than the alternative—laboriously constructing a set of test cases by hand. It also provides good practice for working with trees.

In order to facilitate this style of testing, the root instance variable and the Node class of the RedBlackTree class are package-visible.
The following method produces the template for testing:

```java
/**
 * Makes a template for testing removal.
 * @return a partially complete red black tree for the test.
 * The node to be removed is black.
 */
private static RedBlackTree removalTestTemplate()
{
    RedBlackTree template = new RedBlackTree();

    /*
    n7
    / \
    n1  n8
    / \
    n0  n3
    / \
    n2*  n5
    / \
    n4  n6
    */
    RedBlackTree.Node[] n = new RedBlackTree.Node[9];
    for (int i = 0; i < n.length; i++) { n[i] = new RedBlackTree.Node(); }
    template.root = n[7];
    n[7].setLeftChild(n[1]);
    n[7].setRightChild(n[8]);
    n[1].setLeftChild(n[0]);
    n[1].setRightChild(n[3]);
    n[3].setLeftChild(n[2]);
    n[3].setRightChild(n[5]);
    n[5].setLeftChild(n[4]);
    n[5].setRightChild(n[6]);
    n[2].color = RedBlackTree.BLACK;
    return template;
}
```

Because each test changes the shape of the tree, we want to make a copy of the template in each test. The following recursive method makes a copy of a tree:

```java
/**
 * Copies all nodes of a red-black tree.
 * @param n the root of a red-black tree
 * @return the root node of a copy of the tree
 */
private static RedBlackTree.Node copy(RedBlackTree.Node n)
{
    if (n == null) { return null; }
    newNode.setLeftChild(copy(n.left));
    newNode.setRightChild(copy(n.right));
    newNode.data = n.data;
    newNode.color = n.color;
    return newNode;
}
```
To make a mirror image instead of a copy, just swap the left and right child:

```java
/**
 * Generates the mirror image of a red black tree.
 * @param n the root of the tree to reflect
 * @return the root of the mirror image of the tree
 */
private static RedBlackTree.Node mirror(RedBlackTree.Node n)
{
    if (n == null) { return null; }
    newNode.setLeftChild(mirror(n.right));
    newNode.setRightChild(mirror(n.left));
    newNode.data = n.data;
    newNode.color = n.color;
    return newNode;
}
```

We want to test all possible combinations of red and black nodes in the template. Each pattern of reds and blacks can be represented as a sequence of zeroes and ones, or a binary number between 0 and $2^n - 1$, where $n$ is the number of nodes to be colored.

```java
for (int k = 0; k < Math.pow(2, nodesToColor); k++)
{
    RedBlackTree.Node[] nodes = . . . // The nodes to be colored;

    // Color with the bit pattern of k
    int bits = k;
    for (RedBlackTree.Node n : nodes)
    {
        n.color = bits % 2;
        bits = bits / 2;
    }

    // Now run a test with this tree
    . . .
}
```

We need to have a helper method to get all nodes of a tree into an array. Here it is:

```java
/**
 * Gets all nodes of a tree in sorted order.
 * @param t a red-black tree
 * @return an array of all nodes in t
 */
private static RedBlackTree.Node[] getNodes(RedBlackTree t)
{
    getNodes(t.root, nodes, 0);
    return nodes;
}
```

```java
/**
 * Gets all nodes of a subtree and fills them into an array.
 * @param n the root of the subtree
 * @param nodes the array into which to place the nodes
 * @param start the offset at which to start placing the nodes
 * @return the number of nodes placed
 */
private static int getNodes(RedBlackTree.Node n, RedBlackTree.Node[] nodes, int start)
{
if (n == null) { return 0; }
int leftFilled = getNodes(n.left, nodes, start);
nodes[start + leftFilled] = n;
int rightFilled = getNodes(n.right, nodes, start + leftFilled + 1);
return leftFilled + 1 + rightFilled;
}

Once the tree has been colored, we need to give it a constant black height. For each leaf, we compute the cost to the root:

```java
/**
 * Computes the cost from a node to a root.
 * @param n a node of a red-black tree
 * @return the number of black nodes between n and the root
 */
private static int costToRoot(RedBlackTree.Node n)
{
    int c = 0;
    while (n != null) { c = c + n.color; n = n.parent; }
    return c;
}
```

If that cost is less than the black height of the node to be removed, we add a full tree of black nodes to make up the difference. This method makes these trees:

```java
/**
 * Makes a full tree of black nodes of a given depth.
 * @param depth the desired depth
 * @return the root node of a full black tree
 */
private static RedBlackTree.Node fullTree(int depth)
{
    if (depth <= 0) { return null; }
    RedBlackTree.Node r = new RedBlackTree.Node();
r.color = RedBlackTree.BLACK;
r.setLeftChild(fullTree(depth - 1));
r.setRightChild(fullTree(depth - 1));
return r;
}
```

This loop adds the full trees to the nodes:

```java
int targetCost = costToRoot(toDelete);
for (RedBlackTree.Node n : nodes)
{
    int cost = targetCost - costToRoot(n);
    if (n.left == null) { n.setLeftChild(fullTree(cost)); }
    if (n.right == null) { n.setRightChild(fullTree(cost)); }
}
```

Now we need to fill the tree with values. Because `getNodes` returns the nodes in sorted order, we just populate them with 0, 1, 2, and so on.

```java
/**
 * Populates this tree with the values 0, 1, 2,....
 * @param t a red-black tree
 * @return the number of nodes in t
 */
private static int populate(RedBlackTree t)
{
    RedBlackTree.Node[] nodes = getNodes(t);
    for (int i = 0; i < nodes.length; i++)
    {
```
The resulting tree has constant black height, but it might still not be a valid red-black tree because it might have double-red violations. We could test just that, but we need to have a general method that tests the red-black properties after the removal. We also want to verify that all the parent and child links are not corrupted. Because removal introduces colors other than red or black (e.g., double-black or negative-red), we want to check that those colors are no longer present after the operation has completed. Specifically, we need to check the following for each subtree with root \( n \):

- The left and right subtree of \( n \) have the same black depth.
- \( n \) must be red or black.
- If \( n \) is red, its parent is not.
- If \( n \) has children, then their parent references must equal \( n \).
- \( n.paren \) is null if and only if \( n \) is the root of the tree.
- The root is black.

Moreover, because fixing double-red and negative-red violations reorders nodes, we will check that the tree is still a binary search tree. This can be tested by visiting the tree in order.

Here are the integrity check methods:

```java
/**
 * Checks whether a red-black tree is valid and throws an exception if not.
 * @param t the tree to test
 */
public static void checkRedBlack(RedBlackTree t)
{
    checkRedBlack(t.root, true);

    // Check that it's a BST
    RedBlackTree.Node[] nodes = getNodes(t);
    for (int i = 0; i < nodes.length - 1; i++)
    {
        if (nodes[i].data.compareTo(nodes[i + 1].data) > 0)
        {
            throw new IllegalStateException(nodes[i].data + " is larger than " + nodes[i + 1].data);
        }
    }
}

/**
 * Checks that the tree with the given node is a red-black tree, and throws an exception if a structural error is found.
 * @param n the root of the subtree to check
 * @param isRoot true if this is the root of the tree
 * @return the black depth of this subtree
 */
private static int checkRedBlack(RedBlackTree.Node n, boolean isRoot)
{
    if (n == null) { return 0; }
    int nleft = checkRedBlack(n.left, false);
    int nright = checkRedBlack(n.right, false);
    if (nleft != nright)
    {
        throw new IllegalStateException("The left and right subtree have different black depth.");
    }
    return nleft + 1;
}
```
throw new IllegalStateException("Left and right children of " + n.data + " have different black depths");
}
if (n.parent == null)
{
    if (!isRoot)
    {
        throw new IllegalStateException(n.data + " is not root and has no parent");
    }
    if (n.color != RedBlackTree.BLACK)
    {
        throw new IllegalStateException("Root " + n.data + " is not black");
    }
}
else
{
    if (isRoot)
    {
        throw new IllegalStateException(n.data + " is root and has a parent");
    }
    if (n.color == RedBlackTree.RED && n.parent.color == RedBlackTree.RED)
    {
        throw new IllegalStateException("Parent of red " + n.data + " is red");
    }
}
if (n.left != null && n.left.parent != n)
{
    throw new IllegalStateException("Left child of " + n.data + " has bad parent link");
}
if (n.right != null && n.right.parent != n)
{
    throw new IllegalStateException("Right child of " + n.data + " has bad parent link");
}
if (n.color != RedBlackTree.RED && n.color != RedBlackTree.BLACK)
{
    throw new IllegalStateException(n.data + " has color " + n.color);
}
return n.color + n.left;
}

public static void assertEquals(Object expected, Object actual)
{
    if (expected == null && actual != null || !expected.equals(actual))
    {
        throw new AssertionError("Expected " + expected + " but found " + actual);
    }
}
Now we have all the pieces together. Here is the complete method for testing removal. Note that the outer loop switches between copying and mirroring, and the inner loop iterates over all red/black colorings.

```java
/**
 * Tests removal, given a template for a tree with a black node that is to be deleted. All other nodes should be given all possible combinations of red and black.
 * @param t the template for the test cases
 */
public static void removalTest(RedBlackTree t)
{
    for (int m = 0; m <= 1; m++)
    {
        int nodesToColor = count(t.root) - 2; // We don't recolor the root or toDelete
        for (int k = 0; k < Math.pow(2, nodesToColor); k++)
        {
            RedBlackTree rb = new RedBlackTree();
            if (m == 0) { rb.root = copy(t.root); }
            else { rb.root = mirror(t.root); }

            RedBlackTree.Node[] nodes = getNodes(rb);
            RedBlackTree.Node toDelete = null;

            // Color with the bit pattern of k
            int bits = k;
            for (RedBlackTree.Node n : nodes)
            {
                if (n == rb.root)
                {
                    n.color = RedBlackTree.BLACK;
                }
                else if (n.color == RedBlackTree.BLACK)
                {
                    toDelete = n;
                }
                else
                {
                    n.color = bits % 2;
                    bits = bits / 2;
                }
            }

            // Add children to make equal costs to null
            int targetCost = costToRoot(toDelete);
            for (RedBlackTree.Node n : nodes)
            {
                int cost = targetCost - costToRoot(n);
                if (n.left == null) { n.setLeftChild(fullTree(cost)); }
                if (n.right == null) { n.setRightChild(fullTree(cost)); }
            }

            int filledSize = populate(rb);
            boolean good = true;
            try { checkRedBlack(rb); }
            catch (IllegalStateException ex) { good = false; }
            if (good)
            {
```
Implementing a Red-Black Tree

Comparable d = toDelete.data;
rb.remove(d);
checkRedBlack(rb);
for (Integer j = 0; j < filledSize; j++)
{
    if (!rb.find(j) & id.equals(j))
    {
        throw new IllegalStateException(j + " deleted");
    }
    if (rb.find(d))
    {
        throw new IllegalStateException(d + " not deleted");
    }
}
}

In our main method, we run all three tests. The last line, which only happens if no exceptions have been thrown, proclaims that the tests passed.

public static void main(String[] args)
{
    testFromBook();
    insertionTest("ABCDEFGHIJ");
    removalTest(removalTestTemplate());
    System.out.println("All tests passed.");
}

See ch17/worked_example_2 in your code folder for the complete program.
CHAPTER 18
GENERIC CLASSES

CHAPTER GOALS
To understand the objective of generic programming
To implement generic classes and methods
To explain the execution of generic methods in the virtual machine
To describe the limitations of generic programming in Java

CHAPTER CONTENTS
18.1 GENERIC CLASSES AND TYPE PARAMETERS 824
18.2 IMPLEMENTING GENERIC TYPES 825
SYN Declaring a Generic Class 826
18.3 GENERIC METHODS 829
SYN Declaring a Generic Method 830
18.4 CONSTRAINTING TYPE PARAMETERS 831
CE1 Genericity and Inheritance 833
CE2 The Array Store Exception 833
ST1 Wildcard Types 834
18.5 TYPE ERASURE 835
CE3 Using Generic Types in a Static Context 838
ST2 Reflection 838
WE1 Making a Generic Binary Search Tree Class
In the supermarket, a generic product can be sourced from multiple suppliers. In computer science, generic programming involves the design and implementation of data structures and algorithms that work for multiple types. You have already seen the generic ArrayList class that can be used to collect elements of arbitrary types. In this chapter, you will learn how to implement your own generic classes and methods.

18.1 Generic Classes and Type Parameters

Generic programming is the creation of programming constructs that can be used with many different types. For example, the Java library programmers who implemented the ArrayList class used the technique of generic programming. As a result, you can form array lists that collect elements of different types, such as ArrayList<String>, ArrayList<BankAccount>, and so on.

The LinkedList class that we implemented in Section 16.1 is also an example of generic programming—you can store objects of any class inside a LinkedList. That LinkedList class achieves genericity by using inheritance. It uses references of type Object and is therefore capable of storing objects of any class. For example, you can add elements of type String because the String class extends Object. In contrast, the ArrayList and LinkedList classes from the standard Java library are generic classes. Each of these classes has a type parameter for specifying the type of its elements. For example, an ArrayList<String> stores String elements.

When declaring a generic class, you supply a variable for each type parameter. For example, the standard library declares the class ArrayList<E>, where E is the type variable that denotes the element type. You use the same variable in the declaration of the methods, whenever you need to refer to that type. For example, the ArrayList<E> class declares methods

```java
public void add(E element)
public E get(int index)
```

You could use another name, such as ElementType, instead of E. However, it is customary to use short, uppercase names for type variables.

In order to use a generic class, you need to instantiate the type parameter, that is, supply an actual type. You can supply any class or interface type, for example

```java
ArrayList<BankAccount>
ArrayList<Measurable>
```

However, you cannot substitute any of the eight primitive types for a type parameter. It would be an error to declare an ArrayList<double>. Use the corresponding wrapper class instead, such as ArrayList<Double>.

When you instantiate a generic class, the type that you supply replaces all occurrences of the type variable in the declaration of the class. For example, the add method for ArrayList<BankAccount> has the type variable E replaced with the type BankAccount:

```java
public void add(BankAccount element)
```

Contrast that with the add method of the LinkedList class in Chapter 16:

```java
public void add(Object element)
```
The add method of the generic `ArrayList` class is safer. It is impossible to add a `String` object into an `ArrayList<BankAccount>`, but you can accidentally add a `String` into a `LinkedList` that is intended to hold bank accounts:

```java
ArrayList<BankAccount> accounts1 = new ArrayList<>();
LinkedList accounts2 = new LinkedList(); // Should hold BankAccount objects
accounts1.add("my savings"); // Compile-time error
accounts2.addFirst("my savings"); // Not detected at compile time
```

The latter will result in a class cast exception when some other part of the code retrieves the string, believing it to be a bank account:

```java
BankAccount account = (BankAccount) accounts2.getFirst(); // Run-time error
```

Code that uses the generic `ArrayList` class is also easier to read. When you spot an `ArrayList<BankAccount>`, you know right away that it must contain bank accounts. When you see a `LinkedList`, you have to study the code to find out what it contains.

In Chapters 16 and 17, we used inheritance to implement generic linked lists, hash tables, and binary trees, because you were already familiar with the concept of inheritance. Using type parameters requires new syntax and additional techniques—those are the topic of this chapter.

1. The standard library provides a class `HashMap<K, V>` with key type `K` and value type `V`. Declare a hash map that maps strings to integers.
2. The binary search tree class in Chapter 17 is an example of generic programming because you can use it with any classes that implement the `Comparable` interface. Does it achieve genericity through inheritance or type parameters?
3. Does the following code contain an error? If so, is it a compile-time or run-time error?
   ```java
   ArrayList<Integer> a = new ArrayList<>();
   String s = a.get(0);
   ```
4. Does the following code contain an error? If so, is it a compile-time or run-time error?
   ```java
   ArrayList<Double> a = new ArrayList<>();
   a.add(3);
   ```
5. Does the following code contain an error? If so, is it a compile-time or run-time error?
   ```java
   LinkedList a = new LinkedList();
   a.addFirst("3.14");
   double x = (Double) a.removeFirst();
   ```

Practice It Now you can try these exercises at the end of the chapter: R18.5, R18.6, R18.7.

18.2 Implementing Generic Types

In this section, you will learn how to implement your own generic classes. We will write a very simple generic class that stores `pairs` of objects, each of which can have an arbitrary type. For example,

```java
Pair<String, Integer> result = new Pair<"Harry Morgan", 1729>;
```
Syntax 18.1 Declaring a Generic Class

```java
Syntax  modifier  class  GenericClassName<TypeVariable1, TypeVariable2, . . .>
          { instance  variables  constructors  methods
          }
```

Supply a variable for each type parameter.

```java
public class Pair<T, S>
{
    private T first;
    private S second;
    . . .
    public T getFirst() { return first; }
    . . .
}
```

A method with a variable return type

Instance variables with a variable data type

The `getFirst` and `getSecond` methods retrieve the first and second values of the pair:

```java
String name = result.getFirst();
Integer number = result.getSecond();
```

This class can be useful when you implement a method that computes two values at the same time. A method cannot simultaneously return a `String` and an `Integer`, but it can return a single object of type `Pair<String, Integer>`.

The generic `Pair` class requires two type parameters, one for the type of the first element and one for the type of the second element.

We need to choose variables for the type parameters. It is considered good form to use short uppercase names for type variables, such as those in the following table:

<table>
<thead>
<tr>
<th>Type Variable</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>E</td>
<td>Element type in a collection</td>
</tr>
<tr>
<td>K</td>
<td>Key type in a map</td>
</tr>
<tr>
<td>V</td>
<td>Value type in a map</td>
</tr>
<tr>
<td>T</td>
<td>General type</td>
</tr>
<tr>
<td>S, U</td>
<td>Additional general types</td>
</tr>
</tbody>
</table>

You place the type variables for a generic class after the class name, enclosed in angle brackets (`<` and `> `):

```java
public class Pair<T, S>
```

When you declare the instance variables and methods of the `Pair` class, use the variable `T` for the first element type and `S` for the second element type:

```java
public class Pair<T, S>
{ . . .
```
Implementing Generic Types

private T first;
private S second;

public Pair(T firstElement, S secondElement)
{
    first = firstElement;
    second = secondElement;
}
public T getFirst() { return first; }
public S getSecond() { return second; }

Some people find it simpler to start out with a regular class, choosing some actual types instead of the type parameters. For example,

public class Pair // Here we start out with a pair of String and Integer values
{
    private String first;
    private Integer second;
    public Pair(String firstElement, Integer secondElement)
    {
        first = firstElement;
        second = secondElement;
    }
    public String getFirst() { return first; }
    public Integer getSecond() { return second; }
}

Now it is an easy matter to replace all String types with the type variable T and all Integer types with the type variable S.

This completes the declaration of the generic Pair class. It is ready to use whenever you need to form a pair of two objects of arbitrary types.

The following sample program shows how to make use of a Pair for returning two values from a method.

section_2/Pair.java

```java
/**
 * This class collects a pair of elements of different types.
 */
public class Pair<T, S>
{
    private T first;
    private S second;

    /**
     * Constructs a pair containing two given elements.
     * @param firstElement the first element
     * @param secondElement the second element
     */
    public Pair(T firstElement, S secondElement)
    {
        first = firstElement;
        second = secondElement;
    }
```

Use type parameters for the types of generic instance variables, method parameter variables, and return values.

```java
```
828

Chapter 18 Generic Classes
20
21
22
23
24
25
26
27
28
29
30
31
32
33

/**

Gets the first element of this pair.
@return the first element

*/
public T getFirst() { return first; }
/**

Gets the second element of this pair.
@return the second element

*/
public S getSecond() { return second; }
public String toString() { return "(" + first + ", " + second + ")"; }
}

section_2/PairDemo.java
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
32
33
34

public class PairDemo
{
public static void main(String[] args)
{
String[] names = { "Tom", "Diana", "Harry" };
Pair<String, Integer> result = firstContaining(names, "a");
System.out.println(result.getFirst());
System.out.println("Expected: Diana");
System.out.println(result.getSecond());
System.out.println("Expected: 1");
}
/**

Gets the first String containing a given string, together
with its index.
@param strings an array of strings
@param sub a string
@return a pair (strings[i], i) where strings[i] is the first
strings[i] containing str, or a pair (null, -1) if there is no
match.

*/
public static Pair<String, Integer> firstContaining(
String[] strings, String sub)
{
for (int i = 0; i < strings.length; i++)
{
if (strings[i].contains(sub))
{
return new Pair<>(strings[i], i);
}
}
return new Pair<>(null, -1);
}
}

Program Run
Diana
Expected: Diana
1
Expected: 1


6. How would you use the generic Pair class to construct a pair of strings "Hello" and "World"?

7. How would you use the generic Pair class to construct a pair containing "Hello" and 1729?

8. What is the difference between an ArrayList<Pair<String, Integer>> and a Pair<ArrayList<String>, Integer>?

9. Write a method roots with a Double parameter variable x that returns both the positive and negative square root of x if x ≥ 0 or null otherwise.

10. How would you implement a class Triple that collects three values of arbitrary types?

Practice It Now you can try these exercises at the end of the chapter: E18.1, E18.2, E18.7.

18.3 Generic Methods

A generic method is a method with a type parameter. Such a method can occur in a class that in itself is not generic. You can think of it as a template for a set of methods that differ only by one or more types. For example, we may want to declare a method that can print an array of any type:

```
public class ArrayUtil
{
    /**
     * Prints all elements in an array.
     * @param a the array to print
     */
    public static <T> void print(T[] a)
    {
        . . .
    }
}
```

As described in the previous section, it is often easier to see how to implement a generic method by starting with a concrete example. This method prints the elements in an array of strings:

```
public class ArrayUtil
{
    public static void print(String[] a)
    {
        for (String e : a)
        {
            System.out.print(e + " ");
        }
        System.out.println();
    }
}
```
Syntax 18.2  Declaring a Generic Method

```
Syntax  modifiers <TypeVariable₁, TypeVariable₂, . . .> returnType methodName(parameters)  
{  
    body  
}
```

In order to make the method into a generic method, replace `String` with a type variable, say `E`, to denote the element type of the array. Add a type parameter list, enclosed in angle brackets, between the modifiers (`public static`) and the return type (`void`):

```
public static <E> void print(E[] a)  
{  
    for (E e : a)  
    {  
        System.out.print(e + " ");  
    }  
    System.out.println();  
}
```

When you call the generic method, you need not specify which type to use for the type parameter. (In this regard, generic methods differ from generic classes.) Simply call the method with appropriate arguments, and the compiler will match up the type parameters with the argument types. For example, consider this method call:

```
Rectangle[] rectangles = . . .;  
ArrayUtil.print(rectangles);  
```

The type of the `rectangles` argument is `Rectangle[]`, and the type of the parameter variable is `E[]`. The compiler deduces that `E` is `Rectangle`.

This particular generic method is a static method in an ordinary class. You can also declare generic methods that are not static. You can even have generic methods in generic classes.

As with generic classes, you cannot replace type parameters with primitive types. The generic `print` method can print arrays of any type except the eight primitive types. For example, you cannot use the generic `print` method to print an array of type `int[]`. That is not a major problem. Simply implement a `print(int[] a)` method in addition to the generic `print` method.

**Self Check**

11. Exactly what does the generic `print` method print when you pass an array of `BankAccount` objects containing two bank accounts with zero balances?

12. Is the `getFirst` method of the `Pair` class a generic method?
13. Consider this `fill` method:
   ```java
   public static <T> void fill(List<T> lst, T value)
   {  
      for (int i = 0; i < lst.size(); i++) { lst.set(i, value); }
   }
   ```
   If you have an array list
   ```java
   ArrayList<String> a = new ArrayList<>(10);
   ```
   how do you fill it with ten "*"?

14. What happens if you pass 42 instead of "*" to the `fill` method?

15. Consider this `fill` method:
   ```java
   public static <T> fill(T[] arr, T value)
   {  
      for (int i = 0; i < arr.length; i++) { arr[i] = value; }
   }
   ```
   What happens when you execute the following statements?
   ```java
   String[] a = new String[10];
   fill(a, 42);
   ```

**Practice It**  
Now you can try these exercises at the end of the chapter: E18.3, E18.15.

### 18.4 Constraining Type Parameters

It is often necessary to specify what types can be used in a generic class or method. Consider a generic method that finds the average of the values in an array list of objects. How can you compute averages when you know nothing about the element type? You need to have a mechanism for measuring the elements. In Chapter 10, we designed an interface for that purpose:

```java
public interface Measurable
{  
   double getMeasure();
}
```

We can constrain the type of the elements, requiring that the type implement the `Measurable` type. In Java, this is achieved by adding the clause extends `Measurable` after the type parameter:

```java
public static <E extends Measurable> double average(ArrayList<E> objects)
```

This means, “E or one of its superclasses extends or implements `Measurable`”. In this situation, we say that `E` is a subtype of the `Measurable` type.

Here is the complete average method:

```java
public static <E extends Measurable> double average(ArrayList<E> objects)
{  
   if (objects.size() == 0) { return 0; }
   double sum = 0;
   for (E obj : objects)
   {
```
sum = sum + obj.getMeasure();
return sum / objects.size();

Note the call obj.getMeasure(). The variable obj has type E, and E is a subtype of Measurable. Therefore, we know that it is legal to apply the getMeasure method to obj.

If the BankAccount class implements the Measurable interface, then you can call the average method with an array list of BankAccount objects. But you cannot compute the average of an array list of strings because the String class does not implement the Measurable interface.

Now consider the task of finding the minimum in an array list. We can return the element with the smallest measure (see Self Check 17). However, the Measurable interface was created for this book and is not widely used. Instead, we will use the Comparable interface type that many classes implement. The Comparable interface is itself a generic type. The type parameter specifies the type of the parameter variable of the compareTo method:

```java
public interface Comparable<T>
{
    int compareTo(T other);
}
```

For example, String implements Comparable<String>. You can compare strings with other strings, but not with objects of different classes.

If the array list has elements of type E, then we want to require that E implements Comparable<E>. Here is the method:

```java
public static <E extends Comparable<E>> E min(ArrayList<E> objects)
{
    E smallest = objects.get(0);
    for (int i = 1; i < objects.size(); i++)
    {
        E obj = objects.get(i);
        if (obj.compareTo(smallest) < 0)
        {
            smallest = obj;
        }
    }
    return smallest;
}
```

Because of the type constraint, we know that obj has a method

```java
int compareTo(E other)
```

Therefore, the call

```java
obj.compareTo(smallest)
```

is valid.

Very occasionally, you need to supply two or more type bounds. Then you separate them with the & character, for example

```java
<E extends Comparable<E> & Measurable>
```

The extends reserved word, when applied to type parameters, actually means “extends or implements”. The bounds can be either classes or interfaces, and the type parameter can be replaced with a class or interface type.
16. How would you constrain the type parameter for a generic BinarySearchTree class?

17. Modify the min method to compute the minimum of an array list of elements that implements the Measurable interface.

18. Could we have declared the min method of Self Check 17 without type parameters, like this?
   
   ```java
   public static Measurable min(ArrayList<Measurable> a)
   ```

19. Could we have declared the min method of Self Check 17 without type parameters for arrays, like this?
   
   ```java
   public static Measurable min(Measurable[] a)
   ```

20. How would you implement the generic average method for arrays?

21. Is it necessary to use a generic average method for arrays of measurable objects?

**Practice It**

Now you can try these exercises at the end of the chapter: E18.5, E18.16.

---

**Genericity and Inheritance**

If SavingsAccount is a subclass of BankAccount, is ArrayList<SavingsAccount> a subclass of ArrayList<BankAccount>? Perhaps surprisingly, it is not. Inheritance of type parameters does not lead to inheritance of generic classes. There is no relationship between ArrayList<SavingsAccount> and ArrayList<BankAccount>.

This restriction is necessary for type checking. Without the restriction, it would be possible to add objects of unrelated types to a collection. Suppose it was possible to assign an ArrayList<SavingsAccount> object to a variable of type ArrayList<BankAccount>:

```java
ArrayList<SavingsAccount> savingsAccounts = new ArrayList<>();
ArrayList<BankAccount> bankAccounts = savingsAccounts;

// Not legal, but suppose it was
BankAccount harrysChecking = new CheckingAccount();
// CheckingAccount is another subclass of BankAccount
bankAccounts.add(harrysChecking); // OK—can add BankAccount object
```

But bankAccounts and savingsAccounts refer to the same array list! If the assignment was legal, we would be able to add a CheckingAccount into an ArrayList<SavingsAccount>.

In many situations, this limitation can be overcome by using wildcards—see Special Topic 18.1.

---

**The Array Store Exception**

In Common Error 18.1, you saw that one cannot assign a subclass list to a superclass list. For example, an ArrayList<SavingsAccount> cannot be used where an ArrayList<BankAccount> is expected.

This is surprising, because you *can* perform the equivalent assignment with arrays. For example,

```java
SavingsAccount[] savingsAccounts = new SavingsAccount[10];
BankAccount bankAccounts = savingsAccounts; // Legal
```

But there was a reason the assignment wasn’t legal for array lists—it would have allowed storing a CheckingAccount into savingsAccounts.
Let’s try that with arrays:

```java
BankAccount harrysChecking = new CheckingAccount();
bankAccounts[0] = harrysChecking; // Throws ArrayStoreException
```

This code compiles. The object harrysChecking is a CheckingAccount and hence a BankAccount. But bankAccounts and savingsAccounts are references to the same array—an array of type SavingsAccount[]. When the program runs, that array refuses to store a CheckingAccount, and throws an ArrayStoreException.

Both ArrayList and arrays avoid the type error, but they do it in different ways. The ArrayList class avoids it at compile time, and arrays avoid it at run time. Generally, we prefer a compile-time error notification, but the cost is steep, as you can see from Special Topic 18.1. It is a lot of work to tell the compiler precisely which conversions should be permitted.

### Wildcard Types

Wildcard types were invented for this purpose. There are three kinds of wildcard types:

<table>
<thead>
<tr>
<th>Name</th>
<th>Syntax</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>Wildcard with lower bound</td>
<td>? extends B</td>
<td>Any subtype of B</td>
</tr>
<tr>
<td>Wildcard with upper bound</td>
<td>? super B</td>
<td>Any supertype of B</td>
</tr>
<tr>
<td>Unbounded wildcard</td>
<td>?</td>
<td>Any type</td>
</tr>
</tbody>
</table>

A wildcard type is a type that can remain unknown. For example, we can declare the following method in the LinkedList class:

```java
public void addAll(LinkedList<? extends E> other)
{
    ListIterator<E> iter = other.listIterator();
    while (iter.hasNext())
    {
        add(iter.next());
    }
}
```

The method adds all elements of other to the end of the linked list.

The addAll method doesn’t require a specific type for the element type of other. Instead, it allows you to use any type that is a subtype of E. For example, you can use addAll to add a LinkedList<SavingsAccount> to a LinkedList<BankAccount>.

To see a wildcard with a super bound, have another look at the min method:

```java
public static <E extends Comparable<? super E>> E min(ArrayList<E> objects)
```

However, this bound is too restrictive. Suppose the BankAccount class implements Comparable<BankAccount>. Then the subclass SavingsAccount also implements Comparable<BankAccount> and not Comparable<SavingsAccount>. If you want to use the min method with a SavingsAccount array list, then the type parameter of the Comparable interface should be any supertype of the array list’s element type:

```java
public static <E extends Comparable<? super E>> E min(ArrayList<E> objects)
```

Here is an example of an unbounded wildcard. The Collections class declares a method

```java
public static void reverse(List<?> list)
```
You can think of that declaration as a shorthand for

```java
public static <T> void reverse(List<T> list)
```

Common Error 18.2 compares this limitation with the seemingly more permissive behavior of arrays in Java.

## 18.5 Type Erasure

Because generic types are a fairly recent addition to the Java language, the virtual machine that executes Java programs does not work with generic classes or methods. Instead, type parameters are “erased”, that is, they are replaced with ordinary Java types. Each type parameter is replaced with its bound, or with `Object` if it is not bounded.

For example, the generic class `Pair<T, S>` turns into the following raw class:

```java
public class Pair
{
    private Object first;
    private Object second;

    public Pair(Object firstElement, Object secondElement)
    {
        first = firstElement;
        second = secondElement;
    }

    public Object getFirst() { return first; }
    public Object getSecond() { return second; }
}
```

As you can see, the type parameters `T` and `S` have been replaced by `Object`. The result is an ordinary class.

The same process is applied to generic methods. Consider this method:

```java
public static <E extends Measurable> E min(E[] objects)
{
    E smallest = objects[0];
    for (int i = 1; i < objects.length; i++)
    {
        E obj = objects[i];
        if (obj.getMeasure() < smallest.getMeasure())
        {
            smallest = obj;
        }
    }
    return smallest;
}
```

In the Java virtual machine, generic types are erased.
When erasing the type parameter, it is replaced with its bound, the `Measurable` interface:

```java
public static Measurable min(Measurable[] objects)
{
    Measurable smallest = objects[0];
    for (int i = 1; i < objects.length; i++)
    {
        Measurable obj = objects[i];
        if (obj.getMeasure() < smallest.getMeasure())
        {
            smallest = obj;
        }
    }
    return smallest;
}
```

Knowing about type erasure helps you understand the limitations of Java generics. For example, you cannot construct new objects of a generic type. The following method, which tries to fill an array with copies of default objects, would be wrong:

```java
public static <E> void fillWithDefaults(E[] a)
{
    for (int i = 0; i < a.length; i++)
    {
        a[i] = new E(); // Error
    }
}
```

To see why this is a problem, carry out the type erasure process, as if you were the compiler:

```java
public static void fillWithDefaults(Object[] a)
{
    for (int i = 0; i < a.length; i++)
    {
        a[i] = new Object(); // Not useful
    }
}
```

Of course, if you start out with a `Rectangle[]` array, you don’t want it to be filled with `Object` instances. But that’s what the code would do after erasing types.

In situations such as this one, the compiler will report an error. You then need to come up with another mechanism for solving your problem. In this particular example, you can supply a default object:

```java
public static <E> void fill(E[] a, E defaultValue)
{
    for (int i = 0; i < a.length; i++)
    {
        a[i] = defaultValue;
    }
}
```

Similarly, you cannot construct an array of a generic type:

```java
public class Stack<E>
{
    private E[] elements;
    ...;
    public Stack()
    {
        elements = new E[MAX_SIZE]; // Error
    }
}
```
Because the array construction expression `new E[]` would be erased to `new Object[]`, the compiler disallows it. A remedy is to use an array list instead:

```java
public class Stack<E>
{
    private ArrayList<E> elements;
    
    public Stack()
    {
        elements = new ArrayList<>(); // OK
    }
}
```

Another solution is to use an array of objects and provide a cast when reading elements from the array:

```java
public class Stack<E>
{
    private Object[] elements;
    private int currentSize;
    
    public Stack()
    {
        elements = new Object[MAX_SIZE]; // OK
    }
    
    public E pop()
    {
        size--;
        return (E) elements[currentSize];
    }
}
```

The cast `(E)` generates a warning because it cannot be checked at run time.

These limitations are frankly awkward. It is hoped that a future version of Java will no longer erase types so that the current restrictions due to erasure can be lifted.

---

**SELF CHECK**

22. Suppose we want to eliminate the type bound in the `min` method of Section 18.5, by declaring the parameter variable as an array of `Comparable<E>` objects. Why doesn’t this work?

23. What is the erasure of the `print` method in Section 18.3?

24. Could the `Stack` example be implemented as follows?

```java
public class Stack<E>
{
    private E[] elements;
    
    public Stack()
    {
        elements = (E[]) new Object[MAX_SIZE];
    }
}
```
25. The `ArrayList<E>` class has a method
   `Object[] toArray()`
   Why doesn't the method return an `E[]`?

26. The `ArrayList<E>` class has a second method
   `E[] toArray(E[] a)`
   Why can this method return an array of type `E[]`? (*Hint: Special Topic 18.2.*)

27. Why can’t the method
   `static <T> T[] copyOf(T[] original, int newLength)`
   be implemented without reflection?

**Practice It** Now you can try these exercises at the end of the chapter: R18.12, R18.15, E18.18.

---

**Using Generic Types in a Static Context**

You cannot use type parameters to declare static variables, static methods, or static inner classes. For example, the following would be illegal:

```java
public class LinkedList<E>
{
    private static E defaultValue; // Error
    ...
    public static List<E> replicate(E value, int n) { ... } // Error
    private static class Node { public E data; public Node next; } // Error
}
```

In the case of static variables, this restriction is very sensible. After the generic types are erased, there is only a single variable `LinkedList.defaultValue`, whereas the static variable declaration gives the false impression that there is a separate variable for each `LinkedList<E>`.

For static methods and inner classes, there is an easy workaround; simply add a type parameter:

```java
public class LinkedList<E>
{
    ...
    public static <T> List<T> replicate(T value, int n) { ... } // OK
    private static class Node<T> { public T data; public Node<T> next; } // OK
}
```

---

**Reflection**

As you have seen, type erasure makes it impossible for a generic method to construct a generic array. There is an advanced technique called **reflection** that you can sometimes use to overcome this limitation. Reflection lets you work with classes in a running program.

In Java, the virtual machine keeps a `Class` object for each class that has been loaded. That object has information about each class, as well as methods to construct new objects of the class.

Given an object, you can get its class object by calling `getClass`:

```java
Class objsClass = obj.getClass();
```
You can then make a new instance of that class by calling the newInstance method:

```java
Object newObj = objsClass.newInstance();
```

This method throws an exception if it cannot access a constructor with no arguments.

Given an array, you can get the type of the elements this way:

```java
Class arrayClass = array.getClass();
Class elementClass = arrayClass.getComponentType();
```

If you want to create a new array, use the Array.newInstance method:

```java
Object[] newArray = Array.newInstance(elementClass, length);
```

Using these methods, you can implement the fillWithDefaults method:

```java
public static <E> void fillWithDefaults(E[] a)
{
    Class arrayClass = a.getClass();
    Class elementClass = arrayClass.getComponentType();
    try
    {
        for (int i = 0; i < a.length; i++)
        {
            a[i] = elementClass.newInstance();
        }
    }
    catch (. . .) { . . . }
}
```

Note that we must ask for the element type of a. It does no good asking for a[0].getClass. The array might have length 0, or a[0] might be null, or a[0] might be an instance of a subclass of E.

Here is another example. The Arrays class implements a method

```java
static <T> T[] copyOf(T[] original, int newLength)
```

That method can’t simply call

```java
T[] result = new T[newLength]; // Error
```

Instead, it must construct a new array with the same element type as the original:

```java
Class arrayClass = original.getClass();
Class elementClass = arrayClass.getComponentType();
T[] newArray = (T[]) Array.newInstance(elementClass, newLength);
```

For this technique to work, you must have an element or array of the desired type. You couldn’t use it to build a Stack<E> that uses an E[] array because the stack starts out empty.

---

**WORKED EXAMPLE 18.1**

**Making a Generic Binary Search Tree Class**

Learn how to turn the binary search tree class from Chapter 17 into a generic BinarySearchTree<E> class that stores elements of type E. Go to wiley.com/go/bjeo6examples and download Worked Example 18.1.
Chapter 18  Generic Classes

Describe generic classes and type parameters.

- In Java, generic programming can be achieved with inheritance or with type parameters.
- A generic class has one or more type parameters.
- Type parameters can be instantiated with class or interface types.
- Type parameters make generic code safer and easier to read.

Implement generic classes and interfaces.

- Type variables of a generic class follow the class name and are enclosed in angle brackets.
- Use type parameters for the types of generic instance variables, method parameter variables, and return values.

Implement generic methods.

- A generic method is a method with a type parameter.
- Supply the type parameters of a generic method between the modifiers and the method return type.
- When calling a generic method, you need not instantiate the type parameters.

Specify constraints on type parameters.

- Type parameters can be constrained with bounds.

Recognize how erasure of type parameters places limitations on generic programming in Java.

- The virtual machine erases type parameters, replacing them with their bounds or Objects.
- You cannot construct objects or arrays of a generic type.

REVIEW EXERCISES

- **R18.1** What is a type parameter?
- **R18.2** What is the difference between a generic class and an ordinary class?
- **R18.3** What is the difference between a generic class and a generic method?
- **R18.4** Why is it necessary to provide type arguments when instantiating a generic class, but not when invoking an instantiation of a generic method?
Practice Exercises

- **R18.5** Find an example of a non-static generic method in the standard Java library.
- **R18.6** Find four examples of a generic class with two type parameters in the standard Java library.
- **R18.7** Find an example of a generic class in the standard library that is not a collection class.
- **R18.8** Why is a bound required for the type parameter \( T \) in the following method?
  \[
  \text{int } \text{binarySearch}(T[] a, T key)
  \]
- **R18.9** Why is a bound not required for the type parameter \( E \) in the \texttt{HashSet<E>} class?
- **R18.10** What is an \texttt{ArrayList<Pair<T, T>>}?
- **R18.11** Explain the type bounds of the following method of the \texttt{Collections} class.
  
  \[
  \text{public static } \langle T \text{ extends Comparable? super } T \rangle \text{ void sort(List<T> a)}
  \]
  Why doesn't \( T \text{ extends Comparable} \) or \( T \text{ extends Comparable<T>} \) suffice?
- **R18.12** What happens when you pass an \texttt{ArrayList<String>} to a method with an \texttt{ArrayList} parameter variable? Try it out and explain.
- **R18.13** What happens when you pass an \texttt{ArrayList<String>} to a method with an \texttt{ArrayList} parameter variable, and the method stores an object of type \texttt{BankAccount} into the array list? Try it out and explain.
- **R18.14** What is the result of the following test?
  
  \[
  \text{ArrayList<BankAccount> accounts = new ArrayList<>();}
  \]
  \[
  \text{if (accounts instanceof ArrayList<String>) . . .}
  \]
  Try it out and explain.
- **R18.15** The \texttt{ArrayList<E>} class in the standard Java library must manage an array of objects of type \( E \), yet it is not legal to construct a generic array of type \( E[] \) in Java. Locate the implementation of the \texttt{ArrayList} class in the library source code that is a part of the JDK. Explain how this problem is overcome.

**PRACTICE EXERCISES**

- **E18.1** Modify the generic \texttt{Pair} class so that both values have the same type.
- **E18.2** Add a method \texttt{swap} to the \texttt{Pair} class of Exercise E18.1 that swaps the first and second elements of the pair.
- **E18.3** Implement a static generic method \texttt{PairUtil.swap} whose argument is a \texttt{Pair} object, using the generic class declared in Section 18.2. The method should return a new pair, with the first and second element swapped.
- **E18.4** Implement a static generic method that, given a \texttt{Map<K, V>}, yields a \texttt{List<Pair<K, V>>> of the key/value pairs in the map.
- **E18.5** Implement a generic version of the binary search algorithm.
- **E18.6** Implement a generic version of the selection sort algorithm.
- **E18.7** Implement a generic version of the merge sort algorithm. Your program should compile without warnings.
- **E18.8** Implement a generic version of the \texttt{LinkedList} class of Chapter 16.
**E18.9** Turn the `HashSet` implementation of Chapter 16 into a generic class. Use an array list instead of an array to store the buckets.

**E18.10** Provide suitable `hashCode` and `equals` methods for the `Pair` class of Section 18.2 and implement a `HashMap` class, using a `HashSet<Pair<K, V>>`.

**E18.11** Implement a generic version of the permutation generator in Section 13.4. Generate all permutations of a `List<E>`.

**E18.12** Write a generic static method `print` that prints the elements of any object that implements the `Iterable<E>` interface. The elements should be separated by commas. Place your method into an appropriate utility class.

**E18.13** Turn the `MinHeap` class of Chapter 17 into a generic class. As with the `TreeSet` class of the standard library, allow a `Comparator` to compare elements. If no comparator is supplied, assume that the element type implements the `Comparable` interface.

**E18.14** Make the `Measurer` interface from Chapter 10 into a generic class. Provide a static method `T max(T[] values, Measurer<T> meas)`.

**E18.15** Provide a static method `void append(ArrayList<T> a, ArrayList<T> b)` that appends the elements of `b` to `a`.

**E18.16** Modify the method of Exercise E18.15 so that the second array list can contain elements of a subclass. For example, if `people` is an `ArrayList<Person>` and `students` is an `ArrayList<Student>`, then `append(people, students)` should compile but `append(students, people)` should not.

**E18.17** Modify the method of Exercise E18.15 so that it leaves the first array list unchanged and returns a new array list containing the elements of both array lists.

**E18.18** Modify the method of Exercise E18.17 so that it receives and returns arrays, not array lists. Hint: `Arrays.copyOf`.

**E18.19** Provide a static method that reverses the elements of a generic array list.

**E18.20** Provide a static method that returns the reverse of a generic array list, without modifying the original list.

**E18.21** Provide a static method that checks whether a generic array list is a palindrome; that is, whether the values at index `i` and `n - 1 - i` are equal to each other, where `n` is the size of the array list.

**E18.22** Provide a static method that checks whether the elements of a generic array list are in increasing order. The elements must be comparable.

**Programming Projects**

**P18.1** Write a static generic method `PairUtil.minmax` that computes the minimum and maximum elements of an array of type `T` and returns a pair containing the minimum and maximum value. Require that the array elements implement the `Measurable` interface of Chapter 10.

**P18.2** Repeat Exercise P18.1, but require that the array elements implement the `Comparable` interface.
Repeat Exercise P18.2, but refine the bound of the type parameter to extend the generic Comparable type.

Make the Measurable interface from Chapter 10 into a generic class. Provide a static method that returns the largest element of an ArrayList, provided that the elements are instances of Measurable<T>. Be sure to return a value of type T.

Enhance Exercise P18.4 so that the elements of the ArrayList can implement Measurable<U> for appropriate types U.

Write a static generic method
\[
\text{filter}({\text{List}}<T> \text{ values}, \text{Predicate}? \text{ super } T > p)
\]
that returns a list of all values for which the predicate returns true. Demonstrate how to use this method by getting a list of all strings with length greater than ten from a given list of strings. Use a lambda expression (see Java 8 Note 10.4).

Write a static generic method
\[
\text{map}({\text{List}}<T> \text{ values}, \text{Function}<T, R> f)
\]
that returns a list of the values returned by the function when called with arguments in the values list.

Write a static generic method
\[
\text{map}({\text{List}}<T> \text{ values}, \text{Function}<T, R> f)
\]
that returns a list of pairs (v, f.apply(v)), where v ranges over the given values.

1. HashMap<String, Integer>
2. It uses inheritance.
3. This is a compile-time error. You cannot assign the integer expression a.get(0) to a string.
4. This is a compile-time error. The compiler won’t convert 3 to a Double. Remedy: Call a.add(3.0).
5. This is a run-time error. a.removeFirst() yields a String that cannot be converted into a Double. Remedy: Call a.addFirst(3.14);
6. new Pair<>("Hello", "World")
7. new Pair<>("Hello", 1729)
8. An ArrayList<Pair<String, Integer>> contains multiple pairs, for example [(Tom, 1), (Harry, 3)]. A Pair<ArrayList<String>, Integer> contains a list of strings and a single integer, such as (([Tom, Harry], 1).
9. public static Pair<Double, Double> roots(Double x)
\[
\begin{align*}
\text{if } (x \geq 0) \\
&\quad \text{double } r = \text{Math.sqrt}(x); \\
&\quad \text{return new Pair<}(r, -r); \\
\text{else } \text{return null; }
\end{align*}
\]
10. You have three type parameters: Triple<T, S, U>. Add an instance variable U third, a constructor argument for initializing it, and a method U getThird() for returning it.
11. The output depends on the implementation of the toString method in the BankAccount class.
12. No—the method has no type parameters. It is an ordinary method in a generic class.
13. fill(a, ""s")
14. You get a compile-time error. An integer cannot be converted to a string.
15. You get a run-time error. Unfortunately, the call compiles, with T = Object. This choice is justified because a String[] array is convertible.
to an Object[] array, and 42 becomes new Integer(42), which is convertible to an Object. But when the program tries to store an Integer in the String[] array, an exception is thrown.

16. public class BinarySearchTree<E extends Comparable<E>>
or, if you read Special Topic 18.1,
public class BinarySearchTree<E extends Comparable<? super E>>

17. public static <E extends Measurable> E min(ArraryList<E> objects)
   {
   E smallest = objects.get(0);
   for (int i = 1; i < objects.size(); i++)
   {
   E obj = objects.get(i);
   if (obj.getMeasure() < smallest.getMeasure())
   {
   smallest = obj;
   }
   }
   return smallest;
   }

18. No. As described in Common Error 18.1, you cannot convert an ArrayList<BankAccount> to an ArrayList<Measurable>, even if BankAccount implements Measurable.

19. Yes, but this method would not be as useful. Suppose accounts is an array of BankAccount objects. With this method, min(accounts) would return a result of type Measurable, whereas the generic method yields a BankAccount.

20. public static <E extends Measurable> double average(E[] objects)
   {
   if (objects.length == 0) { return 0; }
   double sum = 0;
   for (E obj : objects)
   {
   sum = sum + obj.getMeasure();
   }
   return sum / objects.length;
   }

21. No. You can define
   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;
   }

For example, if BankAccount implements Measurable, a BankAccount[] array is convertible to a Measurable[] array. Contrast with Self Check 19, where the return type was a generic type. Here, the return type is double, and there is no need for using generic types.

22. public static <E> Comparable<E> min(Comparable<E>[] objects)
is an error. You cannot have an array of a generic type.

23. public static void print(Object[] a)
   {
   for (Object e : a)
   {
   System.out.print(e + " ");
   }
   System.out.println();
   }

24. This code compiles (with a warning), but it is a poor technique. In the future, if type erasure no longer happens, the code will be wrong. The cast from Object[] to String[] will cause a class cast exception.

25. Internally, ArrayList uses an Object[] array. Because of type erasure, it can’t make an E[] array. The best it can do is make a copy of its internal Object[] array.

26. It can use reflection to discover the element type of the parameter a, and then construct another array with that element type (or just call the Arrays.copyOf method).

27. The method needs to construct a new array of type T. However, that is not possible in Java without reflection.
Adding the Type Parameter

The types that we use in a binary search tree must be comparable, so we declare the class as

```java
public class BinarySearchTree<E extends Comparable>
```

We replace the parameter variables of type `Comparable` in the following methods with the type parameter `E`:

```java
public void add(E obj)
public boolean find(E obj)
public void remove(E obj)
```

As it happens, there are no other local variables of type `Comparable` to replace. But the data instance variable of the inner `Node` class needs to be changed from `Comparable` to `E`.

```java
public class BinarySearchTree<E extends Comparable> {
    . . .
    class Node {
        public E data;
        public Node left;
        public Node right;
        . . .
    }
}
```

Note that the `Node` class is not a generic class. It is a regular class that is nested inside the generic `BinarySearchTree<E>` class. For example, if `E` is `String`, we have an inner class `BinarySearchTree<String>.Node` with a data instance variable of type `String`.

In contrast, let us supply an `inorder` method that accepts a visitor, and let’s make `Visitor` a top-level interface (unlike the implementation in Section 17.4 where it was declared inside the tree class.) We need a type parameter for the parameter variable of the `visit` method. Because `Visitor` is not nested inside a generic class, we must make it generic.

```java
public interface Visitor<E> {
    void visit(E data)
}
```

We can then implement the `inorder` method in the usual way:

```java
public void inorder(Visitor<E> v) {
    inorder(root, v);
}
```

```java
private void inorder(Node parent, Visitor<E> v) {
    if (parent == null) { return; }
    inorder(parent.left, v);
    visit(parent.data);
    inorder(parent.right, v);
}
```

Copyright © 2015 John Wiley and Sons, Inc. All rights reserved.
Note that the parent parameter variable doesn’t need a type parameter.
With these modifications, we have a fully functioning BinarySearchTree class. You can try out the TreeTester program, and it will work correctly.

**worked_example_1/TreeTester.java**

```java
public class TreeTester {
    public static void main(String[] args) {
        BinarySearchTree<String> names = new BinarySearchTree<>();
        names.add("Romeo");
        names.add("Juliet");
        names.add("Tom");
        names.add("Dick");
        names.add("Harry");

        class PrintVisitor implements Visitor<String> {
            public void visit(String data) {
                System.out.print(data + " ");
            }
        }

        names.inorder(new PrintVisitor());
        System.out.println();
        System.out.println("Expected: Dick Harry Juliet Romeo Tom");
    }
}
```

**worked_example_1/BinarySearchTree.java**

```java
/**
 * This class implements a binary search tree whose
 * nodes hold objects that implement the Comparable
 * interface.
 */
public class BinarySearchTree<E extends Comparable> {
    private Node root;

    /**
     * Constructs an empty tree.
     */
    public BinarySearchTree() {
        root = null;
    }

    /**
     * Inserts a new node into the tree.
     * @param obj the object to insert
     */
```
public void add(E obj)
{
    Node newNode = new Node();
    newNode.data = obj;
    newNode.left = null;
    newNode.right = null;
    if (root == null) { root = newNode; }
    else { root.addNode(newNode); }
}

/**
   Tries to find an object in the tree.
   @param obj the object to find
   @return true if the object is contained in the tree
*/
public boolean find(E obj)
{
    Node current = root;
    while (current != null)
    {
        int d = current.data.compareTo(obj);
        if (d == 0) { return true; }
        else if (d > 0) { current = current.left; }
        else { current = current.right; }
    }
    return false;
}

/**
   Tries to remove an object from the tree. Does nothing
   if the object is not contained in the tree.
   @param obj the object to remove
*/
public void remove(E obj)
{
    // Find node to be removed
    Node toBeRemoved = root;
    Node parent = null;
    boolean found = false;
    while (!found && toBeRemoved != null)
    {
        int d = toBeRemoved.data.compareTo(obj);
        if (d == 0) { found = true; }
        else
        {
            parent = toBeRemoved;
            if (d > 0) { toBeRemoved = toBeRemoved.left; }
            else { toBeRemoved = toBeRemoved.right; }
        }
    }
    if (!found) { return; }
    // toBeRemoved contains obj
    // If one of the children is empty, use the other
    if (toBeRemoved.left == null || toBeRemoved.right == null)
Node newChild;
    if (toBeRemoved.left == null)
    {
        newChild = toBeRemoved.right;
    }
    else
    {
        newChild = toBeRemoved.left;
    }

    if (parent == null) // Found in root
    {
        root = newChild;
    }
    else if (parent.left == toBeRemoved)
    {
        parent.left = newChild;
    }
    else
    {
        parent.right = newChild;
        return;
    }

    // Neither subtree is empty
    // Find smallest element of the right subtree

    Node smallestParent = toBeRemoved;
    Node smallest = toBeRemoved.right;
    while (smallest.left != null)
    {
        smallestParent = smallest;
        smallest = smallest.left;
    }

    // smallest contains smallest child in right subtree
    // Move contents, unlink child

    toBeRemoved.data = smallest.data;
    if (smallestParent == toBeRemoved)
    {
        smallestParent.right = smallest.right;
    }
    else
    {
        smallestParent.left = smallest.right;
    }

    /**
     * Prints the contents of the tree in sorted order.
     */
    public void inorder(Visitor<E> v)
    {
        inorder(root, v);
    }
**Making a Generic Binary Search Tree Class**

```java
/**
 * Prints a node and all of its descendants in sorted order.
 * @param parent the root of the subtree to print
 */
private void inorder(Node parent, Visitor<E> v)
{
    if (parent == null) { return; }
    inorder(parent.left, v);
    v.visit(parent.data);
    inorder(parent.right, v);
}
```

```java
/**
 * A node of a tree stores a data item and references
 * of the child nodes to the left and to the right.
 */
class Node
{
    public E data;
    public Node left;
    public Node right;
}
```

```java
/**
 * Inserts a new node as a descendant of this node.
 * @param newNode the node to insert
 */
public void addNode(Node newNode)
{
    int comp = newNode.data.compareTo(data);
    if (comp < 0)
    {
        if (left == null) { left = newNode; }
        else { left.addNode(newNode); }
    }
    else if (comp > 0)
    {
        if (right == null) { right = newNode; }
        else { right.addNode(newNode); }
    }
}
```

**worked_example_1/Visitor.java**

```java
public interface Visitor<E>
{
/**
 * This method is called for each visited node.
 * @param data the data of the node
 */
void visit(E data);
}
```
In the following sections, we will discuss additional refinements that are described in Special Topic 18.1 and Common Error 18.2. You can skip this discussion if you are not interested in the finer points of Java generics.
Wildcards

Consider the following simple change to the PrintVisitor class in the TreeTester program. We don’t really need to require that data is a string. The printing code will work for any object:

```java
class PrintVisitor implements Visitor<Object>
{
    public void visit(Object data)
    {
        System.out.print(data + " ");
    }
}
```

Unfortunately, now the inorder method of a BinarySearchTree<String> will no longer accept a new PrintVisitor(). It wants a Visitor<String>, not a Visitor<Object>. That’s a shame. Wildcards were invented to overcome this problem.

There is no harm in passing a String value to a visit method with an Object parameter. In general, the data value of type E can be passed to a visit method that receives a supertype of E. You use a wildcard to spell this out:

```java
public void inorder(Visitor<? super E> v)
```

The inorder method works with a visitor for any supertype of E.

The Generic Comparable Type

The Comparable type is a generic type. A Comparable<T> has a compareTo method with a parameter of type T:

```java
public interface Comparable<T>
{
    int compareTo(T other)
}
```

For example, String implements Comparable<String>.

We should make use of the type parameter in the declaration of the BinarySearchTree class. Instead of

```java
public class BinarySearchTree<E extends Comparable>
```

we can write

```java
public class BinarySearchTree<E extends Comparable<E>>
```

With this change, the unsightly warnings at the calls to compareTo go away.

But that’s not quite good enough. Consider the following class:

```java
public class Person implements Comparable<Person>
{
    
    public int compareTo(Person other)
    {
        return name.compareTo(other.name);
    }
}
```

People are just compared by name.

We have a subclass

```java
public class Student extends Person { . . . }
```

Students are people. How are they compared? Also by name. Note that Student implements Comparable<Person>, not Comparable<Student>.
That means we can’t have a `BinarySearchTree<Student>`! Again, wildcards come to the rescue.
The proper type bound is

```java
public class BinarySearchTree<E extends Comparable<? super E>>
```

### Static Contexts

Look again into the first section, where we implemented the `inorder` method in the “usual way”:

```java
private void inorder(Node parent, Visitor<E> v)
{
    if (parent == null) { return; }
    inorder(parent.left, v);
    v.visit(parent.data);
    inorder(parent.right, v);
}
```

Actually, that wasn’t quite the usual way. In the usual way, the recursive helper method is static. But if you try that, you get an error. In a “static context”, the generic parameters don’t work as expected. There is only a single static method for all parameters `E`, so you can’t use `E` in a static method. (This is a consequence of type erasure. There is only a single `BinarySearchTree` class in which `E` is erased.)

The workaround is to make the method generic, like this:

```java
private static <T> void inorder(Node parent, Visitor<T> v)
```

That is better, but it isn’t quite right because `Node` is defined inside a generic class, so we need to specify what kind of node we need.

```java
private static <T> void inorder(BinarySearchTree<T>.Node parent, Visitor<T> v)
```

We are getting closer, but the compiler now complains that a `BinarySearchTree` is only defined for types `T` that implement `Comparable`. Fair enough:

```java
private static <T extends Comparable<? super T>> void inorder(
    BinarySearchTree<T>.Node parent, Visitor<T> v)
```

This method declaration is unfortunately somewhat complex, but it accurately reflects all requirements that must be fulfilled for the method to work. The type `T` must be comparable. The `Node` must belong to the `BinarySearchTree` with type `T`, and the visitor must be an instance of `Visitor<T>` with the same `T`.

Actually, that’s still not right. As mentioned in the section on wildcards, there is nothing wrong with using a visitor for a supertype of `T`. The most general form is

```java
private static <T extends Comparable<? super T>> void inorder(
    BinarySearchTree<T>.Node parent, Visitor<? super T> v)
```

With generics, the implementor must give precise specifications so that the programmers using the generic construct can do so under the most general circumstances.

```java
/*
 * This class demonstrates the advanced techniques in `BinarySearchTree2`.
 */
public class TreeTester2
{
    public static void main(String[] args)
    {
        BinarySearchTree2<Student> students = new BinarySearchTree2<>();
        // Can form `BinarySearchTree2<Student>` even though `Student`
        // implements `Comparable<Person>` and not `Comparable<Student>`
    }
}
```
Making a Generic Binary Search Tree Class

```java
students.add(new Student("Romeo", "Art History"));
students.add(new Student("Juliet", "CS"));
students.add(new Student("Tom", "Leisure Studies"));
students.add(new Student("Diana", "EE"));
students.add(new Student("Harry", "Biology"));

class PrintVisitor implements Visitor<Object>
{
    public void visit(Object data)
    {
        System.out.println(data);
    }
}

// Can pass a Visitor<Object>, not just a Visitor<Student>
students.inorder(new PrintVisitor());
```

worked_example_1/BinarySearchTree2.java

```java
/**
 * This class implements a binary search tree whose nodes hold objects that implement the Comparable interface for an appropriate type parameter.
 */
public class BinarySearchTree2<E extends Comparable<? super E>>
{
    private Node root;

    /**
     * Constructs an empty tree.
     */
    public BinarySearchTree2()
    {
        root = null;
    }

    /**
     * Inserts a new node into the tree.
     * @param obj the object to insert
     */
    public void add(E obj)
    {
        Node newNode = new Node();
        newNode.data = obj;
        newNode.left = null;
        newNode.right = null;
        if (root == null) { root = newNode; }
        else { root.addNode(newNode); }
    }

    /**
     * Tries to find an object in the tree.
     * @param obj the object to find
     * @return true if the object is contained in the tree
     */
```
public boolean find(E obj) {
    Node current = root;
    while (current != null) {
        int d = current.data.compareTo(obj);
        if (d == 0) { return true; }
        else if (d > 0) { current = current.left; }
        else { current = current.right; }
    }
    return false;
}

/**
 * Tries to remove an object from the tree. Does nothing
 * if the object is not contained in the tree.
 * @param obj the object to remove
 */
public void remove(E obj) {
    // Find node to be removed
    Node toBeRemoved = root;
    Node parent = null;
    boolean found = false;
    while (!found && toBeRemoved != null) {
        int d = toBeRemoved.data.compareTo(obj);
        if (d == 0) { found = true; }
        else {
            parent = toBeRemoved;
            if (d > 0) { toBeRemoved = toBeRemoved.left; }
            else { toBeRemoved = toBeRemoved.right; }
        }
    }
    if (!found) { return; }
    // toBeRemoved contains obj
    // If one of the children is empty, use the other
    if (toBeRemoved.left == null || toBeRemoved.right == null) {
        Node newChild;
        if (toBeRemoved.left == null) {
            newChild = toBeRemoved.right;
        } else {
            newChild = toBeRemoved.left;
        }
        if (parent == null) // Found in root
            root = newChild;
```java
else if (parent.left == toBeRemoved) {
    parent.left = newChild;
} else {
    parent.right = newChild;
}
return;

// Neither subtree is empty
// Find smallest element of the right subtree
Node smallestParent = toBeRemoved;
Node smallest = toBeRemoved.right;
while (smallest.left != null) {
    smallestParent = smallest;
    smallest = smallest.left;
}

// smallest contains smallest child in right subtree
// Move contents, unlink child
toBeRemoved.data = smallest.data;
if (smallestParent == toBeRemoved) {
    smallestParent.right = smallest.right;
} else {
    smallestParent.left = smallest.right;
}

/**
 * Prints the contents of the tree in sorted order.
 */
public void inorder(Visitor<? super E> v) {
    inorder(root, v);
}

/**
 * Prints a node and all of its descendants in sorted order.
 * @param parent the root of the subtree to print
 */
private static <T extends Comparable<? super T>> void inorder(BinarySearchTree2<T>.Node parent, Visitor<? super T> v) {
    if (parent == null) { return; }
    inorder(parent.left, v);
    v.visit(parent.data);
    inorder(parent.right, v);
}
```
A node of a tree stores a data item and references of the child nodes to the left and to the right.

```
/**
   A node of a tree stores a data item and references of the child nodes to the left and to the right.
*/
class Node {
    public E data;
    public Node left;
    public Node right;

    /**
     * Inserts a new node as a descendant of this node.
     * @param newNode the node to insert
     */
    public void addNode(Node newNode) {
        int comp = newNode.data.compareTo(data);
        if (comp < 0) {
            if (left == null) { left = newNode; }
            else { left.addNode(newNode); }
        } else if (comp > 0) {
            if (right == null) { right = newNode; }
            else { right.addNode(newNode); }
        }
    }
}
```
STREAM PROCESSING

CHAPTER GOALS

To be able to develop filter/map/reduce strategies for solving data processing problems
To convert between collections and streams
To use function objects for transformations and predicates
To work with the Optional type
To be able to express common algorithms as stream operations

CHAPTER CONTENTS

19.1 THE STREAM CONCEPT 846
19.2 PRODUCING STREAMS 848
19.3 COLLECTING RESULTS 850
   PT1 One Stream Operation Per Line 851
   ST1 Infinite Streams 851
19.4 TRANSFORMING STREAMS 852
   CE1 Don't Use a Terminated Stream 854
19.5 LAMBDA EXPRESSIONS 855
   SYN Lambda Expressions 855
   PT2 Keep Lambda Expressions Short 856
   ST2 Method and Constructor References 857
   ST3 Higher-Order Functions 858
   ST4 Higher-Order Functions and Comparators 859

19.6 THE OPTIONAL TYPE 859
   CE2 Optional Results Without Values 861
19.7 OTHER TERMINAL OPERATIONS 862
   CE3 Don't Apply Mutations in Parallel Stream Operations 863
19.8 PRIMITIVE-TYPE STREAMS 863
19.9 GROUPING RESULTS 866
19.10 COMMON ALGORITHMS REVISITED 868
   HT1 Working with Streams 871
   WE1 Word Properties
   WE2 A Movie Database
Streams let you process data by specifying what you want to have done, leaving the details to the stream library. The library can execute operations lazily, skipping those that are not needed for computing the result, or distribute work over multiple processors. This is particularly useful when working with very large data sets. Moreover, stream computations are often easier to understand because they express the intent of the programmer more clearly than explicit loops. In this chapter, you will learn how to use Java streams for solving common "big data" processing problems.

19.1 The Stream Concept

When analyzing data, you often write a loop that examines each item and collects the results. For example, suppose you have a list of words and want to know how many words have more than ten letters. You could use the “counting matches” algorithm from Section 6.7.2, like this:

```java
List<String> wordList = ...;
long count = 0;
for (String w : wordList)
{
    if (w.length() > 10) { count++; }
}
```

The stream library, which is a part of Java 8, uses a different approach. With a stream of words, you can use the `filter` method to pick out the words that you want, then the `count` method to count them, like this:

```java
Stream<String> words = ...;
long count = words
    .filter(w -> w.length() > 10)
    .count();
```

As you can see, you tell a stream what you want to achieve, not how to achieve it. That can lead to code that is easier to understand. It also allows the library to optimize the execution. For example, it can take advantage of multiple processors in a computer and have each of them work on a part of the data.

The stream library makes heavy use of `lambda expressions` such as `w -> w.length() > 10` that denote small snippets of computation that a method should execute. For example, the `filter` method subjects each stream element to the test “Is this string longer than ten characters?” Lambda expressions were briefly introduced in Java 8 Note 10.4, and we will discuss them in detail in Section 19.5. For now, you should simply have an intuitive idea what they mean. To the left of the `->` symbol, you have parameters, and to the right, the result of a computation. In this case, each stream element `w` is tested with the expression `w.length() > 10`.

Streams are similar to the collections that you have seen in Chapter 15, but there are significant differences.
Unlike a collection, a stream does not store its data. The data comes from elsewhere—perhaps a collection, a file, a database, or a data source on the Internet.

Unlike a collection, a stream is immutable. You cannot insert, remove, or modify elements. However, you can create a new stream from a given one. For example, `words.filter(w -> w.length() > 10)` is a new stream, containing the long words from the original stream. The original stream has not changed. In general, you work with streams by transforming them into new streams, which contain a subset of the elements or transformations of the elements.

Stream processing is lazy. Transformations are done as late as possible, and only when necessary. For example, suppose you ask for five long words:

```java
Stream<String> fiveLongWords = words
    .filter(w -> w.length() > 10)
    .limit(5);
```

Then the intermediate stream of long words is not actually computed in its entirety. The computation flows backwards. Because only five words are requested in the end, the filtering stops when the fifth match has been found. How does that work? It’s not your problem, but that of the implementors of the stream library. Remember, you specify what you want, not how it should be done.

In this chapter, you will learn how to work with streams: how to create and transform them, and how to harvest results. In particular, you will learn in detail how to specify functionality with lambda expressions, such as

```java
w -> w.length() > 10
```

for expressing “w is a long word”.

As computer scientists work with ever bigger data sets and have machines with multiple processors at their disposal, the “what, not how” principle becomes very important. The Java stream library is a good introduction to “big data” thinking.

The following program contrasts the traditional approach of counting matches with the use of a stream.

```java
public class StreamDemo {

    public static void main(String[] args) throws IOException {
        Scanner in = new Scanner(new File("../countries.txt"));
        // This file contains one country name per line
        List<String> wordList = new ArrayList<>();
        while (in.hasNextLine()) { wordList.add(in.nextLine()); }
        // Now wordList is a list of country names
```
### Chapter 19 Stream Processing

#### Program Run

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Long words: 63</td>
<td>Long words: 63</td>
</tr>
</tbody>
</table>

1. Write a statement to count all overdrawn accounts in a `Stream<BankAccount>`.  
2. Given a stream of strings, how do you remove all empty strings?  
3. How would you collect the first five strings of length greater than ten in a `List<String>` without using streams?  
4. Given a stream of strings, how do you calculate how many have exactly ten characters?  
5. Given a stream of strings, how do you find the first one with length equal to ten?  
6. Given a stream of strings, how can you find out whether it has at least ten strings with three letters, without counting them all if there are more than ten?

**Practice It** Now you can try these exercises at the end of the chapter: R19.1, R19.2, R19.3.

### 19.2 Producing Streams

To do stream processing, you need a stream. The simplest way of getting one is with the static `Stream.of` method. You specify the elements, like this:

```java
Stream<String> words = Stream.of("Mary", "had", "a", "little", "lamb");
Stream<Integer> digits = Stream.of(3, 1, 4, 1, 5, 9);
```

With the same method, you can turn an array of objects into a stream:

```java
Integer[] digitArray = { 3, 1, 4, 1, 5, 9 };  
Stream<Integer> digitStream = Stream.of(digitArray);
```

Note, by the way, that `Stream` is a generic type, just like the collection types. A `Stream<String>` yields strings, and a `Stream<Integer>` yields wrapper objects for integers. (As with all generic types, the type parameter cannot be a primitive type, so you must use wrappers for `int`, `double`, and so on.)
If you have a list, set, or other collection in the Java collections framework, call the stream method to obtain a stream that looks up the elements in the collection.

```java
List<String> wordList = new ArrayList<>();
// Populate wordList
Stream<String> words = wordList.stream();
```

Several utility methods of the Java library yield streams. You can get a stream of the lines in a file with the Files.lines method. That method requires a Path object to describe a file. We will introduce the Path interface in Chapter 21, but for now all you need to know is how to make a path from a file name, as shown in the following code fragment:

```java
String filename = ...;
try (Stream<String> lineStream = Files.lines(Paths.get(filename)))
{
    ...
} // The file is closed here; see Section 11.4
```

You can even make an infinite stream—see Special Topic 19.1.

Moreover, you can make a parallel stream, which causes stream operations to be distributed over the available processors. Today, most computers have multiple processors, and programs run faster if they can take advantage of them. The parallel method turns any stream into a parallel stream. Because distributing the work over multiple processors involves some housekeeping, you should only do that with streams that have many elements (see Exercise P19.1).

<table>
<thead>
<tr>
<th>Table 1 Producing Streams</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Example</strong></td>
</tr>
<tr>
<td>Stream.of(1, 2, 3)</td>
</tr>
<tr>
<td>Collection&lt;String&gt; coll = ...; coll.stream()</td>
</tr>
<tr>
<td>Files.lines(path)</td>
</tr>
<tr>
<td>Stream&lt;String&gt; stream = ...; stream.parallel()</td>
</tr>
<tr>
<td>Stream.generate(() -&gt; 1)</td>
</tr>
<tr>
<td>Stream.iterate(0, n -&gt; n + 1)</td>
</tr>
<tr>
<td>IntStream.range(0, 100)</td>
</tr>
<tr>
<td>Random generator = new Random(); generator.ints(0, 100)</td>
</tr>
<tr>
<td>&quot;Hello&quot;.codePoints()</td>
</tr>
</tbody>
</table>
850

Chapter 19 Stream Processing

Write a statement to create a stream of Color objects.
8. Given a list of String objects, use streams to count how many have length less
SELF CHECK
than or equal to three.
9. Repeat Self Check 8 with an array of strings.
10. Write a statement to count how many lines in the file input.txt have length
© Nicholas Homrich/iStockphoto.
greater than 80.
7.

Practice It

Now you can try these exercises at the end of the chapter: R19.4, E19.1, E19.2.

19.3 Collecting Results

© Jamesmcq24/iStockphoto.

Once you have a stream, you transform it into other streams, as will be shown in
detail in the following sections. Eventually, you will want to harvest a result. If the
result is a number (such as a count, sum, maximum, minimum, or average), there are
methods for computing it—see Section 19.8. But often, you want to put the final
stream values back into an array or collection. Here is how to do the former:
String[] result = stream.toArray(String[]::new);

After processing a
stream, you can col© Jamesmcq24/iStockphoto.
lect its elements in an
array or collection.

To turn a stream into
an array or collection,
use the toArray or
collect methods.

The call looks rather mysterious—see Special Topic 19.2 for an explanation. Of
course, if your stream contains elements of a type other than strings, you need to use
that type instead of String.
If you want to save the stream values in a collection, use the collect method. That
method requires a “collector”, an object that is responsible for placing stream values
into a result object. The Collectors class has static methods that produce collectors for
various result types. Here is how to obtain the stream values as a list or set:
List<String> result = stream.collect(Collectors.toList());
Set<String> result = stream.collect(Collectors.toSet());

When you have a stream of strings, you can use a collector to combine them into one
large string. You call the Collectors.joining method and supply a string that is used to
separate the values. For example, here we join the values of the words stream, separating the words with commas:
String result = words.collect(Collectors.joining(", "));
// A string such as "a, am, an, ant, ..." with comma separators

FULL CODE EXAMPLE

Go to wiley.com/
go/bjeo6code to
© Alex Slobodkin/iStockphoto.
download a program
that shows different
ways of collecting
stream results.

Finally, you will see in Section 19.9 how to collect stream values into maps.

Table 2 Collecting Results from a Stream<T>
Example

Comments

stream.toArray(T[]::new)

Yields a T[] array.

stream.collect(Collectors.toList())
stream.collect(Collectors.toSet())

Yields a List<T> or

stream.collect(Collectors.joining(", ")

Yields a string, joining the elements by the
given separator. Only for Stream<String>.

stream.collect(Collectors.groupingBy(
keyFunction, collector)

Yields a map that associates group keys with
collected group values—see Section 19.9.

Set<T>.


Now you know how to turn a collection into a stream, and a stream back into a collection. In between, you will want to transform the streams, turning the initial values into those that you want to compute. You will see how to do that in the next section.

11. Collect all strings of length greater than ten from a list of strings and store them in another list.
12. Repeat Self Check 11, but collect the result in a set.
13. Find the first string of length greater than ten in a list of strings. Use filter and limit, then convert the stream to a list and retrieve the result. Assume that there is at least one such string.
14. Repeat Self Check 13, but use toArray.
15. The solutions to Self Check 13 and Self Check 14 would work even if you omitted the call to limit. Why would that not be a good idea?

**Practice It**
Now you can try these exercises at the end of the chapter: R19.5, E19.3, E19.4.

### One Stream Operation Per Line
Stream operations are very powerful, and you can achieve a lot of work with a small number of method calls. Your code will be much easier to understand if you put each stream operation in a separate line. Then you can mentally track each step.

```java
List<String> result = list.stream() // Create the stream.
    .filter(w -> w.length() > 10) // Keep long strings.
    .limit(50) // Keep only the first fifty.
    .collect(Collectors.toList()); // Turn into a list.
```

In contrast, if you cram as much as possible in one line, it is tedious to figure out the steps.

```java
List<String> result = list.stream().filter(w -> w.length() > 10).limit(50).collect(Collectors.toList()); // Don’t use this formatting style
```

### Infinite Streams
You can make an infinite stream with the generate method. Provide a lambda expression with no arguments, and it is applied for each stream element. For example,

```java
Stream<Integer> ones = Stream.generate(() -> 1);
```

is an infinite stream of ones, and

```java
Stream<Integer> dieTosses = Stream.generate(() -> 1 + (int)(6 * Math.random()));
```

is an infinite stream of random integers between 1 and 6.

If you want to have more interesting infinite streams, use the iterate method. You provide an initial value and an iteration function that is applied to each preceding value. For example,

```java
Stream<Integer> integers = Stream.iterate(0, n -> n + 1);
```

is an infinite stream with elements 0, 1, 2, 3, and so on.

By filtering, you can get more interesting streams. For example, if isPrime is a static method that checks whether an integer is prime, then

```java
Stream<Integer> primes = integers.filter(n -> isPrime(n));
```

is a stream of prime numbers.
Of course, you cannot generate all elements for such a stream. At some point, you need to limit the results. If you want to find the first 500 primes, call

```java
List<Integer> firstPrimes = primes
    .limit(500)
    .collect(Collectors.toList());
```

What is the advantage of using an infinite stream even though it eventually gets truncated to a finite one? You don’t need to know in advance how many integers to use to end up with the desired number of primes.

## 19.4 Transforming Streams

Suppose you have a stream of words, some uppercase, some lowercase, and you want them all lowercase. In such a situation, you take the stream that you are given and transform it, by applying a function to all elements and collecting the results in a new stream. That is what the `map` method does.

```java
Stream<String> words = Stream.of("A", "Tale", "of", "Two", "Cities");
Stream<String> lowerCaseWords = words.map(w -> w.toLowerCase());
// "a", "tale", "of", "two", "cities"
```

Perhaps you aren’t interested in vowels. Then you can remove them:

```java
Stream<String> consonantsOnly = lowerCaseWords.map(
    w -> w.replaceAll("[aeiou]\", ""));
// ", "tl", "f", "tw", "cts"
```

Maybe you just wanted to know how many consonants each had:

```java
Stream<Integer> consonantCount = consonantsOnly.map(w -> w.length());
// 0, 2, 1, 2, 3
```

As you can see, the `map` method takes a function and applies it to each element of the stream, yielding a new stream with the results. The function syntax is pretty self-explanatory: to the left of the `->` is the parameter variable, and to the right the returned value. We will go over the syntax in detail in Section 19.5.

### Table 3 Stream Transformations

<table>
<thead>
<tr>
<th>Example</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>stream.filter(condition)</code></td>
<td>A stream with the elements matching the condition.</td>
</tr>
<tr>
<td><code>stream.map(function)</code></td>
<td>A stream with the results of applying the function to each element.</td>
</tr>
<tr>
<td><code>stream.mapToInt(function)</code></td>
<td>A primitive-type stream with the results of applying a function with a return value of a primitive type—see Section 19.8.</td>
</tr>
<tr>
<td><code>stream.mapToDouble(function)</code></td>
<td></td>
</tr>
<tr>
<td><code>stream.mapToLong(function)</code></td>
<td></td>
</tr>
<tr>
<td><code>stream.limit(n)</code></td>
<td>A stream consisting of the first <code>n</code>, or all but the first <code>n</code> elements.</td>
</tr>
<tr>
<td><code>stream.skip(n)</code></td>
<td></td>
</tr>
<tr>
<td><code>stream.distinct()</code></td>
<td>A stream of the distinct or sorted elements from the original stream.</td>
</tr>
<tr>
<td><code>stream.sorted()</code></td>
<td></td>
</tr>
<tr>
<td><code>stream.sorted(comparator)</code></td>
<td></td>
</tr>
</tbody>
</table>
You have already seen the filter transformation. It takes a function with a boolean result. (Such a function is called a predicate.) The resulting stream retains all elements that “pass the test”; that is, for which the function returns true.

For example, suppose we have a stream of words and only want to look at those starting with the letter a. You get them as

```java
Stream<String> aWords = words.filter(w -> w.substring(0, 1).equals("a"));
```

Note that the expression to the right of the -> has a boolean value. Either the first letter is an a, or it is not.

The map and filter methods are the “bread and butter” of stream transformations. Usually, you get a stream, filter away what you don’t need, and transform the desired elements to the desired form. Sometimes, you first need to transform the elements so you can apply a filter. You can string together as many map and filter operations as you like. They are evaluated lazily, without creating unnecessary intermediate results.

Here are several more useful stream transformations.

1. You have already seen the limit method. The call
   ```java
   stream.limit(n)
   ```
   yields a new stream containing the first n elements of the original stream (or the entire original stream if it had fewer than n elements).

2. The call stream.skip(n) does the opposite, returning the stream obtained by dropping the first n elements. This operation lets you drop initial elements that don’t contribute to the end result.

3. The distinct method drops replicated elements:
   ```java
   Stream<String> words = Stream.of(
       "how much wood could a wood chuck chuck".split(" "));
   Stream<String> distinctWords = words.distinct();
   // "how", "much", "wood", "could", "a", "chuck"
   ```

4. The sorted method yields a new stream with the elements of the original stream in sorted order.
   ```java
   Stream<String> sortedWords = distinctWords.sorted();
   // "a", "chuck", "could", "how", "much", "wood"
   ```

   For this method to work, the stream elements must belong to a class that implements the Comparable interface. In the next section, you will see how to specify a different comparator.

The following sample program puts several stream transformations to use, producing a list of abbreviated country names.

```java
import java.io.IOException;
import java.nio.file.Files;
import java.nio.file.Paths;
import java.util.List;
import java.util.stream.Collectors;
import java.util.stream.Stream;

public class StreamDemo {
    public static void main(String[] args) throws IOException {
        // The filter method yields a stream of all elements fulfilling a condition.
        // Go to wiley.com/go/bjo6code to download a more detailed sample program.
    }
}
```
try (Stream<String> lines = Files.lines(Paths.get("../countries.txt")))
{
    // Read the lines
    List<String> result = lines
        .filter(w -> w.length() > 10) // Keep only long words
        .map(w -> w.substring(0, 7)) // Truncate to seven characters
        .map(w -> w + "...") // Add ellipses
        .distinct() // Remove duplicates
        .limit(20) // Keep only the first twenty
        .collect(Collectors.toList()); // Collect into a list
    System.out.println(result);
}

Program Run
[Afghani..., America..., Antigua..., Bahamas..., Bosnia ..., British..., Burkina..., Cayman ..., Central..., Christm..., Cocos (......, Congo, ..., Cook Is..., Cote d'..., Czech R..., Dominic..., El Salv..., Equator..., Falklan..., Faroe I...]

Don’t Use a Terminated Stream
Once you obtain a result from a stream, by calling a method such as asArray or collect, the
stream is “used up”, and you can no longer apply other operations to it. For example, you cannot
put the first fifty elements in one list and then take the next fifty:
Stream<String> stream = list.stream();
List<String> result1 = stream.limit(50).collect(Collectors.toList());
stream = stream.skip(50); // Error—the stream can no longer be used

If you want to collect two sets of fifty elements, you need to recreate the stream.
List<String> result1 = list.stream()
    .limit(50)
    .collect(Collectors.toList()); // This stream can no longer be used
List<String> result2 = list.stream() // Create another stream
    .skip(50)
    .limit(50)
    .collect(Collectors.toList());

To avoid this error, it is a good idea to write stream operations as a “pipeline” that starts with
stream creation, is followed by transformations, and ends with collecting a result. Format the
pipeline as described in Programming Tip 19.1.

SELF CHECK
16. Given a stream of words, get a stream of all that start with a or A, converted
to lowercase. Provide two solutions, one applying filter before map and one after.
17. Given a stream of words, produce a stream of Integer values containing the
lengths of the words.
18. Given a list of strings, get a list of the first ten in sorted order.
19. Given a list of words, how do you find how many distinct words there are of
length equal to three?
20. How can you solve Self Check 19 without streams?

Practice It Now you can try these exercises at the end of the chapter: R19.6, E19.5, E19.6.
Lambda Expressions

In the arguments to the `filter` and `map` methods, you have seen lambda expressions such as

\( w \rightarrow w.length() > 10 \)

This is a function, in the mathematical sense, like a static method in Java. It has a parameter variable \( w \) in this example) that is mapped to a result. To the right of the `->` is the result, namely the value of the expression \( w.length() > 10 \), which is true for long strings and false for short ones.

When you use this function with a `Stream<String>`, then the Java compiler can figure out that \( w \) must be a `String`. When the compiler cannot figure out the parameter type from the context, you need to specify it. Then the syntax is

\( (\text{String } w) \rightarrow w.length() > 10 \)

In other situations, you need a function with two parameters. Then you specify both parameters, enclosed in parentheses, like this:

\( (v, w) \rightarrow v.length() - w.length() \)

This function takes two strings and returns the difference of their lengths.

As an application of such a function, consider the task of sorting a sequence of objects. In order to sort, you need to compare elements. You can use the `Comparable` interface for this purpose, but there is another way (shown in Special Topic 14.4). You provide a function that compares two elements and returns an integer. If the result is negative, the first argument comes before the second. If the result is zero, the arguments can come in either order. Otherwise, the first argument comes after the second.

The function that computes the difference between the lengths of two strings is such a comparison function. It compares strings by length.

You can pass this function to the `sorted` method of the `Stream` class, and it yields a stream whose elements are sorted by increasing length. For example,

\[
\text{Stream<String> sortedWords = distinctWords.sorted(}
\quad (v, w) \rightarrow v.length() - w.length();
\quad // "a", "how", "much", "wood", "could", "chuck"
\]

### Syntax 19.1 Lambda Expressions

**Syntax**  \( \text{Parameter variables } \rightarrow \text{body} \)

- **Parameter variables**
- **Optional parameter type**
- **Use braces and a return statement for longer bodies.**
- **The body can be a single expression.**
- **Omit parentheses for a single parameter.**

- \( w \rightarrow w.length() > 10 \)
- \( (\text{String } w) \rightarrow w.length() > 10 \)
- \( (v, w) \rightarrow v.length() - w.length() \)
- \( (v, w) \rightarrow \{
\quad \text{int difference} = v.length() - w.length();
\quad \text{return difference;}
\quad \}
\)
The preceding example showed a typical use of a lambda expression. The `sorted` method has a parameter of type `Comparator`, an interface with a single abstract method or functional interface. You can pass a lambda expression, and it is converted to an instance of the interface.

In all these examples, the result of the function was so simple that the computation happened in a single expression. You can also distribute the code over multiple statements, enclosed in braces. Then use a return statement for the result, as you do with methods:

```
(v, w) ->
{
    int first = v.length();
    int second = w.length();
    return first - second;
}
```

That is all that you have to know about lambda expressions in order to use stream methods such as `filter`, `map`, and `sorted`. When the result expression is just a method call, you can optionally use an even shorter syntax—see Special Topic 19.2.

By the way, the term lambda expression comes from the history of mathematics. In the 1930s, the logician Haskell Curry used the Greek letter lambda (\( \lambda \)) in his notation for functions, \( \lambda \text{parameter} \rightarrow \text{result} \). In Java one uses an arrow instead: `parameter -> result`, but people still call them lambda expressions, not arrow expressions.

21. Write a lambda expression for a function that computes the average of two numbers.

22. Write a lambda expression that tests whether a word starts and ends with the same letter.

23. What does this lambda expression do?
   ```
   s -> s.equals(s.toUpperCase())
   ```

24. Assuming that `words` is a `Stream<String>`, what is the result of this call?
   ```
   words.filter(s -> s.equals(s.toUpperCase()))
   ```

25. Assuming that `words` is a `Stream<String>`, what is the result of this call?
   ```
   words.map(s -> s.equals(s.toUpperCase()))
   ```

**Practice It**

Now you can try these exercises at the end of the chapter: R19.7, E19.7, E19.8.

**Keep Lambda Expressions Short**

Lambda expressions are passed to methods such as `map`, `filter`, and `sorted`, which themselves can occur in sequences of stream operations. If the lambda expression is too long, or if its purpose is not immediately apparent, then it becomes difficult to focus on the sequence of operations. In that case, make a helper method and call it in the lambda expression.

For example, suppose you want to sort countries first by continent and then alphabetically. It is syntactically correct to use a complex expression:

```
List<Country> result = stream.sorted((c, d) ->
{
    int difference = c.getContinent().compareTo(d.getContinent());
    if (difference != 0) { return difference; }
}
```

...
else { return c.getName().compareTo(d.getName); }
})
 .collect(Collectors.toList());

You can make the code easier to read by calling a helper method:

List<Country> result = stream.sorted((c, d) -> byContinentThenByName(c, d))
 .collect(Collectors.toList());

If you don’t want to declare a separate method, use a separate variable whose type is the appropriate functional interface:

Comparator<Country> byContinentThenByName =
 (c, d) -> {
   int difference = c.getContinent().compareTo(d.getContinent());
   if (difference != 0) { return 0; }
   else { return c.getName().compareTo(d.getName); }
  };

List<Country> result = stream.sorted(byContinentThenByName)
 .collect(Collectors.toList());

Also note that it is customary to use single-letter parameter variables in lambda expressions to keep them short. Check out Special Topic 19.2 and Special Topic 19.4 for additional ways of making lambda expressions more compact.

### Special Topic 19.2

**Method and Constructor References**

When a lambda expression consists of just one method call, you can use a very concise syntax, called a **method reference**. A class name followed by a `::` symbol and method name is equivalent to a lambda expression with parameters “at the right places”. For example, the method reference

`String::toUpperCase`

is a shorthand for the lambda expression

`(String w) -> w.toUpperCase()`

There are several forms of method references. You have just seen the case of an instance method of a class with no parameters. If the method has parameters, they are parameters of the equivalent lambda expression. For example, the method reference

`String::compareTo`

is the same as

`(String s, String t) -> s.compareTo(t)`

If the method is a static method, then there is no implicit parameter. The expression

`Double::compare`

is the same as

`(double x, double y) -> Double.compare(x, y)`

Finally, you can have an object to the left of the `::` symbol, for example

`System.out::println`

That is the same as

`x -> System.out.println(x)`, where the type of `x` is inferred from context.

To invoke a constructor, use the keyword `new` after the `::` symbol. For example,

`BankAccount::new`

is the same as

`() -> new BankAccount() or b -> new BankAccount(b)`, depending on where it is used.
One constructor reference that you will see frequently is for creating an array:

```java
String[]::new
```

is the same as `n -> new String[n]`, creating a string array of a given length. When the expression is invoked, one must pass an integer, namely the desired length of the array.

Such a constructor reference is used as the argument of the `toArray` method of streams. When you call

```java
String[] array = words.toArray(String[]::new)
```

then the `toArray` method calls the array constructor with the number of elements of the stream. This sounds more complicated than it should be, but unfortunately, it is not possible in Java to construct an array of a generic type without providing the constructor—see Section 18.5.

You don’t have to use method or constructor references, but many programmers prefer them because they find them clearer and more concise than the equivalent lambda expressions.

### Higher-Order Functions

Methods such as `map` and `filter` consume functions. There is no real difference between a method and a function—a method is just a function where one of the parameters is in a special position. Thus, `map` and `filter` are often called **higher-order functions** because they are functions that consume functions.

To implement a higher-order function in Java, you need to use a **functional interface**, an interface with exactly one abstract method. The `Comparator` interface is one such interface, with a single method `compare`. Another one is the `Predicate<T>` interface. It has an abstract method

```java
boolean test(T t)
```

The `filter` method has a parameter of type `Predicate<T>`, where `T` is the same type as the element type of the stream.

Here is how you would implement such a method. For simplicity, the method works on lists, not streams.

```java
public static <T> List<T> filter(List<T> values, Predicate<T> p)
{
    List<T> result = new ArrayList<>();
    for (T value : values)
    {
        if (p.test(value)) { result.add(value); }
    }
    return result;
}
```

Note how the `filter` method invokes the `test` method on the function parameter `p`.

When you call

```java
List<String> filtered = filter(wordList, w -> w.length() > 10);
```

then the `filter` method receives as its parameter `p` an object of some class that implements `Predicate<String>`. The `test` method of that object returns the `boolean` value `w.length() > 10`.

A higher-order function can also return a function. Consider this example:

```java
public static Predicate<String> contains(String target)
{
    return s -> s.indexOf(target) >= 0;
}
```

The call `contains("and")` yields a predicate that tests whether a string contains the substring "and". You can pass that function to a method that expects a predicate, such as `filter`:

```java
List<String> filtered = filter(wordList, contains("and"));
```
19.6 The Optional Type

Special Topic 19.4

859

Higher-Order Functions and Comparators
A useful higher-order function that receives and returns a function is Comparator.comparing.
You give it an extractor function that extracts a value from an element. The result is a comparison function that compares the extracted values. For example, consider
Comparator<String> comp = Comparator.comparing(t -> t.length())

That is the same function as
Comparator<String> comp = (v, w) -> v.length() - w.length();

You can write it even more succinctly with a method reference:
© Eric Isselé/iStockphoto.
Comparator.comparing(String::length)

This reads quite nicely—the comparator that compares strings by their length.
You can add a secondary comparison with the thenComparing method:
Collections.sort(countries,
Comparator.comparing(Country::getContinent)
.thenComparing(Country::getName));

The countries are compared first by continent. If the continents are the same, they are compared by name.

In Java, it has been common to use the value null to indicate
that a method has no result. However, that’s a bit dangerous. If
a caller doesn’t expect the “no answer” case and uses the result,
a NullPointerException occurs, which can terminate the program.
String result = oldFashionedMethod(searchParameters);
// Returns null if no match
int length = result.length();
// Throws a NullPointerException when result is null

The Optional class is
a wrapper for objects
that may or may not
be present.

The stream library takes a different approach. Whenever a
An Optional value
query might not yield any answer, it returns a result of the may or may not be
type Optional<T>.
present.
Here is a good example. Suppose you want to find the first
and call
long string in a stream of strings. Use the findFirst method
© Tomwang112/iStockphoto.
words.filter(w -> w.length() > 10).findFirst()

But what if words has no elements with more than ten characters? Then there can’t be a
first element in the filtered stream.
For that reason, the findFirst method returns an Optional<String>, not a String:
Optional<String> optResult = words
.filter(w -> w.length() > 10)
.findFirst();

An Optional object is a wrapper that holds a value or an indication that no value is
present.
Use the orElse method to extract the value, or the provided alternative:
int length = optResult.orElse("").length();

© Tomwang112/iStockphoto.

19.6 The Optional Type


Table 4 Working with Optional Values

<table>
<thead>
<tr>
<th>Example</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>result = optional.orElse(&quot;&quot;);</td>
<td>Extracts the wrapped value or the specified default if no value is present.</td>
</tr>
<tr>
<td>optional.ifPresent(v -&gt; Process v);</td>
<td>Processes the wrapped value if present or does nothing if no value is present.</td>
</tr>
<tr>
<td>if (optional.isPresent())</td>
<td>Processes the wrapped value if present, or deals with the situation when it is not present.</td>
</tr>
<tr>
<td>{ process optional.get() }</td>
<td></td>
</tr>
<tr>
<td>else</td>
<td></td>
</tr>
<tr>
<td>{ Handle the absence of a value. }</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>double average = pstream.average().getAsDouble();</td>
<td>Gets the wrapped value from a primitive-type stream—see Section 19.8.</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>if (there is a result)</td>
<td>Returns an Optional value from a method.</td>
</tr>
<tr>
<td>{ return Optional.of(result); }</td>
<td></td>
</tr>
<tr>
<td>else</td>
<td></td>
</tr>
<tr>
<td>{ return Optional.empty(); }</td>
<td></td>
</tr>
</tbody>
</table>

If there is a result, then its length is computed. Otherwise, the orElse method returns an alternative, "", on which it is safe to invoke the length method.

Another alternative is to call the ifPresent method and give it a function that consumes the value. For example,

```
optResult.ifPresent(v -> results.add(v));
```

If a value v is present, it is added to the results collection. If not, nothing happens.

The isPresent method tests whether there really was a result:

```
if (!optResult.isPresent())
    System.out.println("No element has more than ten characters");
```

If you know there is a result, you can use the get method to retrieve it. But if you were wrong and there was no value, a NoSuchElementException is thrown.

Now you have learned how to use the Optional type. If you write a method that returns an Optional value, return either Optional.of(result), if there is a result, or Optional.empty() if there is none.

**Self Check**

26. Set word to the first word in the list wordList containing the letter a, or to the empty string if there is no match.

27. Repeat Self Check 26 using ifPresent.
28. Repeat Self Check 26 using isPresent.

29. Set word to the tenth word in the list wordList containing the letter a, or to the empty string if there is no such string. Don’t use collect.

30. Write a method reciprocal that receives a parameter x of type double and returns an Optional<Double> containing 1 / x if x is not zero.

**Practice It** Now you can try these exercises at the end of the chapter: R19.8, E19.9, E19.10.

**Optional Results Without Values**

Methods such as findFirst and max return an Optional result for a good reason — there might not be any result. If you ignore that possibility and just call get to retrieve the wrapped value, you run the risk of a NoSuchElementException.

It is best to avoid the get method and use orElse or ifPresent if at all possible.

Of course, you can call isPresent followed by get, but the resulting code is usually more complex.

Consider the example of finding a long word. Don’t just call

```java
String result = stream
    .filter(w -> w.length() > 10)
    .findFirst()
    .get(); // Throws an exception if no value is present
```

What do you want to do if there is no long word? If you want to print either the word or a message, you can use orElse:

```java
String result = stream
    .filter(w -> w.length() > 10)
    .findFirst()
    .orElse("None");
System.out.println("Long word: "+result);
```

If you want to only process the result when it exists and otherwise do nothing, use ifPresent:

```java
List<String> results = . . . ;
stream
    .filter(w -> w.length() > 10)
    .findFirst()
    .ifPresent(w -> results.add(w));
```

Reserve the use of isPresent and get for complex situations where you carry out entirely different actions depending on the outcome:

```java
Optional<String> result = stream
    .filter(w -> w.length() > 10)
    .findFirst();
if (result.isPresent())
{
    results.add(result.get()); // Safe to call get
}
else
{
    System.out.println("No long words");
}
```
Methods such as count and findFirst are **terminal operations**. A terminal operation forces the execution of all pending lazy operations and yields a value that is not a stream. Afterward, the stream is no longer usable.

Another terminal operation is findAny. It works like findFirst, but it returns any match, not necessarily the first one. That is faster on parallel streams. Suppose you want to find a long word that ends with y, and you don’t care which word is chosen if there is more than one. Then call

```
result = words
           .parallel()
           .filter(w -> w.length() > 10)
           .filter(w -> w.endsWith("y"))
           .findAny()
           .orElse;
```

Other terminal operations are max and min. You need to supply a comparator when calling these methods. For example, to get the longest string (or, if there are multiple strings of maximum length, one of them), call

```
Optional<String> result = words
                           .max((v, w) -> v.length() - w.length());
```

Because it is possible that the stream is empty, max and min return Optional values.

The toArray and collect methods from Section 19.3 are also terminal operations. Another one is forEach, which doesn’t yield a result but instead applies a function to all elements. For example,

```
words.forEach(w -> System.out.println(w))
```

prints all words in the stream. You need to be cautious with the forEach method on a parallel stream. For example, printing all words in a parallel stream can result in a garbled output.

Finally, there are three terminal operations that return a boolean value, to test whether all, any, or no stream elements match a condition: allMatch, anyMatch, and noneMatch.

```
boolean result = words.allMatch(w -> w.contains("e"));
// result is true if all words contain the letter e
// Use anyMatch or noneMatch to check for some or no matches
```

**SELF CHECK**

31. Rewrite the example for the findAny operation at the beginning of this section so that the filter method is only called once.

32. How can you check whether any words start with the letter q and end with the letter y without calling findAny?
33. What is wrong with the following code?

```java
Stream<String> qys = wordList.stream()
    .filter(w -> w.startsWith("q"))
    .filter(w -> w.endsWith("y"));
if (qys.count() > 0)
{
    System.out.println(qys.findAny().get());
}
```

34. How can you get two words starting with q and ending with y?

35. An operation short circuits if it stops looking at inputs that can no longer change the result. For example, the Boolean operator && short circuits when the first operand is false. Which of allMatch, anyMatch, and noneMatch can short circuit?


Don’t Apply Mutations in Parallel Stream Operations

Often, you can speed up stream operations by making the stream parallel. Then the work is distributed over multiple processors. However, that only works when the operations don’t interfere with each other. If you print output, it may appear interleaved. If you store values in a data structure, it can get corrupted. Here is an example of something that will not work:

```java
List<String> longWords = new ArrayList<>();
wordList.stream()
    .parallel()
    .forEach(w ->
    {
        if (w.length() > 10)
        {
            longWords.add(w); // Error—don’t mutate in a parallel stream
        }
    });
```

If you try this with a sufficiently long list of words on a computer with more than one processor, chances are great that your program will crash when the array list is corrupted. And if your program happens not to crash, the list will probably not contain all results. Chapter 21 has much more to say about the dangers of accessing shared data at the same time.

In this example, you should just use filter and collect to safely return the result. In general, it is safe to use parallel streams when none of the stream operations mutate shared objects.

19.8 Primitive-Type Streams

It is inefficient to have streams of wrappers to primitive types, such as Stream<Integer>, because each individual int element needs a separate wrapper. There are three specialized stream interfaces, IntStream, LongStream, and DoubleStream, that store elements of type int, long, and double. (The remaining five primitive types (float, byte, char, short, and boolean) don’t occur often enough in streams to warrant special cases.)
19.8.1 Creating Primitive-Type Streams

You can create an IntStream from individual integers, or from an array:

```java
IntStream stream = IntStream.of(3, 1, 4, 1, 5, 9);
int[] values = ...;
stream = IntStream.of(values);
```

To get a stream consisting of the integers \(a, a + 1, a + 2\), up to, but not including \(b\), call

```java
IntStream stream = IntStream.range(a, b);
```

The doubles method of the Random class yields a DoubleStream of random values between 0 and 1. The ints method yields random numbers between an inclusive lower bound and an exclusive upper bound:

```java
Random generator = new Random();
dieTosses = generator.ints(1, 7);
```

You can turn a String object \(str\) into an IntStream of its Unicode code points with

```java
IntStream codePoints = str.codePoints();
```

Often, you will create an IntStream from another stream by applying a function with \(int\) results to the elements, using the mapToInt method. For example, if you want to process word lengths, you get a stream of them with

```java
IntStream lengths = words.mapToInt(w -> w.length());
```

Similarly, there are methods mapToDouble and mapToLong that yield a DoubleStream and a LongStream.

19.8.2 Mapping a Primitive-Type Stream

The map method of an IntStream yields another IntStream. For example,

```java
IntStream stream = IntStream.range(0, 20)
    .map(n -> Math.min(n, 10));
```

yields a stream with twenty elements 0, 1, 2, ..., 9, 10, 10, ..., 10.

If you want to convert an IntStream to a stream of objects, call mapToObj. This is often useful to imitate a loop over a range of integers. Here, we generate a stream of strings of increasing length:

```java
String river = "Mississippi";
int n = river.length();
Stream<String> prefixes = IntStream.range(0, n)
    .mapToObj(i -> river.substring(0, i));
// "", "M", "Mi", "Miss", "Missi", ...
```

The IntStream class also has methods mapToDouble and mapToLong. The DoubleStream and LongStream classes have similar methods.

19.8.3 Processing Primitive-Type Streams

An IntStream has all the methods for streams that you have seen in the preceding sections. Some of them are adjusted for the fact that the stream contains \(int\) values and not objects. For example, the toArray method yields an int[] array.
In addition to the `count` method, primitive-type streams have four additional methods that yield a numeric result: `sum`, `average`, `max`, and `min`. For example,

```java
int sumOfLengths = words
    .mapToInt(w -> w.length())
    .sum();
```

yields the sum of all word lengths. Unlike the `max` and `min` methods for arbitrary streams, the `max` and `min` methods for primitive streams do not require a comparator.

The `average` method returns an `OptionalDouble`, because the stream might be empty. That type is just like an `Optional<Double>`, except that it holds a `double` value, not a `Double` wrapper. As with all `Optional` values, it is best to use `orElse` to obtain the wrapped value or an alternative.

```java
double average = words
    .mapToInt(w -> w.length())
    .average()
    .orElse(0);
```

Similarly, the `max` and `min` methods of the `IntStream` class return an `OptionalInt` because they might be invoked on an empty stream.

The `DoubleStream` and `LongStream` classes have equivalent methods, with return types that match the element type.

<table>
<thead>
<tr>
<th>Example</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>stream.count()</code></td>
<td>Yields the number of elements as a <code>long</code> value.</td>
</tr>
<tr>
<td><code>stream.findFirst()</code></td>
<td>Yields the first, or an arbitrary element as an <code>Optional&lt;T&gt;</code>—see Section 19.6.</td>
</tr>
<tr>
<td><code>stream.findAny()</code></td>
<td></td>
</tr>
<tr>
<td><code>stream.max(comparator)</code></td>
<td>Yields the largest or smallest element as an <code>Optional&lt;T&gt;</code>—see Section 19.7.</td>
</tr>
<tr>
<td><code>stream.min(comparator)</code></td>
<td></td>
</tr>
<tr>
<td><code>psstream.sum()</code></td>
<td>The sum, average, maximum, or minimum of a primitive-type stream—see Section 19.8.</td>
</tr>
<tr>
<td><code>psstream.average()</code></td>
<td></td>
</tr>
<tr>
<td><code>psstream.max()</code></td>
<td></td>
</tr>
<tr>
<td><code>psstream.min()</code></td>
<td></td>
</tr>
<tr>
<td><code>stream.allMatch(condition)</code></td>
<td>Yields a boolean variable indicating whether all, any, or no elements match the condition—see Section 19.7.</td>
</tr>
<tr>
<td><code>stream.anyMatch(condition)</code></td>
<td></td>
</tr>
<tr>
<td><code>stream.noneMatch(condition)</code></td>
<td></td>
</tr>
<tr>
<td><code>stream.forEach(action)</code></td>
<td>Carries out the action on all stream elements—see Section 19.7.</td>
</tr>
</tbody>
</table>

36. Given a list of `BankAccount` objects, use streams to find the sum of all balances.
37. Given a list of `BankAccount` objects, use streams to find the average balance.
38. Given a list of words, find the length of the longest one.
39. Given a list of words, find the length of the shortest word starting with the letter z.

**Practice It** Now you can try these exercises at the end of the chapter: R19.9, R19.10, E19.11.
So far, you have seen how to put data into a stream, transform it, and obtain a result. Sometimes, you would like to split a stream into groups and then get a result for each group.

To form a group, call

\[
\text{stream.collect(Collectors.groupingBy(function))}
\]

The function determines a “key” for each element in the stream. Elements with the same key form a group.

Like all calls to collect, this is a terminal operation, yielding a non-stream result. The result is a map that associates each key with its group.

Let’s try this with a stream of words and a key function \( w \to w.\text{substring}(0, 1) \). That is, the key for each element is the first letter.

```java
Map<String, List<String>> groups = Stream.of(words)
    .collect(Collectors.groupingBy(
        w -> w.substring(0, 1))); // The function for extracting the keys
```

The result is a map, where \( \text{groups.get("a")} \) is a list of all words starting with the letter a, \( \text{groups.get("b")} \) has all words starting with b, and so on.

It is nice to be able to split a stream into groups, but you can do even more. You can process each group with a collector. In Section 19.3, you saw the methods Collectors.toSet and Collectors.joining that yield collectors.

For example, if you prefer to have sets of strings instead of lists, use

```java
Map<String, Set<String>> groupOfSets = Stream.of(words)
    .collect(Collectors.groupingBy(
        w -> w.substring(0, 1), // The function for extracting the keys
        Collectors.toSet()));  // The group collector
```

There are two collectors. The groupingBy collector collects the stream into a map. The toSet collector collects each group into a set.

There are several collectors for computing a count, sum, average, maximum, or minimum in each group. (You can’t use the stream methods that you have already seen because the groups are lists, not streams.)

To count each group, pass Collectors.counting():

```java
Map<String, Long> groupCounts = Stream.of(words)
    .collect(Collectors.groupingBy(
        w -> w.substring(0, 1),
        Collectors.counting()));
```

This is a pretty interesting result. You get to find out how many words start with each letter.

The summingInt, summingDouble, and summingLong collectors apply a function to each element in a group. The sum is associated with the group key.

As an example, here is how to compute the total population of each continent from a stream of countries:

```java
Map<String, Long> groupSum = countries.collect(
    Collectors.groupingBy(
        c -> c.getContinent(), // The function for extracting the keys
        Collectors.summingLong(
            c -> c.getPopulation()))); // The function for getting the summands
```
Note that there are two functions: one for extracting the key, and one for getting the values that are being summed.

The averagingInt, averagingDouble, and averagingLong collectors work just like the summing collectors, but they collect the averages for each group.

To get the largest or smallest element in each group, use maxBy or minBy, with a comparison function. Here is a map of the longest strings for each letter:

```java
Map<String, Optional<String>> groupLongest = Stream.of(words)
    .collect(Collectors.groupingBy(
        w -> w.substring(0, 1), // The function for extracting the keys
        Collectors.maxBy(
            (v, w) -> v.length() - w.length())); // The comparator function
```

The following program shows the grouping operations in action. See Worked Example 19.1 and Worked Example 19.2 for examples of forming groups with real-world data.

```java
section_9/GroupDemo.java
import java.util.List;
import java.util.Map;
import java.util.Optional;
import java.util.Set;
import java.util.stream.Collectors;

public class GroupDemo {
    public static void main(String[] args) {
        String[] words = ("how much wood would a woodchuck chuck " + 
                        "if a woodchuck could chuck wood").split(" ");

        Map<String, List<String>> groups = Stream.of(words)
            .collect(Collectors.groupingBy(
                w -> w.substring(0, 1)));
        System.out.println("Lists by first letter: "+ groups);

        Map<String, Set<String>> groupOfSets = Stream.of(words)
            .collect(Collectors.groupingBy(
                w -> w.substring(0, 1), // The function for extracting the keys
                Collectors.toSet())); // The group collector
        System.out.println("Sets by first letter: "+ groupOfSets);

        Map<String, Long> groupCounts = Stream.of(words)
            .collect(Collectors.groupingBy(
                w -> w.substring(0, 1),
                Collectors.counting()));
        System.out.println("Counts by first letter: "+ groupCounts);

        Map<String, Optional<String>> groupLongest = Stream.of(words)
            .collect(
                Collectors.groupingBy(
                    w -> w.substring(0, 1), // The function for extracting the keys
                    Collectors.maxBy(
                        (v, w) -> v.length() - w.length()))); // The comparator function
```
Suppose `words` contains the strings "Mary", "had", "a", "little", "lamb". What are the contents of `groups` in the first example of this section?

With the same contents for `words`, what are the contents of `groupCounts` in the third example?

Given a list of strings, make a map with keys 1, 2, 3, …, so that the value for the key `n` is a list of all words of length `n`.

Associate with each letter the average length of words in `words` that start with that letter.

Associate with each letter a string containing all words in `words` starting with that letter, separated by commas.

Now you can try these exercises at the end of the chapter: R19.12, E19.15, E19.16.

### 19.10 Common Algorithms Revisited

In Sections 6.7 and 7.3, you saw a number of common algorithms for sequences of values. Many of them are quite a bit easier with streams. Let’s revisit them and see how using streams simplifies them.

#### 19.10.1 Filling

To fill an array with squares (0, 1, 4, 9, 16, …), you can use a loop

```java
int[] squares = new int[n];
for (int i = 0; i < squares.length; i++)
    squares[i] = i * i;
```

With streams, you start out with a range and map it to the desired form.

```java
int[] squares = IntStream.range(0, n)
    .map(i -> i * i)
    .toArray();
```

The stream form is shorter and, with a bit of practice, easier to understand.
19.10.2 Sum, Average, Maximum, and Minimum

Chapter 7 had explicit loops for what are simply method calls with streams.

```java
double[] values = . . .;
double total = DoubleStream.of(values).sum();
double average = DoubleStream.of(values).average().orElse(0);
double largest = DoubleStream.of(values).max().orElse(Double.MIN_VALUE);
double smallest = DoubleStream.of(values).min().orElse(Double.MAX_VALUE);
```

The elegance of streams becomes even more convincing when you process objects, not numbers. Here is the loop for comparing the average area from a list of countries:

```java
double total = 0;
for (Country country : countries)
{
    total = total + country.getArea();
} 
double average = 0;
if (values.length > 0) { average = total / countries.size(); } 
```

With streams, that computation is much simpler:

```java
double average = countries.stream()
    .mapToDouble(c -> c.getArea())
    .average()
    .orElse(0);
```

19.10.3 Counting Matches

In Chapter 6, you saw how to count the number of spaces in a string.

```java
int spaces = 0;
for (int i = 0; i < str.length(); i++)
{
    char ch = str.charAt(i);
    if (ch == ' ')
    {
        spaces++;
    }
}
```

In general, it is easy to count matches in a stream by filtering and counting. To express this example with streams, turn the string into a stream instead of looping over the positions.

```java
long spaces = str.codePoints() 
    .filter(ch -> ch == ' ') 
    .count();
```

19.10.4 Element Separators

You need one fewer separator than elements to separate values with commas or vertical lines, such as

```
32 | 54 | 67.5 | 29 | 35
```
In Chapter 7, this was solved like this:

```java
double[] values = . . .;
String result = "";
for (int i = 0; i < values.length; i++)
{
    if (i > 0)
    {
        result = result + " | ";
    }
    result = result + values[i];
}
```

With streams, this is easily done with the `joining` collector, but you first have to convert the values to strings:

```java
String result = DoubleStream.of(values)
    .mapToObj(v -> "" + v)
    .collect(Collectors.joining(" | "));
```

### 19.10.5 Linear Search

With streams, it is very easy to search for a match. Suppose you are asked for the first value larger than 100 in an array. That is

```java
OptionalDouble result = DoubleStream.of(values)
    .filter(v -> v > 100)
    .findFirst();
```

But the example in Chapter 7 asked for the first element that is equal to 100. When you call `filter(v -> v == 100).findFirst()`, you either get 100 or no match, whereas in Chapter 7, you also got the position of the match. If you want that with streams, use a stream of positions:

```java
int n = values.length;
int pos = IntStream.range(0, n)
    .filter(i -> values[i] == 100)
    .findFirst()
    .orElse(-1);
```

### 19.10.6 Comparing Adjacent Values

Section 6.7.6 has an algorithm for finding adjacent values in a sequence of inputs:

```java
double input = in.nextDouble();
while (in.hasNextDouble())
{
    double previous = input;
    input = in.nextDouble();
    if (input == previous) { System.out.println("Duplicate input"); }
}
```

This is an example of an algorithm that cannot easily be adapted to Java streams. Stream operations such as `map`, `filter`, and `findFirst` process their elements in isolation, without looking at their neighbors. It is possible to overcome this by storing the previous value (see Exercise E19.20). But that solution doesn’t work for parallel streams.

Streams are not a solution for all data processing problems, but as you have seen, in many common cases they offer a solution that is clearer than a loop.
45. How do you compute the sum of all positive values in an array of integers?

46. How do you find the position of the last space in a string, using streams?

47. How do you get the smallest area of any country from a list of Country objects, assuming that the Country class has a method public double getArea()? How do you get the country with that area?

48. Someone proposes the following way to find the smallest element in a stream:

```java
smallest = stream.sorted().limit(1).findAny().get();
```

Will it work? If so, is it a good idea?

49. Why can’t one use the distinct method to solve the problem of removing adjacent duplicates from a stream?

**Practice It**  
Now you can try these exercises at the end of the chapter: R19.13, E19.18, E19.19.

---

**HOW TO 19.1**

**Working with Streams**

Streams are particularly useful for extracting information from large real-world data sets. This How To tells you how to process a data set by transforming the data and extracting the features of interest.

**Step 1**  
Get the data.

Generally, when you are asked to process a data set, you need to work at getting it into a form that a Java program can handle. A stream is a sequence of objects of a particular class that describes the data, so you need to design that class first. Suppose you process information about movies. Then you need a class Movie, with the information that is in your data set, such as the title, director, actors, and so on. (See Worked Example 19.2 for more on this particular example.) Similarly, if you process data about countries, you will want a Country class.

Next, you need to read the data. If you are lucky, the data is in a text file in a format that you can easily process, using the principles that you learned in Chapter 11. But sometimes, data is in a spreadsheet. Then you should export the data into a text format called CSV (comma-separated values). A row in such a file might look like

```
United States,North America,318892103
```

You can easily read such a row, split it into fields, and build a Country object.

If the fields can contain commas, choose a different field separator (such as the | character) before exporting your data.

You can now read the data into a list, such as an ArrayList<Country> if you are processing Country objects.

**Step 2**  
Make a stream.

If you read your data into an array list, just call the stream method and you have your stream.

```java
List<Country> countryList = new ArrayList<>();
// Fill countryList
Stream<Country> countries = countryList.stream();
```

Alternatively, if it happens that your input file has one line per item, then you can use a more efficient approach. First read in a stream of lines and then transform each line into an object.
For example,
```java
try (Stream<String> lines = Files.lines(path))
{
    Stream<Country> countries = lines.map(line -> Country.parse(line));
    . . .
}
```
The parse method is a static helper method that breaks an input line into fields and constructs a Country object.

**Step 3** Transform the stream.

Often, you will want to analyze just a part of the provided data; perhaps only the movies since 1990 or only countries in Africa. Then you use filter to pick the desired items.
```java
Stream<Country> africanCountries = countries
    .filter(c -> c.getContinent().equals("Africa"));
```
At times, you may only be interested in a part of the data, or you want the data in a different form. Then you use the map method to transform the items. For example, if you only care about the continents,
```java
Stream<String> continents = countries.map(c -> c.getContinent());
```
When you map to a numeric type, you should use the methods mapToInt, mapToLong, or mapToDouble. The resulting primitive-type streams store their values more efficiently than object streams, and you can compute their sums and averages.

The sorted method sorts the elements of the stream.

Another useful transformation is the distinct method that drops duplicates from the stream.

Of course, you can combine these operations. Here are the continents that have countries with at least a hundred million inhabitants:
```java
Stream<String> continentsWithPopulousCountries = countries
    .filter(c -> c.getPopulation() >= 100_000_000)
    .map(c -> c.getContinent())
    .distinct();
```
Sometimes, it is not so obvious what you want to do with the stream elements. You may want to skip ahead to the next step and figure out what the stream should look like so that you can easily collect the desired result, then figure out how to transform it.

**Step 4** Collect the results.

There are four kinds of result that you can harvest from a stream, and we'll look at each in turn:

1. A single element from the stream
2. A collection of values
3. A count, sum, or average
4. A map that associates groups with group properties

A single element can be one of the following:

- The largest or smallest, by some ordering: max, min
- The first, or any, element fulfilling a condition: filter followed by findFirst or findAny

For example, in a stream of countries, you may want to find the one with the largest population, or any country with a population of at least a hundred million:
```java
Optional<Country> aPopulousCountry = countries
    .filter(country -> country.getPopulation() > 100_000_000)
    .findFirst();
```
Now suppose you want to collect multiple values. If you like, you can restrict the number of values with the limit method before you collect the result. For example, adding limit(10) to the processing pipeline returns at most ten elements.
You need to decide where you want your answer:

- In an array: `toArray( ElementType::new )`
- In a list: `collect( Collectors.toList() )`
- In a set: `collect( Collectors.toSet() )`
- In a string: `collect( Collectors.joining( "", "" ) )`

For example, suppose you want the ten most populous countries in a list. This requires a little cleverness. Because `limit( 10 )` yields the first ten values, you want to sort so that the most populous elements are at the beginning of the stream. You can flip the comparison like this:

```
List<Country> mostPopulous = countries
    .sorted( (c, d) -> Double.compare( d.getPopulation(), c.getPopulation() ) )
    .limit( 10 )
    .collect( Collectors.toList() );
```

In the preceding cases, the result consisted of one or more stream values. Sometimes, you want a summary value instead. You have these choices:

- The count of the elements: `count`
- The sum, average, maximum, or minimum of a numeric stream: `sum`, `average`, `max`, `min`
- A boolean indicating whether all, some, or none of the stream elements fulfill a condition: `allMatch`, `anyMatch`, `noneMatch`

For example, to get the average population, first transform countries to populations, and then get the average:

```
double average = countries
    .mapToInt( country -> country.getPopulation() )
    .average()
    .orElse( 0 );
```

The call to `orElse` is necessary because `average` returns an `OptionalDouble`.

Finally, let's look at the most complex case, where each element belongs to a group, and you want to collect results for each group. First, you need to come up with a function that yields the group of an element. If you want to group countries by continent, that function is `c -> c.getContinent()`. If you want to group them by the first letter of the name, use the function `c -> c.getName().substring( 0, 1 )`.

You pass that function to the `groupingBy` method. The result is a map. If you do nothing else, each group key is associated with a list of the group's elements. The call

```
Map<String, List<Country>> countriesByContinent = countries
    .collect( Collectors.groupingBy( c -> c.getContinent() ) );
```

yields a map, where, for example, `countriesByContinent.get( "Africa" )` is a list of all countries in Africa.

You can apply another collector to each group, computing

- The number of group elements: `counting`
- The sum of a numeric attribute: `summingInt`, `summingDouble`, `summingLong`
- The average of a numeric attribute: `averagingInt`, `averagingDouble`, `averagingLong`
- The largest or smallest element, as determined by a comparator: `maxBy`, `minBy`

Here is how you can get the average population per continent:

```
Map<String, Double> averagePopulationByContinent = countries
    .collect( Collectors.groupingBy( c -> c.getContinent(), Collectors.averagingInt( c -> c.getPopulation() ) ) );
```
Understand the concept of streams.

- A stream is an immutable sequence of values that are processed lazily.
- Lazy processing means to defer operations until they are needed, and to skip those that are not needed.

Be able to create streams.

- The `Stream.of` static method and the `stream` methods of collection classes yield streams.

Collect results from streams.

- To turn a stream into an array or collection, use the `toArray` or `collect` methods.

Determine how to transform streams into a form from which you can collect results.

- The `map` method applies a function to all elements of a stream, yielding another stream.
- The `filter` method yields a stream of all elements fulfilling a condition.

Master the syntax of lambda expressions.

- A lambda expression consists of one or more parameter variables, an arrow `->`, and an expression or block yielding the result.
- A functional interface is an interface with a single abstract method.
- A lambda expression can be converted to an instance of a functional interface.

Work with values of the `Optional` type.

- The `Optional` class is a wrapper for objects that may or may not be present.
- Use the `orElse` method to obtain the value of an `Optional` or, if no value is present, an alternative.
Know the terminal stream operations.

- A terminal operation triggers the lazy operations on a stream and yields a non-stream value.

Work with streams that contain values of primitive types.

- The `IntStream.range` method yields a stream of consecutive integers.
- The `mapToInt` method applies an int-valued function to stream elements and yields an `IntStream`.
- Primitive-type streams have methods `sum`, `average`, `max`, and `min`.

Group stream results with common characteristics.

- Using grouping collectors, you can group together elements with the same key.
- A grouping collector can apply another collector to each group.

Express common algorithms with stream operations.

- Many common processing tasks that involve sequences of values can be carried out easily with streams.
- Streams are not easily applicable to algorithms that compare adjacent elements of a sequence.

**Standard Library Items Introduced in this Chapter**

<table>
<thead>
<tr>
<th>Java Class</th>
<th>Method/Constructor</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>java.util.Collection&lt;T&gt;</code></td>
<td><code>stream</code></td>
</tr>
<tr>
<td><code>java.util.Comparator&lt;T&gt;</code></td>
<td><code>comparing</code>, <code>thenComparing</code></td>
</tr>
<tr>
<td><code>java.util.stream.Collectors</code></td>
<td><code>averagingDouble</code>, <code>averagingInt</code>, <code>averagingLong</code></td>
</tr>
<tr>
<td><code>java.util.stream.Collectors</code></td>
<td><code>counting</code>, <code>groupingBy</code>, <code>joining</code>, <code>maxBy</code>, <code>minBy</code></td>
</tr>
<tr>
<td><code>java.util.stream.Collectors</code></td>
<td><code>summingDouble</code>, <code>summingInt</code>, <code>summingLong</code></td>
</tr>
<tr>
<td><code>java.util.stream.Collectors</code></td>
<td><code>toList</code>, <code>toSet</code></td>
</tr>
<tr>
<td><code>java.util.stream.DoubleStream</code></td>
<td><code>of</code></td>
</tr>
<tr>
<td><code>java.util.stream.IntStream</code></td>
<td><code>range</code> (use with <code>IntStream</code> only)</td>
</tr>
<tr>
<td><code>java.util.stream.LongStream</code></td>
<td></td>
</tr>
<tr>
<td><code>java.nio.file.Files</code></td>
<td><code>lines</code></td>
</tr>
<tr>
<td><code>java.util.Optional&lt;T&gt;</code></td>
<td><code>get</code>, <code>getAsDouble</code>, <code>getAsInt</code>, <code>getAsLong</code></td>
</tr>
<tr>
<td><code>java.util.OptionalDouble</code></td>
<td><code>ifPresent</code>, <code>isPresent</code>, <code>orElse</code></td>
</tr>
<tr>
<td><code>java.util.OptionalInt</code></td>
<td></td>
</tr>
<tr>
<td><code>java.util.OptionalLong</code></td>
<td></td>
</tr>
<tr>
<td><code>java.util.function.Predicate&lt;T&gt;</code></td>
<td><code>test</code></td>
</tr>
<tr>
<td><code>java.util.stream.Stream&lt;T&gt;</code></td>
<td><code>allMatch</code>, <code>anyMatch</code>, <code>collect</code>, <code>count</code>, <code>distinct</code>, <code>filter</code>, <code>findFirst</code>, <code>findAny</code>, <code>generate</code>, <code>forEach</code>, <code>limit</code>, <code>map</code>, <code>mapToDouble</code>, <code>mapToInt</code>, <code>mapToLong</code>, <code>noneMatch</code>, <code>of</code>, <code>parallel</code>, <code>skip</code>, <code>sorted</code>, <code>toList</code>, <code>toSet</code>, <code>toArray</code>, <code>toArray</code>, <code>String</code>, <code>codePoints</code></td>
</tr>
</tbody>
</table>
R19.1 Provide expressions that compute the following information about a `Stream<String>`.

a. How many elements start with the letter a?

b. How many elements of length greater than ten start with the letter a?

c. Are there at least 100 elements that start with the letter a? (Don’t count them all if there are more.)

R19.2 How can you collect five long words (that is, with more than ten letters) from an `ArrayList<String>` without using streams? Compare your solution with the code fragment in Section 19.1. Which is easier to understand? Why?

R19.3 What is the difference between these two expressions?

```java
words.filter(w -> w.length() > 10).limit(100).count()
words.limit(100).filter(w -> w.length() > 10).count()
```

R19.4 Give three ways of making a `Stream<String>` (or five, including the ones described in Special Topic 19.1).

R19.5 How can you place all elements from a `Stream<Integer>` into

a. a `List<Integer>`?

b. an `Integer[]` array?

c. an `int[]` array?

R19.6 How do you turn a `Stream<Double>` into a `Stream<String>`, with each number turned into the equivalent string? How do you turn it back into a `Stream<Double>`?

R19.7 Give three ways of making a lambda expression that applies the `length` method to a `String` (or four if you read Special Topic 19.2).

R19.8 Given an `Optional<String>`, what are three different ways of printing it when it is present and not printing anything when it isn’t? Which of these can be adapted to print the string “None” if no string is present?

R19.9 Describe five different ways of producing an `IntStream`. Which of them can be adapted to producing a `DoubleStream`?

R19.10 Suppose you want to find the length of the longest string in a stream. Describe two approaches: using `mapToInt` followed by the `max` method of the `IntStream` class, and calling the `max` method of `Stream<String>`. What are the advantages and disadvantages of each approach?

R19.11 List all terminal operations on object streams and primitive-type streams that have been discussed in this chapter.

R19.12 List all collectors that were introduced in this chapter.

R19.13 Explain the values used in the `orElse` clauses in Section 19.10.2.
• **E19.1** Write a program that reads all lines from a file and, using a `Stream<String>`, prints how many of them contain the word “the”.

• **E19.2** Write a program that reads all words from a file and, using a `Stream<String>`, prints how many of them are the word “the”.

• **E19.3** Write a program that reads all lines from a file and, using a `Stream<String>`, prints all lines containing the word “the”.

• **E19.4** Write a program that reads all words from a file and, using a `Stream<String>`, prints all distinct words with at most four letters (in some order).

• **E19.5** Write a method
  
  ```java
  public static <T> String toString(Stream<T> stream, int n)
  ```

  that turns a `Stream<T>` into a comma-separated list of its first `n` elements.

• **E19.6** The static `getAvailableCurrencies` method of the `java.util.Currency` class yields a set of `Currency` objects. Turn it into a stream and transform it into a stream of the currency display names. Print them in sorted order.

• **E19.7** Write a lambda expression for a function that turns a string into a string made of the first letter, three periods, and the last letter, such as "W...d". (Assume the string has at least two letters.) Then write a program that reads words into a stream, applies the lambda expression to each element, and prints the result. Filter out any words with fewer than two letters.

• **E19.8** Write a program that sorts an array of bank accounts by increasing balance. Pass an appropriate lambda expression to `Arrays.sort`.

• **E19.9** Write a program that reads in words from a file and prompts the user for another word. Print the longest word from the file that contains the given word, or “No match” if the word does not occur in the file. Use the `max` method of `Stream<String>`.

• **E19.10** Write a method
  
  ```java
  public static Optional<Integer> smallestProperDivisor(int n)
  ```

  that returns the smallest proper divisor of `n` or, if `n` is a prime number, a value indicating that no result is present.

• **E19.11** Write a program that reads an integer `n` and then prints all squares of the integers from 1 to `n` that are palindromes (that is, their decimal representation equals its reverse). Use `IntStream.range`, `map`, and `filter`.

• **E19.12** Write a method
  
  ```java
  public static Stream<String> characters(String str)
  ```

  that yields a stream of strings of length 1 that contains the characters of the string `str`. Use the `codePoints` method and skip code points greater than 65535. Extra credit if you don’t skip them and instead produce strings of length 2.
E19.13 Read all words from a file and print the one with the maximum number of vowels. Use a `Stream<String>` and the `max` method. Extra credit if you define the comparator with the `Comparator.comparing` method described in Special Topic 19.4.

E19.14 Read all words from a file into an `ArrayList<String>`, then turn it into a parallel stream. Use the dictionary file `words.txt` provided with the book’s companion code. Use filters and the `findAny` method to find any palindrome that has at least five letters, then print the word. What happens when you run the program multiple times?

E19.15 Read all words in a file and group them by length. Print out how many words of each length are in the file. Use `collect` and `Collectors.groupingBy`.

E19.16 Read all words in a file and group them by the first letter (in lowercase). Print the average word length for each initial letter. Use `collect` and `Collectors.groupingBy`.

E19.17 Assume that a `BankAccount` class has methods for yielding the account owner’s name and the current balance. Write a function that, given a list of bank accounts, produces a map that associates owner names with the total balance in all their accounts. Use `collect` and `Collectors.groupingBy`.

E19.18 Write a program that reads a `Stream<Country>` from a file that contains country names and numbers for the population and area. Print the most densely populated country.

E19.19 Write a function that returns a list of all positions of a given character in a string. Produce two versions—one with streams and one without. Which one is easier to implement?

E19.20 Find all adjacent duplicates of a `Stream<String>`, by using a predicate that compares each element against the previous one (stashed away in an array of length 1), updates the array, and returns the result of the comparison. You have to be careful with the first element.

P19.1 In a stream of random integers, filter out the even ones, call `limit(n)`, and count the result. Set `n` to 10, 100, 1000, and so on. Measure the amount of time these operations take with a regular stream and a parallel stream. How big does `n` have to be for parallel streams to be faster on your computer?

P19.2 Write a program that generates an infinite stream of integers that are perfect squares and then displays the first `n` of them that are palindromes (that is, their decimal representation equals its reverse). Extra credit if you use `BigInteger` so that you can find solutions of arbitrary length.

P19.3 Repeat Exercise P19.2 with prime numbers instead of perfect squares.

P19.4 Produce an infinite stream that contains the factorials 1!, 2!, 3!, and so on. Hint: First produce a stream containing arrays [1, 1!], [2, 2!], [3, 3!], and so on. Use `BigInteger` values for the factorials.

P19.5 Worked Example 19.1 showed you how to find all words with five distinct vowels (which might occur more than once). Using a similar approach, find all words in which each vowel occurs exactly once.
P19.6 Using an approach similar to that in Worked Example 19.1, find all words with length of at least ten in which no letter is repeated. What is the longest one? How many such words exist for each length?

P19.7 Using an approach similar to that in Worked Example 19.1, find all words with exactly one vowel (which might be repeated). What is the longest one? How many such words exist for each length?

P19.8 Perhaps the reason that so many movie titles start with the letter A is that their first word is “A” or “An”? Count how many movies in the data set of Worked Example 19.2 start with these words.

P19.9 What are the 100 most common initial words in movie titles contained in the data set in Worked Example 19.2?

P19.10 Write a program to determine how many actors there are in the data set in Worked Example 19.2. Note that many actors are in multiple movies. The challenge in this assignment is that each movie has a list of actors, not a single actor, and there is no ready-made collector to form the union of these lists. However, there is another collect method that has three parameters:
- A function to generate an instance of the target
- A function to add an element to a target
- A function to merge two targets into one

For example,
```
stream.collect(
    () -> 0,
    (t, e) -> t + e,
    (t, u) -> t + u)
```
computes the sum of elements in a `Stream<Integer>`. Note that the last function is only needed for parallel streams.

Define methods for generating a set, adding a list of actors into one, and for combining two sets.

P19.11 Write a program to determine the 100 actors with the most movies, and the number of movies in which they appear. For each movie, produce a map whose keys are the actors, all with value 1. Merge those maps as in Exercise P19.10. Then extract the top 100 actors from a stream of actors.

P19.12 Find an online database with a large number of cities and their locations. Write a program that prints all cities within a given distance from a location. (You will need to find a formula for computing the distance between two points on Earth.)

---

ANSWERS TO SELF-CHECK QUESTIONS

1. long count = stream
   .filter(b -> b.getBalance() < 0)
   .count();

2. Stream<String> result = stream.filter(
    w -> w.length() > 0)

3. List<String> result = new ArrayList<>();
   int i = 0;
   while (i < strings.size() && result.size() < 5)
   {
     String s = strings.get(i);
     if (s.length() > 10) { result.add(s); }
   }

4. long result = stream.filter(
    w -> w.length() == 10).count();
As a stream, that is
Stream<String> result = stream.filter(
    w -> w.length() == 10).limit(1);
You will see in Section 19.3 how to get the
answer as a string. And Section 19.6 will pres-
ent an easier way to obtain this result.

boolean atLeastTen = stream.filter(
    w -> w.length() == 3)
    .limit(10).count() == 10;
Because stream processing is lazy, the limit
operation stops filtering as soon as ten
matches have been found.

For example,
Stream<Color> colors = Stream.of(
    Color.RED, Color.WHITE, Color.BLUE);

long count = list.stream()
    .filter(w -> w.length() <= 3).count();
long count = Stream.of(array)
    .filter(w -> w.length() <= 3).count();

try (Stream<String> lines = Files.lines(
    Paths.get("input.txt")))
{
    long count = lines.filter(
        l -> l.length() > 80).count();
    ...
}

List<String> result = list.stream()
    .filter(w -> w.length() > 10)
    .collect(Collectors.toList());
Set<String> result = list.stream()
    .filter(w -> w.length() > 10)
    .collect(Collectors.toSet());
String result = list.stream()
    .filter(w -> w.length() > 10)
    .limit(1)
    .collect(Collectors.toList())
    .get(0);

String result = list.stream()
    .filter(w -> w.length() > 10)
    .limit(1)
    .collect(Collectors.toList())
    .toArray(String[]::new)[0];

If you omitted the call to limit(1), all strings
of length greater than 10 would be collected
and converted to a list or array. With the call
to limit, collecting stops as soon as the first
match has been found.

Stream<String> result = words
    .map(w -> w.toLowerCase())
    .filter(w -> w.substring(0, 1)
        .equalsIgnoreCase("a"));

or
Stream<String> result = words
    .map(w -> w.toLowerCase())
    .filter(w -> w.substring(0, 1)
        .equals("a"));

Stream<Integer> result = words
    .map(w -> w.length());
List<String> result = list.stream()
    .sorted()
    .limit(10)
    .collect(Collectors.toList());

int result = list.stream()
    .filter(w -> w.length() == 3)
    .distinct()
    .count();

Put all words of length 3 into a set and get its
size:
Set<String> words = new HashSet<>();
for (String w : list)
{
    if (w.length() == 3)
    {
        words.add(w);
    }
}
int result = words.size();

(x, y) -> (x + y) / 2.0
w -> w.substring(0, 1).equals(
    w.substring(w.length() - 1))

It is a predicate that tests whether a string is in
uppercase.

It is a stream consisting of all words that are
entirely in uppercase.

It is a Stream<Boolean> with values Boolean.TRUE
and Boolean.FALSE (the wrappers for true and
false), depending on whether the elements of
words were entirely in uppercase or not.

String word = wordList.stream()
    .findFirst(w -> w.contains("a"))
    .orElse(""); or
Stream<String> result = words
    .map(w -> w.toLowerCase())
    .filter(w -> w.substring(0, 1)
        .equals("a");

String word = "";
wordList.stream()
    .findFirst(w -> w.contains("a"))
    .ifPresent(v -> { word = v; });
Note that the previous solution was better
because it did not involve any “side effect”.

Optional<String> optResult = wordList.stream()
    .findFirst(w -> w.contains("a");
String word = "";
if (optResult.isPresent())
{
Answers to Self-Check Questions

29. `word = wordList.stream()
    .filter(w -> w.contains("a"))
    .skip(9)
    .findFirst()
    .orElse(""`;

30. `public static Optional<Double> reciprocal(double x) {
    if (x == 0) { return Optional.empty(); }
    else { return Optional.of(1 / x); }
} `;

31. `result = words.parallel()
    .filter(w -> w.length() > 10
            && w.endsWith("y"))
    .findFirst()
    .orElse(""`;

32. `boolean result = words.anyMatch(
    w -> w.startsWith("q")
            && w.endsWith("y"));`;

33. `Once you invoke the terminal operation qys. count(), you can no longer invoke any operations on the stream.`

34. `You can’t call findAny twice, so you should use limit to limit the stream to two results and then collect it to an array or list. `

35. `They all can short circuit: allMatch returns false as soon as it finds an element that doesn’t match, and anyMatch and noneMatch return as soon as they find an element that matches, with return values true and false respectively.`

36. `double sum = accounts.stream()
    .mapToDouble(a -> a.getBalance())
    .sum();`;

37. `double average = accounts.stream()
    .mapToDouble(a -> a.getBalance())
    .average();
    .orElse(0);`;

38. `int longestLength = wordList.stream()
    .mapToInt(w -> w.length())
    .max()
    .orElse(0);`;

39. `int longestLength = wordList.stream()
    .filter(w -> w.startsWith("z"))
    .mapToInt(w -> w.length())
    .min()
    .orElse(0);`

40. It is a map `{ "M" -> ["Mary"], "h" -> ["had"], "a" -> ["a"], "l" -> ["little", "lamb"] }.`

41. It is a map `{ "M" -> 1, "h" -> 1, "a" -> 1, "l" -> 2 }.`

42. `Map<Integer, List<String>> groups = wordList.stream()
    .collect(Collectors.groupingBy(
    w -> w.length));`;

43. `Map<String, Double> averages = Stream.of(words)
    .collect(Collectors.groupingBy(
    w -> w.substring(0, 1),
    Collectors.averagingInt(
    w -> w.length())));`;

44. `Map<String, Double> averages = Stream.of(words)
    .collect(Collectors.groupingBy(
    w -> w.substring(0, 1),
    Collectors.averagingInt(
    w -> w.length())));`;

45. `int sum = IntStream.of(values)
    .filter(n -> n > 0)
    .sum();`;

46. `There are two possible approaches. You can collect all matches and then pick the last one. `

47. `double smallest = countries.stream()
    .mapToDouble(c -> c.getArea())
    .min();`;

To get the country, you could now search:

Optional<Country> smallestCountry = countries.stream()
    .findAny(c -> c.getArea() == smallest);

But it is more efficient to search for the country with the minimal area. Then you need to specify a comparator.

Optional<Country> smallestCountry = countries.stream()
    .min((c, d) -> Double.compare(c.getArea(), d.getArea()));`
Or, using Special Topic 19.4,

```java
Optional<Country> smallestCountry = countries.stream()
    .min(Comparator.comparing(c -> c.getArea()));
```

48. Yes, it will work, provided there is at least one element in the stream. But it’s not a good idea because sorting is much less efficient than computing the minimum.

49. The `distinct` method removes all duplicates, not just adjacent ones.
WORKED EXAMPLE 19.1  Word Properties

It is fun to find words with interesting properties. To see how streams make this easy, download the code for Worked Example 19.1 from the companion code for this book.

Problem Statement  The French word oiseau has five distinct vowels, which is pretty nifty. Are there English words like that? Just a few or a lot? Which words are the shortest and longest among them?

Step 1  Get the data.

In this case, we need a list of English words. The companion code for this book contains a copy of the words.txt file that is available on computers with a Unix-based operating system.

Step 2  Make a stream.

In this case, it is very easy to get a stream of words because the input file has one word per line. Simply call

```java
try (Stream<String> lines = Files.lines(Paths.get("words.txt")))
{
    ...
}
```

Step 3  Transform the stream.

First, the input contains many words ending with 's, such as Alice's. We don't want them:

```java
try (Stream<String> lines = Files.lines(Paths.get("words.txt")))
{
    Stream<String> words = lines.filter(w -> !w.endsWith("'s"));
    ...
}
```

And we only want words that have all five vowels. This is complex enough that we should write a separate method:

```java
public static boolean hasFiveVowels(String word)
```

This method needs to check whether the word contains all five vowels. As an aside, this too can be done with streams:

```java
return word.toLowerCase().codePoints() // A stream of code points
    .filter(c -> c == 'a' || c == 'e' || c == 'i' || c == 'o' || c == 'u')
    .distinct() // The distinct vowels in word
    .count() == 5;
```

Now that we have this method implemented, we add a second filter to our stream of words:

```java
Stream words = lines
    .filter(w -> !w.endsWith("'s"))
    .filter(w -> hasFiveVowels(w));
```

Step 4  Collect the results.

Once we have the words that we want, we can ask questions about them. How many are there?

```java
long count = words.count();
```

It turns out that there are 469. Let's see a few.

```java
List<String> examples = words
    .limit(20)
    .collect(Collectors.toList());
```
The result is a list of these words: Aurelio, Aureomycin, Australopithecus, Austronesian, Barquisimeto, Beaujolais, Beauvoir, Byelorussia, Carboniferous, Cointreau, Ecuadorian, Ecuadorians, Figueroa, Hermaphroditus, Milquetoast, Mozambique, Teotihuacan, abstemious, accentuation, adulteration.

What is the shortest one?

```java
String shortest = words
    .min((s, t) -> s.length() < t.length())
    .orElse("");
```

That’s the word Aurelio. Are there others of the same length?

```java
List<String> allShortest = words
    .filter(w -> w.length() == 7)
    .collect(Collectors.toList());
```

This yields two words: Aurelio and sequoia. Not bad—just one letter longer than the French *oiseau*. I’ll leave it to you to figure out how to get the longest word—it is counterrevolutionaries.
WORKED EXAMPLE 19.2  A Movie Database

In this Worked Example, we analyze a large database of movies and use streams to obtain interesting statistics from the data. To follow along, download the companion code for Worked Example 19.2.

Problem Statement  The file movies.txt in the book's companion code has information about 23,000 movies, taken from the database of facts at http://freebase.com. Each movie has a year, title, and lists of directors, producers, and actors. What interesting facts can you find?

Step 1  Get the data.

The movies.txt file has five lines for each movie which look like this:

Name: Five Easy Pieces
Year: 1970
Directed by: Bob Rafelson
Produced by: Bob Rafelson, Richard Wechsler, Harold Schneider
Actors: Jack Nicholson, Karen Black, Billy Green Bush, more...

First, let’s come up with a class that describes a movie:

```java
public class Movie{
    private String title;
    private int year;
    private List<String> directors;
    private List<String> producers;
    private List<String> actors;

    public Movie(String title, int year, List<String> directors, List<String> producers, List<String> actors) { . . . }

    public String getTitle() { return title; }
    // Accessors for the other fields ...
}
```

Next, we need to read in the movies. Because we need to consume five input lines per movie, there is nothing to be gained by reading the input as a stream of lines. Instead, we just put the movies into an ArrayList:

```java
public static List<Movie> readMovies(String filename) throws IOException
{
    List<Movie> movies = new ArrayList<>();
    try (Scanner in = new Scanner(new File(filename)))
    {
        while (in.hasNextLine())
        {
            String nameLine = in.nextLine();
            String yearLine = in.nextLine();
            String directorsLine = in.nextLine();
            String producersLine = in.nextLine();
            String actorsLine = in.nextLine();
            movies.add(new Movie(getString(nameLine),
                               Integer.parseInt(getString(yearLine)),
                               getList(directorsLine),
                               getList(producersLine),
                               getList(actorsLine)));
```
private static String getString(String line) {
    int colon = line.indexOf(":");
    return line.substring(colon + 1).trim();
}

private static List<String> getList(String line) {
    return Stream.of(getString(line).split(",")).
        collect(Collectors.toList());
}

Here, getString is a helper method that strips off the field header, and getList is a helper that
breaks up a comma-separated list:

Step 2 Make a stream.
Because we have a method for reading a collection of Movie objects, simply call

List<Movie> movieList = readMovies("movies.txt");
Stream<Movie> movies = movieList.stream();

Step 3 Transform the stream.
Now we are ready to work with the data. The problem statement was rather vague. What
interesting facts might be hidden in the data? Let’s start with something simple: Are there any
movie titles that start with the letter X?

This is a simple application of map and filter: Map each movie to its title, and filter the ones
that start with an X.

List<String> result1 = movieList.stream()
    .map(m -> m.getTitle())
    .filter(t -> t.startsWith("X"))
    .collect(Collectors.toList());

Indeed, they are a few: XX/XY, Xiu Xiu: The Sent Down Girl, X-15, X Marks the Spot,
X-Men: First Class, and so on. In the next step, you will see how many movies start with a
given letter.

Is it common for a director to also be an actor? To answer this question, we want to check
for each movie whether the list of directors and the list of actors have an element in common.
You can map a movie to the intersection of the two lists (using a helper method), and then
count the ones with nonempty intersections:

long count = movieList.stream()
    .map(m -> intersect(m.getDirectors(), m.getActors()))
    .filter(l -> l.size() > 0)
    .count();

The Java library doesn’t have a method for computing the intersection of two collections, but
it is easy to provide one:

public static Set<String> intersect(Collection<String> a, Collection<String> b) {
    Set<String> intersection = new HashSet<>(a);
    intersection.retainAll(b);
    return intersection;
However, it is simpler to filter on the criterion without actually computing the intersection. For a given movie \( m \), we want to know if any of the directors is also an actor. Perhaps surprisingly, this can be expressed more concisely with a stream than with methods from the Collection interface:

```java
public static boolean commonActorAndDirector(Movie m) {
    return m.getDirectors().stream().anyMatch(d -> m.getActors().contains(d));
}
```

Then simply compute

```java
long count = movieList.stream()
        .filter(m -> commonActorAndDirector(m))
        .count();
```

You could dispense with the `commonActorAndDirector` helper method, but then it would be quite hard to follow what is going on.

Which movie has the most actors? It is easy to get the maximum number. First map each movie to the size of the actor list, then get the maximum:

```java
int result2 = movieList.stream()
        .mapToInt(m -> m.getActors().size())
        .max()
        .orElse(0);
```

The call to `orElse` is necessary because `max` returns an `OptionalInt`.

As it turns out, the maximum is 100. But that only tells us that there is a movie with a hundred actors, not which one it is. To get that answer, we need to refine our strategy. Instead of mapping movies to numbers, we need to compute the “largest” movie, where one movie is larger than another if it has more actors:

```java
movieList.stream()
        .max((a, b) -> a.getActors().size() - b.getActors().size())
        .ifPresent(m -> System.out.println("Movie with most actors: " + m));
```

Note the call to the `ifPresent` method. The `max` method returns an `Optional<Movie>`. In general, it is not a good idea to call `get` because that would cause an exception if there was no result. In our case, we know there has to be one, so we could call `get` anyway. But it is just as easy to use the safe call to `ifPresent`, passing a function to print the movie.

Also, if you find the comparator cumbersome, you may want to read Special Topic 19.4. There is a more elegant way to describe it, as

```java
Comparator.comparing(m -> m.getActors().size())
```

The point of this example is that it isn’t always necessary to process a stream. Sometimes you can obtain the result directly.

**Step 4** Collect the results.

In the preceding examples, it was easy to collect the results. Let’s look at a couple of examples that are more challenging. First, we want to know how many movies start with a given letter. This is a typical use of `groupingBy` with a secondary collector:

```java
Map<String, Long> firstLetters = movieList.stream()
        .collect(Collectors.groupingBy(
            m -> m.getTitle().substring(0, 1),
            Collectors.counting()));
```

Almost 6,000 movies start with the letter T. There is a simple reason that you can verify by running the query

```java
movieList.stream()
        .filter(m -> m.getTitle().startsWith("The "))
        .count();
```
Almost 5,000 titles start with the word “The”.
Interestingly, the letters A, B, and C have high frequencies, probably because there is some
incentive in picking a title that shows up at the top of alphabetical listings.
Who is the most prolific director? This is not so easy to answer because a movie can have
more than one director. Fortunately, this only happens with about five percent of movies, so
let’s just pick the first one. There are, however, a number of movies with no directors in the
data set. We filter those out first. Then we can group by the first director:

```java
Map<String, List<Movie>> moviesByDirector = movieList.stream()
    .filter(m -> m.getDirectors().size() > 0)
    .collect(Collectors.groupingBy(
        m -> m.getDirectors().get(0)));
```

This map associates all directors with a list of the movies that they directed. Unfortunately,
that’s a large map with about 10,000 entries. How can we find out which director had the most
movies? It’s the map entry with the longest list.

There is no need to use streams. The `Collections.max` method yields the largest value of a

```java
String mostProlificDirector = Collections.max(
    moviesByDirector.entrySet(),
    Comparator.comparing(e -> e.getValue().size())).getKey();
```

It turns out that this director is D. W. Griffith, a pioneer of silent films who directed over 150
movies. Which movies? To extract the titles, it is easy to use `map` with a stream:

```java
List<String> titles = moviesByDirector.get(mostProlificDirector)
    .stream()
    .map(m -> m.getTitle())
    .collect(Collectors.toList());
```

This example has shown you how you can discover facts in a data set. Once you start finding
interesting results, you can issue additional queries to dig deeper. Streams are a good tool for
exploring data because they let you focus on the “what, not how”, allowing you to generate
and refine queries quickly.
CHAPTER 20

GRAPHICAL USER INTERFACES

CHAPTER GOALS

To use layout managers to arrange user-interface components in a container
To use text components to capture and display text in a graphical application
To become familiar with common user-interface components, such as radio buttons, check boxes, and menus
To browse the Java documentation effectively

CHAPTER CONTENTS

20.1 LAYOUT MANAGEMENT 884
   CE1 By Default, Components Have Zero Width and Height 887
   ST1 Adding the main Method to the Frame Class 888

20.2 PROCESSING TEXT INPUT 888

20.3 CHOICES 894
   HT1 Laying Out a User Interface 901
   WE1 Programming a Working Calculator
   PT1 Use a GUI Builder 904

20.4 MENUS 905

20.5 EXPLORING THE SWING DOCUMENTATION 911
The graphical applications with which you are familiar have many visual gadgets for information entry: text fields, buttons, scroll bars, menus, and so on. In this chapter, you will learn how to create programs that use the most common user-interface components in the Java Swing toolkit. We also show you how to effectively use the documentation for Swing components, so you can make use of additional components in your applications.

20.1 Layout Management

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. In this section, you will see how to achieve more sophisticated arrangements.

20.1.1 Using Layout Managers

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());
```

![Figure 1](image.png)

Components Expand to Fill Space in the Border Layout
Now the panel is controlled by a border layout, not the flow layout. When adding a component, you specify the position, like this:

```java
    panel.add(component, BorderLayout.NORTH);
```

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:

```java
    JPanel buttonPanel = new JPanel();
    buttonPanel.setLayout(new GridLayout(4, 3));
    buttonPanel.add(button7);
    buttonPanel.add(button8);
    buttonPanel.add(button9);
    buttonPanel.add(button4);
    ... 
```

### 20.1.2 Achieving Complex Layouts

Sometimes you want to have a tabular arrangement of 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 20.1 on page 904.

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 label is in the northern position of the larger panel.
The following code produces the arrangement in Figure 3:

```java
JPanel keypadPanel = new JPanel();
keypadPanel.setLayout(new BorderLayout());
buttonPanel = new JPanel();
buttonPanel.setLayout(new GridLayout(4, 3));
buttonPanel.add(button7);
buttonPanel.add(button8);
// ... 
keypadPanel.add(buttonPanel, BorderLayout.CENTER);
JLabel display = new JLabel("0");
keypadPanel.add(display, BorderLayout.NORTH);
```

### 20.1.3 Using Inheritance to Customize Frames

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,

```java
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:

```java
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);
}

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?

6. Why does the FilledFrameViewer2 class declare the frame variable to have class JFrame, not FilledFrame?

7. How many Java source files are required by the application in Section 20.1.3 when we use inheritance to declare the frame class?

8. 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: R20.1, R20.3, E20.1.

By Default, Components Have Zero Width and Height

You must be careful when you add a painted component, such as a component displaying a car, to a panel. You add the component in the same way as a button or label:

    panel.add(button);
    panel.add(label);
    panel.add(carComponent);

However, the default size for a component is 0 by 0 pixels, and the car component will not be visible. The remedy is to call the setPreferredSize method, like this:

    carComponent.setPreferredSize(new Dimension(CAR_COMPONENT_WIDTH, CAR_COMPONENT_HEIGHT));

This is an issue only for painted components. Buttons, labels, and so on know how to compute their preferred size.
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:

```java
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.

20.2 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 4 and 6). 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.

20.2.1 Text Fields

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:

```java
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:

```java
JLabel rateLabel = new JLabel("Interest Rate: ");
```
20.2 Processing Text Input

You want to give the user an opportunity to enter all information into the text field before processing it. Therefore, you should supply a button that the user can press to indicate that the input is ready for processing.

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.

```java
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.

`section_2_1/InvestmentFrame2.java`

```java
/**
 * A frame that shows the growth of an investment with variable interest.
 */
public class InvestmentFrame2 extends JFrame {
    private static final int FRAME_WIDTH = 450;
    private static final int FRAME_HEIGHT = 100;
    private static final double DEFAULT_RATE = 5;
    private static final double INITIAL_BALANCE = 1000;
    private JLabel rateLabel;
    private JTextField rateField;
    private JButton button;
```
private JLabel resultLabel;
private double balance;

public InvestmentFrame2()
{
    balance = INITIAL_BALANCE;
    resultLabel = new JLabel("Balance: " + balance);

    createTextField();
    createButton();
    createPanel();

    setSize(FRAME_WIDTH, FRAME_HEIGHT);
}

private void createTextField()
{
    rateLabel = new JLabel("Interest Rate: ");

    final int FIELD_WIDTH = 10;
    rateField = new JTextField(FIELD_WIDTH);
    rateField.setText("" + DEFAULT_RATE);
}

/**
 * Adds interest to the balance and updates the display.
 */
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);
    }
}

private void createButton()
{
    button = new JButton("Add Interest");

    ActionListener listener = new AddInterestListener();
    button.addActionListener(listener);
}

private void createPanel()
{
    JPanel panel = new JPanel();
    panel.add(rateLabel);
    panel.add(rateField);
    panel.add(button);
    panel.add(resultLabel);
    add(panel);
}
20.2.2 Text Areas

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.

When constructing a text area, you can specify the number of rows and columns:

```java
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:

```java
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

```java
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 5, 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.

---

**Figure 5**
A Part of the Hierarchy of Swing User-Interface Components
To add scroll bars to a text area, use a JScrollPane, like this:

```java
JTextArea textArea = new JTextArea(ROWS, COLUMNS);
JScrollPane scrollPane = new JScrollPane(textArea);
```

Then add the scroll pane to the panel. Figure 6 shows the result.

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.

**section_2_2/InvestmentFrame3.java**

```java
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import javax.swing.JButton;
import javax.swing.JFrame;
import javax.swing.JLabel;
import javax.swing.JPanel;
import javax.swing.JScrollPane;
import javax.swing.JTextArea;
import javax.swing.JTextField;

/**
   A frame that shows the growth of an investment with variable interest, 
   using a text area.
*/
public class InvestmentFrame3 extends JFrame {
    private static final int FRAME_WIDTH = 400;
    private static final int FRAME_HEIGHT = 250;
    private static final int AREA_ROWS = 10;
    private static final int AREA_COLUMNS = 30;
    private static final double DEFAULT_RATE = 5;
    private static final double INITIAL_BALANCE = 1000;

    private JLabel rateLabel;
    private JTextField rateField;
    private JButton button;
    private JTextArea resultArea;
    private double balance;
    ...
public InvestmentFrame3()
{
    balance = INITIAL_BALANCE;
    resultArea = new JTextArea(AREA_ROWS, AREA_COLUMNS);
    resultArea.setText(balance + "\n");
    resultArea.setEditable(false);

    createTextField();
    createButton();
    createPanel();

    setSize(FRAME_WIDTH, FRAME_HEIGHT);
}

private void createTextField()
{
    rateLabel = new JLabel("Interest Rate: ");

    final int FIELD_WIDTH = 10;
    rateField = new JTextField(FIELD_WIDTH);
    rateField.setText("" + DEFAULT_RATE);
}

class AddInterestListener implements ActionListener
{
    public void actionPerformed(ActionEvent event)
    {
        double rate = Double.parseDouble(rateField.getText());
        double interest = balance * rate / 100;
        balance = balance + interest;
        resultArea.append(balance + "\n");
    }
}

private void createButton()
{
    button = new JButton("Add Interest");

    ActionListener listener = new AddInterestListener();
    button.addActionListener(listener);
}

private void createPanel()
{
    JPanel = new JPanel();
    panel.add(rateLabel);
    panel.add(rateField);
    panel.add(button);
    JScrollPane scrollPane = new JScrollPane(resultArea);
    panel.add(scrollPane);
    add(panel);
}

9. What happens if you omit the first JLabel object in the program of Section 20.2.1?

10. If a text field holds an integer, what expression do you use to read its contents?
### 20.3 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.

#### 20.3.1 Radio Buttons

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, the old station is automatically deselected.) For example, in Figure 7, 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, then add all buttons in the set to a **ButtonGroup** object:

```java
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,

```java
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 7, for example, the panels containing the Size radio buttons and Style check boxes have borders.

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:

```java
JPanel panel = new JPanel();
panel.setBorder(new EtchedBorder());
```

If you want to add a title to the border (as in Figure 7), 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:

```java
panel.setBorder(new TitledBorder(new EtchedBorder(), "Size"));
```

## 20.3.2 Check Boxes

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 7 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:

```java
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.

### 20.3.3 Combo Boxes

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 8).

![Figure 8 An Open Combo Box](image)

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.

```java
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`:

```java
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 9 shows how the components are arranged in the frame.
import javax.swing.JFrame;

/**
 * This program allows the user to view font effects.
 */
public class FontViewer {

    public static void main(String[] args) {
        JFrame frame = new FontFrame();
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.setTitle("FontViewer");
        frame.setVisible(true);
    }
}

import java.awt.BorderLayout;
import java.awt.Font;
import java.awt.GridLayout;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import javax.swing.ButtonGroup;
import javax.swing.JButton;
import javax.swing.JCheckBox;
import javax.swing.JComboBox;
import javax.swing.JFrame;
import javax.swing.JLabel;
import javax.swing.JPanel;
import javax.swing.JRadioButton;
import javax.swing.border.EtchedBorder;
import javax.swing.border.TitledBorder;

/**
   * This frame contains a text sample and a control panel
to change the font of the text.
*/
public class FontFrame extends JFrame {
   {
      private static final int FRAME_WIDTH = 300;
      private static final int FRAME_HEIGHT = 400;

      private JLabel label;
      private JCheckBox italicCheckBox;
      private JCheckBox boldCheckBox;
      private JRadioButton smallButton;
      private JRadioButton mediumButton;
      private JRadioButton largeButton;
      private JComboBox facenameCombo;
      private ActionListener listener;

      /**
       * Constructs the frame.
       */
      public FontFrame()
      {
         // Construct text sample
         label = new JLabel("Big Java");
         add(label, BorderLayout.CENTER);

         // This listener is shared among all components
         listener = new ChoiceListener();

         createControlPanel();
         setLabelFont();
         setSize(FRAME_WIDTH, FRAME_HEIGHT);
      }

      class ChoiceListener implements ActionListener
      {
         public void actionPerformed(ActionEvent event)
         {
            setLabelFont();
         }
      }

      /**
       * Creates the control panel to change the font.
       */
      public void createControlPanel()
      {
         JPanel facenamePanel = createComboBox();
         JPanel sizeGroupPanel = createCheckBoxes();
         JPanel styleGroupPanel = createRadioButtons();

         // Line up component panels
         JPanel controlPanel = new JPanel();
         controlPanel.setLayout(new GridLayout(3, 1));
         controlPanel.add(facenamePanel);
      }
}
controlPanel.add(sizeGroupPanel);
controlPanel.add(styleGroupPanel);

// Add panels to content pane

add(controlPanel, BorderLayout.SOUTH);
}

/**
 * Creates the combo box with the font style choices.
 * @return the panel containing the combo box
 */
public JPanel createComboBox()
{
    facenameCombo = new JComboBox();
    facenameCombo.addItem("Serif");
    facenameCombo.addItem("SansSerif");
    facenameCombo.addItem("Monospaced");
    facenameCombo.setEditable(true);
    facenameCombo.addActionListener(listener);

    JPanel panel = new JPanel();
    panel.add(facenameCombo);
    return panel;
}

/**
 * Creates the check boxes for selecting bold and italic styles.
 * @return the panel containing the check boxes
 */
public JPanel createCheckboxes()
{
    italicCheckBox = new JCheckBox("Italic");
    italicCheckBox.addActionListener(listener);

    boldCheckBox = new JCheckBox("Bold");
    boldCheckBox.addActionListener(listener);

    JPanel panel = new JPanel();
    panel.add(italicCheckBox);
    panel.add(boldCheckBox);
    panel.setBorder(new TitledBorder(new EtchedBorder(), "Style"));

    return panel;
}

/**
 * Creates the radio buttons to select the font size.
 * @return the panel containing the radio buttons
 */
public JPanel createRadioButtons()
{
    smallButton = new JRadioButton("Small");
    smallButton.addActionListener(listener);

    mediumButton = new JRadioButton("Medium");
    mediumButton.addActionListener(listener);

    largeButton = new JRadioButton("Large");
largeButton.addActionListener(listener);
largeButton.setSelected(true);

// Add radio buttons to button group
ButtonGroup group = new ButtonGroup();
group.add(smallButton);
group.add(mediumButton);
group.add(largeButton);

JPanel panel = new JPanel();
panel.add(smallButton);
panel.add(mediumButton);
panel.add(largeButton);
panel.setBorder(new TitledBorder(new EtchedBorder(), "Size"));

return panel;
}

/**
 * Gets user choice for font name, style, and size and sets the font of the text sample.
 */
public void setLabelFont()
{
    // Get font name
    String facename = (String) facenameCombo.getSelectedItem();
    // Get font style
    int style = 0;
    if (italicCheckBox.isSelected())
    {
        style = style + Font.ITALIC;
    }
    if (boldCheckBox.isSelected())
    {
        style = style + Font.BOLD;
    }
    // Get font size
    int size = 0;
    final int SMALL_SIZE = 24;
    final int MEDIUM_SIZE = 36;
    final int LARGE_SIZE = 48;
    if (smallButton.isSelected()) { size = SMALL_SIZE; }
    else if (mediumButton.isSelected()) { size = MEDIUM_SIZE; }
    else if (largeButton.isSelected()) { size = LARGE_SIZE; }
    // Set font of text field
    label.setFont(new Font(facename, style, size));
    label.repaint();
}
14. What is the advantage of a JComboBox over a set of radio buttons? What is the disadvantage?

15. What happens when you put two check boxes into a button group? Try it out if you are not sure.

16. How can you nest two etched borders, like this?

17. Why do all user-interface components in the FontFrame class share the same listener?

18. Why was the combo box placed inside a panel? What would have happened if it had been added directly to the control panel?

19. How could the following user interface be improved?

Practice It  Now you can try these exercises at the end of the chapter: R20.11, E20.3, E20.4.

### HOW TO 20.1 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

<table>
<thead>
<tr>
<th>Size</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Small</td>
<td></td>
</tr>
<tr>
<td>Medium</td>
<td></td>
</tr>
<tr>
<td>Large</td>
<td></td>
</tr>
</tbody>
</table>

- Pepperoni
- Anchovies

Your Price: _
**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:

```
<table>
<thead>
<tr>
<th>Size</th>
<th>Pepperoni</th>
<th>Anchovies</th>
</tr>
</thead>
<tbody>
<tr>
<td>Small</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Medium</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Large</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

**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 ordering user interface:

```java
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);
```

**Worked Example 20.1** Programming a Working Calculator

Learn how to implement arithmetic and scientific operations for a calculator. The sample program in Section 20.1 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/bjeo6examples and download Worked Example 20.1.
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.

GridLayout is 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 10.

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.
Anyone who has ever used a graphical user interface is familiar with pull-down menus (see Figure 11). 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:

```java
public class MyFrame extends JFrame {
    public MyFrame()
    {
        JMenuBar menuBar = new JMenuBar();
        setJMenuBar(menuBar);
    }
}
```

Menus are then added to the menu bar:

```java
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:

```java
JMenuItem exitItem = new JMenuItem("Exit");
fileMenu.add(exitItem);

JMenu styleMenu = new JMenu("Style");
fontMenu.add(styleMenu); // A submenu
```

A menu provides a list of available choices.
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:

```java
ActionListener listener = new ExitItemListener();
extItem.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,

```java
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:

```java
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:

```java
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 inside the createFaceItem method (see Section 10.5). Then the actionPerformed method can access the name parameter variable directly. However, we need to observe a technical rule. Because name is a local variable, it must be declared as `final` to be accessible from an inner class method.
public JMenuItem createFaceItem(final String name)
// Final variables can be accessed from an inner class method
{
    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.

section_4/FontViewer2.java

```java
import javax.swing.JFrame;

/**
   This program uses a menu to display font effects.
*/
public class FontViewer2
{
    public static void main(String[] args)
    {
        JFrame frame = new FontFrame2();
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.setTitle("FontViewer");
        frame.setVisible(true);
    }
}
```

section_4/FontFrame2.java

```java
import java.awt.BorderLayout;
import java.awt.Font;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import javax.swing.JFrame;
import javax.swing.JLabel;
import javax.swing.JMenu;
import javax.swing.JMenuBar;
import javax.swing.JMenuItem;

/**
   This frame has a menu with commands to change the font
   of a text sample.
*/
public class FontFrame2 extends JFrame
{
    private static final int FRAME_WIDTH = 300;
    private static final int FRAME_HEIGHT = 400;

    private JLabel label;
    private String facename;
```
private int fontstyle;
private int fontsize;

/**
   * Constructs the frame.
   */
public FontFrame2()
{
    // Construct text sample
    label = new JLabel("Big Java");
    add(label, BorderLayout.CENTER);

    // Construct menu
    JMenuBar menuBar = new JMenuBar();
    setJMenuBar(menuBar);
    menuBar.add(createFileMenu());
    menuBar.add(createFontMenu());

    facename = "Serif";
    fontsize = 24;
    fontstyle = Font.PLAIN;

    setLabelFont();
    setSize(FRAME_WIDTH, FRAME_HEIGHT);
}

class ExitItemListener implements ActionListener
{
    public void actionPerformed(ActionEvent event)
    {
        System.exit(0);
    }
}

/**
   * Creates the File menu.
   * @return the menu
   */
public JMenu createFileMenu()
{
    JMenu menu = new JMenu("File");
    JMenuItem exitItem = new JMenuItem("Exit");
    ActionListener listener = new ExitItemListener();
    exitItem.addActionListener(listener);
    menu.add(exitItem);
    return menu;
}

/**
   * Creates the Font submenu.
   * @return the menu
   */
public JMenu createFontMenu()
{
    JMenuItem faceMenu = new JMenuItem("Face");
    JMenuItem sizeMenu = new JMenuItem("Size");
    JMenuItem styleMenu = new JMenuItem("Style");
    ActionListener listener = new ExitItemListener();
    faceMenu.addActionListener(listener);
    sizeMenu.addActionListener(listener);
    styleMenu.addActionListener(listener);
    menu.add(faceMenu);
    menu.add(sizeMenu);
    menu.add(styleMenu);
    return menu;
}
public JMenu createFaceMenu()
{
    JMenu menu = new JMenu("Face");
    menu.add(createFaceItem("Serif"));
    menu.add(createFaceItem("SansSerif"));
    menu.add(createFaceItem("Monospaced"));
    return menu;
}

public JMenu createSizeMenu()
{
    JMenu menu = new JMenu("Size");
    menu.add(createSizeItem("Smaller", -1));
    menu.add(createSizeItem("Larger", 1));
    return menu;
}

public JMenu createStyleMenu()
{
    JMenu menu = new JMenu("Style");
    menu.add(createStyleItem("Plain", Font.PLAIN));
    menu.add(createStyleItem("Bold", Font.BOLD));
    menu.add(createStyleItem("Italic", Font.ITALIC));
    menu.add(createStyleItem("Bold Italic", Font.BOLD + Font.ITALIC));
    return menu;
}

class FaceItemListener implements ActionListener
{
    public void actionPerformed(ActionEvent event)
    {
        facename = name;
        setLabelFont();
    }
}

JMenuItem item = new JMenuItem(name);
ActionListener listener = new FaceItemListener();
item.addActionListener(listener);
return item;

/**
 * Creates a menu item to change the font size
 * and set its action listener.
 * @param name the name of the menu item
 * @param increment the amount by which to change the size
 * @return the menu item
 */
public JMenuItem createSizeItem(String name, final int increment) {
    class SizeItemListener implements ActionListener {
        public void actionPerformed(ActionEvent event) {
            fontsize = fontsize + increment;
            setLabelFont();
        }
    }

    JMenuItem item = new JMenuItem(name);
    ActionListener listener = new SizeItemListener();
    item.addActionListener(listener);
    return item;
}

/**
 * Creates a menu item to change the font style
 * and set its action listener.
 * @param name the name of the menu item
 * @param style the new font style
 * @return the menu item
 */
public JMenuItem createStyleItem(String name, final int style) {
    class StyleItemListener implements ActionListener {
        public void actionPerformed(ActionEvent event) {
            fontstyle = style;
            setLabelFont();
        }
    }

    JMenuItem item = new JMenuItem(name);
    ActionListener listener = new StyleItemListener();
    item.addActionListener(listener);
    return item;
}

/**
 * Sets the font of the text sample.
 */
public void setLabelFont() {
    Font f = new Font(facename, fontstyle, fontsize);
    label.setFont(f);
}
20. Why do JMenu objects not generate action events?

21. Can you add a menu item directly to the menu bar? Try it out. What happens?

22. Why is the increment parameter variable in the createSizeItem method declared as final?

23. Why can’t the createFaceItem method simply set the faceName instance variable, like this:
   ```java
   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;
   }
   ```

24. 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: R20.12, E20.6, E20.7.

20.5 Exploring the Swing Documentation

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 12).
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 13). Or you can look at the names of all of the classes that start with \texttt{J} and decide that \texttt{JSlider} may be a good candidate.

Next, you need to ask yourself a few questions:

- How do I construct a \texttt{JSlider}?
- How can I get notified when the user has moved it?
- How can I tell to which value the user has set it?

When you look at the documentation of the \texttt{JSlider} class, you will probably not be happy. There are over 50 methods in the \texttt{JSlider} class and over 250 inherited methods, and some of the method descriptions look downright scary, such as the one in Figure 14. Apparently some folks out there are concerned about the \texttt{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 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. There are six constructors for the \texttt{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

\begin{verbatim}
public JSlider()

    Creates a horizontal slider with the range 0 to 100 and an initial value of 50.
\end{verbatim}

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

```java
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

```java
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

```java
public void addChangeListener(ChangeListener l)
```

Click on the ChangeListener link to find out more about this interface. It has a single method

```java
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

```java
public int getValue()

Returns the slider’s value.
```

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 E20.12.

Figure 15 shows how the components are arranged in the frame.
section_5/ColorViewer.java

```java
import javax.swing.JFrame;

public class ColorViewer
{
    public static void main(String[] args)
    {
        JFrame frame = new ColorFrame();
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.setVisible(true);
    }
}
```

section_5/ColorFrame.java

```java
import java.awt.BorderLayout;
import java.awt.Color;
import java.awt.GridLayout;
import javax.swing.JFrame;
import javax.swing.JLabel;
import javax.swing.JPanel;
import javax.swing.JSlider;
import javax.swing.event.ChangeListener;
import javax.swing.event.ChangeEvent;

public class ColorFrame extends JFrame
{
    private static final int FRAME_WIDTH = 300;
    private static final int FRAME_HEIGHT = 400;

    private JPanel colorPanel;
    private JSlider redSlider;
    private JSlider greenSlider;
    private JSlider blueSlider;

    public ColorFrame()
    {
        colorPanel = new JPanel();
        add(colorPanel, BorderLayout.CENTER);
        createControlPanel();
        setSize(FRAME_WIDTH, FRAME_HEIGHT);
    }

    class ColorListener implements ChangeListener
    {
        public void stateChanged(ChangeEvent event)
        {
            setSampleColor();
        }
    }

    public void createControlPanel()
    {
        ChangeListener listener = new ColorListener();
        redSlider = new JSlider(0, 255, 255);
        redSlider.addChangeListener(listener);
    }
}
45 greenSlider = new JSlider(0, 255, 175);
46 greenSlider.addChangeListener(listener);
47
48 blueSlider = new JSlider(0, 255, 175);
49 blueSlider.addChangeListener(listener);
50
51 JPanel controlPanel = new JPanel();
52 controlPanel.setLayout(new GridLayout(3, 2));
53
54 controlPanel.add(new JLabel("Red"));
55 controlPanel.add(redSlider);
56
57 controlPanel.add(new JLabel("Green"));
58 controlPanel.add(greenSlider);
59
60 controlPanel.add(new JLabel("Blue"));
61 controlPanel.add(blueSlider);
62
63 add(controlPanel, BorderLayout.SOUTH);
64 }
65
two
66 /**
67 \s The slider values and sets the panel to
68 \s the selected color.
69 */
70 public void setSampleColor()
71 {
72     // Read slider values
73
74     int red = redSlider.getValue();
75     int green = greenSlider.getValue();
76     int blue = blueSlider.getValue();
77
78     // Set panel background to selected color
79     colorPanel.setBackground(new Color(red, green, blue));
80     colorPanel.repaint();
81 }
82 }

SELF CHECK

25. 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.

26. 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: R20.14, E20.2, E20.12.
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.

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.

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.
Can you use a flow layout for the components in a frame? If yes, how?

What is the advantage of a layout manager over telling the container “place this component at position (x, y)”?

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.

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.

What happens when you add a button to a container that uses a border layout and omit the position? Try it out and explain.

What happens when you try to add a button to another button? Try it out and explain.

The ColorFrame in Section 20.5 uses a grid layout manager. Explain a drawback of the grid that is apparent from Figure 15. What could you do to overcome this drawback?

What is the difference between the grid layout and the grid bag layout?
• R20.9 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.

• R20.10 What is the difference between radio buttons and check boxes?

• R20.11 Why do you need a button group for radio buttons but not for check boxes?

• R20.12 What is the difference between a menu bar, a menu, and a menu item?

• R20.13 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?

• R20.14 How do you construct a vertical slider? Consult the Swing documentation for an answer.

• R20.15 Why doesn’t a JComboBox send out change events?

• R20.16 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.

• R20.17 How many Swing user-interface components are there? Look at the Java documentation to get an approximate answer.

• R20.18 How many methods does the JProgressBar component have? Be sure to count inherited methods. Look at the Java documentation.

• R20.19 Is it a requirement to use inheritance for frames, as described in Section 20.1.3? (Hint: Consider Special Topic 20.1.)

• R20.20 What is the difference between a label, a text field, and a text area?

• R20.21 Name a method that is declared in JTextArea, a method that JTextArea inherits from JTextComponent, and a method that JTextArea inherits from JComponent.

• R20.22 Why did the program in Section 20.2.2 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?

**PRACTICE EXERCISES**

• E20.1 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.

• E20.2 Add icons to the buttons of Exercise E20.1. Use a JButton constructor with an Icon argument and supply an ImageIcon.

• E20.3 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.

• E20.4 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.
Chapter 20  Graphical User Interfaces

- **E20.5** 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.

- **E20.6** 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.

- **E20.7** 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.

- **E20.8** 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.

- **E20.9** Write a program that lets users design charts such as the following:

  - Golden Gate
  - Brooklyn
  - Delaware Memorial
  - Mackinac

  Use appropriate components to ask for the length, label, and color, then apply them when the user clicks an “Add Item” button.

- **E20.10** 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.

- **E20.11** Write a graphical application describing an earthquake, as in Section 5.3. Supply a text field and button for entering the strength of the earthquake. Display the earthquake description in a label.

- **E20.12** Modify the slider test program in Section 20.5 to add a set of tick marks to each slider that show the exact slider position.

---

**Programming Projects**

- **P20.1** Modify the program of Exercise E20.7 to replace the buttons with a slider to generate more or fewer random rectangles.

- **P20.2** 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.

- **P20.3** 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.
P20.4 In the application from Exercise P20.3, replace the text area with a bar chart that shows the balance after the end of each year.

Business P20.5 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. Use the following conversion rates:

1 EUR is equal to 1.42 USD.
1 GBP is equal to 1.64 USD.
1 GBP is equal to 1.13 EUR.

Business P20.6 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 user name and password, separated by a space. You should make a users.txt file for testing your program.

Science P20.7 In Exercise P20.6, 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 Business P20.6 by using this component instead of a text field. Each time the user types a letter, show a ■ character.

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. 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.
7. Two: FilledFrameViewer2, FilledFrame.
8. It’s an instance method of FilledFrame, so the frame is the implicit parameter.
9. Then the text field is not labeled, and the user will not know its purpose.
10. Integer.parseInt(textField.getText())
11. A text field holds a single line of text; a text area holds multiple lines.
12. The text area is intended to display the program output. It does not collect user input.
13. Don’t construct a JScrollPane but add the resultArea object directly to the panel.

14. 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.

15. If one of them is checked, the other one is unchecked. You should use radio buttons if that is the behavior you want.

16. You can’t nest borders, but you can nest panels with borders:
   ```java
   JPanel p1 = new JPanel();
   p1.setBorder(new EtchedBorder());
   JPanel p2 = new JPanel();
   p2.setBorder(new EtchedBorder());
   p1.add(p2);
   ```

17. When any of the component settings is changed, the program simply queries all of them and updates the label.

18. To keep it from growing too large. It would have grown to the same width and height as the two panels below it.

19. Instead of using radio buttons with two choices, use a checkbox.

20. When you open a menu, you have not yet made a selection. Only JMenuItem objects correspond to selections.

21. 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.

22. The parameter variable is accessed in a method of an inner class.

23. Then the faceName variable is set when the menu item is added to the menu, not when the user selects the menu.

24. 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.

25. JColorChooser.

26. Action events describe one-time changes, such as button clicks. Change events describe continuous changes.
Arithmetic

In the calculator program of Section 20.1, 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 16, you saw 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:

<table>
<thead>
<tr>
<th>Button Clicked</th>
<th>Action</th>
<th>Display</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Show 1 in display.</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>Add 3 to end of display.</td>
<td>13</td>
</tr>
<tr>
<td>+</td>
<td>Store 13 and + for later use.</td>
<td>13</td>
</tr>
<tr>
<td>4</td>
<td>Clear display, add 4.</td>
<td>4</td>
</tr>
<tr>
<td>*</td>
<td>Replace display with result of 13 + 4. Store 17 and * for later use.</td>
<td>17</td>
</tr>
<tr>
<td>2</td>
<td>Clear display, add 2.</td>
<td>2</td>
</tr>
<tr>
<td>=</td>
<td>Replace display with result of 17 * 2.</td>
<td>34</td>
</tr>
</tbody>
</table>

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.
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 variable.

```java
public class CalculatorFrame extends JFrame
{
    private JLabel 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 is the first digit after an operator:

```java
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:

```java
class DigitButtonListener implements ActionListener
{
    private String digit;

    public DigitButtonListener(String aDigit)
    {
        digit = aDigit;
    }
    ...
}
```

We construct digit buttons with this helper method:

```java
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("");
        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 $3 + 4 = 5 * 6$. 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:

```java
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);
    return new JPanel();
}
```
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.

```java
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:

```java
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();
        if (!radian && (mathOp.equals("sin")
        {
```
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

```java
import javax.swing.JFrame;

public class CalculatorViewer
{
    public static void main(String[] args)
    {
        JFrame frame = new CalculatorFrame();
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.setTitle("Calculator");
        frame.setVisible(true);
    }
}
```

worked_example_1/CalculatorFrame.java

```java
import java.awt.BorderLayout;
import java.awt.GridLayout;
import java.awt.event.ActionListener;
import java.awt.event.ActionEvent;
import javax.swing.ButtonGroup;
import javax.swing.JButton;
import javax.swing.JCheckBox;
```
import javax.swing.JComboBox;
import javax.swing.JFrame;
import javax.swing.JPanel;
import javax.swing.JRadioButton;
import javax.swing.JLabel;
import javax.swing.border.EtchedBorder;
import javax.swing.border.TitledBorder;

/**
 * This frame contains a panel that displays buttons for a calculator and a panel with a text fields to specify the result of calculation.
 */
public class CalculatorFrame extends JFrame {
    private JLabel display;
    private JCheckBox radianCheckBox;
    private JRadioButton baseeButton;
    private JRadioButton base10Button;
    private JRadioButton base2Button;
    private JComboBox mathOpCombo;
    private JButton mathOpButton;

    private double lastValue;
    private String lastOperator;
    private boolean startNewValue;

    private static final int FRAME_WIDTH = 400;
    private static final int FRAME_HEIGHT = 300;

    public CalculatorFrame() {
        createButtonPanel();
        createControlPanel();

        display = new JLabel("0");
        add(display, BorderLayout.NORTH);

        lastValue = 0;
        lastOperator = ";
        startNewValue = true;

        setSize(FRAME_WIDTH, FRAME_HEIGHT);
    }

    /**
     * Creates the control panel with the text field and buttons on the frame.
     */
    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"));
        buttonPanel.add(makeOperatorButton("/"));
        buttonPanel.add(makeDigitButton("4"));
        buttonPanel.add(makeDigitButton("5"));
buttonPanel.add(makeDigitButton("6"));
buttonPanel.add(makeOperatorButton("*"));
buttonPanel.add(makeDigitButton("1"));
buttonPanel.add(makeDigitButton("2"));
buttonPanel.add(makeDigitButton("3"));
buttonPanel.add(makeOperatorButton("-")));
buttonPanel.add(makeDigitButton("0"));
buttonPanel.add(makeOperatorButton("."));
buttonPanel.add(makeOperatorButton("="));
buttonPanel.add(makeOperatorButton("+"));
add(buttonPanel, BorderLayout.CENTER);
}

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();
        {
            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("");
        startNewValue = true;
    }
}

private JPanel createBaseButtons()
{
```java
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;
}

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);
}

/**
 * Combines two values with an operator.
 * @param value1 the first value
 * @param value2 the second value
 * @param op an operator (+, -, *, /, or =)
 */
public double calculate(double value1, double value2, String op)
{
    if (op.equals("+"))
    {
        return value1 + value2;
    }
    else if (op.equals("-"))
    {
        return value1 - value2;
    }
    //...
```java
else if (op.equals("*"))
{
    return value1 * value2;
}
else if (op.equals("/"))
{
    return value1 / value2;
}
else // "+"
{
    return value2;
}
}

class DigitButtonListener implements ActionListener
{
    private String digit;

    /**
     * Constructs a listener whose actionPerformed method adds a digit to the display.
     * @param aDigit the digit to add
     */
    public DigitButtonListener(String aDigit)
    {
        digit = aDigit;
    }

    public void actionPerformed(ActionEvent event)
    {
        if (startNewValue)
        {
            display.setText("";
            startNewValue = false;
        }
        display.setText(display.getText() + digit);
    }

    /**
     * Makes a button representing a digit of a calculator.
     * @param digit the digit of the calculator
     * @return the button of the calculator
     */
    public JButton makeDigitButton(String digit)
    {
        JButton button = new JButton(digit);
        ActionListener listener = new DigitButtonListener(digit);
        button.addActionListener(listener);
        return button;
    }
```

---

*Big Java, 6e, Cay Horstmann, Copyright © 2015 John Wiley and Sons, Inc. All rights reserved.*
class OperatorButtonListener implements ActionListener
{
    private String operator;

    /**
     * Constructs a listener whose actionPerformed method
     * schedules an operator for execution.
     */
    public OperatorButtonListener(String anOperator)
    {
        operator = anOperator;
    }

    public void actionPerformed(ActionEvent event)
    {
        if (!startNewValue)
        {
            double value = Double.parseDouble(display.getText());
            lastValue = calculate(lastValue, value, lastOperator);
            display.setText("");
            startNewValue = true;
        }
        lastOperator = operator;
    }
}

/**
 * Makes a button representing an operator of a calculator.
 * @param op the operator of the calculator
 * @return the button of the calculator
 */
public JButton makeOperatorButton(String op)
{
    JButton button = new JButton(op);
    ActionListener listener = new OperatorButtonListener(op);
    button.addActionListener(listener);
    return button;
}
CHAPTER 21
ADVANCED INPUT/OUTPUT

CHAPTER GOALS

To become familiar with text and binary file formats
To learn about encryption
To understand when to use sequential and random file access
To read and write objects using serialization

CHAPTER CONTENTS

21.1 READERS, WRITERS, AND INPUT/OUTPUT STREAMS W924
21.2 BINARY INPUT AND OUTPUT W925
   CE1 Negative byte Values W929
21.3 RANDOM ACCESS W929
21.4 OBJECT INPUT AND OUTPUT STREAMS W934
   HT1 Choosing a File Format W938
21.5 FILE AND DIRECTORY OPERATIONS W939
In this chapter you will learn more about how to write Java programs that interact with files and other sources of bytes and characters. You will learn about reading and writing text and binary data, and the differences between sequential and random access to data in a file. As an application of file processing, you will study a program that encrypts and decrypts data stored in a binary format. Next, you will see how you can use object serialization to save and load complex objects with very little effort. Finally, you will learn how to work with files and directories.

21.1 Readers, Writers, and Input/Output Streams

There are two fundamentally different ways to store data: in text format or binary format. In text format, data items are represented in human-readable form, as a sequence of characters. For example, in text form, the integer 12,345 is stored as the sequence of five characters:

'1' '2' '3' '4' '5'

In binary form, data items are represented in bytes. A byte is composed of 8 bits and can denote one of 256 values \(256 = 2^8\). For example, in binary format, the integer 12,345 is stored as a sequence of four bytes:

0 0 48 57

(because \(12,345 = 48 \cdot 256 + 57\)).

The Java library provides two sets of classes for handling input and output. Input and output streams handle binary data. Readers and writers handle data in text form. Figure 1 shows a part of the hierarchy of the Java classes for input and output.

Text input and output are more convenient for humans, because it is easier to produce input (just use a text editor) and it is easier to check that output is correct (just look at the output file in an editor). However, binary storage is more compact and more efficient.

The Reader and Writer classes were designed to process information in text form. You have already used the PrintWriter class in Chapter 11. However, for reading

![Figure 1: Java Classes for Input and Output](image_url)
text, the Scanner class is more convenient than the Reader class. Internally, the Scanner class makes use of readers to read characters.

If you store information in binary form, as a sequence of bytes, use the InputStream and OutputStream classes and their subclasses.

Why use two sets of classes? Characters are made up of bytes, but there is some variation in how each character is represented. For example, the character 'é' is encoded as a single byte with value 223 in the ISO-8859-1 encoding that has been commonly used in North America and Western Europe. However, in the UTF-8 encoding that is capable of encoding all Unicode characters, the character is represented by two bytes, 195 and 169. In the UTF-16 encoding, another encoding for Unicode, the same character is encoded as 0 223.

The Reader and Writer classes have the responsibility of converting between bytes and characters. By default, these classes use the character encoding of the computer executing the program. You can specify a different encoding in the constructor of the Scanner or PrintWriter, like this:

```java
Scanner in = new Scanner(input, "UTF-8");
// input can be a File or InputStream
PrintWriter out = new PrintWriter(output, "UTF-8");
// output can be a File or OutputStream
```

Unfortunately, there is no way of automatically determining the character encoding that is used in a particular text. You need to know which character encoding was used when the text was written. If you only exchange data with users from the same country, then you can use the default encoding of your computer. Otherwise, it is a good idea to use the UTF-8 encoding.

You learned in Chapter 11 how to process text files. In the remainder of this chapter, we will focus on binary files.

1. Suppose you need to read an image file that contains color values for each pixel in the image. Will you use a Reader or an InputStream?

2. Special Topic 11.1 introduced the openStream method of the URL class, which returns an InputStream:

   ```java
   URL locator = new URL("http://bigjava.com/index.html");
   InputStream in = locator.openStream();
   
   Why doesn’t the URL class provide a Reader instead?
   ``
The InputStream read method returns an integer, either -1 to indicate end of input, or a byte between 0 and 255.

The OutputStream write method writes a single byte.

InputStream.read method returns an int, not a byte, so that it can signal either that a byte has been read or that the end of input has been reached. It returns the byte read as an integer between 0 and 255 or, when it is at the end of the input, it returns -1.

You should test the return value. Only process the input when it is not -1:

```java
InputStream in = ...;
int next = in.read();
if (next != -1)
{
    Process next. // A value between 0 and 255
}
```

The OutputStream class has a write method to write a single byte. The parameter variable of the write method has type int, but only the lowest eight bits of the argument are written to the output stream:

```java
OutputStream out = ...;
int value = ...; // Should be between 0 and 255
out.write(value);
```

Use the try-with-resources statement to ensure that all opened files are closed:

```java
try (InputStream in = ..., OutputStream out = ...) {
    Read from in.
    Write to out.
}
```

These basic methods are the only input and output methods that the input and output stream classes provide. The Java input/output package is built on the principle that each class should have a very focused responsibility. The job of an input stream is to get bytes, not to analyze them. If you want to read numbers, strings, or other objects, you have to combine the class with other classes whose responsibility it is to group individual bytes or characters together into numbers, strings, and objects. You will see an example of those classes in Section 21.4.

As an application of a task that involves reading and writing individual bytes, we will implement an encryption program. The program scrambles the bytes in a file so that the file is unreadable except to those who know the decryption method and the secret keyword. We will use the Caesar cipher that you saw in Section 11.3, but now we will encode all bytes. The person performing any encryption chooses an encryption key; here the key is a number between 1 and 255 that indicates the shift to be used in encrypting each byte. (Julius Caesar used a key of 3, replacing A with D, B with E, and so on—see Figure 2).

To decrypt, simply use the negative of the encryption key. For example, to decrypt a message encoded with a key of 3, use a key of -3.

In this program we read each value separately, encrypt it, and write the encrypted value:

```java
int next = in.read();
if (next == -1)
{
    done = true;
}
else
{
    int encrypted = encrypt(next);
    out.write(encrypted);
}
```
In a more complex encryption program, you would read a block of bytes, encrypt the block, and write it out.

Try out the program on a file of your choice. You will find that the encrypted file is unreadable. In fact, because the newline characters are transformed, you may not be able to read the encrypted file in a text editor. To decrypt, simply run the program again and supply the negative of the encryption key.

**section_2/CaesarCipher.java**

```java
import java.io.InputStream;
import java.io.OutputStream;
import java.io.IOException;

/**
 * This class encrypts files using the Caesar cipher.
 * For decryption, use an encryptor whose key is the negative of the encryption key.
 */
public class CaesarCipher {
    private int key;

    /**
     * Constructs a cipher object with a given key.
     * @param aKey the encryption key
     */
    public CaesarCipher(int aKey) {
        key = aKey;
    }

    /**
     * Encrypts the contents of an input stream.
     * @param in the input stream
     * @param out the output stream
     */
    public void encryptStream(InputStream in, OutputStream out) throws IOException {
        boolean done = false;
        while (!done) {
            int next = in.read();
            if (next == -1) {
                done = true;
            } else {
                int encrypted = encrypt(next);
                out.write(encrypted);
            }
        }
    }
}
```
/**
 * Encrypts a value.
 * @param b the value to encrypt (between 0 and 255)
 * @return the encrypted value
 */
public int encrypt(int b)
{
    return (b + key) % 256;
}

section_2/CaesarEncryptor.java

import java.io.File;
import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.InputStream;
import java.io.IOException;
import java.io.OutputStream;
import java.util.Scanner;

/**
 * This program encrypts a file, using the Caesar cipher.
 */
public class CaesarEncryptor
{
    public static void main(String[] args)
    {
        Scanner in = new Scanner(System.in);
        System.out.print("Input file: ");
        String inFile = in.next();
        System.out.print("Output file: ");
        String outFile = in.next();
        System.out.print("Encryption key: ");
        int key = in.nextInt();

        try (InputStream inStream = new FileInputStream(inFile);
             OutputStream outStream = new FileOutputStream(outFile))
        {
            CaesarCipher cipher = new CaesarCipher(key);
            cipher.encryptStream(inStream, outStream);
        }
        catch (IOException exception)
        {
            System.out.println("Error processing file: " + exception);    
        }
    }
}

3. Why does the read method of the InputStream class return an int and not a byte?
4. Decrypt the following message: Khoor/#Zruog$.
5. Can you use the sample program from this section to encrypt a binary file, for example, an image file?

Practice It  Now you can try these exercises at the end of the chapter: R21.6, E21.1, P21.1.
**21.3 Random Access**

Reading a file sequentially from beginning to end can be inefficient. In this section, you will learn how to directly access arbitrary locations in a file. Consider a file that contains a set of bank accounts. We want to change the balances of some of the accounts. We could read all account data into an array list, update the information that has changed, and save the data out again. But if the data set in the file is very large, we may end up doing a lot of reading and writing just to update a handful of records. It would be better if we could locate the changed information in the file and simply replace it.

This is quite different from the file access you programmed in Chapter 11, where you read from a file, starting at the beginning and reading the entire contents until you reached the end. That access pattern is called **sequential access**. Now we would like to access specific locations in a file and change only those locations. This access pattern is called **random access** (see Figure 3). There is nothing “random” about random access—the term simply means that you can read and modify any byte stored at any location in the file.

---

**Negative byte Values**

The `read` method of the `InputStream` class returns –1 or the byte that was read, a value between 0 and 255. It is tempting to place this value into a variable of type `byte`, but that turns out not to be a good idea.

```java
int next = in.read();
if (next != -1)
{
    byte input = (byte) next; // Not recommended
    ...
}
```

In Java, the `byte` type is a `signed` type. There are 256 values of the `byte` type, from –128 to 127. When converting an `int` value between 128 and 255 to a byte, the result is a negative value. This can be inconvenient. For example, consider this test:

```java
int next = in.read();
byte input = (byte) next;
if (input == 'é') . . .
```

The condition is never true, even if `next` is equal to the Unicode value for the 'é' character. That Unicode value happens to be 233, but a single byte is always a value between –128 and 127.

The remedy is to work with `int` values. Don’t use the `byte` type.

---

**Figure 3**  Sequential and Random Access
Only disk files support random access; the input and output streams that correspond to the keyboard and the terminal window do not. Each disk file has a special file pointer position. Normally, the file pointer is at the end of the file, and any output is appended to the end. However, if you move the file pointer to the middle of the file and write to the file, the output overwrites what is already there. The next read command starts reading input at the file pointer location. You can move the file pointer just beyond the last byte currently in the file but no further.

In Java, you use a RandomAccessFile object to access a file and move a file pointer. To open a random access file, you supply a file name and a string to specify the open mode. You can open a file either for reading only (“r”) or for reading and writing (“rw”). For example, the following command opens the file bank.dat for both reading and writing:

```java
RandomAccessFile f = new RandomAccessFile("bank.dat", "rw");
```

The method call

```java
f.seek(position);
```

moves the file pointer to the given position, counted from the beginning of the file. The first byte of a file has position 0. To find out the current position of the file pointer (counted from the beginning of the file), use

```java
long position = f.getFilePointer();
```

Because files can be very large, the file pointer values are long integers. To determine the number of bytes in a file, use the length method:

```java
long fileLength = f.length();
```

In the example program at the end of this section, we use a random access file to store a set of bank accounts, each of which has an account number and a current balance. The test program lets you pick an account and deposit money into it.

If you want to manipulate a data set in a file, you have to pay special attention to the formatting of the data. Suppose you just store the data as text. Say account 1001 has a balance of $900, and account 1015 has a balance of 0.

We want to deposit $100 into account 1001. Suppose we move the file pointer to the first character of the old value:

```
1 0 0 1 | 9 0 0 | 1 0 1 5 | 0
```

If we now simply write out the new value, the result is

```
1 0 0 1 | 1 0 0 0 | 1 0 1 5 | 0
```

That is not working too well. The update is overwriting the space that separates the values.

In order to be able to update values in a file, you must give each value a fixed size that is sufficiently large. As a result, every record in the file has the same size. This has another advantage: It is then easy to skip quickly to, say, the 50th record, without having to read the first 49 records in. Just set the file pointer to 49 × the record size.
When storing numbers in a file with fixed record sizes, it is easier to access them in binary form, rather than text form. For that reason, the RandomAccessFile class stores binary data. The readInt and writeInt methods read and write integers as four-byte quantities. The readDouble and writeDouble methods process double-precision floating-point numbers as eight-byte quantities.

```java
double x = f.readDouble();
f.writeDouble(x);
```

If we save the account number as an integer and the balance as a double value, then each bank account record consists of 12 bytes: 4 bytes for the integer and 8 bytes for the double-precision floating-point value.

Now that we have determined the file layout, we can implement our random access file methods. In the program at the end of this section, we use a BankData class to translate between the random access file format and bank account objects. The size method determines the total number of accounts by dividing the file length by the size of a record.

```java
public int size() throws IOException
{
    return (int) (file.length() / RECORD_SIZE);
}
```

To read the nth account in the file, the read method positions the file pointer to the offset \( n \times \text{RECORD\_SIZE} \), then reads the data, and constructs a bank account object:

```java
public BankAccount read(int n) throws IOException
{
    file.seek(n * RECORD_SIZE);
    int accountNumber = file.readInt();
    double balance = file.readDouble();
    return new BankAccount(accountNumber, balance);
}
```

Writing an account works the same way:

```java
public void write(int n, BankAccount account) throws IOException
{
    file.seek(n * RECORD_SIZE);
    file.writeInt(account.getAccountNumber());
    file.writeDouble(account.getBalance());
}
```

The test program asks the user to enter an account number and an amount to deposit. If the account does not currently exist, it is created. The money is deposited, and then the user can choose to continue or quit. The bank data are saved and reloaded when the program is run again.

**section_3/BankSimulator.java**

```java
import java.io.IOException;
import java.util.Scanner;

/**
 * This program demonstrates random access. You can access existing accounts and deposit money, or create new accounts.
 * The accounts are saved in a random access file.
 */
public class BankSimulator
{

```
public static void main(String[] args) throws IOException
{
    try (Scanner in = new Scanner(System.in);
         BankData data = new BankData())
    {
        data.open("bank.dat");

        boolean done = false;
        while (!done)
        {
            System.out.print("Account number: ");
            int accountNumber = in.nextInt();
            System.out.print("Amount to deposit: ");
            double amount = in.nextDouble();

            int position = data.find(accountNumber);
            BankAccount account;
            if (position >= 0)
            {
                account = data.read(position);
                account.deposit(amount);
                System.out.println("New balance: " + account.getBalance());
            }
            else // Add account
            {
                account = new BankAccount(accountNumber, amount);
                position = data.size();
                System.out.println("Adding new account.");
                data.write(position, account);

                System.out.print("Done? (Y/N) ");
                String input = in.next();
                if (input.equalsIgnoreCase("Y")) { done = true; }
            }
        }
    }
}

section_3/BankData.java

import java.io.IOException;
import java.io.RandomAccessFile;

/**
 * This class is a conduit to a random access file containing bank account records.
 * /
public class BankData implements AutoCloseable
{
    private RandomAccessFile file;

    public static final int INT_SIZE = 4;
    public static final int DOUBLE_SIZE = 8;
    public static final int RECORD_SIZE = INT_SIZE + DOUBLE_SIZE;

    /**
     * Constructs a BankData object that is not associated with a file.
     * /

public BankData()
{
    file = null;
}

/**
   * Opens the data file.
   * @param filename the name of the file containing bank account information
   */
   public void open(String filename)
   throws IOException
   {
     if (file != null) { file.close(); }
     file = new RandomAccessFile(filename, "rw");
   }

/**
   * Gets the number of accounts in the file.
   * @return the number of accounts
   */
   public int size()
   throws IOException
   {
     return (int) (file.length() / RECORD_SIZE);
   }

/**
   * Closes the data file.
   */
   public void close()
   throws IOException
   {
     if (file != null) { file.close(); }
     file = null;
   }

/**
   * Reads a bank account record.
   * @param n the index of the account in the data file
   * @return a bank account object initialized with the file data
   */
   public BankAccount read(int n)
   throws IOException
   {
     file.seek(n * RECORD_SIZE);
     int accountNumber = file.readInt();
     double balance = file.readDouble();
     return new BankAccount(accountNumber, balance);
   }

/**
   * Finds the position of a bank account with a given number.
   * @param accountNumber the number to find
   * @return the position of the account with the given number, or -1 if there is no such account
   */
   public int find(int accountNumber)
   throws IOException
   {
for (int i = 0; i < size(); i++)
{
    file.seek(i * RECORD_SIZE);
    int a = file.readInt();
    if (a == accountNumber) { return i; }
        // Found a match
}
return -1; // No match in the entire file

/**
  * Writes a bank account record to the data file.
  * @param n the index of the account in the data file
  * @param account the account to write
  */
  public void write(int n, BankAccount account)
    throws IOException
  {
    file.seek(n * RECORD_SIZE);
    file.writeInt(account.getAccountNumber());
    file.writeDouble(account.getBalance());
  }

Program Run

Account number: 1001
Amount to deposit: 100
Adding new account.
Done? (Y/N) N
Account number: 1018
Amount to deposit: 200
Adding new account.
Done? (Y/N) N
Account number: 1001
Amount to deposit: 1000
New balance: 1100.0
Done? (Y/N) Y

6. Why doesn’t System.out support random access?
7. What is the advantage of the binary format for storing numbers? What is the disadvantage?

Practice It


21.4 Object Input and Output Streams

In the program in Section 21.3 you read BankAccount objects by reading each input value separately. Actually, there is an easier way. The ObjectOutputStream class can save entire objects out to disk. Objects are saved in binary format; hence, you use output streams and not writers. You use the ObjectInputStream class to read the saved objects.
For example, you can write a BankAccount object to a file as follows:

```java
BankAccount b = ...;
ObjectOutputStream out = new ObjectOutputStream(
    new FileOutputStream("bank.dat"));
out.writeObject(b);
```

The object output stream automatically saves all instance variables of the object to
the output stream. When reading the object back in, you use the readObject method
of the ObjectInputStream class. That method returns an object reference, so you need to
remember the types of the objects that you saved and use a cast:

```java
ObjectInputStream in = new ObjectInputStream(
    new FileInputStream("bank.dat"));
BankAccount b = (BankAccount) in.readObject();
```

The readObject method can throw a ClassNotFoundException—it is a checked exception,
so you need to catch or declare it.

You can do even better than that, though. You can store a whole bunch of objects
in an array list or array, or inside another object, and then save that object:

```java
ArrayList<BankAccount> a = new ArrayList<>();
// Now add many BankAccount objects into a
out.writeObject(a);
```

With one instruction, you can save the array list and all the objects that it references.
You can read all of them back with one instruction:

```java
ArrayList<BankAccount> a = (ArrayList<BankAccount>) in.readObject();
```

Of course, if the Bank class contains an ArrayList of bank accounts, then you can simply
save and restore a Bank object. Then its array list, and all the BankAccount objects
that it contains, are automatically saved and restored as well. The sample program at
the end of this section uses this approach.

This is a truly amazing capability that is highly recommended.

To place objects of a particular class into an object output stream, the class must
implement the Serializable interface. That interface has no methods, so there is no
effort involved in implementing it:

```java
public class BankAccount implements Serializable
{
    ...
}
```

The process of saving objects to an object output stream is called serialization
because each object is assigned a serial number. If the same object is saved twice, only
the serial number is written out the second time. When the objects are read back in,
duplicate serial numbers are restored as references to the same object.

Following is a sample program that puts serialization to work. The Bank class manages
a collection of bank accounts. Both the Bank and BankAccount classes implement
the Serializable interface. Run the program several times. Whenever the program
exits, it saves the Bank object (and all bank account objects that the bank contains) into
a file bank.dat. When the program starts again, the file is loaded, and the changes from
the preceding program run are automatically reflected. However, if the file is missing
(either because the program is running for the first time, or because the file was erased),
then the program starts with a new bank.
import java.io.Serializable;
import java.util.ArrayList;

/**
 * This bank contains a collection of bank accounts.
 */
public class Bank implements Serializable {
    private ArrayList<BankAccount> accounts;

    /**
     * Constructs a bank with no bank accounts.
     */
    public Bank() {
        accounts = new ArrayList<>();
    }

    /**
     * Adds an account to this bank.
     * @param a the account to add
     */
    public void addAccount(BankAccount a) {
        accounts.add(a);
    }

    /**
     * Finds a bank account with a given number.
     * @param accountNumber the number to find
     * @return the account with the given number, or null if there
     * is no such account
     */
    public BankAccount find(int accountNumber) {
        for (BankAccount a : accounts) {
            if (a.getAccountNumber() == accountNumber) // Found a match
                return a;
        }
        return null; // No match in the entire array list
    }
}

import java.io.File;
import java.io.IOException;
import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.ObjectInputStream;
import java.io.ObjectOutputStream;

/**
 * This program demonstrates serialization of a Bank object.
 * If a file with serialized data exists, then it is loaded.
 */
Object Input and Output Streams

Otherwise the program starts with a new bank.
Bank accounts are added to the bank. Then the bank object is saved.

```
*/
public class SerialDemo {
    public static void main(String[] args)
        throws IOException, ClassNotFoundException
    {  
    Bank firstBankOfJava;
    File f = new File("bank.dat");
    if (f.exists())
    {  
        try (ObjectInputStream in = new ObjectInputStream(
            new FileInputStream(f)))
        {  
            firstBankOfJava = (Bank) in.readObject();
        }
        else
        {  
            firstBankOfJava = new Bank();
            firstBankOfJava.addAccount(new BankAccount(1001, 20000));
            firstBankOfJava.addAccount(new BankAccount(1015, 10000));
        }
    // Deposit some money
    BankAccount a = firstBankOfJava.find(1001);
    a.deposit(100);
    System.out.println(a.getAccountNumber() + "\n" + a.getBalance());
    a = firstBankOfJava.find(1015);
    System.out.println(a.getAccountNumber() + "\n" + a.getBalance());
    try (ObjectOutputStream out = new ObjectOutputStream(
        new FileOutputStream(f)))
    {  
        out.writeObject(firstBankOfJava);
    }
    }
}
```

**Program Run**

```
1001:20100.0
1015:10000.0
```

**Second Program Run**

```
1001:20200.0
1015:10000.0
```

8. Why is it easier to save an object with an ObjectOutputStream than a RandomAccessFile?

9. What do you have to do to the Country class from Section 10.1 so that its objects can be saved in an ObjectOutputStream?

**Practice It** Now you can try these exercises at the end of the chapter: R21.8, R21.9, E21.8.
Choosing a File Format

Many programs allow users to save their work in files. Program users can later load those files and continue working on the data, or they can send the files to other users. When you develop such a program, you need to decide how to store the data.

This How To shows you how to choose the appropriate mechanisms for saving and loading your program’s data.

Step 1
Select a data format.

The most important questions you need to ask yourself concern the format to use for saving your data:

- Does your program manipulate text, such as plain text files? If so, use readers and writers.
- Does your program update portions of a file? Then use random access.
- Does your program read or write individual bytes of binary data, such as image files or encrypted data? Then use input and output streams.
- Does your program save and restore objects? Then use object output and input streams.

Step 2
Use scanners and writers if you are processing text.

Use a scanner to read the input.

```java
Scanner in = new Scanner(new File("input.txt"));
```

Then use the familiar methods `next`, `nextInt`, and so on. See Chapter 11 for details.

To write output, turn the file output stream into a PrintWriter:

```java
PrintWriter out = new PrintWriter("output.txt");
```

Then use the familiar `print` and `println` methods:

```java
out.println(text);
```

Step 3
Use the `RandomAccessFile` class if you need random access.

The `RandomAccessFile` class has methods for moving a file pointer to an arbitrary position:

```java
file.seek(position);
```

You can then read or write individual bytes, characters, binary integers, and binary floating-point numbers.

Step 4
Use input and output streams if you are processing bytes.

Use this loop to process input one byte at a time:

```java
InputStream in = new FileInputStream("input.bin");
boolean done = false;
while (!done)
{
    int next = in.read();
    if (next == -1)
    {
        done = true;
    }
    else
    {
        Process next. // next is between 0 and 255
    }
}
Similarly, write the output one byte at a time:

```java
try (OutputStream out = new FileOutputStream("output.bin")) {
    . . .
    while (. . .) {
        int b = . . .; // b is between 0 and 255
        out.write(b);
    }
}
```

Use input and output streams only if you are ready to process the input one byte at a time. This makes sense for encryption/decryption or processing the pixels in an image.

### Step 5

Use object input and output streams if you are processing objects.

First go through your classes and tag them with `implements Serializable`. You don’t need to add any additional methods.

Also go to the online API documentation to check that the library classes that you are using implement the `Serializable` interface. Fortunately, many of them do. In particular, `String` and `ArrayList` are serializable.

Next, put all the objects you want to save into a class (or an array or array list— but why not make another class containing that?).

Saving all program data is a trivial operation:

```java
ProgramData data = . . .;
try (ObjectOutputStream out = new ObjectOutputStream(
    new FileOutputStream("program.dat"))) {
    out.writeObject(data);
}
```

Similarly, to restore the program data, you use an `ObjectInputStream` and call

```java
ProgramData data = (ProgramData) in.readObject();
```

The `readObject` method can throw a `ClassNotFoundException`. You must catch or declare that exception.

## 21.5 File and Directory Operations

You have now seen how to read and write data in files. In the following sections, you will learn how to manipulate files and directories.

### 21.5.1 Paths

To work with files and directories, you need to specify them. Up to now, you have specified file names using strings, or, when constructing a `Scanner`, you used a `File` object. The `Path` interface provides a more sophisticated way of working with names for files and directories.

You obtain a `Path` object with the static `Paths.get` method:

```java
Path inputPath = Paths.get("input.txt");
Path dirPath = Paths.get("/home/myname");
```
You combine paths with the `resolve` method:

```java
Path fullPath = dirPath.resolve(inputPath); // The path /home/myname/input.txt
```

The argument to `resolve` can also be a string: `dirPath.resolve("output.txt")`.

A path can be `absolute` (starting at the root of the file system) or `relative` (only meaningful when resolved against some other path). For example, `/home/myname` is an absolute path, and `input.txt` is relative. When you have a relative path, you can turn it into an absolute path by calling the `toAbsolutePath` method. For example,

```java
Path absolutePath = Paths.get("input.txt").toAbsolutePath();
```
yields a path that resolves `input.txt` against the directory from which the program was started; something similar to `/home/myname/cs2/project10/input.txt`.

There are many methods for taking paths apart. Some of the most useful are:

```java
Path parent = fullPath.getParent(); // The path /home/myname
Path fileName = fullPath.getFileName(); // The path input.txt
```

To get all components of a `Path`, you can use an enhanced `for` loop because the `Path` interface extends the `Iterable<Path>` interface:

```java
for (Path p : fullPath)
{
    Analyze p;
}
```

In our example, `p` is set to `home`, `myname`, and `input.txt`.

Path objects are not strings. If you need to convert them to strings (for example, to change the file extension), use the `toString` method.

### 21.5.2 Creating and Deleting Files and Directories

The `Files` class has a large number of useful static methods for working with files and directories. You create an empty file or directory with

```java
Files.createFile(path);
Files.createDirectory(path);
```

Here, `path` is a `Path` object. You saw in the preceding section how to obtain such an object.

If you try to create a file or directory that already exists, an exception is thrown. Also, the parent of the path must already exist. You can test whether a path exists by calling

```java
boolean pathExists = Files.exists(path);
```

To find out whether an existing path is a file or a directory, call `Files.isRegularFile` or `Files.isDirectory`.

If you want to delete a file or an empty directory, call

```java
Files.delete(path);
```

Sometimes, you want to create a temporary file or directory. You don’t care what it is called, but you want a unique fresh name. Call

```java
Path tempFile = Files.createTempFile(prefix, extension);
Path tempDir = Files.createTempDirectory(prefix);
```

Here, `prefix` is a string that helps you find the file or directory. It is located in the temporary directory of your operating system (for example, `/tmp` in Linux). You don’t
need to delete the file or directory when you are done. The operating system automatically cleans the temporary directory.

21.5.3 Useful File Operations

The `Files` class has several useful operations for common tasks. The `Files.size` method yields the size of a file in bytes:

```java
long size = Files.size(path);
```

You can read an entire file into a list of lines (if it is a text file) or a byte array (if it is a binary file):

```java
List<String> lines = Files.readAllLines(path);
byte[] bytes = Files.readAllBytes(path);
```

To write a collection of lines or an array of bytes to a file, call

```java
Files.write(path, lines);
Files.write(path, bytes);
```

If you want to read a file into a single string, call

```java
String contents = new String(Files.readAllBytes(path), "UTF-8");
```

Conversely, here is how you can save a string to a file:

```java
Files.write(path, contents.getBytes("UTF-8"));
```

You can also obtain the lines of a text file as a stream (see Chapter 19). Then the lines are read lazily, as needed by the stream operations. You need to be careful to use a `try-with-resources` block so that the underlying file is closed after stream processing is complete. For example, this code snippet gets the first ten lines containing a given string and then automatically closes the file without reading more lines:

```java
String target = " and ";
final int MAX_LINES = 10;
List<String> result = null;
try (Stream<String> lines = Files.lines(path))
{
  result = lines
    .filter(s -> s.contains(target))
    .limit(MAX_LINES)
    .collect(Collectors.toList());
}
```

To copy or move a file, call

```java
Files.copy(fromPath, toPath);
Files.move(fromPath, toPath);
```

If `fromPath` doesn’t exist or `toPath` exists, the methods throw an exception.

You can use the `Files.move` method to move an empty directory. But to move a non-empty directory, you need to move all descendants—see Exercise P21.11.

21.5.4 Visiting Directories

In order to read all files in a directory, call the `Files.list` method. It returns a `Stream<Path>` with the files and directories of the given directory.
If you just want to have a list of files, call
try (Stream<Path> entries = Files.list(dirPath))
{
    List<Path> paths = entries.collect(Collectors.toList());
    Process the list paths.
}

If you are familiar with streams, you can instead use stream operations such as filter, map, and collect.
try (Stream<Path> entries = Files.list(dirPath))
{
    Process the stream entries.
}

The API uses streams because some applications have directories with huge numbers of files. A stream visits them lazily, but a list must be big enough to hold them all. The Files.list method does not visit subdirectories. In order to get all descendant files and directories, call Files.walk instead. It also returns a Stream<Path>, containing the descendants in depth-first order.

**SELF CHECK**

10. Construct a Path object downloads with the path to the directory on your computer that contains downloaded files.

11. Call a method of the Files class to create a subdirectory bigjava in your downloads directory.

12. Given a Path object p for /home/cay/output.txt, how do you get a path to /home/cay/output.bak?

13. How can you make a backup copy of a file with path p before writing to it?

14. What happens in Self Check 13 if the backup file already exists? How can you overcome that problem?

**Practice It**

Now you can try these exercises at the end of the chapter: R21.15, E21.9, E21.10.

**CHAPTER SUMMARY**

**Describe the Java class hierarchy for handling input and output.**

- Input and output streams access sequences of bytes. Readers and writers access sequences of characters.

**Write programs that carry out input and output of binary data.**

- Use FileInputStream and FileOutputStream classes to read and write binary data from and to disk files.
- The InputStream.read method returns an integer, either -1 to indicate end of input, or a byte between 0 and 255.
- The OutputStream.write method writes a single byte.
Describe random access and use the `RandomAccessFile` class.

- In sequential file access, a file is processed one byte at a time.
- Random access allows access at arbitrary locations in the file, without first reading the bytes preceding the access location.
- A file pointer is a position in a random access file. Because files can be very large, the file pointer is of type `long`.
- The `RandomAccessFile` class reads and writes numbers in binary form.

Use object streams to automatically read and write entire objects.

- Use object output streams to save all instance variables of an object automatically, and use object input streams to load the saved objects.
- Objects saved to an object output stream must belong to classes that implement the `Serializable` interface.

Use paths to manipulate files and directories.

- A `Path` describes the location of a file or directory.
- The `Files` class has many static methods for working with files and directories.
- The `Files.list` and `Files.walk` methods yield the children and descendants of a directory.

**STANDARD LIBRARY ITEMS INTRODUCED IN THIS CHAPTER**

```
java.io.FileInputStream
java.io.FileOutputStream
java.io.InputStream
close
read
java.io.ObjectInputStream
readObject
java.
io.ObjectOutputSteam
writeObject
java.io.OutputStream
close
write
java.io.RandomAccessFile
getFilePointer
length
readChar
readDouble
readInt
seek
writeChar
writeChars
writeDouble
writeInt
java.io.Serializable
java.nio.file.Files
copy
createDirectory
createFile
delete
exists
isDirectory
isRegularFile
lines
list
move
java.nio.file.Paths
get
java.nio.file.Path
getFileLine
getFileName
getParent
resolve
```

**REVIEW EXERCISES**

- **R21.1** What is the difference between an input stream and a reader?
- **R21.2** Write a few lines of text to a new `PrintWriter("output1.txt", "UTF-8")` and the same text to a new `PrintWriter("output2.txt", "UTF-16")`. How do the output files differ?
- **R21.3** How can you open a file for both reading and writing in Java?
- **R21.4** What happens if you try to write to a file reader?
- **R21.5** What happens if you try to write to a random access file that you opened only for reading? Try it out if you don’t know.
- **R21.6** How can you break the Caesar cipher? That is, how can you read a document that was encrypted with the Caesar cipher, even though you don’t know the key?
Chapter 21  Advanced Input/Output

R21.7 What happens if you try to save an object that is not serializable in an object output stream? Try it out and report your results.

R21.8 Of the classes in the java.lang and java.io packages that you have encountered in this book, which implement the Serializable interface?

R21.9 Why is it better to save an entire ArrayList to an object output stream instead of programming a loop that writes each element?

R21.10 What is the difference between sequential access and random access?

R21.11 What is the file pointer in a file? How do you move it? How do you tell the current position? Why is it a long integer?

R21.12 How do you move the file pointer to the first byte of a file? To the last byte? To the exact middle of the file?

R21.13 What happens if you try to move the file pointer past the end of a file? Try it out and report your result.

R21.14 Can you move the file pointer of System.in?

R21.15 Paths can be absolute or relative. An absolute path name begins with a root element (/ on Unix-like systems, a drive letter on Windows). Look at the Path API to see how one can create and recognize absolute and relative paths. What does the resolve method do when its argument is an absolute path?

R21.16 Look up the relativize method in the Path API and explain in which sense it is the opposite of resolve. Give two examples when it is useful; one for files and one for directories.

R21.17 What exactly does it mean that Files.walk yields the results in depth-first order? Are directory children listed before or after parents? Are files listed before directories? Are either listed alphabetically? Run some experiments to find out.

PRACTICE EXERCISES

E21.1 Write a program that opens a binary file and prints all ASCII characters from that file, that is, all bytes with values between 32 and 126. Print a new line after every 64 characters. What happens when you use your program with word processor documents? With Java class files?

E21.2 Write a method public static void copy(String infile, String outfile) that copies all bytes from one file to another, without using Files.copy.

E21.3 Write a method that reverses all lines in a file. Read all lines, reverse each line, and write the result.

E21.4 Repeat Exercise E21.3 by using a random access file, reversing each line in place.

E21.5 Repeat Exercise E21.3, reading one line at a time and writing the reversed lines to a temporary file. Then erase the original and move the temporary file into its place.

E21.6 Modify the BankSimulator program in Section 21.3 so that it is possible to delete an account. To delete a record from the data file, fill the record with zeroes.

E21.7 The data file in Exercise E21.6 may end up with many deleted records that take up space. Write a program that compacts such a file, moving all active records to the
beginning and shortening the file length. **Hint:** Use the `setLength` method of the `RandomAccessFile` class to truncate the file length. Look up the method’s behavior in the API documentation.

**E21.8** Enhance the `SerialDemo` program from Section 21.4 to demonstrate that it can save and restore a bank that contains a mixture of savings and checking accounts.

**E21.9** Write a method that, given a path to a file that doesn’t yet exist, creates all intermediate directories and the file.

**E21.10** Write a method `public static void swap(Path p, Path q)` that swaps two files. **Hint:** Use a temporary file.

---

**P21.1** *Random monoalphabet cipher.* The Caesar cipher, which shifts all letters by a fixed amount, is far too easy to crack. Here is a better idea. For the key, don’t use numbers but words. Suppose the keyword is `FEATHER`. Then first remove duplicate letters, yielding `FEATHR`, and append the other letters of the alphabet in reverse order. Now encrypt the letters as follows:

```
A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
F E A T H R Z Y X W V U S Q P O N M L K J I G D C B
```

Write a program that encrypts or decrypts a file using this cipher. The keyword is specified with the `-k` command line option. The `-d` command line option specifies decryption. For example,

```
java Encryptor -d -k FEATHER encrypt.txt output.txt
```
decrypts a file using the keyword `FEATHER`. It is an error not to supply a keyword.

**P21.2** *Letter frequencies.* If you encrypt a file using the cipher of Exercise P21.1, it will have all of its letters jumbled up, and will look as if there is no hope of decrypting it without knowing the keyword. Guessing the keyword seems hopeless, too. There are just too many possible keywords. However, someone who is trained in decryption will be able to break this cipher in no time at all. The average letter frequencies of English letters are well known. The most common letter is `E`, which occurs about 13 percent of the time. Here are the average frequencies of English letters:

<table>
<thead>
<tr>
<th>Letter</th>
<th>Frequency</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>8%</td>
</tr>
<tr>
<td>B</td>
<td>&lt;1%</td>
</tr>
<tr>
<td>C</td>
<td>3%</td>
</tr>
<tr>
<td>D</td>
<td>4%</td>
</tr>
<tr>
<td>E</td>
<td>13%</td>
</tr>
<tr>
<td>F</td>
<td>3%</td>
</tr>
<tr>
<td>G</td>
<td>2%</td>
</tr>
<tr>
<td>H</td>
<td>4%</td>
</tr>
<tr>
<td>I</td>
<td>7%</td>
</tr>
<tr>
<td>J</td>
<td>&lt;1%</td>
</tr>
<tr>
<td>K</td>
<td>&lt;1%</td>
</tr>
<tr>
<td>L</td>
<td>4%</td>
</tr>
<tr>
<td>M</td>
<td>3%</td>
</tr>
<tr>
<td>N</td>
<td>8%</td>
</tr>
<tr>
<td>O</td>
<td>7%</td>
</tr>
<tr>
<td>P</td>
<td>3%</td>
</tr>
<tr>
<td>Q</td>
<td>&lt;1%</td>
</tr>
<tr>
<td>R</td>
<td>8%</td>
</tr>
<tr>
<td>S</td>
<td>6%</td>
</tr>
<tr>
<td>T</td>
<td>9%</td>
</tr>
<tr>
<td>U</td>
<td>3%</td>
</tr>
<tr>
<td>V</td>
<td>1%</td>
</tr>
<tr>
<td>W</td>
<td>2%</td>
</tr>
<tr>
<td>X</td>
<td>&lt;1%</td>
</tr>
<tr>
<td>Y</td>
<td>2%</td>
</tr>
<tr>
<td>Z</td>
<td>&lt;1%</td>
</tr>
</tbody>
</table>

Write a program that reads an input file and prints the letter frequencies in that file. Such a tool will help a code breaker. If the most frequent letters in an encrypted file are `H` and `K`, then there is an excellent chance that they are the encryptions of `E` and `T`. 
• P21.3 Vigenère cipher. The trouble with a monoalphabetic cipher is that it can be easily broken by frequency analysis. The so-called Vigenère cipher overcomes this problem by encoding a letter into one of several cipher letters, depending on its position in the input document. Choose a keyword, for example TIGER. Then encode the first letter of the input text like this:

```
A B C D E F G H I J K L M N O
T U V W X Y Z A B C D E F G H
P I Q J R S T U V W X Y Z S
```

That is, the encoded alphabet is just the regular alphabet shifted to start at T, the first letter of the keyword TIGER. The second letter is encrypted according to this map:

```
A B C D E F G H I J K L M N O
I J K L M N O P Q R S T U V W X Y Z
T U V W X Y Z A B C D E F G H
```

The third, fourth, and fifth letters in the input text are encrypted using the alphabet sequences beginning with characters G, E, and R. Because the key is only five letters long, the sixth letter of the input text is encrypted in the same way as the first.

Write a program that encrypts or decrypts an input text using this cipher. Use command line arguments as in Exercise P21.1.

• P21.4 Playfair cipher. Another way of thwarting a simple letter frequency analysis of an encrypted text is to encrypt pairs of letters together. A simple scheme to do this is the Playfair cipher. You pick a keyword and remove duplicate letters from it. Then you fill the keyword, and the remaining letters of the alphabet, into a 5 \times 5 square. (Because there are only 25 squares, I and J are considered the same letter.) Here is such an arrangement with the keyword PLAYFAIR:

```
P L A Y F
I R B C D
E G H K M
N O Q S T
U V W X Z
```

To encrypt a letter pair, say AT, look at the rectangle with corners A and T:

```
P L A Y F
I R B C D
E G H K M
N O Q S T
U V W X Z
```

The encoding of this pair is formed by looking at the other two corners of the rectangle—in this case, FQ. If both letters happen to be in the same row or column, such as GO, simply swap the two letters. Decryption is done in the same way.

Write a program that encrypts or decrypts an input text using this cipher. Use command line arguments as in Exercise P21.1.

Business P21.5 Write a program that manipulates a database of product records. Records are stored in a binary file. Each record consists of these items:

- Product name: 30 characters at two bytes each = 60 bytes
- Price: one double = 8 bytes
- Quantity: one int = 8 bytes
The program should allow the user to add a record, find a record that matches a product name, and change the price and quantity of a product by a given amount.

- Graphics P21.6 Implement a graphical user interface for the BankSimulator program in Section 21.3.

- Graphics P21.7 Write a graphical application in which the user clicks on a panel to add shapes (rectangles, ellipses, cars, etc.) at the mouse click location. The shapes are stored in an array list. When the user selects File->Save from the menu, save the selection of shapes in a file. When the user selects File->Open, load in a file. Use serialization.

- P21.8 Write a toolkit that helps a cryptographer decrypt a file that was encrypted using a monoalphabet cipher. A monoalphabet cipher encrypts each character separately. Examples are the Caesar cipher and the cipher in Exercise P21.1. Analyze the letter frequencies as in Exercise P21.2. Use brute force to try all Caesar cipher keys, and check the output against a dictionary file. Allow the cryptographer to enter some substitutions and show the resulting text, with the unknown characters represented as ?. Try out your toolkit by decrypting files that you get from your classmates.

- P21.9 In the BMP format for 24-bit true-color images, an image is stored in binary format. The start of the file has the following information:
  - The position of the first pixel of the image data, starting at offset 10
  - The width of the image, starting at offset 18
  - The height of the image, starting at offset 22
Each of these is a 4-byte integer value stored in little-endian format. That is, you need to get the integer value as \( n = b_k + 256 \cdot b_{k+1} + 256^2 \cdot b_{k+2} + 256^3 \cdot b_{k+3} \), where \( k \) is the starting offset.

Each pixel of the image occupies three bytes; one each for red, green, and blue. At the end of each row are between 0 and 3 padding bytes to make the row lengths multiples of four. For example, if an image has a width of 101 pixels, there is one padding byte per row.

Using a RandomAccessFile, turn each pixel of such a BMP file into its negative.

- P21.10 Write a method `public static copyFiles(Path fromDir, Path toDir)` that copies all files (but none of the directories) from one directory to another.

- P21.11 Write a method `public static copyDirectories(Path fromDir, Path toDir)` that copies all files and directories from one directory to another.

- P21.12 Write a method `public static clearDirectory(Path dir)` that removes all files (but none of the directories) from a directory. Be careful when testing it!

- P21.13 Write a method `public static clearAllDirectories(Path dir)` that removes all files and all subdirectories from a directory. Be very careful when testing it!

- P21.14 You can use a zip file system to look into the contents of a .zip file. Call

  ```java
  FileSystem zipfs = FileSystems.newFileSystem(path, null);
  ```

  where `path` is the `Path` to the zip file. Then call `zipfs.getPath(p)` to get any `Path` inside the zip file, as if it were a path in a regular directory. You can read, copy, or move it. To inspect all files and directory trees, call `Files.walk(zipfs.getPath("/"))`.

  Write a program that opens a zip file and shows the names and the first ten lines of all files in it.
ANSWERS TO SELF-CHECK QUESTIONS

1. Image data is stored in a binary format—try loading an image file into a text editor, and you won’t see much text. Therefore, you should use an InputStream.

2. For HTML files, a reader would be useful. But URLs can also point to binary files, such as http://horstmann.com/bigjava/duke.gif.

3. It returns a special value of -1 to indicate that no more input is available. If the return type were byte, no special value would be available that could be distinguished from a legal data value.

4. It is "Hello, World!", encrypted with a key of 3.

5. Yes—the program uses input and output streams and encrypts each byte.

6. Suppose you print something, and then you call seek(0), and print again to the same location. It would be difficult to reflect that behavior in the console window.

7. Advantage: The numbers use a fixed amount of storage space, making it possible to change their values without affecting surrounding data. Disadvantage: You cannot read a binary file with a text editor.

8. You can save the entire object with a single writeObject call. With a RandomAccessFile, you have to save each instance variable separately.

9. Add implements Serializable to the class definition.

10. The details depend on your operating system. On my computer, it is
    Path downloads = Paths.get("/home/cay/Downloads");

11. Files.createDirectory(
        downloads.resolve("bigjava"));

12. Path q = Paths.get(p.toString()).replace(".txt", ".bak");
    Or, if you are worried that the directory name of p might contain the string .txt,
    Path q = p.getParent().resolve(p.getFileName().replace(".txt", ".bak");

13. First get a path for the backup file and change the suffix as in the preceding answer or, as is common in Linux, add a ~ to the filename:
    Path q = Paths.get(p.toString() + "~");
    Then copy the file:
    Files.copy(p, q);

14. Then the call to copy throws an exception. Delete the file first if it exists:
    if (Files.exists(q)) { Files.delete(q); }
    Files.copy(p, q);
CHAPTER 22
MULTITHREADING

CHAPTER GOALS
To understand how multiple threads can execute in parallel
To learn to implement threads
To understand race conditions and deadlocks
To avoid corruption of shared objects by using locks and conditions
To use threads for programming animations

CHAPTER CONTENTS

22.1 RUNNING THREADS W950
   PT1 Use the Runnable Interface W954
   ST1 Thread Pools W954

22.2 TERMINATING THREADS W955
   PT2 Check for Thread Interruptions in the run Method of a Thread W957

22.3 RACE CONDITIONS W957

22.4 SYNCHRONIZING OBJECT ACCESS W963

22.5 AVOIDING DEADLOCKS W965
   CE1 Calling await Without Calling signalAll W970
   CE2 Calling signalAll Without Locking the Object W971
   ST2 Object Locks and Synchronized Methods W971
   ST3 The Java Memory Model W972

22.6 APPLICATION: ALGORITHM ANIMATION W972
It is often useful for a program to carry out two or more tasks at the same time. For example, a web browser can load multiple images on a web page at the same time. Or an animation program can show moving figures, with separate tasks computing the positions of each separate figure. In this chapter, you will see how to implement this behavior by running tasks in multiple threads, and how you can ensure that the tasks access shared data in a controlled fashion.

### 22.1 Running Threads

A **thread** is a program unit that is executed independently of other parts of the program. The Java virtual machine executes each thread for a short amount of time and then switches to another thread. This gives the illusion of executing the threads in parallel to each other. Actually, if a computer has multiple central processing units (CPUs), then some of the threads can run in parallel, one on each processor.

Running a thread is simple in Java—follow these steps:

1. Write a class that implements the `Runnable` interface. That interface has a single method called `run`:
   ```java
   public interface Runnable
   {
     void run();
   }
   ```

2. Place the code for your task into the `run` method of your class:
   ```java
   public class MyRunnable implements Runnable
   {
     public void run()
     {
       Task statements.
     }
   }
   ```

3. Create an object of your runnable class:
   ```java
   Runnable r = new MyRunnable();
   ```

4. Construct a `Thread` object from the runnable object:
   ```java
   Thread t = new Thread(r);
   ```

5. Call the `start` method to start the thread:
   ```java
   t.start();
   ```

Let’s look at a concrete example. We want to print ten greetings of “Hello, World!”, one greeting every second. We add a time stamp to each greeting to see when it is printed.

```
Wed Apr 15 23:12:03 PST 2015 Hello, World!
Wed Apr 15 23:12:04 PST 2015 Hello, World!
Wed Apr 15 23:12:05 PST 2015 Hello, World!
Wed Apr 15 23:12:06 PST 2015 Hello, World!
Wed Apr 15 23:12:07 PST 2015 Hello, World!
Wed Apr 15 23:12:08 PST 2015 Hello, World!
```
Using the instructions for creating a thread, define a class that implements the Runnable interface:

```java
public class GreetingRunnable implements Runnable {
    private String greeting;

    public GreetingRunnable(String aGreeting) {
        greeting = aGreeting;
    }

    public void run() {
        Task statements.
        ...
    }
}
```

The run method should loop ten times through the following task actions:

- Print a time stamp.
- Print the greeting.
- Wait a second.

Get the time stamp by constructing an object of the java.util.Date class. The Date constructor without arguments produces a date that is set to the current date and time.

```java
Date now = new Date();
System.out.println(now + " " + greeting);
```

To wait a second, we use the static sleep method of the Thread class. The call

```java
Thread.sleep(milliseconds)
```

puts the current thread to sleep for a given number of milliseconds. In our case, it should sleep for 1,000 milliseconds, or one second.

There is, however, one technical problem. Putting a thread to sleep is potentially risky—a thread might sleep for so long that it is no longer useful and should be terminated. As you will see in Section 22.2, to terminate a thread, you interrupt it. When a sleeping thread is interrupted, an InterruptedException is generated. You need to catch that exception in your run method and terminate the thread.

The simplest way to handle thread interruptions is to give your run method the following form:

```java
public void run() {
    try {
        Task statements.
    } catch (InterruptedException exception) {
    }
    Clean up, if necessary.
}
```
We follow that structure in our example. Here is the complete code for our runnable class:

```java
import java.util.Date;

/**
 * A runnable that repeatedly prints a greeting.
 */
public class GreetingRunnable implements Runnable {
    private static final int REPETITIONS = 10;
    private static final int DELAY = 1000;
    private String greeting;

    /**
     * Constructs the runnable object.
     * @param aGreeting the greeting to display
     */
    public GreetingRunnable(String aGreeting) {
        greeting = aGreeting;
    }

    public void run() {
        try {
            for (int i = 1; i <= REPETITIONS; i++)
            {
                Date now = new Date();
                System.out.println(now + " " + greeting);
                Thread.sleep(DELAY);
            }
        } catch (InterruptedException exception) {
        }
    }
}
```

To start a thread, first construct an object of the runnable class.

```java
Runnable r = new GreetingRunnable("Hello, World!");
```

Then construct a thread and call the start method.

```java
Thread t = new Thread(r);
t.start();
```

Now a new thread is started, executing the code in the run method of your runnable class in parallel with any other threads in your program.

In the GreetingThreadRunner program, we start two threads: one that prints “Hello” and one that prints “Goodbye”.
section_1/GreetingThreadRunner.java

```java
/**
 * This program runs two greeting threads in parallel.
 */
public class GreetingThreadRunner {
    public static void main(String[] args) {
        GreetingRunnable r1 = new GreetingRunnable("Hello");
        GreetingRunnable r2 = new GreetingRunnable("Goodbye");
        Thread t1 = new Thread(r1);
        Thread t2 = new Thread(r2);
        t1.start();
        t2.start();
    }
}
```

Program Run

Wed Apr 15 12:04:46 PST 2015 Hello
Wed Apr 15 12:04:46 PST 2015 Goodbye
Wed Apr 15 12:04:47 PST 2015 Hello
Wed Apr 15 12:04:47 PST 2015 Goodbye
Wed Apr 15 12:04:48 PST 2015 Hello
Wed Apr 15 12:04:48 PST 2015 Goodbye
Wed Apr 15 12:04:49 PST 2015 Hello
Wed Apr 15 12:04:49 PST 2015 Goodbye
Wed Apr 15 12:04:50 PST 2015 Hello
Wed Apr 15 12:04:50 PST 2015 Goodbye
Wed Apr 15 12:04:51 PST 2015 Hello
Wed Apr 15 12:04:51 PST 2015 Goodbye
Wed Apr 15 12:04:52 PST 2015 Goodbye
Wed Apr 15 12:04:52 PST 2015 Hello
Wed Apr 15 12:04:53 PST 2015 Hello
Wed Apr 15 12:04:53 PST 2015 Goodbye
Wed Apr 15 12:04:54 PST 2015 Hello
Wed Apr 15 12:04:54 PST 2015 Goodbye
Wed Apr 15 12:04:55 PST 2015 Goodbye
Wed Apr 15 12:04:55 PST 2015 Hello

Because both threads are running in parallel, the two message sets are interleaved. However, if you look closely, you will find that the two threads aren’t exactly interleaved. Sometimes, the second thread seems to jump ahead of the first thread. This shows an important characteristic of threads. The thread scheduler gives no guarantee about the order in which threads are executed. Each thread runs for a short amount of time, called a time slice. Then the scheduler activates another thread. However, there will always be slight variations in running times, especially when calling operating system services (such as input and output). Thus, you should expect that the order in which each thread gains control is somewhat random.
1. What happens if you change the call to the sleep method in the run method to
   `Thread.sleep(1)`?

2. What would be the result of the program if the main method called
   ```java
   r1.run();
   r2.run();
   ```
   instead of starting threads?

**Practice It** Now you can try these exercises at the end of the chapter: R22.2, R22.3, E22.7.

---

**Use the Runnable Interface**

In Java, you can define the task statements of a thread in two ways. As you have seen already,
you can place the statements into the run method of a class that implements the Runnable
interface. Then you use an object of that class to construct a Thread object. You can also form
a subclass of the Thread class, and place the task statements into the run method of your subclass:

```java
public class MyThread extends Thread {
    public void run() {
        Task statements.
    }
}
```

Then you construct an object of the subclass and call the start method:

```java
Thread t = new MyThread();
t.start();
```

This approach is marginally easier than using a Runnable, and it also seems quite intuitive.
However, if a program needs a large number of threads, or if a program executes in a resource-
constrained device, such as a cell phone, it can be quite expensive to construct a separate thread
for each task. Special Topic 22.1 shows how to use a thread pool to overcome this problem. A
thread pool uses a small number of threads to execute a larger number of runnables.

The Runnable interface is designed to encapsulate the concept of a sequence of statements
that can run in parallel with other tasks, without equating it with the concept of a thread, a
potentially expensive resource that is managed by the operating system.

---

**Thread Pools**

A program that creates a huge number of short-lived threads can be inefficient. Threads are
managed by the operating system, and there is a cost for creating threads. Each thread requires
memory, and thread creation takes time. This cost can be reduced by using a thread pool. A
thread pool creates a number of threads and keeps them alive. When you add a Runnable object
to the thread pool, the next idle thread executes its run method.

For example, the following statements submit two runnables to a thread pool:

```java
Runnable r1 = new GreetingRunnable("Hello");
Runnable r2 = new GreetingRunnable("Goodbye");
ExecutorService pool = Executors.newFixedThreadPool(MAX_THREADS);
pool.execute(r1);
pool.execute(r2);
```
If many runnables are submitted for execution, then the pool may not have enough threads available. In that case, some runnables are placed in a queue until a thread is idle. As a result, the cost of creating threads is minimized. However, the runnables that are run by a particular thread are executed sequentially, not in parallel.

Thread pools are particularly important for server programs, such as database and web servers, that repeatedly execute requests from multiple clients. Rather than spawning a new thread for each request, the requests are implemented as runnable objects and submitted to a thread pool.

### 22.2 Terminating Threads

When the run method of a thread has finished executing, the thread terminates. This is the normal way of terminating a thread—implement the run method so that it returns when it determines that no more work needs to be done.

However, sometimes you need to terminate a running thread. For example, you may have several threads trying to find a solution to a problem. As soon as the first one has succeeded, you may want to terminate the other ones. In the initial release of the Java library, the Thread class had a stop method to terminate a thread. However, that method is now deprecated—computer scientists have found that stopping a thread can lead to dangerous situations when multiple threads share objects. (We will discuss access to shared objects in Section 22.3.) Instead of simply stopping a thread, you should notify the thread that it should be terminated. The thread needs to cooperate, by releasing any resources that it is currently using and doing any other required cleanup. In other words, a thread should be in charge of terminating itself.

To notify a thread that it should clean up and terminate, you use the interrupt method.

```java
  t.interrupt();
```

This method does not actually cause the thread to terminate—it merely sets a boolean variable in the thread data structure.

The run method can check whether that flag has been set, by calling the interrupted method. In that case, it should do any necessary cleanup and exit. For example, the run method of the GreetingRunnable could check for interruptions at the beginning of each loop iteration:

```java
  public void run()
  {
    for (int i = 1; i <= REPETITIONS && !Thread.interrupted(); i++)
    {
      Do work.
    }
    Clean up.
  }
```

However, if a thread is sleeping, it can’t execute code that checks for interruptions. Therefore, the sleep method is terminated with an InterruptedException whenever a sleeping thread is interrupted. The sleep method also throws an InterruptedException when it is called in a thread that is already interrupted. If your run method calls sleep in each loop iteration, simply use the InterruptedException to find out whether the thread is terminated. The easiest way to do that is to surround the entire work portion of the run method with a try block, like this:
Chapter 22  Multithreading

```java
public void run()
{
    try
    {
        for (int i = 1; i <= REPETITIONS; i++)
        {
            Do work.
            Sleep.
        }
    }
    catch (InterruptedException exception)
    {
    }
Clean up.
}
```

Strictly speaking, there is nothing in the Java language specification that says that a thread must terminate when it is interrupted. It is entirely up to the thread what it does when it is interrupted. Interrupting is a general mechanism for getting the thread’s attention, even when it is sleeping. However, in this chapter, we will always terminate a thread that is being interrupted.

### SELF CHECK

3. Suppose a web browser uses multiple threads to load the images on a web page. Why should these threads be terminated when the user hits the “Back” button?

4. Consider the following runnable.

   ```java
   public class MyRunnable implements Runnable
   {
       public void run()
       {
           try
           {
               System.out.println(1);
               Thread.sleep(1000);
               System.out.println(2);
           }
           catch (InterruptedException exception)
           {
           }
       System.out.println(3);
   }
   System.out.println(4);
   }
   ```

Suppose a thread with this runnable is started and immediately interrupted:

```java
Thread t = new Thread(new MyRunnable());
t.start();
t.interrupt();
```

What output is produced?

### Practice It

Now you can try these exercises at the end of the chapter: R22.4, R22.5, R22.6.
Check for Thread Interruptions in the run Method of a Thread

By convention, a thread should terminate itself (or at least act in some other well-defined way) when it is interrupted. You should implement your threads to follow this convention.

To do so, put the thread action inside a try block that catches the InterruptedException. That exception occurs when your thread is interrupted while it is not running, for example inside a call to sleep. When you catch the exception, do any required cleanup and exit the run method.

Some programmers don’t understand the purpose of the InterruptedException and muzzle it by placing only the call to sleep inside a try block:

```java
public void run()
{
    while (. . .)
    {
        ... 
        try 
        {
            Thread.sleep(delay);
        } 
        catch (InterruptedException exception) {} // DON’T 
        ... 
    }
}
```

Don’t do that. If you do, users of your thread class can’t get your thread’s attention by interrupting it. It is just as easy to place the entire thread action inside a single try block. Then interrupting the thread terminates the thread action.

```java
public void run()
{
    try 
    {
        while (. . .)
        {
            ... 
            Thread.sleep(delay);
            ... 
        }
    } 
    catch (InterruptedException exception) {} // OK
}
```

22.3 Race Conditions

When threads share access to a common object, they can conflict with each other. To demonstrate the problems that can arise, we will investigate a sample program in which multiple threads manipulate a bank account.

We construct a bank account that starts out with a zero balance. We create two sets of threads:

- Each thread in the first set repeatedly deposits $100.
- Each thread in the second set repeatedly withdraws $100.
Here is the run method of the DepositRunnable class:

```java
public void run()
{
    try
    {
        for (int i = 1; i <= count; i++)
        {
            account.deposit(amount);
            Thread.sleep(DELAY);
        }
    }
    catch (InterruptedException exception)
    {
    }
}
```

The WithdrawRunnable class is similar—it withdraws money instead.

The deposit and withdraw methods of the BankAccount class have been modified to print messages that show what is happening. For example, here is the code for the deposit method:

```java
public void deposit(double amount)
{
    System.out.print("Depositing " + amount);
    double newBalance = balance + amount;
    System.out.println("", new balance is " + newBalance);
    balance = newBalance;
}
```

You can find the complete source code at the end of this section.

Normally, the program output looks somewhat like this:

```
Depositing 100.0, new balance is 100.0
Withdrawing 100.0, new balance is 0.0
Depositing 100.0, new balance is 100.0
Depositing 100.0, new balance is 200.0
Withdrawing 100.0, new balance is 100.0
...  
Withdrawing 100.0, new balance is 0.0
```

In the end, the balance should be zero. However, when you run this program repeatedly, you may sometimes notice messed-up output, like this:

```
Depositing 100.0
Withdrawing 100.0, new balance is 100.0
, new balance is -100.0
```

And if you look at the last line of the output, you will notice that the final balance is not always zero. Clearly, something problematic is happening. You may have to try the program several times to see this effect.

Here is a scenario that explains how a problem can occur.

1. A deposit thread executes the lines

   ```java
   System.out.print("Depositing " + amount);
   double newBalance = balance + amount;
   ```

   in the deposit method of the BankAccount class. The value of the balance variable is still 0, and the value of the newBalance local variable is 100.

2. Immediately afterward, the deposit thread reaches the end of its time slice, and the second thread gains control.
3. A withdraw thread calls the `withdraw` method, which prints a message and withdraws $100 from the balance variable. It is now –100.

4. The withdraw thread goes to sleep.

5. The deposit thread regains control and picks up where it was interrupted. It now executes the lines
   ```java
   System.out.println("new balance is " + newBalance);
   balance = newBalance;
   ```

   The value of `balance` is now 100 (see Figure 1).

Thus, not only are the messages interleaved, but the balance is wrong. The balance after a withdrawal and deposit should again be 0, not 100. Because the `deposit` method

![Diagram of thread interactions.](image-url)
A race condition occurs if the effect of multiple threads on shared data depends on the order in which the threads are scheduled.

was interrupted, it used the old balance (before the withdrawal) to compute the value of its local newBalance variable. Later, when it was activated again, it used that newBalance value to overwrite the changed balance variable.

As you can see, each thread has its own local variables, but all threads share access to the balance instance variable. That shared access creates a problem. This problem is often called a race condition. All threads, in their race to complete their respective tasks, manipulate a shared variable, and the end result depends on which of them happens to win the race.

You might argue that the reason for this problem is that we made it too easy to interrupt the balance computation. Suppose the code for the deposit method is reorganized like this:

```java
public void deposit(double amount) {
    balance = balance + amount;
    System.out.print("Depositing " + amount + ", new balance is " + balance);
}
```

Suppose further that you make the same change in the withdraw method. If you run the resulting program, everything seems to be fine.

However, that is a dangerous illusion. The problem hasn’t gone away; it has become much less frequent, and, therefore, more difficult to observe. It is still possible for the deposit method to reach the end of its time slice after it has computed the right-hand-side value

```
    balance + amount
```

but before it performs the assignment

```
    balance = the right-hand-side value
```

When the method regains control, it finally carries out the assignment, putting the wrong value into the balance variable.

**section_3/BankAccountThreadRunner.java**

```java
/**
 * This program runs threads that deposit and withdraw money from the same bank account.
 */
public class BankAccountThreadRunner {
    public static void main(String[] args) {
        BankAccount account = new BankAccount();
        final double AMOUNT = 100;
        final int REPETITIONS = 100;
        final int THREADS = 100;

        for (int i = 1; i <= THREADS; i++) {
            DepositRunnable d = new DepositRunnable(account, AMOUNT, REPETITIONS);
            WithdrawRunnable w = new WithdrawRunnable(account, AMOUNT, REPETITIONS);
            Thread dt = new Thread(d);
            Thread wt = new Thread(w);
        }
    }
}
```
22.3 Race Conditions

section_3/DepositRunnable.java

```java
/**
 * A deposit runnable makes periodic deposits to a bank account.
 */
public class DepositRunnable implements Runnable {
    private static final int DELAY = 1;
    private BankAccount account;
    private double amount;
    private int count;

    /**
     * Constructs a deposit runnable.
     * @param anAccount the account into which to deposit money
     * @param anAmount the amount to deposit in each repetition
     * @param aCount the number of repetitions
     */
    public DepositRunnable(BankAccount anAccount, double anAmount, int aCount) {
        account = anAccount;
        amount = anAmount;
        count = aCount;
    }

    public void run() {
        try {
            for (int i = 1; i <= count; i++)
                account.deposit(amount);
        }
        catch (InterruptedException exception) {
        }
    }
}
```

section_3/WithdrawRunnable.java

```java
/**
 * A withdraw runnable makes periodic withdrawals from a bank account.
 */
public class WithdrawRunnable implements Runnable {
    private static final int DELAY = 1;
    private BankAccount account;
    private double amount;
    private int count;
```
Chapter 22  Multithreading

```java
/**
 * Constructs a withdraw runnable.
 * @param anAccount the account from which to withdraw money
 * @param anAmount the amount to withdraw in each repetition
 * @param aCount the number of repetitions
 */
public WithdrawRunnable(BankAccount anAccount, double anAmount, int aCount) {
    account = anAccount;
    amount = anAmount;
    count = aCount;
}

public void run() {
    try {
        for (int i = 1; i <= count; i++) {
            account.withdraw(amount);
            Thread.sleep(Delay);
        }
    } catch (InterruptedException exception) {} }
```
22.4 Synchronizing Object Access

To solve problems such as the one that you observed in the preceding section, use a lock object. The lock object is used to control the threads that want to manipulate a shared resource.

The Java library defines a Lock interface and several classes that implement this interface. The ReentrantLock class is the most commonly used lock class, and the only one that we cover in this book. (Locks are a feature added in Java version 5.0. Earlier versions of Java have a lower-level facility for thread synchronization—see Special Topic 22.2).

Program Run

<table>
<thead>
<tr>
<th>Depositing 100.0, new balance is 100.0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Withdrawing 100.0, new balance is 0.0</td>
</tr>
<tr>
<td>Depositing 100.0, new balance is 100.0</td>
</tr>
<tr>
<td>Withdrawing 100.0, new balance is 0.0</td>
</tr>
<tr>
<td>...</td>
</tr>
<tr>
<td>Withdrawing 100.0, new balance is 400.0</td>
</tr>
<tr>
<td>Depositing 100.0, new balance is 500.0</td>
</tr>
<tr>
<td>Withdrawing 100.0, new balance is 400.0</td>
</tr>
<tr>
<td>Withdrawing 100.0, new balance is 300.0</td>
</tr>
</tbody>
</table>

SELF CHECK

5. Give a scenario in which a race condition causes the bank balance to be –100 after one iteration of a deposit thread and a withdraw thread.

6. Suppose two threads simultaneously insert objects into a linked list. Using the implementation in Chapter 16, explain how the list can be damaged in the process.

Practice It

Now you can try these exercises at the end of the chapter: R22.8, R22.9, E22.1.
Typically, a lock object is added to a class whose methods access shared resources, like this:

```java
public class BankAccount {
    private Lock balanceChangeLock;
    ...
    public BankAccount() {
        balanceChangeLock = new ReentrantLock();
        ...
    }
}
```

All code that manipulates the shared resource is surrounded by calls to lock and unlock the lock object:

```java
balanceChangeLock.lock();
Manipulate the shared resource.
balanceChangeLock.unlock();
```

However, this sequence of statements has a potential flaw. If the code between the calls to `lock` and `unlock` throws an exception, the call to `unlock` never happens. This is a serious problem. After an exception, the current thread continues to hold the lock, and no other thread can acquire it. To overcome this problem, place the call to `unlock` into a `finally` clause:

```java
balanceChangeLock.lock();
try {
    Manipulate the shared resource.
} finally {
    balanceChangeLock.unlock();
}
```

For example, here is the code for the `deposit` method:

```java
public void deposit(double amount) {
    balanceChangeLock.lock();
    try {
        System.out.print("Depositing "+ amount);
        double newBalance = balance + amount;
        System.out.println("new balance is "+ newBalance);
        balance = newBalance;
    } finally {
        balanceChangeLock.unlock();
    }
}
```

When a thread calls the `lock` method, it *owns the lock* until it calls the `unlock` method. If a thread calls `lock` while another thread owns the lock, the first thread is temporarily deactivated. The thread scheduler periodically reactivates such a thread so that it can again try to acquire the lock. If the lock is still unavailable, the thread is again deactivated. Eventually, when the lock is available because the original thread unlocked it, the waiting thread can acquire the lock.
22.5  Avoiding Deadlocks

You can use lock objects to ensure that shared data are in a consistent state when several threads access them. However, locks can lead to another problem. It can happen that one thread acquires a lock and then waits for another thread to do some essential work. If that other thread is currently waiting to acquire the same lock, then...
neither of the two threads can proceed. Such a situation is called a deadlock or deadly
embrace. Let’s look at an example.

Suppose we want to disallow negative bank balances in our program. Here’s a
naive way of doing that. In the run method of the WithdrawRunnable class, we can check
the balance before withdrawing money:

```java
if (account.getBalance() >= amount)
{
    account.withdraw(amount);
}
```

This works if there is only a single thread running that withdraws money. But suppose
we have multiple threads that withdraw money. Then the time slice of the current
thread may expire after the check account.getBalance() >= amount passes, but before the
withdraw method is called. If, in the interim, another thread withdraws more money,
then the test was useless, and we still have a negative balance.

Clearly, the test should be moved inside the withdraw method. That ensures that
the test for sufficient funds and the actual withdrawal cannot be separated. Thus, the
withdraw method could look like this:

```java
public void withdraw(double amount)
{
    balanceChangeLock.lock();
    try
    {
        while (balance < amount)
        {
            Wait for the balance to grow.
        }
        . . .
    }
    finally
    {
        balanceChangeLock.unlock();
    }
}
```

But how can we wait for the balance to grow? We can’t simply call sleep inside the
withdraw method. If a thread sleeps after acquiring a lock, it blocks all other threads
that want to use the same lock. In particular, no other thread can successfully execute
the deposit method. Other threads will call deposit, but they will simply be blocked
until the withdraw method exits. But the withdraw method doesn’t exit until it has funds
available. This is the deadlock situation that we mentioned earlier.

To overcome this problem, we use a condition object. Condition objects allow a
thread to temporarily release a lock, so that another thread can proceed, and to regain
the lock at a later time.

In the telephone booth analogy, suppose that the coin reservoir of the telephone
is completely filled, so that no further calls can be made until a service technician
removes the coins. You don’t want the person in the booth to go to sleep with the
door closed. Instead, think of the person leaving the booth temporarily. That gives
another person (hopefully a service technician) a chance to enter the booth.

Each condition object belongs to a specific lock object. You obtain a condition
object with the newCondition method of the Lock interface. For example,

```java
public class BankAccount
{
```
private Lock balanceChangeLock;
private Condition sufficientFundsCondition;
...
public BankAccount()
{
    balanceChangeLock = new ReentrantLock();
    sufficientFundsCondition = balanceChangeLock.newCondition();
    ...
}

It is customary to give the condition object a name that describes the condition that you want to test (such as “sufficient funds”). You need to implement an appropriate test. For as long as the test is not fulfilled, call the await method on the condition object:

```java
public void withdraw(double amount)
{
    balanceChangeLock.lock();
    try
    {
        while (balance < amount)
        {
            sufficientFundsCondition.await();
        }
    }
    finally
    {
        balanceChangeLock.unlock();
    }
}
```

When a thread calls await, it is not simply deactivated in the same way as a thread that reaches the end of its time slice. Instead, it is in a blocked state, and it will not be activated by the thread scheduler until it is unblocked. To unblock, another thread must execute the signalAll method on the same condition object. The signalAll method unblocks all threads waiting on the condition. They can then compete with all other threads that are waiting for the lock object. Eventually, one of them will gain access to the lock, and it will exit from the await method.

In our situation, the deposit method calls signalAll:

```java
public void deposit(double amount)
{
    balanceChangeLock.lock();
    try
    {
        ...
        sufficientFundsCondition.signalAll();
    }
    finally
    {
        balanceChangeLock.unlock();
    }
}
```

The call to signalAll notifies the waiting threads that sufficient funds may be available, and that it is worth testing the loop condition again.
In the telephone booth analogy, the thread calling `await` corresponds to the person who enters the booth and finds that the phone doesn’t work. That person then leaves the booth and waits outside, depressed, doing absolutely nothing, even as other people enter and leave the booth. The person knows it is pointless to try again. At some point, a service technician enters the booth, empties the coin reservoir, and shouts a signal. Now all the waiting people stop being depressed and again compete for the telephone booth.

There is also a `signal` method, which randomly picks just one thread that is waiting on the object and unblocks it. The `signal` method can be more efficient, but it is useful only if you know that every waiting thread can actually proceed. In general, you don’t know that, and `signal` can lead to deadlocks. For that reason, we recommend that you always call `signalAll`.

The `await` method can throw an `InterruptedException`. The `withdraw` method propagates that exception, because it has no way of knowing what the thread that calls the `withdraw` method wants to do if it is interrupted.

With the calls to `await` and `signalAll` in the `withdraw` and `deposit` methods, we can launch any number of withdrawal and deposit threads without a deadlock. If you run the sample program, you will note that all transactions are carried out without ever reaching a negative balance.

```
section_5/BankAccount.java

import java.util.concurrent.locks.Condition;
import java.util.concurrent.locks.Lock;
import java.util.concurrent.locks.ReentrantLock;

/**
   A bank account has a balance that can be changed by deposits and withdrawals.
*/
public class BankAccount {

    private double balance;
    private Lock balanceChangeLock;
    private Condition sufficientFundsCondition;

    /**
     * Constructs a bank account with a zero balance.
     */
    public BankAccount() {
        balance = 0;
        balanceChangeLock = new ReentrantLock();
        sufficientFundsCondition = balanceChangeLock.newCondition();
    }

    /**
     * Deposits money into the bank account.
     * @param amount the amount to deposit
     */
    public void deposit(double amount) {
        balanceChangeLock.lock();
        try {
            
```
section_5/BankAccountThreadRunner.java

```java
/**
 * This program runs threads that deposit and withdraw money from the same bank account.
 */
public class BankAccountThreadRunner {
    public static void main(String[] args) {
        BankAccount account = new BankAccount();
        final double AMOUNT = 100;
    }
}
```
final int REPETITIONS = 100;
final int THREADS = 100;

for (int i = 1; i <= THREADS; i++)
{
    DepositRunnable d = new DepositRunnable(
        account, AMOUNT, REPETITIONS);
    WithdrawRunnable w = new WithdrawRunnable(
        account, AMOUNT, REPETITIONS);

    Thread dt = new Thread(d);
    Thread wt = new Thread(w);

    dt.start();
    wt.start();
}

Program Run

Depositing 100.0, new balance is 100.0
Withdraw 100.0, new balance is 0.0
Depositing 100.0, new balance is 100.0
Depositing 100.0, new balance is 200.0
...  
Withdraw 100.0, new balance is 100.0
Depositing 100.0, new balance is 200.0
Withdraw 100.0, new balance is 100.0
Withdraw 100.0, new balance is 0.0

9. What is the essential difference between calling sleep and await?

10. Why is the sufficientFundsCondition object an instance variable of the BankAccount class and not a local variable of the withdraw and deposit methods?

Practice It

Now you can try these exercises at the end of the chapter: R22.12, E22.3, E22.4, E22.5.

Calling await Without Calling signalAll

It is intuitively clear when to call await. If a thread finds out that it can’t do its job, it has to wait. But once a thread has called await, it temporarily gives up all hope and doesn’t try again until some other thread calls signalAll on the condition object for which the thread is waiting. In the telephone booth analogy, if the service technician who empties the coin reservoir doesn’t notify the waiting people, they’ll wait forever.

A common error is to have threads call await without matching calls to signalAll by other threads. Whenever you call await, ask yourself which call to signalAll will signal your waiting thread.
22.5  Avoiding Deadlocks

**Calling `signalAll` Without Locking the Object**

The thread that calls `signalAll` must own the lock that belongs to the condition object on which `signalAll` is called. Otherwise, an `IllegalMonitorStateException` is thrown.

In the telephone booth analogy, the service technician must shout the signal while inside the telephone booth after emptying the coin reservoir.

In practice, this should not be a problem. Remember that `signalAll` is called by a thread that has just changed the state of some shared data in a way that may benefit waiting threads. That change should be protected by a lock in any case. As long as you use a lock to protect all access to shared data, and you are in the habit of calling `signalAll` after every beneficial change, you won’t run into problems. But if you use `signalAll` in a haphazard way, you may encounter the `IllegalMonitorStateException`.

### Object Locks and Synchronized Methods

The Lock and Condition interfaces were added in Java version 5.0. They overcome limitations of the thread synchronization mechanism in earlier Java versions. In this note, we discuss that classic mechanism.

Every Java object has one built-in lock and one built-in condition variable. The lock works in the same way as a `ReentrantLock` object. However, to acquire the lock, you call a **synchronized method**.

You simply tag all methods that contain thread-sensitive code (such as the `deposit` and `withdraw` methods of the `BankAccount` class) with the `synchronized` reserved word.

```java
public class BankAccount {
    public synchronized void deposit(double amount) {
        System.out.print("Depositing "+ amount);
        double newBalance = balance + amount;
        System.out.println(" , new balance is "+ newBalance);
        balance = newBalance;
    }

    public synchronized void withdraw(double amount) {
        . . .
    }
    . . .
}
```

When a thread calls a synchronized method on a `BankAccount` object, it owns that object’s lock until it returns from the method and thereby unlocks the object. When an object is locked by one thread, no other thread can enter a synchronized method for that object. When another thread makes a call to a synchronized method for that object, the calling thread is automatically deactivated and needs to wait until the first thread has unlocked the object again.

In other words, the `synchronized` reserved word automatically implements the `lock/try/finally/unlock` idiom for the built-in lock.

The object lock has a single condition variable that you manipulate with the `wait`, `notifyAll`, and `notify` methods of the `Object` class. If you call `x.wait()`, the current thread is added to the
set of threads that is waiting for the condition of the object $x$. Most commonly, you will call `wait()`, which makes the current thread wait on this. For example,

```java
public synchronized void withdraw(double amount)
throws InterruptedException {
    while (balance < amount) {
        wait();
    }
    ...
}
```

The call `notifyAll()` unblocks all threads that are waiting for this:

```java
public synchronized void deposit(double amount) {
    ...
    notifyAll();
}
```

This classic mechanism is undeniably simpler than using explicit locks and condition variables. However, there are limitations. Each object lock has one condition variable, and you can’t test whether another thread holds the lock. If these limitations are not a problem, by all means, go ahead and use the `synchronized` reserved word. If you need more control over threads, the `Lock` and `Condition` interfaces give you additional flexibility.

### The Java Memory Model

In a computer with multiple CPUs, you have to be particularly careful when multiple threads access shared data. Because modern processors are quite a bit faster than RAM memory, each CPU has its own memory cache that stores copies of frequently used memory locations. If a thread changes shared data, another thread may not see the change until both processor caches are synchronized. The same effect can happen even on a computer with a single CPU—occasionally, memory values are cached in CPU registers.

The Java language specification contains a set of rules, called the *memory model*, that describes under which circumstances the virtual machine must ensure that changes to shared data are visible in other threads. One of the rules states the following:

- If a thread changes shared data and then releases a lock, and another thread acquires the same lock and reads the same data, then it is guaranteed to see the changed data. However, if the first thread does not release a lock, then the virtual machine is not required to write cached data back to memory. Similarly, if the second thread does not acquire the lock, the virtual machine is not required to refresh its cache from memory.

Thus, you should always use locks or synchronized methods when you access data that is shared among multiple threads, even if you are not concerned about race conditions.

### 22.6 Application: Algorithm Animation

One popular use for thread programming is animation. A program that displays an animation shows different objects moving or changing in some way as time progresses. This is often achieved by launching one or more threads that compute how parts of the animation change.
22.6 Application: Algorithm Animation

You can use the Swing Timer class for simple animations without having to do any thread programming—see Exercise P22.7 for an example. However, more advanced animations are best implemented with threads.

In this section you will see a particular kind of animation, namely the visualization of the steps of an algorithm. Algorithm animation is an excellent technique for gaining a better understanding of how an algorithm works. Many algorithms can be animated—type “Java algorithm animation” into your favorite web search engine, and you’ll find lots of links to web pages with animations of various algorithms.

All algorithm animations have a similar structure. The algorithm runs in a separate thread that periodically updates an image of the current state of the algorithm and then pauses so that the user can view the image. After a short amount of time, the algorithm thread wakes up again and runs to the next point of interest in the algorithm. It then updates the image and pauses again. This sequence is repeated until the algorithm has finished.

Let’s take the selection sort algorithm of Chapter 14 as an example. That algorithm sorts an array of values. It first finds the smallest element, by inspecting all elements in the array and bringing the smallest element to the leftmost position. It then finds the smallest element among the remaining elements and brings it into the second position. It keeps going in that way. As the algorithm progresses, the sorted part of the array grows.

How can you visualize this algorithm? It is useful to show the part of the array that is already sorted in a different color. Also, we want to show how each step of the algorithm inspects another element in the unsorted part. That demonstrates why the selection sort algorithm is so slow—it first inspects all elements of the array, then all but one, and so on. If the array has \( n \) elements, the algorithm inspects

\[
n + (n - 1) + (n - 2) + \cdots = \frac{n(n + 1)}{2}
\]

or \( O(n^2) \) elements. To demonstrate that, we mark the currently visited element in red.

Thus, the algorithm state is described by three items:

- The array of values
- The size of the already sorted area
- The currently marked element

We add this state to the SelectionSorter class.

```java
public class SelectionSorter {
    // This array is being sorted
    private int[] a;
    // These instance variables are needed for drawing
    private int markedPosition = -1;
    private int alreadySorted = -1;
    // ...
}
```

The array that is being sorted is now an instance variable, and we will change the sort method from a static method to an instance method.

This state is accessed by two threads: the thread that sorts the array and the thread that paints the frame. We use a lock to synchronize access to the shared state.
Finally, we add a `component` instance variable to the algorithm class and augment the constructor to set it. That instance variable is needed for repainting the component and finding out the dimensions of the component when drawing the algorithm state.

```java
public class SelectionSorter
{
    private JComponent component;
    . . .
    public SelectionSorter(int[] anArray, JComponent aComponent)
    {
        a = anArray;
        sortStateLock = new ReentrantLock();
        component = aComponent;
    }
}
```

At each point of interest, the algorithm needs to pause so that the user can admire the graphical output. We supply the `pause` method shown below, and call it at various places in the algorithm. The `pause` method repaints the component and sleeps for a small delay that is proportional to the number of steps involved.

```java
public void pause(int steps) throws InterruptedException
{
    component.repaint();
    Thread.sleep(steps * DELAY);
}
```

We add a `draw` method to the algorithm class that can draw the current state of the data structure, with the items of special interest highlighted. The `draw` method is specific to the particular algorithm. This `draw` method draws the array elements as a sequence of sticks in different colors. The already sorted portion is blue, the marked position is red, and the remainder is black (see Figure 3).

```java
public void draw(Graphics g)
{
    sortStateLock.lock();
    try
    {
        int deltaX = component.getWidth() / a.length;
        for (int i = 0; i < a.length; i++)
        {
            if (i == markedPosition)
            {
                g.setColor(Color.RED);
            } else if (i <= alreadySorted)
            {
                g.setColor(Color.BLUE);
            } else
            {
                g.setColor(Color.BLACK);
            }
            g.drawLine(i * deltaX, 0, i * deltaX, a[i]);
        }
    }
    finally
    {
        sortStateLock.unlock();
    }
}
```
You need to update the special positions as the algorithm progresses and pause the animation whenever something interesting happens. The pause should be proportional to the number of steps that are being executed. For a sorting algorithm, pause one unit for each visited array element.

Here is the minimumPosition method from Chapter 14:

```java
public static int minimumPosition(int[] a, int from)
{
    int minPos = from;
    for (int i = from + 1; i < a.length; i++)
    {
        if (a[i] < a[minPos]) { minPos = i; }
    }
    return minPos;
}
```

After each iteration of the for loop, update the marked position of the algorithm state; then pause the program. To measure the cost of each step fairly, pause for two units of time, because two array elements were inspected. Because we need to access the marked position and call the pause method, we need to change the method to an instance method:

```java
private int minimumPosition(int from)
    throws InterruptedException
{
    int minPos = from;
    for (int i = from + 1; i < a.length; i++)
    {
        sortStateLock.lock();
        try
        {
            if (a[i] < a[minPos]) { minPos = i; }
            // For animation
            markedPosition = i;
        }
        finally
        {
            sortStateLock.unlock();
        }
    }
    return minPos;
}
```
The sort method is augmented in the same way. You will find the code at the end of this section. This concludes the modification of the algorithm class. Let us now turn to the component class.

The component’s paintComponent method calls the draw method of the algorithm object.

```java
public class SelectionSortComponent extends JComponent {
    private SelectionSorter sorter;

    public void paintComponent(Graphics g) {
        sorter.draw(g);
    }
}
```

The SelectionSortComponent constructor constructs a SelectionSorter object, which supplies a new array and the this reference to the component that displays the sorted values:

```java
public SelectionSortComponent() {
    int[] values = ArrayUtil.randomIntArray(30, 300);
    sorter = new SelectionSorter(values, this);
}
```

The startAnimation method constructs a thread that calls the sorter’s sort method:

```java
public void startAnimation() {
    class AnimationRunnable implements Runnable {
        public void run() {
            try {
                sorter.sort();
            } catch (InterruptedException exception) {
            }
        }
    }
    Runnable r = new AnimationRunnable();
    Thread t = new Thread(r);
    t.start();
}
```

The class for the viewer program that displays the animation is at the end of this example. Run the program and the animation starts.
Exercise P22.8 asks you to animate the merge sort algorithm of Chapter 14. If you do that exercise, then start both programs and run them in parallel to see which algorithm is faster. Actually, you may find the result surprising. If you build fair delays into the merge sort animation to account for the copying from and to the temporary array, you will find that it doesn’t perform all that well for small arrays. But if you increase the array size, then the advantage of the merge sort algorithm becomes clear.

**section_6/SelectionSortViewer.java**

```java
import java.awt.BorderLayout;
import javax.swing.JButton;
import javax.swing.JFrame;

public class SelectionSortViewer {
    public static void main(String[] args) {
        JFrame frame = new JFrame();

        final int FRAME_WIDTH = 300;
        final int FRAME_HEIGHT = 400;

        frame.setSize(FRAME_WIDTH, FRAME_HEIGHT);
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

        final SelectionSortComponent component = new SelectionSortComponent();
        frame.add(component, BorderLayout.CENTER);

        frame.setVisible(true);
        component.startAnimation();
    }
}
```

**section_6/SelectionSortComponent.java**

```java
import java.awt.Graphics;
import javax.swing.JComponent;

/**
 * A component that displays the current state of the selection sort algorithm.
 */
public class SelectionSortComponent extends JComponent {
    private SelectionSorter sorter;

    /**
     * Constructs the component.
     */
    public SelectionSortComponent() {
        int[] values = ArrayUtil.randomIntArray(30, 300);
        sorter = new SelectionSorter(values, this);
    }

    public void paintComponent(Graphics g) {
        sorter.draw(g);
    }
}
/**
   * Starts a new animation thread.
   */
   public void startAnimation()
   {
      class AnimationRunnable implements Runnable
      {
         public void run()
         {
            try
            {
               sorter.sort();
            }
            catch (InterruptedException exception)
            {
            }
         }
      }

      Runnable r = new AnimationRunnable();
      Thread t = new Thread(r);
      t.start();
   }
public SelectionSorter(int[] anArray, JComponent aComponent) {
    a = anArray;
    sortStateLock = new ReentrantLock();
    component = aComponent;
}
/**
 * Sorts the array managed by this selection sorter.
 */
public void sort() throws InterruptedException {
    for (int i = 0; i < a.length - 1; i++) {
        int minPos = minimumPosition(i);
        sortStateLock.lock();
        try {
            ArrayUtil.swap(a, minPos, i);
            // For animation
            alreadySorted = i;
        } finally {
            sortStateLock.unlock();
        }
    }
    pause(2);
}
/**
 * Finds the smallest element in a tail range of the array.
 * @param from the first position in a to compare
 * @return the position of the smallest element in the range a[from] ... a[a.length - 1]
 */
private int minimumPosition(int from) throws InterruptedException {
    int minPos = from;
    for (int i = from + 1; i < a.length; i++) {
        sortStateLock.lock();
        try {
            if (a[i] < a[minPos]) { minPos = i; }
            // For animation
            markedPosition = i;
        } finally {
            sortStateLock.unlock();
        }
    }
    return minPos;
}
Draws the current state of the sorting algorithm.
@param g the graphics context
*/
public void draw(Graphics g)
{
    sortStateLock.lock();
    try
    {
        int deltaX = component.getWidth() / a.length;
        for (int i = 0; i < a.length; i++)
        {
            if (i == markedPosition)
            {
                g.setColor(Color.RED);
            }
            else if (i <= alreadySorted)
            {
                g.setColor(Color.BLUE);
            }
            else
            {
                g.setColor(Color.BLACK);
            }
            g.drawLine(i * deltaX, 0, i * deltaX, a[i]);
        }
    }
    finally
    {
        sortStateLock.unlock();
    }
}

/**
Pauses the animation.
@param steps the number of steps to pause
*/
public void pause(int steps)
throws InterruptedException
{
    component.repaint();
    Thread.sleep(steps * DELAY);
}

11. Why is the draw method added to the SelectionSorter class and not the SelectionSortComponent class?
12. Would the animation still work if the startAnimation method simply called sorter.sort() instead of spawning a thread that calls that method?

Practice It Now you can try these exercises at the end of the chapter: R22.14, P22.5, P22.7.
Describe how multiple threads execute concurrently.

- A thread is a program unit that is executed concurrently with other parts of the program.
- The `start` method of the `Thread` class starts a new thread that executes the `run` method of the associated `Runnable` object.
- The `sleep` method puts the current thread to sleep for a given number of milliseconds.
- When a thread is interrupted, the most common response is to terminate the `run` method.
- The thread scheduler runs each thread for a short amount of time, called a time slice.

Choose appropriate mechanisms for terminating threads.

- A thread terminates when its `run` method terminates.
- The `run` method can check whether its thread has been interrupted by calling the `interrupted` method.

Recognize the causes and effects of race conditions.

- A race condition occurs if the effect of multiple threads on shared data depends on the order in which the threads are scheduled.

Use locks to control access to resources that are shared by multiple threads.

- By calling the `lock` method, a thread acquires a `Lock` object. Then no other thread can acquire the lock until the first thread releases the lock.

Explain how deadlocks occur and how they can be avoided with condition objects.

- A deadlock occurs if no thread can proceed because each thread is waiting for another to do some work first.
- Calling `await` on a condition object makes the current thread wait and allows another thread to acquire the lock object.
- A waiting thread is blocked until another thread calls `signalAll` or `signal` on the condition object for which the thread is waiting.

Use multiple threads to display an animation of an algorithm.

- Use a separate thread for running the algorithm that is being animated.
- The algorithm state needs to be safely accessed by the algorithm and painting threads.
Run a program with the following instructions:

```java
GreetingRunnable r1 = new GreetingRunnable("Hello");
GreetingRunnable r2 = new GreetingRunnable("Goodbye");
r1.run();
r2.run();
```

Note that the threads don’t run in parallel. Explain.

In the program of Section 22.1, is it possible that both threads are sleeping at the same time? Is it possible that neither of the two threads is sleeping at a particular time? Explain.

In Java, a program with a graphical user interface has more than one thread. Explain how you can prove that.

Why is the `stop` method for stopping a thread deprecated? How do you terminate a thread?

Give an example of why you would want to terminate a thread.

Suppose you surround each call to the `sleep` method with a `try/catch` block to catch an `InterruptedException` and ignore it. What problem do you create?

What is a race condition? How can you avoid it?

Consider the `ArrayList` implementation from Section 16.2. Describe two different scenarios in which race conditions can corrupt the data structure.

Consider a stack that is implemented as a linked list, as in Section 16.3.1. Describe two different scenarios in which race conditions can corrupt the data structure.

Consider a queue that is implemented as a circular array, as in Section 16.3.4. Describe two different scenarios in which race conditions can corrupt the data structure.

What is a deadlock? How can you avoid it?

What is the difference between a thread that sleeps by calling `sleep` and a thread that waits by calling `wait`?
R22.13 What happens when a thread calls await and no other thread calls signalAll or signal?

R22.14 In the algorithm animation program of Section 22.6, we do not use any conditions. Why not?

E22.1 Write a program in which multiple threads add and remove elements from a java.util.LinkedList. Demonstrate that the list is being corrupted.

E22.2 Implement a stack as a linked list in which the push, pop, and isEmpty methods can be safely accessed from multiple threads.

E22.3 Implement a Queue class whose add and remove methods are synchronized. Supply one thread, called the producer, which keeps inserting strings into the queue as long as there are fewer than ten elements in it. When the queue gets too full, the thread waits. As sample strings, simply use time stamps new Date().toString(). Supply a second thread, called the consumer, that keeps removing and printing strings from the queue as long as the queue is not empty. When the queue is empty, the thread waits. Both the consumer and producer threads should run for 100 iterations.

E22.4 Enhance the program of Exercise E22.3 by supplying a variable number of producer and consumer threads. Prompt the program user for the numbers.

E22.5 Reimplement Exercise E22.4 by using the ArrayBlockingQueue class from the standard library.

E22.6 Modify the ArrayList implementation of Section 16.2 so that all methods can be safely accessed from multiple threads.

E22.7 Write a program WordCount that counts the words in one or more files. Start a new thread for each file. For example, if you call

```java
java WordCount report.txt address.txt Homework.java
```

then the program might print

```
address.txt: 1052
Homework.java: 445
report.txt: 2099
```

E22.8 Enhance the program of Exercise E22.7 so that the last active thread also prints a combined count. Use locks to protect the combined word count and a counter of active threads.

E22.9 Add a condition to the deposit method of the BankAccount class in Section 22.5, restricting deposits to $100,000 (the insurance limit of the U.S. government). The method should block until sufficient money has been withdrawn by another thread. Test your program with a large number of deposit threads.
Write a program `Find` that searches all files specified on the command line and prints out all lines containing a reserved word. Start a new thread for each file. For example, if you call

```java
java Find Buff report.txt address.txt Homework.java
```

then the program might print

```
report.txt: Buffet style lunch will be available at the
address.txt: Buffet, Warren|11801 Trenton Court|Dallas|TX
Homework.java: BufferedReader in;
address.txt: Walters, Winnie|59 Timothy Circle|Buffalo|MI
```

Implement the merge sort algorithm of Chapter 14 by spawning a new thread for each smaller `MergeSorter`. *Hint:* Use the `join` method of the `Thread` class to wait for the spawned threads to finish. Look up the method’s behavior in the API documentation.

Write a program that shows two cars moving across a window. Use a separate thread for each car.

Modify Exercise P22.3 so that the cars change direction when they hit an edge of the window.

Enhance the `SelectionSorter` of Section 22.6 so that the current minimum is painted in yellow.

Enhance the `SelectionSortViewer` of Section 22.6 so that the sorting only starts when the user clicks a “Start” button.

Instead of using a thread and a `pause` method, use the `Timer` class introduced in Chapter 10 to animate an algorithm. Whenever the timer sends out an action event, run the algorithm to the next step and display the state. That requires a more extensive recoding of the algorithm. You need to implement a `runToNextStep` method that is capable of running the algorithm one step at a time. Add sufficient instance variables to the algorithm to remember where the last step left off. For example, in the case of the selection sort algorithm, if you know the values of `alreadySorted` and `markedPosition`, you can determine the next step.

Implement an animation of the merge sort algorithm of Chapter 14. Reimplement the algorithm so that the recursive calls sort the elements inside a subrange of the original array, rather than in their own arrays:

```java
public void mergeSort(int from, int to) {
    if (from == to) { return; }
    int mid = (from + to) / 2;
    mergeSort(from, mid);
    mergeSort(mid + 1, to);
    merge(from, mid, to);
}
```

The `merge` method merges the sorted ranges `a[from] ... a[mid]` and `a[mid + 1] ... a[to]`. Merge the ranges into a temporary array, then copy back the temporary array into the combined range.
Pause in the `merge` method whenever you inspect an array element. Color the range `a[from] ... a[to]` in blue and the currently inspected element in red.

Enhance Exercise P22.8 so that it shows two frames, one for a merge sorter and one for a selection sorter. They should both sort arrays with the same values.

Reorganize the code of the sorting animation in Section 22.6 so that it can be used for generic animations. Provide a class `Animated` with abstract methods

```java
public void run()
public void draw(Graphics g, int width, int height)
```

and concrete methods

```java
public void lock()
public void unlock(int steps)
public void setComponent(JComponent component)
```

so that the `SelectionSorter` can be implemented as

```java
public class SelectionSorter extends Animated {
    private int[] a;
    private int markedPosition = -1;
    private int alreadySorted = -1;

    public SelectionSorter(int[] anArray) { a = anArray; }

    public void run() {
        for (int i = 0; i < a.length - 1; i++) {
            int minPos = minimumPosition(i);
            lock();
            ArrayUtil.swap(a, minPos, i);
            alreadySorted = i;
            unlock(2);
        }
    }

    private int minimumPosition(int from) {
        int minPos = from;
        for (int i = from + 1; i < a.length; i++) {
            lock();
            if (a[i] < a[minPos]) { minPos = i; }
            markedPosition = i;
            unlock(2);
        }
        return minPos;
    }

    public void draw(Graphics g, int width, int height) {
        int deltaX = width / a.length;
        int deltaX = width / a.length;
        for (int i = 0; i < a.length; i++) {
            if (i == markedPosition) { g.setColor(Color.RED); }
            else if (i <= alreadySorted) { g.setColor(Color.BLUE); }
            else { g.setColor(Color.BLACK); }
            g.drawLine(i * deltaX, 0, i * deltaX, a[i]);
        }
    }
}
```
1. The messages are printed about one millisecond apart.
2. The first call to run would print ten “Hello” messages, and then the second call to run would print ten “Goodbye” messages.
3. If the user hits the “Back” button, the current web page is no longer displayed, and it makes no sense to expend network resources to fetch additional image data.
4. The run method prints the values 1, 3, and 4. The call to interrupt merely sets the interruption flag, but the sleep method immediately throws an InterruptedException.
5. There are many possible scenarios. Here is one:
   a. The first thread loses control after the first print statement.
   b. The second thread loses control just before the assignment balance = newBalance.
   c. The first thread completes the deposit method.
   d. The second thread completes the withdraw method.
6. One thread calls addFirst and is preempted just before executing the assignment first = newNode. Then the next thread calls addFirst, using the old value of first. Then the first thread completes the process, setting first to its new node. As a result, the links are not in sequence.
7. Two, one for each bank account object. Each lock protects a separate balance variable.
8. When a thread calls deposit, it continues to own the lock, and any other thread trying to deposit or withdraw money in the same bank account is blocked forever.
9. A sleeping thread is reactivated when the sleep delay has passed. A waiting thread is only reactivated if another thread has called signalAll or signal.
10. The calls to await and signal/signalAll must be made to the same object.
11. The draw method uses the array values and the values that keep track of the algorithm’s progress. These values are available only in the SelectionSorter class.
12. Yes, provided you only show a single frame. If you modify the SelectionSortViewer program to show two frames, you want the sorters to run in parallel.
CHAPTER GOALS
To understand the concept of sockets
To send and receive data through sockets
To implement network clients and servers
To communicate with web servers and server-side applications through the Hypertext Transfer Protocol (HTTP)

CHAPTER CONTENTS

23.1 THE INTERNET PROTOCOL W988
23.2 APPLICATION LEVEL PROTOCOLS W990
23.3 A CLIENT PROGRAM W993
23.4 A SERVER PROGRAM W996
23.5 URL CONNECTIONS W1004

PT1 Use High-Level Libraries W1007
HT1 Designing Client/Server Programs W1003
You probably have quite a bit of experience with the Internet, the global network that links together millions of computers. In particular, you use the Internet whenever you browse the World Wide Web. Note that the Internet is not the same as the “Web”. The World Wide Web is only one of many services offered over the Internet. E-mail, another popular service, also uses the Internet, but its implementation differs from that of the Web. In this chapter, you will see what goes on “under the hood” when you send an e-mail message or when you retrieve a web page from a remote server. You will also learn how to write programs that fetch data from sites across the Internet and how to write server programs that can serve information to other programs.

23.1 The Internet Protocol

Computers can be connected with each other through a variety of physical media. In a computer lab, for example, computers are connected by network cabling. Electrical impulses representing information flow across the cables. If you use a DSL modem to connect your computer to the Internet, the signals travel across a regular telephone wire, encoded as tones. On a wireless network, signals are sent by transmitting a modulated radio frequency. The physical characteristics of these transmissions differ widely, but they ultimately consist of sending and receiving streams of zeroes and ones along the network connection.

These zeroes and ones represent two kinds of information: application data, the data that one computer actually wants to send to another, and network protocol data, the data that describe how to reach the intended recipient and how to check for errors and data loss in the transmission. The protocol data follow certain rules set forth by the Internet Protocol Suite, also called TCP/IP, after the two most important protocols in the suite. These protocols have become the basis for connecting computers around the world over the Internet. We will discuss TCP and IP in this chapter.

Suppose that a computer A wants to send data to a computer B, both on the Internet. The computers aren’t connected directly with a cable, as they could be if both were on the same local area network. Instead, A may be someone’s home computer and connected to an Internet service provider (ISP), which is in turn connected to an Internet access point; B might be a computer on a local area network belonging to a large firm that has an Internet access point of its own, which may be half a world away from A. The Internet itself, finally, is a complex collection of pathways on which a message can travel from one Internet access point to, eventually, any other Internet access point (see Figure 1). Those connections carry millions of messages, not just the data that A is sending to B.

For the data to arrive at its destination, it must be marked with a destination address. In IP, addresses are denoted by sequences of four numbers, each one byte (that is, between 0 and 255); for example, 130.65.86.66. (Because there aren’t enough four-byte addresses for all devices that would like to connect to the Internet, these addresses have been extended to sixteen bytes. For simplicity, we use the classic four-byte addresses in this chapter.) In order to send data, A needs to know the Internet
address of B and include it in the protocol portion when sending the data across the Internet. The routing software that is distributed across the Internet can then deliver the data to B.

Of course, addresses such as 130.65.86.66 are not easy to remember. You would not be happy if you had to use number sequences every time you sent e-mail or requested information from a web server. On the Internet, computers can have so-called *domain names* that are easier to remember, such as cs.sjsu.edu or horstmann.com. A special service called the *Domain Name System (DNS)* translates between domain names and Internet addresses. Thus, if computer A wants to have information from horstmann.com, it first asks the DNS to translate this domain name into a numeric Internet address; then it includes the numeric address with the request.

One interesting aspect of IP is that it breaks large chunks of data up into more manageable *packets*. Each packet is delivered separately, and different packets that are part of the same transmission can take different routes through the Internet. Packets are numbered, and the recipient reassembles them in the correct order.

The Internet Protocol is used when attempting to deliver data from one computer to another across the Internet. If some data get lost or garbled in the process, IP has safeguards built in to make sure that the recipient is aware of that unfortunate fact and doesn’t rely on incomplete data. However, IP has no provision for retrying an incomplete transmission. That is the job of a higher-level protocol, the *Transmission Control Protocol (TCP)*. This protocol attempts reliable delivery of data, with retries if there are failures, and it notifies the sender whether or not the attempt succeeded. Most, but not all, Internet programs use TCP for reliable delivery. (Exceptions are “streaming media” services, which bypass the slower TCP for the highest possible throughput and tolerate occasional information loss. However, the most popular Internet services—the World Wide Web and e-mail—use TCP.) TCP is independent of the Internet Protocol; it could in principle be used with another lower-level
network protocol. However, in practice, TCP over IP (often called TCP/IP) is the most commonly used combination. We will focus on TCP/IP networking in this chapter.

A computer that is connected to the Internet may have programs for many different purposes. For example, a computer may run both a web server program and a mail server program. When data are sent to that computer, they need to be marked so that they can be forwarded to the appropriate program. TCP uses port numbers for this purpose. A port number is an integer between 0 and 65,535. The sending computer must know the port number of the receiving program and include it with the transmitted data. Some applications use “well-known” port numbers. For example, by convention, web servers use port 80, whereas mail servers running the Post Office Protocol (POP) use port 110. A TCP connection, therefore, requires

- The Internet address of the recipient.
- The port number of the recipient.
- The Internet address of the sender.
- The port number of the sender.

You can think of a TCP connection as a “pipe” between two computers that links the two ports together. Data flow in either direction through the pipe. In practical programming situations, you simply establish a connection and send data across it without worrying about the details of the TCP/IP mechanism. You will see how to establish such a connection in Section 23.3.

1. What is the difference between an IP address and a domain name?
2. Why do some streaming media services not use TCP?

Practice It Now you can try these exercises at the end of the chapter: R23.1, R23.2, R23.3.

23.2 Application Level Protocols

In the preceding section you saw how the TCP/IP mechanism can establish an Internet connection between two ports on two computers so that the two computers can exchange data. Each Internet application has a different application protocol, which describes how the data for that particular application are transmitted.

Consider, for example, HTTP: the Hypertext Transfer Protocol, which is used for the World Wide Web. Suppose you type a web address, called a Uniform Resource Locator (URL), such as http://horstmann.com/index.html, into the address window of your browser and ask the browser to load the page.

The browser now takes the following steps:

1. It examines the part of the URL between the double slash and the first single slash (“horstmann.com”), which identifies the computer to which you want to connect. Because this part of the URL contains letters, it must be a domain name rather than an Internet address, so the browser sends a request to a DNS
server to obtain the Internet address of the computer with domain name horstmann.com.

2. From the http: prefix of the URL, the browser deduces that the protocol you want to use is HTTP, which by default uses port 80.

3. It establishes a TCP/IP connection to port 80 at the Internet address it obtained in Step 1.

4. It deduces from the /index.html suffix that you want to see the file /index.html, so it sends a request, formatted as an HTTP command, through the connection that was established in Step 3. The request looks like this:

   ```
   GET /index.html HTTP/1.1
   Host: horstmann.com
   blank line
   ```

   (The host is needed because a web server can host multiple domains with the same Internet address.)

5. The web server running on the computer whose Internet address is the one the browser obtained in Step 1 receives the request and decodes it. It then fetches the file /index.html and sends it back to the browser on your computer.

6. The browser displays the contents of the file. Because it happens to be an HTML file, the browser translates the HTML tags into fonts, bullets, separator lines, and so on. If the HTML file contains images, then the browser makes more GET requests, one for each image, through the same connection, to fetch the image data. (Appendix J contains a summary of the most frequently used HTML tags.)

You can try the following experiment to see this process in action. The “Telnet” program enables a user to type characters for sending to a remote computer and view characters that the remote computer sends back. On Windows, you need to enable the Telnet program in the control panel. UNIX, Linux, and Mac OS X systems normally have Telnet preinstalled.

For this experiment, you want to start Telnet with a host of horstmann.com and port 80. To start the program from the command line, simply type

   ```
telnet horstmann.com 80
   ```

<table>
<thead>
<tr>
<th>Command</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>GET</td>
<td>Return the requested item</td>
</tr>
<tr>
<td>HEAD</td>
<td>Request only the header information of an item</td>
</tr>
<tr>
<td>OPTIONS</td>
<td>Request communications options of an item</td>
</tr>
<tr>
<td>POST</td>
<td>Supply input to a server-side command and return the result</td>
</tr>
<tr>
<td>PUT</td>
<td>Store an item on the server</td>
</tr>
<tr>
<td>DELETE</td>
<td>Delete an item on the server</td>
</tr>
<tr>
<td>TRACE</td>
<td>Trace server communication</td>
</tr>
</tbody>
</table>
Once the program starts, type very carefully, without making any typing errors and without pressing the backspace key,

```
GET / HTTP/1.1
Host: horstmann.com
```

Then press the Enter key twice.

The first `/` denotes the root page of the web server. Note that there are spaces before and after the first `/`, but there are no spaces in `HTTP/1.1`.

On Windows, you will not see what you type, so you should be extra careful when typing in the commands.

The server now sends a response to the request—see Figure 2. The response, of course, consists of the root web page that you requested. The Telnet program is not a browser and does not understand HTML tags, so it simply displays the HTML file—text, tags, and all.

The `GET` command is one of the commands of HTTP. Table 1 shows the other commands of the protocol. As you can see, the protocol is pretty simple.

By the way, be sure not to confuse HTML with HTTP. **HTML** is a *document format* (with commands such as `<h1>` or `<ul>`) that describes the structure of a document, including headings, bulleted lists, images, hyperlinks, and so on. **HTTP** is a *protocol* (with commands such as `GET` and `POST`) that describes the command set for web server requests. Web *browsers* know how to display HTML documents and how to issue HTTP commands. Web *servers* know nothing about HTML. They merely understand HTTP and know how to fetch the requested items. Those items may be HTML documents, GIF or JPEG images, or any other data that a web browser can display.

HTTP is just one of many application protocols in use on the Internet. Another commonly used protocol is the Post Office Protocol (POP), which is used to download received messages from e-mail servers. To *send* messages, you use yet another protocol called the Simple Mail Transfer Protocol (SMTP). We don’t want to go into...
23.3 A Client Program

In this section you will see how to write a Java program that establishes a TCP connection to a server, sends a request to the server, and prints the response.

In the terminology of TCP/IP, there is a socket on each side of the connection (see Figure 4). In Java, a client establishes a socket with a call

```java
Socket s = new Socket(hostname, portnumber);
```

For example, to connect to the HTTP port of the server horstmann.com, you use

```java
final int HTTP_PORT = 80;
Socket s = new Socket("horstmann.com", HTTP_PORT);
```

Once you have a socket, you obtain its input and output streams:

```java
InputStream instream = s.getInputStream();
OutputStream outstream = s.getOutputStream();
```

A socket is an object that encapsulates a TCP connection. To communicate with the other end point of the connection, use the input and output streams attached to the socket.

Black = mail client requests
Color = mail server responses

![Figure 3](image)

A Sample POP Session

the details of these protocols, but Figure 3 gives you a flavor of the commands used by the Post Office Protocol.

Both HTTP and POP use plain text, which makes it particularly easy to test and debug client and server programs (see How To 23.1).

3. Why don’t you need to know about HTTP when you use a web browser?

4. Why is it important that you don’t make typing errors when you type HTTP commands in Telnet?

Practice It Now you can try these exercises at the end of the chapter: R23.13, R23.14, R23.15.
When you send data to `outstream`, the socket automatically forwards it to the server. The socket catches the server’s response, and you can read the response through `instream` (see Figure 4).

When you are done communicating with the server, you should close the socket. This is best done with a try-with-resources statement:

```java
try (Socket s = ...) {
    // ...} // s.close() called here
```

In Chapter 21, you saw that the `InputStream` and `OutputStream` classes are used for reading and writing bytes. If you want to communicate with the server by sending and receiving text, you should turn the streams into scanners and writers, as follows:

```java
Scanner in = new Scanner(instream);
PrintWriter out = new PrintWriter(outstream);
```

A print writer buffers the characters that you send to it. That is, characters are not immediately sent to their destination. Instead, they are placed into an array. When the array is full, then the print writer sends all characters in the array to its destination. The advantage of buffering is increased performance—it takes some amount of time to contact the destination and send it data, and it is expensive to pay for that contact time for every character. However, when communicating with a server that responds to requests, you want to make sure that the server gets a complete request. Therefore, you need to `flush` the buffer manually whenever you send a command:

```java
out.print(command);
out.flush();
```

The `flush` method empties the buffer and forwards all waiting characters to the destination.

The `WebGet` program at the end of this section lets you retrieve any item from a web server. You need to specify the host and the item from the command line. For example,

```java
java WebGet horstmann.com /
```

The `/` item denotes the root page of the web server that listens to port 80 of the host `horstmann.com`. Note that there is a space before the `/`.

The `WebGet` program establishes a connection to the host, sends a `GET` command to the host, and then receives input from the server until the server closes its connection.
public class WebGet
{
    public static void main(String[] args) throws IOException
    {
        // Get command-line arguments
        String host;
        String resource;

        if (args.length == 2)
        {
            host = args[0];
            resource = args[1];
        }
        else
        {
            System.out.println("Getting / from horstmann.com");
            host = "horstmann.com";
            resource = "/";
        }

        // Open socket
        final int HTTP_PORT = 80;
        try (Socket s = new Socket(host, HTTP_PORT))
        {

            // Get streams
            InputStream instream = s.getInputStream();
            OutputStream outstream = s.getOutputStream();

            // Turn streams into scanners and writers
            Scanner in = new Scanner(instream);
            PrintWriter out = new PrintWriter(outstream);

            // Send command
            String command = "GET " + resource + " HTTP/1.1\n" + "Host: " + host + "\n\n";
            out.print(command);
            out.flush();

            // Read server response
while (in.hasNextLine())
{
    String input = in.nextLine();
    System.out.println(input);
}

// The try-with-resources statement closes the socket
}

Program Run

Getting / from horstmann.com
HTTP/1.1 200 OK
Date: Thu, 09 Apr 2015 14:15:04 GMT
Server: Apache/1.3.41 (Unix) Sun-ONE-ASP/4.0.2
... Content-Length: 6654
Content-Type: text/html

<html>
<head><title>Cay Horstmann's Home Page</title></head>
<body>
<h1>Welcome to Cay Horstmann's Home Page</h1>
... <head><title>Cay Horstmann's Home Page</title></head>

5. What happens if you call WebGet with a nonexistent resource, such as wombat.html at horstmann.com?

6. How do you open a socket to read e-mail from the POP server at e-mail.sjsu.edu?

Practice It Now you can try these exercises at the end of the chapter: R23.7, R23.8, E23.1, E23.2.

23.4 A Server Program

Now that you have seen how to write a network client, we will turn to the server side. In this section we will develop a server program that enables clients to manage a set of bank accounts in a bank.

Whenever you develop a server application, you need to specify some application-level protocol that clients can use to interact with the server. For the purpose of this example, we will create a “Simple Bank Access Protocol”. Table 2 shows the protocol format. Of course, this is just a toy protocol to show you how to implement a server.

The server program waits for clients to connect to a particular port. We choose port 8888 for this service. This number has not been preassigned to another service, so it is unlikely to be used by another server program. To listen to incoming connec-
tions, you use a *server socket*. To construct a server socket, you need to supply the port number:

```java
ServerSocket server = new ServerSocket(8888);
```

The `accept` method of the `ServerSocket` class waits for a client connection. When a client connects, then the server program obtains a socket through which it communicates with the client:

```java
Socket s = server.accept();
BankService service = new BankService(s, bank);
```

The `BankService` class carries out the service. This class implements the `Runnable` interface, and its `run` method will be executed in each thread that serves a client connection. The `run` method gets a scanner and writer from the socket in the same way as we discussed in the preceding section. Then it executes the following method:

```java
public void doService() throws IOException {
    while (true) {
        if (!in.hasNext()) { return; }
        String command = in.next();
        if (command.equals("QUIT")) { return; }
        executeCommand(command);
    }
}
```

The `executeCommand` method processes a single command. If the command is `DEPOSIT`, then it carries out the deposit:

```java
int account = in.nextInt();
double amount = in.nextDouble();
bank.deposit(account, amount);
```

The `WITHDRAW` command is handled in the same way. After each command, the account number and new balance are sent to the client:

```java
out.println(account + " " + bank.getBalance(account));
```

The `doService` method returns to the `run` method if the client closed the connection or the command equals "QUIT". Then the `run` method closes the socket and exits.

> Let us go back to the point where the server socket accepts a connection and constructs the `BankService` object. At this point, we could simply call the `run` method. But then our server program would have a serious limitation: only one client could connect to it at any point in time. To overcome that limitation, server programs spawn a new thread whenever a client connects. Each thread is responsible for serving one client.

### Table 2  A Simple Bank Access Protocol

<table>
<thead>
<tr>
<th>Client Request</th>
<th>Server Response</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>BALANCE (n)</td>
<td>(n) and the balance</td>
<td>Get the balance of account (n)</td>
</tr>
<tr>
<td>DEPOSIT (n\ \ a)</td>
<td>(n) and the new balance</td>
<td>Deposit amount (a) into account (n)</td>
</tr>
<tr>
<td>WITHDRAW (n\ \ a)</td>
<td>(n) and the new balance</td>
<td>Withdraw amount (a) from account (n)</td>
</tr>
<tr>
<td>QUIT</td>
<td>None</td>
<td>Quit the connection</td>
</tr>
</tbody>
</table>
Our BankService class implements the Runnable interface. Therefore, the server program BankServer simply starts a thread with the following instructions:

```java
Thread t = new Thread(service);
t.start();
```

The thread dies when the client quits or disconnects and the run method exits. In the meantime, the BankServer loops back to accept the next connection.

```java
while (true)
{
  try (Socket s = server.accept())
  {
    BankService service = new BankService(s, bank);
    Thread t = new Thread(service);
    t.start();
  }
}
```

The server program never stops. When you are done running the server, you need to kill it. For example, if you started the server in a shell window, press Ctrl+C.

To try out the program, run the server. Then use Telnet to connect to localhost, port number 8888. Start typing commands. Here is a typical dialog (see Figure 5):

```
DEPOSIT 3 1000
3 1000.0
WITHDRAW 3 500
3 500.0
QUIT
```

Alternatively, you can use a client program that connects to the server. You will find a sample client program at the end of this section.

---

**Figure 5** Using the Telnet Program to Connect to the Bank Server
section_4/BankServer.java

```java
import java.io.IOException;
import java.net.ServerSocket;
import java.net.Socket;

/**
   A server that executes the Simple Bank Access Protocol. 
*/
public class BankServer {
    public static void main(String[] args) throws IOException {
        final int ACCOUNTS_LENGTH = 10;
        Bank bank = new Bank(ACCOUNTS_LENGTH);
        final int SBAP_PORT = 8888;
        ServerSocket server = new ServerSocket(SBAP_PORT);
        System.out.println("Waiting for clients to connect...");

        while (true) {
            try (Socket s = server.accept()) {
                System.out.println("Client connected.");
                BankService service = new BankService(s, bank);
                Thread t = new Thread(service);
                t.start();
            }
        }
    }
}
```

section_4/BankService.java

```java
import java.io.InputStream;
import java.io.IOException;
import java.io.OutputStream;
import java.io.PrintWriter;
import java.net.Socket;
import java.util.Scanner;

/**
*/
public class BankService implements Runnable {
    private Socket s;
    private Scanner in;
    private PrintWriter out;
    private Bank bank;

    /**
       Constructs a service object that processes commands from a socket for a bank.
       @param aSocket the socket 
       @param aBank the bank
    */
    public BankService(Socket aSocket, Bank aBank) {
    }
```
Chapter 23  Internet Networking

```java
s = aSocket;
bank = aBank;
}

public void run()
{
   try
   {
      in = new Scanner(s.getInputStream());
      out = new PrintWriter(s.getOutputStream());
      doService();
   }
   catch (IOException exception)
   {
      exception.printStackTrace();
   }
}

/**
   * Executes all commands until the QUIT command or the end of input.
   */
public void doService() throws IOException
{
   while (true)
   {
      if (!in.hasNext()) { return; }
      String command = in.next();
      if (command.equals("QUIT")) { return; }
      else { executeCommand(command); }
   }
}

/**
   * Executes a single command.
   * @param command the command to execute
   */
public void executeCommand(String command)
{
   int account = in.nextInt();
   if (command.equals("DEPOSIT"))
   {
      double amount = in.nextDouble();
      bank.deposit(account, amount);
   }
   else if (command.equals("WITHDRAW"))
   {
      double amount = in.nextDouble();
      bank.withdraw(account, amount);
   }
   else if (!command.equals("BALANCE"))
   {
      out.println("Invalid command");
      out.flush();
      return;
   }
   out.println(account + " " + bank.getBalance(account));
   out.flush();
}
```
section_4/Bank.java

```java
/**
 * A bank consisting of multiple bank accounts.
 */
public class Bank {
    private BankAccount[] accounts;

    /**
     * Constructs a bank account with a given number of accounts.
     * @param size the number of accounts
     */
    public Bank(int size) {
        accounts = new BankAccount[size];
        for (int i = 0; i < accounts.length; i++) {
            accounts[i] = new BankAccount();
        }
    }

    /**
     * Deposits money into a bank account.
     * @param accountNumber the account number
     * @param amount the amount to deposit
     */
    public void deposit(int accountNumber, double amount) {
        BankAccount account = accounts[accountNumber];
        account.deposit(amount);
    }

    /**
     * Withdraws money from a bank account.
     * @param accountNumber the account number
     * @param amount the amount to withdraw
     */
    public void withdraw(int accountNumber, double amount) {
        BankAccount account = accounts[accountNumber];
        account.withdraw(amount);
    }

    /**
     * Gets the balance of a bank account.
     * @param accountNumber the account number
     * @return the account balance
     */
    public double getBalance(int accountNumber) {
        BankAccount account = accounts[accountNumber];
        return account.getBalance();
    }
}
```

section_4/BankClient.java

```java
import java.io.InputStream;
import java.io.IOException;
```
import java.io.InputStream;
import java.io.OutputStream;
import java.io.PrintWriter;
import java.net.Socket;
import java.util.Scanner;

/**
 * This program tests the bank server.
 */
public class BankClient {
    public static void main(String[] args) throws IOException {
        final int SBAP_PORT = 8888;
        try (Socket s = new Socket("localhost", SBAP_PORT)) {
            InputStream instream = s.getInputStream();
            OutputStream outstream = s.getOutputStream();
            Scanner in = new Scanner(instream);
            PrintWriter out = new PrintWriter(outstream);

            String command = "DEPOSIT 3 1000\n";
            System.out.print("Sending: " + command);
            out.print(command);
            out.flush();
            String response = in.nextLine();
            System.out.println("Receiving: " + response);

            command = "WITHDRAW 3 500\n";
            System.out.print("Sending: " + command);
            out.print(command);
            out.flush();
            response = in.nextLine();
            System.out.println("Receiving: " + response);

            command = "QUIT\n";
            System.out.print("Sending: " + command);
            out.print(command);
            out.flush();
        }
    }
}

Program Run
Sending: DEPOSIT 3 1000
Receiving: 3 1000.0
Sending: WITHDRAW 3 500
Receiving: 3 500.0
Sending: QUIT

7. Why didn’t we choose port 80 for the bank server?
8. Can you read data from a server socket?

Practice It Now you can try these exercises at the end of the chapter: E23.3, E23.4, P23.2.
The bank server of this section is a typical example of a client/server program. A web browser/web server is another example. This How To outlines the steps to follow when designing a client/server application.

**Step 1** Determine whether it really makes sense to implement a stand-alone server and a matching client.

Many times it makes more sense to build a web application instead. Chapter 26 discusses the construction of web applications in detail. For example, the bank application of this section could easily be turned into a web application, using an HTML form with Withdraw and Deposit buttons. However, programs for chat or peer-to-peer file sharing cannot easily be implemented as web applications.

**Step 2** Design a communication protocol.

Figure out exactly what messages the client and server send to each other and what the success and error responses are.

With each request and response, ask yourself how the end of data is indicated.

- Do the data fit on a single line? Then the end of the line serves as the data terminator.
- Can the data be terminated by a special line (such as a blank line after the HTTP header or a line containing a period in SMTP)?
- Does the sender of the data close the socket? That’s what a web server does at the end of a GET request.
- Can the sender indicate how many bytes are contained in the request? Web browsers do that in POST requests.

Use text, not binary data, for the communication between client and server. A text-based protocol is easier to debug.

**Step 3** Implement the server program.

The server listens for socket connections and accepts them. It starts a new thread for each connection. Supply a class that implements the `Runnable` interface. The `run` method receives commands, interprets them, and sends responses back to the client.

**Step 4** Test the server with the Telnet program.

Try out all commands in the communication protocol.

**Step 5** Once the server works, write a client program.

The client program interacts with the program user, turns user requests into protocol commands, sends the commands to the server, receives the response, and displays the response for the program user.
In Section 23.3, you saw how to use sockets to connect to a web server and how to retrieve information from the server by sending HTTP commands. However, because HTTP is such an important protocol, the Java library contains a `URLConnection` class, which provides convenient support for the HTTP. The `URLConnection` class takes care of the socket connection, so you don’t have to fuss with sockets when you want to retrieve from a web server. As an additional benefit, the `URLConnection` class can also handle FTP, the file transfer protocol.

The `URLConnection` class makes it very easy to fetch a file from a web server given the file’s URL as a string. First, you construct a `URL` object from the URL in the familiar format, starting with the `http` or `ftp` prefix. Then you use the `URL` object’s `openConnection` method to get the `URLConnection` object itself:

```java
URL u = new URL("http://horstmann.com/index.html");
URLConnection connection = u.openConnection();
```

Then you call the `getInputStream` method to obtain an input stream:

```java
InputStream instream = connection.getInputStream();
```

You can turn the stream into a scanner in the usual way, and read input from the scanner.

The `URLConnection` class can give you additional useful information. To understand those capabilities, we need to have a closer look at HTTP requests and responses. You saw in Section 23.2 that the command for getting an item from the server is

```
GET item HTTP/1.1
Host: hostname

```

You may have wondered why you need to provide a blank line. This blank line is a part of the general request format. The first line of the request is a command, such as `GET` or `POST`. The command is followed by `request properties` (such as `Host:`). Some commands—in particular, the `POST` command—send input data to the server. The reason for the blank line is to denote the boundary between the request property section and the input data section.

A typical request property is `If-Modified-Since`. If you request an item with

```
GET item HTTP/1.1
Host: hostname
If-Modified-Since: date

```

the server sends the item only if it is newer than the date. Browsers use this feature to speed up redisplay of previously loaded web pages. When a web page is loaded, the browser stores it in a `cache` directory. When the user wants to see the same web page again, the browser asks the server to get a new page only if it has been modified since the date of the cached copy. If it hasn’t been, the browser simply rediscovers the cached copy and doesn’t spend time downloading another identical copy.

The `URLConnection` class has methods to set request properties. For example, you can set the `If-Modified-Since` property with the `setIfModifiedSince` method:

```java
connection.setIfModifiedSince(date);
```

You need to set request properties before calling the `getInputStream` method. The `URLConnection` class then sends to the web server all the request properties that you set.
Similarly, the response from the server starts with a status line followed by a set of response parameters. The response parameters are terminated by a blank line and followed by the requested data (for example, an HTML page). Here is a typical response:

```
HTTP/1.1 200 OK
Date: Thu, 09 Apr 2015 00:15:48 GMT
Server: Apache/1.3.3 (Unix)
Last-Modified: Tue, 03 Mar 2015 20:53:38 GMT
Content-Length: 4813
Content-Type: text/html

requested data
```

Normally, you don’t see the response code. However, you may have run across bad links and seen a page that contained a response code 404 Not Found. (A successful response has status 200 OK.)

To retrieve the response code, you need to cast the `URLConnection` object to the `HttpURLConnection` subclass. You can retrieve the response code (such as the number 200 in this example, or the code 404 if a page was not found) and response message with the `getResponseCode` and `getResponseMessage` methods:

```
HttpURLConnection httpConnection = (HttpURLConnection) connection;
int code = httpConnection.getResponseCode(); // e.g., 404
String message = httpConnection.getResponseMessage(); // e.g., “Not found”
```

As you can see from the response example, the server sends some information about the requested data, such as the content length and the content type. You can request this information with methods from the `URLConnection` class:

```
int length = connection.getContentLength();
String type = connection.getContentType();
```

You need to call these methods after calling the `getInputStream` method.

To summarize: You don’t need to use sockets to communicate with a web server, and you need not master the details of the HTTP protocol. Simply use the `URLConnection` and `HttpURLConnection` classes to obtain data from a web server, to set request properties, or to obtain response information.

The program at the end of this section puts the `URLConnection` class to work. The program fulfills the same purpose as that of Section 23.3—to retrieve a web page from a server—but it works at a higher level of abstraction. There is no longer a need to issue an explicit GET command. The `URLConnection` class takes care of that. Similarly, the parsing of the HTTP request and response headers is handled transparently to the programmer. Our sample program takes advantage of that fact. It checks whether the server response code is 200. If not, it exits. You can try that out by testing the program with a bad URL, like `http://horstmann.com/wombat.html`. Then the program prints a server response, such as 404 Not Found.

This program completes our introduction to Internet programming with Java. You have seen how to use sockets to connect client and server programs. You also saw how to use the higher-level `URLConnection` class to obtain information from web servers.

**section_5/URLGet.java**

```java
import java.io.InputStream;
import java.io.IOException;
import java.io.OutputStream;
import java.io.PrintWriter;
```
import java.net.HttpURLConnection;
import java.net.URL;
import java.net.URLConnection;
import java.util.Scanner;

/**
   * This program demonstrates how to use a URL connection
   * to communicate with a web server. Supply the URL on
   * the command line, for example
   * java URLGet http://horstmann.com/index.html
   */
public class URLGet {
    public static void main(String[] args) throws IOException {
        // Get command-line arguments
        String urlString;
        if (args.length == 1) {
            urlString = args[0];
        } else {
            urlString = "http://horstmann.com/";
            System.out.println("Using " + urlString);
        }

        // Open connection
        URL u = new URL(urlString);
        URLConnection connection = u.openConnection();

        // Check if response code is HTTP_OK (200)
        HttpURLConnection httpConnection
            = (HttpURLConnection) connection;
        int code = httpConnection.getResponseCode();
        String message = httpConnection.getResponseMessage();
        System.out.println(code + " " + message);
        if (code != HttpURLConnection.HTTP_OK) {
            return;
        }

        // Read server response
        InputStream instream = connection.getInputStream();
        Scanner in = new Scanner(instream);
        while (in.hasNextLine()) {
            String input = in.nextLine();
            System.out.println(input);
        }
    }
}
9. Why is it better to use a `URLConnection` instead of a socket when reading data from a web server?

10. What happens if you use the `URLGet` program to request an image (such as `http://horstmann.com/cay-tiny.gif`)?

**Practice It** Now you can try these exercises at the end of the chapter: P23.5, P23.6, P23.7.

---

**Use High-Level Libraries**

When you communicate with a web server to obtain data, you have two choices. You can make a socket connection and send GET and POST commands to the server over the socket. Or you can use the `URLConnection` class and have it issue the commands on your behalf.

Similarly, to communicate with a mail server, you can write programs that send SMTP and POP commands, or you can learn how to use the Java mail extensions. (See [http://oracle.com/technetwork/java/javamail/index.html](http://oracle.com/technetwork/java/javamail/index.html) for more information on the Java Mail API.)

In such a situation, you may be tempted to use the low-level approach and send commands over a socket connection. It seems simpler than learning a complex set of classes. However, that simplicity is often deceptive. Once you go beyond the simplest cases, the low-level approach usually requires hard work. For example, to send binary e-mail attachments, you may need to master complex data encodings. The high-level libraries have all that knowledge built in, so you don’t have to reinvent the wheel.

For that reason, you should not actually use sockets to connect to web servers. Always use the `URLConnection` class instead. Why did this book teach you about sockets if you aren’t expected to use them? There are two reasons. Some client programs don’t communicate with web or mail servers, and you may need to use sockets when a high-level library is not available. And, just as importantly, knowing what the high-level library does under the hood helps you understand it better. For the same reason, you saw in Chapter 16 how to implement linked lists, even though you probably will never program your own lists and will just use the standard `LinkedList` class.
Describe the IP and TCP protocols.

- The Internet is a worldwide collection of networks, routing equipment, and computers using a common set of protocols to define how each party will interact with each other.
- TCP/IP is the abbreviation for Transmission Control Protocol and Internet Protocol, the pair of communication protocols designed to establish reliable transmission of data between two computers on the Internet.
- A TCP connection requires the Internet addresses and port numbers of both end points.

Describe the HTTP protocol.

- HTTP, or Hypertext Transfer Protocol, is the protocol that defines communication between web browsers and web servers.
- A URL, or Uniform Resource Locator, is a pointer to an information resource (such as a web page or an image) on the World Wide Web.
- The Telnet program is a useful tool for establishing test connections with servers.
- The HTTP GET command requests information from a web server. The web server returns the requested item, which may be a web page, an image, or other data.

Implement programs that use network sockets for reading data.

- A socket is an object that encapsulates a TCP connection. To communicate with the other end point of the connection, use the input and output streams attached to the socket.
- When transmission over a socket is complete, remember to close the socket.
- For text protocols, turn the socket streams into scanners and writers.
- Flush the writer attached to a socket at the end of every command. Then the command is sent to the server, even if the writer’s buffer is not completely filled.

Implement programs that serve data over a network.

- The ServerSocket class is used by server applications to listen for client connections.

Use the URLConnection class to read data from a web server.

- The URLConnection class makes it easy to communicate with a web server without having to issue HTTP commands.
- The URLConnection and HttpURLConnection classes can give you additional information about HTTP requests and responses.
Review Exercises

What is the IP address of the computer that you are using at home? Does it have a domain name?

Can a computer somewhere on the Internet establish a network connection with the computer at your home? If so, what information does the other computer need to establish the connection?

What is a port number? Can the same computer receive data on two different ports?

What is a server? What is a client? How many clients can connect to a server at one time?

What is a socket? What is the difference between a Socket object and a ServerSocket object?

Under what circumstances would an UnknownHostException be thrown?

What happens if the Socket constructor’s second argument is not the same as the port number at which the server waits for connections?

When a socket is created, which of the following Internet addresses is used?

- The address of the computer to which you want to connect
- The address of your computer
- The address of your ISP

What is the purpose of the accept method of the ServerSocket class?

After a socket establishes a connection, which of the following mechanisms will your client program use to read data from the server computer?

- The Socket will fill a buffer with bytes.
- You will use a Reader obtained from the Socket.
- You will use an InputStream obtained from the Socket.

Why is it not common to work directly with the InputStream and OutputStream objects obtained from a Socket object?

When a client program communicates with a server, it sometimes needs to flush the output stream. Explain why.
**R23.13** What is the difference between HTTP and HTML?

**R23.14** Try out the `HEAD` command of the HTTP protocol. What command did you use? What response did you get?

**R23.15** Connect to a POP server that hosts your e-mail and retrieve a message. Provide a record of your session (but remove your password). If your mail server doesn't allow access on port 110, access it through SSL encryption (usually on port 995). Get a copy of the `openssl` utility and use the command

```
openssl s_client -connect servername:995
```

**R23.16** How can you communicate with a web server without using sockets?

**R23.17** What is the difference between a `URL` instance and a `URLConnection` instance?

**R23.18** What is a URL? How do you create an object of class `URL`? How do you connect to a URL?

---

**Practice Exercises**

**E23.1** Modify the `WebGet` program to print only the HTTP header of the returned HTML page. The HTTP header is the beginning of the response data. It consists of several lines, such as

```
HTTP/1.1 200 OK
Date: Tue, 14 Apr 2015 16:10:34 GMT
Server: Apache/1.3.19 (Unix)
Cache-Control: max-age=86400
Expires: Wed, 15 Apr 2015 16:10:34 GMT
Connection: close
Content-Type: text/html
```

followed by a blank line.

**E23.2** Modify the `WebGet` program to print only the title of the returned HTML page. An HTML page has the structure

```
<html><head><title> ... </title></head><body> ... </body></html>
```

For example, if you run the program by typing at the command line

```
java WebGet horstmann.com /
```

the output should be the title of the root web page at `horstmann.com`, such as Cay Horstmann's Home Page.

**E23.3** Modify the `BankServer` program so that it can be terminated more elegantly. Provide another socket on port 8889 through which an administrator can log in. Support the commands `LOGIN password`, `STATUS`, `PASSWORD newPassword`, `LOGOUT`, and `SHUTDOWN`. The `STATUS` command should display the total number of clients that have logged in since the server started.

**E23.4** Modify the `BankServer` program to provide complete error checking. For example, the program should check to make sure that there is enough money in the account when withdrawing. Send appropriate error reports back to the client. Enhance the protocol to be similar to HTTP, in which each server response starts with a number
indicating the success or failure condition, followed by a string with response data or an error description.

**E23.5** Write a program to display the protocol, host, port, and file components of a URL. *Hint:* Look at the API documentation of the `URL` class.

**P23.1** Write a client application that executes an infinite loop that
a. Prompts the user for a number.
b. Sends that value to the server.
c. Receives a number from the server.
d. Displays the new number.

Also write a server that executes an infinite loop whose body accepts a client connection, reads a number from the client, computes its square root, and writes the result to the client.

**P23.2** Implement a client-server program in which the client will print the date and time given by the server. Two classes should be implemented: `DateClient` and `DateServer`. The `DateServer` simply prints `new Date().toString()` whenever it accepts a connection and then closes the socket.

**P23.3** Write a simple web server that recognizes only the `GET` request (without the `Host:` request parameter and blank line). When a client connects to your server and sends a command, such as `GET filename HTTP/1.1`, then return a header

```
HTTP/1.1 200 OK
```

followed by a blank line and all lines in the file. If the file doesn’t exist, return 404 Not Found instead.

Your server should listen to port 8080. Test your web server by starting up your web browser and loading a page, such as `localhost:8080/c:\cs1\myfile.html`.

**P23.4** Write a chat server and client program. The chat server accepts connections from clients. Whenever one of the clients sends a chat message, it is displayed for all other clients to see. Use a protocol with three commands: `LOGIN name`, `CHAT message`, and `LOGOUT`.

**P23.5** A query such as

```
```

returns a page containing the moon phases in a given year. Write a program that asks the user for a year, month, and day and then prints the phase of the moon on that day.

**P23.6** A page such as

```
http://www.mws.noaa.gov/view/states.php
```

contains links to pages showing the weather reports for many cities in the fifty states. Write a program that asks the user for a state and city and then prints the weather report.
A page such as
countrytemplate_ca.html
contains information about a country (here Canada, with the symbol ca—see
the country symbols). Write a program that asks the user for a country name and
then prints the area and population.

1. An IP address is a numerical address, consisting of four or sixteen bytes. A domain name is
an alphanumeric string that is associated with an IP address.

2. TCP is reliable but somewhat slow. When sending sounds or images in real time, it is
acceptable if a small amount of the data is lost. But there is no point in transmitting data that
is late.

3. The browser software translates your requests (typed URLs and mouse clicks on links) into
HTTP commands that it sends to the appropriate web servers.

4. Some Telnet implementations send all keystrokes that you type to the server, including the backspace key. The server does not recognize a character sequence such as G W Backspace E T as a valid command.

5. The program makes a connection to the server, sends the GET request, and prints the error mes-

6. Socket s = new Socket("e-mail.sjsu.edu", 110);

7. Port 80 is the standard port for HTTP. If a web server is running on the same computer, then
one can't open a server socket on an open port.

8. No, a server socket just waits for a connection and yields a regular Socket object when a client
has connected. You use that socket object to read the data that the client sends.

9. The URLConnection class understands the HTTP protocol, freeing you from assembling
requests and analyzing response headers.

10. The bytes that encode the images are displayed on the console, but they will appear to be ran-
don gibberish.
CHAPTER 24

RELATIONAL DATABASES

CHAPTER GOALS

To understand how relational databases store information

To learn to query a database with the Structured Query Language (SQL)

To connect to a database with Java Database Connectivity (JDBC)

To write database programs that insert, update, and query data in a relational database

CHAPTER CONTENTS

24.1 ORGANIZING DATABASE INFORMATION  W1014
  PT1 Stick with the Standard  W1019
  PT2 Avoid Unnecessary Data Replication  W1020
  PT3 Don’t Replicate Columns in a Table  W1020

24.2 QUERIES  W1021
  CE1 Joining Tables Without Specifying a Link Condition  W1027

24.3 INSTALLING A DATABASE  W1028

24.4 DATABASE PROGRAMMING IN JAVA  W1032
  CE2 Constructing Queries from Arbitrary Strings  W1038

24.5 APPLICATION: ENTERING AN INVOICE  W1040
  ST2 Transactions  W1048
  ST3 Object-Relational Mapping  W1049
  WE1 Programming a Bank Database  W1050

© Jason Allen/iStockphoto.
When you store data in a file, you want to be able to add and remove data, change data items, and find items that match certain criteria. However, if you have a lot of data, it can be difficult to carry out these operations quickly and efficiently. Because data storage is such a common task, special *database management systems* have been invented that let the programmer think in terms of the data rather than how it is stored. In this chapter, you will learn how to use SQL, the Structured Query Language, to query and update information in a relational database, and how to access database information from Java programs.

### 24.1 Organizing Database Information

#### 24.1.1 Database Tables

A relational database stores information in *tables*. Figure 1 shows a typical table. As you can see, each *row* in this table corresponds to a product. The *column headers* correspond to attributes of the product: the product code, description, and unit price. Note that all items in a particular column have the same type: product codes and descriptions are strings, unit prices are floating-point numbers. The allowable column types differ somewhat from one database to another. Table 1 shows types that are commonly available in relational databases that follow the SQL (for Structured Query Language; often pronounced “sequel”) standard.

Most relational databases follow the SQL standard. There is no relationship between SQL and Java—they are different languages. However, as you will see later in this chapter, you can use Java to send SQL commands to a database. You will see in the next section how to use SQL commands to carry out queries, but there are other SQL commands.

For example, here is the SQL command to create a table named Product:

```sql
CREATE TABLE Product
(
    Product_Code CHAR(7),
    Description VARCHAR(40),
    Price DECIMAL(10, 2)
)
```

<table>
<thead>
<tr>
<th>Product Code</th>
<th>Description</th>
<th>Price</th>
</tr>
</thead>
<tbody>
<tr>
<td>116-064</td>
<td>Toaster</td>
<td>24.95</td>
</tr>
<tr>
<td>257-535</td>
<td>Hair dryer</td>
<td>29.95</td>
</tr>
<tr>
<td>643-119</td>
<td>Car vacuum</td>
<td>19.99</td>
</tr>
</tbody>
</table>

*Figure 1*  A Product Table in a Relational Database
### Table 1 Some Standard SQL Types and Their Corresponding Java Types

<table>
<thead>
<tr>
<th>SQL Data Type</th>
<th>Java Data Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>INTEGER or INT</td>
<td>int</td>
</tr>
<tr>
<td>REAL</td>
<td>float</td>
</tr>
<tr>
<td>DOUBLE</td>
<td>double</td>
</tr>
<tr>
<td>DECIMAL((m, n))</td>
<td>Fixed-point decimal numbers with (m) total digits and (n) digits after the decimal point; similar to BigDecimal</td>
</tr>
<tr>
<td>BOOLEAN</td>
<td>boolean</td>
</tr>
<tr>
<td>VARCHAR((n))</td>
<td>Variable-length String of length up to (n)</td>
</tr>
<tr>
<td>CHARACTER((n)) or CHAR((n))</td>
<td>Fixed-length String of length (n)</td>
</tr>
</tbody>
</table>

Unlike Java, SQL is not case sensitive. For example, you could spell the command `create table` instead of `CREATE TABLE`. However, as a matter of convention, we will use uppercase letters for SQL keywords and mixed case for table and column names.

To insert rows into the table, use the `INSERT INTO` command. Issue one command for each row, such as:

```sql
INSERT INTO Product
VALUES ('257-535', 'Hair dryer', 29.95)
```

SQL uses single quotes ('), not double quotes, to delimit strings. What if you have a string that contains a single quote? Rather than using an escape sequence (such as `\'`) as in Java, you just write the single quote twice, such as:

`'Sam''s Small Appliances'`

If you create a table and subsequently want to remove it, use the `DROP TABLE` command with the name of the table. For example,

```sql
DROP TABLE Test
```

### 24.1.2 Linking Tables

If you have objects whose instance variables are strings, numbers, dates, or other types that are permissible as table column types, then you can easily store them as rows in a database table. For example, consider a Java class `Customer`:

```java
public class Customer
{
    private String name;
    private String address;
    private String city;
    private String state;
    private String zip;
    . . .
}
```
It is simple to come up with a database table structure that allows you to store customers—see Figure 2.

For other objects, it is not so easy. Consider an invoice. Each invoice object contains a reference to a customer object:

```java
public class Invoice
{
    private int invoiceNumber;
    private Customer theCustomer;
    . . .
}
```

Because `Customer` isn’t a standard SQL type, you might consider simply entering all the customer data into the invoice table—see Figure 3. However, this is not a good idea. If you look at the sample data in Figure 3, you will notice that Sam’s Small Appliances had two invoices, numbers 11731 and 11733. Yet all information for the customer was replicated in two rows.

This replication has two problems. First, it is wasteful to store the same information multiple times. If the same customer places many orders, then the replicated information can take up a lot of space. More importantly, the replication is dangerous. Suppose the customer moves to a new address. Then it would be an easy mistake to update the customer information in some of the invoice records and leave the old address in place in others.

In a Java program, neither of these problems occurs. Multiple `Invoice` objects can contain references to a single shared `Customer` object.

### Table: Customer

<table>
<thead>
<tr>
<th>Name</th>
<th>Address</th>
<th>City</th>
<th>State</th>
<th>Zip</th>
</tr>
</thead>
<tbody>
<tr>
<td>VARCHAR(40)</td>
<td>VARCHAR(40)</td>
<td>VARCHAR(30)</td>
<td>CHAR(2)</td>
<td>CHAR(5)</td>
</tr>
<tr>
<td>Sam’s Small Appliances</td>
<td>100 Main Street</td>
<td>Anytown</td>
<td>CA</td>
<td>98765</td>
</tr>
</tbody>
</table>

**Figure 2  A Customer Table**

### Table: Invoice

<table>
<thead>
<tr>
<th>Invoice_Number</th>
<th>Customer_Name</th>
<th>Customer_Address</th>
<th>Customer_City</th>
<th>Customer_State</th>
<th>Customer_Zip</th>
</tr>
</thead>
<tbody>
<tr>
<td>INTEGER</td>
<td>VARCHAR(40)</td>
<td>VARCHAR(40)</td>
<td>VARCHAR(30)</td>
<td>CHAR(2)</td>
<td>CHAR(5)</td>
</tr>
<tr>
<td>11731</td>
<td>Sam’s Small Appliances</td>
<td>100 Main Street</td>
<td>Anytown</td>
<td>CA</td>
<td>98765</td>
</tr>
<tr>
<td>11732</td>
<td>Electronics Unlimited</td>
<td>1175 Liberty Ave</td>
<td>Pleasantville</td>
<td>MI</td>
<td>45066</td>
</tr>
<tr>
<td>11733</td>
<td>Sam’s Small Appliances</td>
<td>100 Main Street</td>
<td>Anytown</td>
<td>CA</td>
<td>98765</td>
</tr>
</tbody>
</table>

**Figure 3  A Poor Design for an Invoice Table with Replicated Customer Data**
The first step in achieving the same effect in a database is to organize your data into multiple tables as in Figure 4. Dividing the columns into two tables solves the replication problem. The customer data are no longer replicated—the Invoice table contains no customer information, and the Customer table contains a single record for each customer. But how can we refer to the customer to which an invoice is issued? Notice in Figure 4 that there is now a Customer_Number column in both the Customer table and the Invoice table. Now all invoices for Sam’s Small Appliances share only the customer number. The two tables are linked by the Customer_Number field. To find out more details about this customer, you need to use the customer number to look up the customer in the Customer table.

Note that the customer number is a unique identifier. We introduced the customer number because the customer name by itself may not be unique. For example, there may well be multiple Electronics Unlimited stores in various locations. Thus, the customer name alone does not uniquely identify a record (a row of data), so we cannot use the name as a link between the two tables.

In database terminology, a column (or combination of columns) that uniquely identifies a row in a table is called a primary key. In our Customer table, the Customer_Number column is a primary key. You need a primary key if you want to establish a link from another table. For example, the Customer table needs a primary key so that you can link customers to invoices.

When a primary key is linked to another table, the matching column (or combination of columns) in that table is called a foreign key. For example, the Customer_Number in the Invoice table is a foreign key, linked to the primary key in the Customer table. Unlike primary keys, foreign keys need not be unique. For example, in our Invoice table we have several records that have the same value for the Customer_Number foreign key.
24.1.3 Implementing Multi-Valued Relationships

Each invoice is linked to exactly one customer. That is called a single-valued relationship. On the other hand, each invoice has many line items. (As in Chapter 12, a line item identifies the product, quantity, and unit price.) Thus, there is a multi-valued relationship between invoices and line items. In the Java class, the LineItem objects are stored in an array list:

```java
public class Invoice {
    private int invoiceNumber;
    private Customer theCustomer;
    private ArrayList<LineItem> items;
    private double payment;
    ...
}
```

However, in a relational database, you need to store the information in tables. Surprisingly many programmers, when faced with this situation, commit a major faux pas and replicate columns, one for each line item, as in Figure 5 below.

Clearly, this design is not satisfactory. What should we do if there are more than three line items on an invoice? Perhaps we should have 10 line items instead? But that is wasteful if the majority of invoices have only a couple of line items, and it still does not solve our problem for the occasional invoice with lots of line items.

Instead, distribute the information into two tables: one for invoices and another for line items. Link each line item back to its invoice with an Invoice_Number foreign key in the LineItem table—see Figure 6.

<table>
<thead>
<tr>
<th>Invoice_Keyword</th>
<th>Customer_Keyword</th>
<th>Product_Code1</th>
<th>Quantity1</th>
<th>Product_Code2</th>
<th>Quantity2</th>
<th>Product_Code3</th>
<th>Quantity3</th>
<th>Payment</th>
</tr>
</thead>
<tbody>
<tr>
<td>INTEGER</td>
<td>INTEGER</td>
<td>CHAR(7)</td>
<td>INTEGER</td>
<td>CHAR(7)</td>
<td>INTEGER</td>
<td>CHAR(7)</td>
<td>INTEGER</td>
<td>DECIMAL (10, 2)</td>
</tr>
<tr>
<td>11731</td>
<td>3175</td>
<td>116-064</td>
<td>3</td>
<td>257-535</td>
<td>1</td>
<td>643-119</td>
<td>2</td>
<td>0</td>
</tr>
</tbody>
</table>

**Figure 5** A Poor Design for an Invoice Table with Replicated Columns

<p>| LineItem |
|-----------------|---------------|----------|----------|</p>
<table>
<thead>
<tr>
<th>Invoice_Number</th>
<th>Product_Code</th>
<th>Quantity</th>
</tr>
</thead>
<tbody>
<tr>
<td>11731</td>
<td>116-064</td>
<td>3</td>
</tr>
<tr>
<td>11731</td>
<td>257-535</td>
<td>1</td>
</tr>
<tr>
<td>11731</td>
<td>643-119</td>
<td>2</td>
</tr>
</tbody>
</table>

**Figure 6** Linked Invoice and LineItem Tables Implement a Multi-Valued Relationship
In a similar fashion, the LineItem table links to the Product table via the Product table’s Product_Code primary key. Our database now consists of four tables:

- Invoice
- Customer
- LineItem
- Product

Figure 7 shows the links between these tables. In the next section you will see how to query this database for information about invoices, customers, and products. The queries will take advantage of the links between the tables.

1. Would a telephone number be a good primary key for a customer table?
2. In the database of Section 24.1.3, what are all the products that customer 3176 ordered?

**Practice It** Now you can try these exercises at the end of the chapter: R24.2, R24.4, R24.5.

**Stick with the Standard**

The Java language is highly standardized. You will rarely find compilers that allow you to specify Java code that differs from the standard, and if they do, it is always a compiler bug. However, SQL implementations are often much more forgiving. For example, many SQL vendors allow you to use a Java-style escape sequence such as

'Sam\'s Small Appliances'

in a SQL string. The vendor probably thought that this would be “helpful” to programmers who are familiar with Java or C. (The C language uses the same escape mechanism for denoting special characters.)
However, this is an illusion. Deviating from the standard limits portability. Suppose you later want to move your database code to another vendor, perhaps to improve performance or to lower the cost of the database software. If the other vendor hasn’t implemented a particular deviation, then your code will no longer work and you need to spend time fixing it.

To avoid these problems, you should stick with the standard. With SQL, you cannot rely on your database to flag all errors—some of them may be considered “helpful” extensions. That means that you need to know the standard and have the discipline to follow it. (See *A Guide to the SQL Standard: A User’s Guide to the Standard Database Language*, by Chris J. Date and Hugh Darwen (Addison-Wesley, 1996), for more information.)

### Avoid Unnecessary Data Replication

It is very common for beginning database designers to replicate data. When replicating data in a table, ask yourself if you can move the replicated data into a separate table and use a key, such as a code or ID number, to link the tables.

Consider this example in an Invoice table:

```
<table>
<thead>
<tr>
<th>...</th>
<th>Product_Code</th>
<th>Description</th>
<th>Price</th>
</tr>
</thead>
<tbody>
<tr>
<td>...</td>
<td>CHAR(7)</td>
<td>VARCHAR(40)</td>
<td>DECIMAL(10, 2)</td>
</tr>
<tr>
<td>...</td>
<td>116-064</td>
<td>Toaster</td>
<td>24.95</td>
</tr>
<tr>
<td>...</td>
<td>116-064</td>
<td>Toaster</td>
<td>24.95</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>
```

As you can see, some product information is replicated. Is this replication an error? It depends. The product description for the product with code 116-064 is always going to be “Toaster.” Therefore, that correspondence should be stored in an external Product table.

The product price, however, can change over time. When it does, the old invoices don’t automatically use the new price. Thus, it makes sense to store the price that the customer was actually charged in an Invoice table. The current list price, however, is best stored in an external Product table.

### Don’t Replicate Columns in a Table

If you find yourself numbering columns in a table with suffixes 1, 2, and so forth (such as Quantity1, Quantity2, Quantity3), then you are probably on the wrong track. How do you know there are exactly three quantities? In that case, it’s time for another table.

Add a table to hold the information for which you replicated the columns. In that table, add a column that links back to a key in the first table, such as the invoice number in our example. By using an additional table, you can implement a multi-valued relationship.
Primary Keys and Indexes

Recall that a **primary key** is a column (or combination of columns) that uniquely identifies a row in a table. When a table has a primary key, then the database can build an index file: a file that stores information on how to access a row quickly when the primary key is known. Indexing can greatly increase the speed of database queries.

If the primary key is contained in a single column, then you can tag the column with the `PRIMARY KEY` attribute, like this:

```sql
CREATE TABLE Product
(
    Product_Code CHAR(7) PRIMARY KEY,
    Description VARCHAR(40),
    Price DECIMAL(10, 2)
)
```

If the primary key is contained in multiple columns, then add a `PRIMARY KEY` clause to the end of the `CREATE TABLE` command, like this:

```sql
CREATE TABLE LineItem
(
    Invoice_Number INTEGER,
    Product_Code CHAR(7),
    Quantity INTEGER,
    PRIMARY KEY (Invoice_Number, Product_Code)
)
```

Occasionally, one can speed queries up by building **secondary indexes**: index files that index other column sets, which are not necessarily unique. That is an advanced technique that we will not discuss here.

24.2 Queries

Let’s assume that the tables in our database have been created and that records have been inserted. Once a database is filled with data, you will want to **query** the database for information, such as

- What are the names and addresses of all customers?
- What are the names and addresses of all customers in California?
- What are the names and addresses of all customers who bought toasters?
- What are the names and addresses of all customers with unpaid invoices?

In this section you will learn how to formulate simple and complex queries in SQL. We will use the data shown in Figure 8 for our examples.
24.2.1 Simple Queries

In SQL, you use the `SELECT` command to issue queries. For example, the command to select all data from the `Customer` table is

```sql
SELECT * FROM Customer
```

The result is

<table>
<thead>
<tr>
<th>Customer_Number</th>
<th>Name</th>
<th>Address</th>
<th>City</th>
<th>State</th>
<th>Zip</th>
</tr>
</thead>
<tbody>
<tr>
<td>3175</td>
<td>Sam's Small Appliances</td>
<td>100 Main Street</td>
<td>Anytown</td>
<td>CA</td>
<td>98765</td>
</tr>
<tr>
<td>3176</td>
<td>Electronics Unlimited</td>
<td>1175 Liberty Ave</td>
<td>Pleasantville</td>
<td>MI</td>
<td>45066</td>
</tr>
</tbody>
</table>

Use the SQL `SELECT` command to query a database.
The outcome of the query is a *view* — a set of rows and columns that provides a “window” through which you can see some of the database data. If you select all rows and columns from a single table, of course you get a view into just that table.

Many database systems have tools that let you issue interactive SQL commands — Figure 9 shows a typical example. When you issue a `SELECT` command, the tool displays the resulting view. You may want to skip ahead to Section 24.3 and install a database. Or perhaps your computer lab has a database installed already. Then you can run the interactive SQL tool of your database and try out some queries.

### 24.2.2 Selecting Columns

Often, you don’t care about all columns in a table. Suppose your traveling salesperson is planning a trip to all customers. To plan the route, the salesperson wants to know the cities and states of all customers. Here is the query:

```
SELECT City, State FROM Customer
```

The result is

<table>
<thead>
<tr>
<th>City</th>
<th>State</th>
</tr>
</thead>
<tbody>
<tr>
<td>Anytown</td>
<td>CA</td>
</tr>
<tr>
<td>Pleasantville</td>
<td>MI</td>
</tr>
</tbody>
</table>
As you can see, the syntax for selecting columns is straightforward. Simply specify the names of the columns you want, separated by commas.

### 24.2.3 Selecting Subsets

You just saw how you can restrict a view to show selected columns. Sometimes you want to select certain rows that fit a particular criterion. For example, you may want to find all customers in California. Whenever you want to select a subset, you use the `WHERE` clause, followed by the condition that describes the subset. Here is an example:

```sql
SELECT * FROM Customer WHERE State = 'CA'
```

The result is

<table>
<thead>
<tr>
<th>Customer_Number</th>
<th>Name</th>
<th>Address</th>
<th>City</th>
<th>State</th>
<th>Zip</th>
</tr>
</thead>
<tbody>
<tr>
<td>3175</td>
<td>Sam’s Small Appliances</td>
<td>100 Main Street</td>
<td>Anytown</td>
<td>CA</td>
<td>98765</td>
</tr>
</tbody>
</table>

You have to be a bit careful with expressing the condition in the `WHERE` clause, because SQL syntax differs from the Java syntax. As you already know, in SQL you use single quotes to delimit strings, such as ‘CA’. You also use a single `=`, not a double `==`, to test for equality. To test for inequality, you use the `<>` operator. For example

```sql
SELECT * FROM Customer WHERE State <> 'CA'
```

selects all customers that are not in California.

You can match patterns with the `LIKE` operator. The right-hand side must be a string that can contain the special symbols `_` (match exactly one character) and `%` (match any character sequence). For example, the expression

```sql
Name LIKE '_o%'
```

matches all strings whose second character is an “o”. Thus, “Toaster” is a match but “Crowbar” is not.

You can combine expressions with the logical connectives `AND`, `OR`, and `NOT`. (Do not use the Java `&&`, `||`, and `!` operators.) For example

```sql
SELECT * FROM Product WHERE Price < 100 AND Description <> 'Toaster'
```

selects all products with a price less than 100 that are not toasters.

Of course, you can select both row and column subsets, such as

```sql
SELECT Name, City FROM Customer WHERE State = 'CA'
```

### 24.2.4 Calculations

Suppose you want to find out how many customers there are in California. Use the `COUNT` function:

```sql
SELECT COUNT(*) FROM Customer WHERE State = 'CA'
```
In addition to the \texttt{COUNT} function, there are four other functions: \texttt{SUM}, \texttt{AVG} (average), \texttt{MAX}, and \texttt{MIN}.

The \texttt{*} means that you want to calculate entire records. That is appropriate only for the \texttt{COUNT} function. For other functions, you have to access a specific column. Put the column name inside the parentheses:

\begin{verbatim}
SELECT AVG(Price) FROM Product
\end{verbatim}

### 24.2.5 Joins

The queries that you have seen so far all involve a single table. However, the information that you want is usually distributed over multiple tables. For example, suppose you are asked to find all invoices that include a line item for a car vacuum. From the Product table, you can issue a query to find the product code:

\begin{verbatim}
SELECT Product_Code
FROM Product
WHERE Description = 'Car vacuum'
\end{verbatim}

You will find out that the car vacuum has product code 643-119. Then you can issue a second query:

\begin{verbatim}
SELECT Invoice_Number
FROM LineItem
WHERE Product_Code = '643-119'
\end{verbatim}

But it makes sense to combine these two queries so that you don’t have to keep track of the intermediate result. When combining queries, note that the two tables are linked by the Product\_Code field. We want to look at matching rows in both tables. In other words, we want to restrict the search to rows where

\begin{verbatim}
\end{verbatim}

Here, the syntax

\begin{verbatim}
TableName.ColumnName
\end{verbatim}

denotes the column in a particular table. Whenever a query involves multiple tables, you should specify both the table name and the column name. Thus, the combined query is

\begin{verbatim}
SELECT LineItem.Invoice_Number
FROM Product, LineItem
WHERE Product.Description = 'Car vacuum'
\end{verbatim}

The result is

\begin{tabular}{|c|}
\hline
Invoice\_Number \\
11731 \\
11733 \\
\hline
\end{tabular}

In this query, the \texttt{FROM} clause contains the names of multiple tables, separated by commas. (It doesn’t matter in which order you list the tables.) Such a query is often called a \textit{join} because it involves joining multiple tables.
You may want to know in what cities hair dryers are popular. Now you need to add the Customer table to the query—it contains the customer addresses. The customers are referenced by invoices, so you need that table as well. Here is the complete query:

```sql
FROM Product, LineItem, Invoice, Customer
WHERE Product.Description = 'Hair dryer'
  AND LineItem.Invoice_Number = Invoice.Invoice_Number
  AND Invoice.Customer_Number = Customer.Customer_Number
```

The result is

<table>
<thead>
<tr>
<th>City</th>
<th>State</th>
<th>Zip</th>
</tr>
</thead>
<tbody>
<tr>
<td>Anytown</td>
<td>CA</td>
<td>98765</td>
</tr>
</tbody>
</table>

Whenever you formulate a query that involves multiple tables, remember to:

- List all tables that are involved in the query in the `FROM` clause.
- Use the `TableName.ColumnName` syntax to refer to column names.
- List all join conditions (`TableName1.ColumnName1 = TableName2.ColumnName2`) in the `WHERE` clause.

As you can see, these queries can get a bit complex. However, database management systems are very good at answering these queries (see Programming Tip 24.5 on page W1039). One remarkable aspect of SQL is that you describe what you want, not how to find the answer. It is entirely up to the database management system to come up with a plan for how to find the answer to your query in the shortest number of steps.

Commercial database manufacturers take great pride in coming up with clever ways to speed up queries: query optimization strategies, caching of prior results, and so on. In this regard, SQL is a very different language from Java. SQL statements are descriptive and leave it to the database to determine how to execute them. Java statements are prescriptive—you spell out exactly the steps you want your program to carry out.

### 24.2.6 Updating and Deleting Data

Up to now, you have been shown how to formulate increasingly complex `SELECT` queries. The outcome of a `SELECT` query is a result set that you can view and analyze. Two related statement types, `UPDATE` and `DELETE`, don’t produce a result set. Instead, they modify the database. The `DELETE` statement is the easier of the two. It simply deletes the rows that you specify. For example, to delete all customers in California, you issue the statement

```sql
DELETE FROM Customer WHERE State = 'CA'
```

The `UPDATE` query allows you to update columns of all records that fulfill a certain condition. For example, here is how you can add another unit to the quantity of every line item in invoice number 11731:

```sql
UPDATE LineItem
SET Quantity = Quantity + 1
WHERE Invoice_Number = '11731'
```
You can update multiple column values by specifying multiple update expressions in the `SET` clause, separated by commas.

Both the `DELETE` and the `UPDATE` statements return a value, namely the number of rows that are deleted or updated.

3. How do you query the names of all customers that are not from Alaska or Hawaii?
4. How do you query all invoice numbers of all customers in Hawaii?

**Practice It**


---

**Joining Tables Without Specifying a Link Condition**

If you select data from multiple tables without a restriction, the result is somewhat surprising—you get a result set containing *all combinations* of the values, whether or not one of the combinations exists with actual data. For example, the query

```sql
SELECT Invoice.Invoice_Number, Customer.Name
FROM Invoice, Customer
```

returns the result set

<table>
<thead>
<tr>
<th>Invoice.Invoice_Number</th>
<th>Customer.Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>11731</td>
<td>Sam's Small Appliances</td>
</tr>
<tr>
<td>11732</td>
<td>Sam's Small Appliances</td>
</tr>
<tr>
<td>11733</td>
<td>Sam's Small Appliances</td>
</tr>
<tr>
<td>11731</td>
<td>Electronics Unlimited</td>
</tr>
<tr>
<td>11732</td>
<td>Electronics Unlimited</td>
</tr>
<tr>
<td>11733</td>
<td>Electronics Unlimited</td>
</tr>
</tbody>
</table>

As you can see, the result set contains all six combinations of invoice numbers (11731, 11732, 11733) and customer names (Sam's Small Appliances and Electronics Unlimited), even though three of those combinations don’t occur with real invoices. You need to supply a `WHERE` clause to restrict the set of combinations. For example, the query

```sql
SELECT Invoice.Invoice_Number, Customer.Name
FROM Invoice, Customer
WHERE Invoice.Customer_Number = Customer.Customer_Number
```

yields

<table>
<thead>
<tr>
<th>Invoice.Invoice_Number</th>
<th>Customer.Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>11731</td>
<td>Sam’s Small Appliances</td>
</tr>
<tr>
<td>11732</td>
<td>Electronics Unlimited</td>
</tr>
<tr>
<td>11733</td>
<td>Sam’s Small Appliances</td>
</tr>
</tbody>
</table>
24.3 Installing a Database

A wide variety of database systems are available. Among them are

- Production-quality databases, such as Oracle, IBM DB2, Microsoft SQL Server, PostgreSQL, or MySQL.
- Lightweight Java databases, such as Apache Derby.
- Desktop databases, such as Microsoft Access.

Which one should you choose for learning database programming? That depends greatly on your available budget, computing resources, and experience with installing complex software. In a laboratory environment with a trained administrator, it makes a lot of sense to install a production-quality database. Lightweight Java databases are much easier to install and work on a variety of platforms. This makes them a good choice for the beginner. Desktop databases have limited SQL support and can be difficult to configure for Java programming.

In addition to a database, you need a JDBC driver. The acronym JDBC stands for Java Database Connectivity, the name of the technology that enables Java programs to interact with databases. When your Java program issues SQL commands, the driver forwards them to the database and lets your program analyze the results (see Figure 10).

Different databases require different drivers, which may be supplied by either the database manufacturer or a third party. You need to locate and install the driver that matches your database.

If you work in a computing laboratory, someone will have installed a database for you, and you should ask your lab for instructions on how to use it. If you need to provide your own database, we suggest that you choose Apache Derby. It is included with the Java Development Kit. You can also download it separately from http://db.apache.org/derby/.

You should run a test program to check that your database is working correctly. You will find the code for the test program at the end of this section. The following section describes the implementation of the test program in detail.

If you use Apache Derby, then follow these instructions:

1. Locate the JDBC driver file derby.jar and copy it into the ch24/section_3 directory of the companion code for this book.
2. Open a shell window, change to the ch24/section_3 directory, and run
   
   javac TestDB.java
   java -classpath derby.jar:; TestDB database.properties
   
   If you run Linux, UNIX, or Mac OS X, use a colon, not a semicolon, as a path separator:
   
   java -classpath derby.jar:; TestDB database.properties
   
3. If you followed the test instructions precisely, you should see one line of output with the name “Romeo”. You may then skip the remainder of this section.

If you install a database other than the one included with Java, you will need to set aside some time to carry out the installation process. Detailed instructions for installing a database vary widely.
Here we give you a general sequence of steps on how to install a database and test your installation:

1. Install the database program.

2. Start the database. With most database systems (but not some of the lightweight Java database systems), you need to start the database server before you can carry out any database operations. Read the installation instructions for details.

3. Set up user accounts. This typically involves running an administration program, logging in as administrator with a default administration account, and adding user names and passwords. If you are the only user of the database, you may simply be able to use a default account. Again, details vary greatly among databases, and you should consult the documentation.

4. Run a test. Locate the program that allows you to execute interactive SQL instructions. Run the program and issue the following SQL instructions:

   `CREATE TABLE Test (Name VARCHAR(20))`
   `INSERT INTO Test VALUES ('Romeo')`
   `SELECT * FROM Test`
   `DROP TABLE Test`

   At this point, you should get a display that shows a single row and column of the Test database, containing the string “Romeo”. If not, carefully read the documentation of your SQL tool to see how you need to enter SQL statements. For example, with some SQL tools, you need a special terminator for each SQL statement.

Next, locate the JDBC driver and run a sample Java program to verify that the installation was successful. Here are the steps for testing the JDBC driver:

1. Every JDBC driver contains some Java code that your Java programs require to connect to the database. From the JDBC driver documentation, find the `class path` for the driver. Here is a typical example—the class path component for the Apache Derby JDBC driver that is included in the Java Development Kit.

   `c:\jdk1.7.0\db\lib\derby.jar`

   One version of the Oracle database uses a class path

   `/usr/local/oracle/jdbc/classes11g.zip`

   You will find this information in the documentation of your database system.
2. If your JDBC driver is not fully compliant with the JDBC4 standard, you need to know the name of the driver class. For example, the Oracle database uses `oracle.jdbc.driver.OracleDriver`.
Your database documentation will have this information.

3. Find the name of the *database URL* that your driver expects. All database URLs have the format

```
jdbc:subprotocol:driver-specific data
```

The subprotocol is a code that identifies the driver manufacturer, such as `derby` or `oracle`. The driver-specific data encode the database name and the location of the database. Here are typical examples:

```
jdbc:derby:InvoiceDB;create=true
jdbc:oracle:thin:@larry.mathcs.sjsu.edu:1521:InvoiceDB
```

Again, consult your JDBC driver information for details on the format of the database URL and how to specify the database that you use.

4. In order to run the `TestDB.java` program at the end of this section, edit the file `database.properties` and supply

- The driver class name (if required).
- The database URL.
- Your database user name.
- Your database password.

With lightweight Java databases such as Apache Derby, you usually specify a blank user name and password.

5. Compile the program as

```
javac TestDB.java
```

6. Run the program as

```
java -classpath driver_class_path:. TestDB database.properties
```

In UNIX/Linux/Mac OS X, use a `:` separator in the class path:

```
java -classpath driver_class_path:. TestDB database.properties
```

If everything works correctly, you will get an output that lists all data in the Test table. If you followed the test instructions precisely, you will see one line of output with the name “Romeo”.

Here is the test program. We will explain the Java instructions in this program in the following section.

`section_3/TestDB.java`

```java
import java.io.File
import java.sql.Connection;
import java.sql.ResultSet;
import java.sql.Statement;

/**
 * Tests a database installation by creating and querying a sample table. Call this program as
 * java -classpath driver_class_path:. TestDB propertiesFile
 */
```
24.3 Installing a Database

public class TestDB {
    public static void main(String[] args) throws Exception {
        if (args.length == 0) {
            System.out.println("Usage: java -classpath driver_class_path" + File.pathSeparator + ". TestDB propertiesFile");
            return;
        }

        SimpleDataSource.init(args[0]);
        try (Connection conn = SimpleDataSource.getConnection()) {
            Statement stat = conn.createStatement();
            stat.execute("CREATE TABLE Test (Name VARCHAR(20))");
            stat.execute("INSERT INTO Test VALUES ('Romeo')");

            ResultSet result = stat.executeQuery("SELECT * FROM Test");
            result.next();
            System.out.println(result.getString("Name"));

            stat.execute("DROP TABLE Test");
        }
    }
}

section_3/SimpleDataSource.java

import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.SQLException;
import java.io.FileInputStream;
import java.io.IOException;
import java.util.Properties;

/**
 * A simple data source for getting database connections.
 */
public class SimpleDataSource {
    private static String url;
    private static String username;
    private static String password;

    /**
     * Initializes the data source.
     * @param fileName the name of the property file that contains the database driver, URL, username, and password
     */
    public static void init(String fileName) throws IOException, ClassNotFoundException {
        Properties props = new Properties();
        FileInputStream in = new FileInputStream(fileName);
        props.load(in);
    }
}
To connect to a database, you need an object of the `Connection` class. The following shows you how to obtain such a connection. With older versions of the JDBC standard, you first need to manually load the database driver class. Starting with JDBC4 (which is a part of Java 6), the driver is loaded automatically. If you use Java 6 or later and a fully JDBC4 compatible driver, you can skip the loading step. Otherwise, use the following code:

```java
String driver = ...;
Class.forName(driver);  // Load driver
```
Next, you ask the DriverManager for a connection. You need to initialize the url, username, and password strings with the values that apply to your database:

```java
String url = "";
String username = "";
String password = "";
Connection conn = DriverManager.getConnection(url, username, password);
```

When you are done issuing your database commands, the database connection needs to be closed. Use the `try-with-resources` statement:

```java
try (Connection conn = . . .) {
  Work with conn.
}
```

Larger programs (such as the bank example in Worked Example 24.1) need to connect to the database from many classes. You don’t want to propagate the database login information to a large number of classes. Also, it is usually not feasible to use a single connection for all database requests. In particular, as you will see in Chapter 26, a container for web applications can run many simultaneous web page requests from different browsers. Each page request needs its own database connection. But because opening a database connection is quite slow and page requests come so frequently, database connections need to be pooled rather than closed and reopened.

In order to decouple connection management from the other database code, we supply a SimpleDataSource class for this purpose. The implementation is at the end of Section 24.3. This class is a very simple tool for connection management. At the beginning of your program, call the static `init` method with the name of the database configuration file, for example

```java
SimpleDataSource.init("database.properties");
```

The configuration file is a text file that may contain the following lines:

```ini
jdbc.driver= ...
jdbc.url= ...
jdbc.username= ...
jdbc.password= ...
```

The `init` method uses the `Properties` class, which is designed to make it easy to read such a file. The `Properties` class has a `load` method to read a file of key/value pairs from a stream:

```java
Properties props = new Properties();
FileInputStream in = new FileInputStream(fileName);
props.load(in);
```

The `getProperty` method returns the value of a given key:

```java
String driver = props.getProperty("jdbc.driver");
```

You don’t actually have to think about this—the `init` method takes care of the details. Whenever you need a connection, call

```java
try (Connection conn = . . .) {
  Work with conn.
}
```

Real-world connection managers have slightly different methods, but the basic principle is the same.
24.4.2 Executing SQL Statements

Once you have a connection, you can use it to create `Statement` objects. You need `Statement` objects to execute SQL statements.

```java
Statement stat = conn.createStatement();
```

The execute method of the `Statement` class executes a SQL statement. For example,

```java
stat.execute("CREATE TABLE Test (Name CHAR(20))");
stat.execute("INSERT INTO Test VALUES ('Romeo')");
```

To issue a query, use the `executeQuery` method of the `Statement` class. The query result is returned as a `ResultSet` object. For example,

```java
String query = "SELECT * FROM Test";
ResultSet result = stat.executeQuery(query);
```

You will see in the next section how to use the `ResultSet` object to analyze the result of the query.

For `UPDATE` statements, you can use the `executeUpdate` method. It returns the number of rows affected by the statement:

```java
String command = "UPDATE LineItem"
    + " SET Quantity = Quantity + 1"
    + " WHERE Invoice_Number = '11731'";
int count = stat.executeUpdate(command);
```

If your statement has variable parts, then you should use a `PreparedStatement` instead:

```java
String query = "SELECT * WHERE Account_Number = ?";
PreparedStatement stat = conn.prepareStatement(query);
```

The `?` symbols in the query string denote variables that you fill in when you make an actual query. You call one of several `set` methods for that purpose, for example

```java
stat.setString(1, accountNumber);
```

The first parameter of the `set` methods denotes the variable position: 1 is the first `?`, 2 the second, and so on. There are also methods `setInt` and `setDouble` for setting numerical variables. After you set all variables, you call `executeQuery` or `executeUpdate`.

Finally, you can use the generic `execute` method to execute arbitrary SQL statements. It returns a `boolean` value to indicate whether the SQL command yields a result set. If so, you can obtain it with the `getResultSet` method. Otherwise, you can get the update count with the `getUpdateCount` method.

```java
String command = . . . ;
boolean hasResultSet = stat.execute(command);
if (hasResultSet)
{
    ResultSet result = stat.getResultSet();
    . . .
}
else
{
    int count = stat.getUpdateCount();
    . . .
}
```

You can reuse a `Statement` or `PreparedStatement` object to execute as many SQL commands as you like. However, for each statement, you should only have one active
ResultSet. If your program needs to look at several result sets at the same time, then you need to create multiple Statement objects.

When you are done with a Statement object, be sure that it is closed by declaring it in a try-with-resources statement:

```java
try (PreparedStatement stat = conn.prepareStatement(query)) {
    Configure stat.
    ResultSet result = stat.getResultSet();
    Analyze result.
}
```

You do not need to close the result set—it is automatically closed when the statement is closed. However, if you make multiple queries with the same statement, close each result set before making a new query.

When you close a connection, it automatically closes all statements and result sets.

### 24.4.3 Analyzing Query Results

A ResultSet lets you fetch the query result, one row at a time. You iterate through the rows, and for each row, you can inspect the column values. Like the collection iterators that you saw in Chapter 15, the ResultSet class has a `next` method to visit the next row. However, the behavior of the `next` method is somewhat different. The `next` method does not return any data; it returns a boolean value that indicates whether more data are available. Moreover, when you first get a result set from the `executeQuery` method, no row data are available. You need to call `next` to move to the first row. This appears curious, but it makes the iteration loop simple:

```java
while (result.next()) {
    Inspect column data from the current row.
}
```

If the result set is completely empty, then the first call to `result.next()` returns `false`, and the loop is never entered. Otherwise, the first call to `result.next()` fetches the data for the first row from the database. As you can see, the loop ends when the `next` method returns `false`, which indicates that all rows have been fetched.

Once the result set object has fetched a particular row, you can inspect its columns. Various get methods return the column value formatted as a number, string, date, and so on. In fact, for each data type, there are two get methods. One of them has an integer argument that indicates the column position. The other has a string argument for the column name. For example, you can fetch the product code as

```java
String productCode = result.getString(1);
```

or

```java
String productCode = result.getString("Product_Code");
```

Note that the integer index starts at one, not at zero; that is, `getString(1)` inspects the first column. Database column indexes are different from array subscripts.

Accessing a column by an integer index is marginally faster and perfectly acceptable if you explicitly named the desired columns in the `SELECT` statement, such as

```sql
SELECT Invoice_Number FROM Invoice WHERE Payment = 0
```
However, if you make a `SELECT *` query, it is a good idea to use a column name instead of a column index. It makes your code easier to read, and you don’t have to update the code when the column layout changes.

In this example, you saw the `getString` method in action. To fetch a number, use the `getInt` and `getDouble` methods instead, for example:

```java
int quantity = result.getInt("Quantity");
double unitPrice = result.getDouble("Price");
```

### 24.4.4 Result Set Metadata

When you have a result set from an unknown table, you may want to know the names of the columns. You can use the `ResultSetMetaData` class to find out about properties of a result set. Start by requesting the metadata object from the result set:

```java
ResultSetMetaData metaData = result.getMetaData();
```

Then you can get the number of columns with the `getColumnCount` method. The `getColumnLabel` method gives you the column name for each column. Finally, the `getColumnDisplaySize` method returns the column width, which is useful if you want to print table rows and have the columns line up. Note that the indexes for these methods start with 1. For example,

```java
for (int i = 1; i <= metaData.getColumnCount(); i++)
{
    String columnName = metaData.getColumnLabel(i);
    int columnSize = metaData.getColumnDisplaySize(i);
    //...}
```

`ExecSQL.java` is a useful sample program that puts these concepts to work. The program reads a file containing SQL statements and executes them all. When a statement has a result set, the result set is printed, using the result set metadata to determine the column count and column labels.

For example, suppose you have the following file:

**section_4/Product.sql**

```sql
CREATE TABLE Product
(  Product_Code CHAR(7), Description VARCHAR(40), Price DECIMAL(10, 2))
INSERT INTO Product VALUES ('116-064', 'Toaster', 24.95)
INSERT INTO Product VALUES ('257-535', 'Hair dryer', 29.95)
INSERT INTO Product VALUES ('643-119', 'Car vacuum', 19.95)
SELECT * FROM Product
```

Run the `Exec.SQL` program as:

```java
java -classpath derby.jar;. ExecSQL database.properties Product.sql
```

The program executes the statements in the `Product.sql` file and prints out the result of the `SELECT` query.

You can also use the `Exec.SQL` program as an interactive testing tool. Run:

```java
java -classpath derby.jar;. ExecSQL database.properties
```

Then type in SQL commands at the command line. Every time you press the Enter key, the command is executed.
**Executes all SQL statements from a file or the console.**

```java
import java.sql.Connection;
import java.sql.ResultSet;
import java.sql.ResultSetMetaData;
import java.sql.Statement;
import java.sql.SQLException;
import java.io.File;
import java.io.IOException;
import java.util.Scanner;

/**
 * Executes all SQL statements from a file or the console.
 */
public class ExecSQL {
    public static void main(String[] args)
            throws SQLException, IOException, ClassNotFoundException {
        if (args.length == 0) {
            System.out.println(
                    "Usage: java -classpath driver_class_path"
                    + File.pathSeparator
                    + ", ExecSQL propertiesFile [SQLcommandFile]"");
            return;
        }

        SimpleDataSource.init(args[0]);

        Scanner in;
        if (args.length > 1) {
            in = new Scanner(new File(args[1]));
        } else {
            in = new Scanner(System.in);
        }

        try (Connection conn = SimpleDataSource.getConnection(),
                Statement stat = conn.createStatement()) {
            while (in.hasNextLine()) {
                String line = in.nextLine();
                try {
                    boolean hasResultSet = stat.execute(line);
                    if (hasResultSet) {
                        try (ResultSet result = stat.getResultSet()) {
                            showResultSet(result);
                        }
                    } catch (SQLException ex) {
                        System.out.println(ex);
                    }
                } catch (SQLException ex) {
                    System.out.println(ex);
                }
            }
        }
    }
}
```
7. Suppose you want to test whether there are any customers in Hawaii. Issue the statement

```java
ResultSet result = stat.executeQuery("SELECT * FROM Customer WHERE State = 'HI'" restrain)
```

Which Boolean expression answers your question?

8. Suppose you want to know how many customers are in Hawaii. What is an efficient way to get this answer?

**Practice It**

Now you can try these exercises at the end of the chapter: R24.22, E24.3, E24.5.

### Constructing Queries from Arbitrary Strings

Suppose you need to issue the following query with different names.

```sql
SELECT * FROM Customer WHERE Name = customerName
```

Many students try to construct a SELECT statement manually, like this:

```java
String customerName = ...;
String query = "SELECT * FROM Customer WHERE Name = " + customerName + "";
ResultSet result = stat.executeQuery(query);
```
However, this code will fail if the name contains single quotes, such as "Sam's Small Appliances". The query string has a syntax error: a mismatched quote. More serious failures can be introduced by hackers who deliberately enter names or addresses with SQL control characters, changing the meanings of queries. These “SQL injection attacks” have been responsible for many cases of data theft. Never add a string to a query that you didn’t type yourself.

The remedy is to use a PreparedStatement instead:

```java
String query = "SELECT * FROM Customer WHERE Name = ?";
PreparedStatement stat = conn.prepareStatement(query);
stat.setString(1, aName);
ResultSet result = stat.executeQuery(query);
```

The setString method of the PreparedStatement class will properly handle quotes and other special characters in the string.

### Don’t Hardwire Database Connection Parameters into Your Program

It is considered inelegant to hardwire the database parameters into a program:

```java
public class MyProg {
    public static void main(String[] args) {
        // Don't do this:
        String driver = "oracle.jdbc.driver.OracleDriver";
        String url = "jdbc:oracle:thin:@larry.mathcs.sjsu.edu:1521:InvoiceDB";
        String username = "admin";
        String password = "secret";
        . . .
    }
}
```

If you want to change to a different database, you must locate these strings, update them, and recompile.

Instead, place the strings into a separate configuration file (such as `database.properties` in our sample program). The `SimpleDataSource.java` file reads in the configuration file with the database connection parameters. To connect to a different database, you simply supply a different configuration file name on the command line.

### Let the Database Do the Work

You now know how to issue a SQL query from a Java program and iterate through the result set. A common error that students make is to iterate through one table at a time to find a result. For example, suppose you want to find all invoices that contain car vacuums. You could use the following plan:

1. Issue the query `SELECT * FROM Product` and iterate through the result set to find the product code for a car vacuum.
2. Issue the query `SELECT * FROM LineItem` and iterate through the result set to find the line items with that product code.

However, that plan is extremely inefficient. Such a program does in very slow motion what a database has been designed to do quickly.
Instead, you should let the database do all the work. Give the complete query to the
database:

```
SELECT LineItem.Invoice_Number
FROM Product, LineItem
WHERE Product.Description = 'Car vacuum'
```

Then iterate through the result set to read all invoice numbers.

Beginners are often afraid of issuing complex SQL queries. However, you are throwing
away a major benefit of a relational database if you don’t take advantage of SQL.

24.5 Application: Entering an Invoice

In this section, we develop a program for entering an invoice into the database shown
in Figure 8. Here is a sample program run:

Name: Robert Lee
Street address: 833 Lyon Street
City: San Francisco
State: CA
Zip: 94155
Product code (D=Done, L=List): L
116-064 Toaster
257-535 Hair dryer
643-119 Car vacuum
Product code (D=Done, L=List): 116-064
Quantity: 2
Product code (D=Done, L=List): 257-535
Quantity: 3
Product code (D=Done, L=List): D
Robert Lee
833 Lyon Street
San Francisco, CA 94155
2 x 116-064 Toaster
3 x 257-535 Hair dryer

This program puts the concepts of the preceding sections to work.

Before running the program, we assume that the Customer, Product, Invoice, and
LineItem tables have been created. To do so, you can run the ExecSQL program of Sec-
tion 24.4 with the following files (provided with the book’s companion code):

- Customer.sql
- Product.sql
- Invoice.sql
- LineItem.sql

As in the previous programs, we use our SimpleDataSource helper class to get a database
connection. Then we call addInvoice so it can prompt for the invoice information.

```
SimpleDataSource.init(args[0]);
try (Connection conn = SimpleDataSource.getConnection(),
    Scanner in = new Scanner(System.in))
{
    addInvoice(in, conn);
}
```
The `newCustomer` method prompts for the new customer information and adds it to the database. The `nextLine` method is a convenience method for prompting the user and reading a string from a `Scanner`—see the code at the end of this section.

```java
private static int newCustomer(Connection conn, Scanner in)
    throws SQLException
{
    String name = nextLine(in, "Name");
    String address = nextLine(in, "Street address");
    String city = nextLine(in, "City");
    String state = nextLine(in, "State");
    String zip = nextLine(in, "Zip");
    int id = ...;
    try (PreparedStatement stat = conn.prepareStatement(
            "INSERT INTO Customer VALUES (?, ?, ?, ?, ?, ?)"))
    {
        stat.setInt(1, id);
        stat.setString(2, name);
        stat.setString(3, address);
        stat.setString(4, city);
        stat.setString(5, state);
        stat.setString(6, zip);
        stat.executeUpdate();
    }
    return id;
}
```

The method gathers the new customer data and issues an `INSERT INTO` statement to store them in the database.

But how do we provide the ID? We don’t want to ask the program user to come up with IDs. They should be automatically assigned. We will query the largest ID that has been used so far, and use the next larger value as the new ID:

```java
try (Statement stat = conn.createStatement())
{
    ResultSet result = stat.executeQuery(
            "SELECT max(Customer_Number) FROM Customer");
    result.next();
    int id = result.getInt(1) + 1;
}
```

There is just one potential problem. If two users access the database simultaneously, it is possible that both of them create a customer with the same ID at the same time. The remedy is to place the code for adding a customer inside a transaction—see Special Topic 24.2. This is an important requirement in a database with simultaneous users. However, we will skip this step to keep the program simple.

Later, we also need to generate new IDs for invoices. We provide a method `getNewId` that works for both tables.

This completes the customer portion of the invoice entry. We now add a row for the invoice, calling the `getNewId` method to get a new invoice number.

```java
int id = getNewId(conn, "Invoice");
try (PreparedStatement stat = conn.prepareStatement(
        "INSERT INTO Invoice VALUES (?, ?, 0)")
)
{
    stat.setInt(1, id);
    stat.setInt(2, customerNumber);
    stat.executeUpdate();
}
```
Next, the user needs to enter the product codes. When a user provides a code, we will check that it is valid. That is a very simple `SELECT` query. We don’t even look at the result set—if it has a row, we have found the product.

```java
try (PreparedStatement stat = conn.prepareStatement(
    "SELECT * FROM Product WHERE Product_Code = ?")
{
    stat.setString(1, code);
    ResultSet result = stat.executeQuery();
    boolean found = result.next();
}
```

You will find this code in the `findProduct` method.

When the user chooses to see a list of products, we issue a simple query and list the result, as shown here:

```java
try (Statement stat = conn.createStatement())
{
    ResultSet result = stat.executeQuery(
        "SELECT Product_Code, Description FROM Product";
    while (result.next())
    {
        String code = result.getString(1);
        String description = result.getString(2);
        System.out.println(code + " " + description);
    }
}
```

Whenever the user has supplied a product code and a quantity, we add another row to the `LineItem` table. That is yet another `INSERT INTO` statement—you will find it in the `addLineItem` method below.

The following loop keeps asking for product codes and quantities:

```java
boolean done = false;
while (!done)
{
    String productCode = nextLine(in, "Product code (D=Done, L=List)"");
    if (productCode.equals("D")) { done = true; } 
    else if (productCode.equals("L") } { listProducts(conn); } 
    else if (findProduct(conn, productCode))
    {
        int quantity = nextInt(in, "Quantity");
        addLineItem(conn, id, productCode, quantity);
    } else { System.out.println("Invalid product code."); }
}
```

When the loop ends, we print the invoice. Here, the queries are more interesting. The `showInvoice` method has the invoice ID as a parameter. It needs to find the matching customer data by joining the `Invoice` and `Customer` tables:

```java
try (PreparedStatement stat = conn.prepareStatement(
    "SELECT Customer.Name, Customer.Address, "
    + "FROM Customer, Invoice "
    + "WHERE Customer.Customer_Number = Invoice.Customer_Number "
    + "AND Invoice.Invoice_Number = ?")
{
    stat.setInt(1, id);
    
```
The query result contains the customer information which we print. Then we need to get all line items and the product descriptions, again linking two tables in a query:

```java
try (PreparedStatement stat = conn.prepareStatement(
    "SELECT Product.Product_Code, Product.Description, LineItem.Quantity 
    + "FROM Product, LineItem 
    + "AND LineItem.Invoice_Number = ?")
)
{
    stat.setInt(1, id);
    . . .
}
```

Our program simply prints the data in a simple form, as you saw at the beginning of this section; Exercise E24.5 asks you to format it better.

Following is the complete invoice entry program.

**section_5/InvoiceEntry.java**

```java
import java.sql.Connection;
import java.sql.PreparedStatement;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.Statement;
import java.io.IOException;
import java.io.File;
import java.util.Scanner;

/**
 * Enters an invoice into the database.
 * Be sure to add Customer.sql, Product.sql, Invoice.sql, and LineItem.sql to the database before running this program.
 */
public class InvoiceEntry
{
    public static void main(String args[])
    {
        if (args.length == 0)
        {
            System.out.println(
                "Usage: java -classpath driver_class_path"
            + File.pathSeparator
            + ". InvoiceEntry propertiesFile");
            return;
        }
        try
        {
            SimpleDataSource.init(args[0]);
            try (Connection conn = SimpleDataSource.getConnection(),
                Scanner in = new Scanner(System.in))
            {
                addInvoice(in, conn);
            }
        }
        catch (SQLException ex)
        {
            System.out.println("Database error");
            ex.printStackTrace();
        }
    }
}
```
public static void addInvoice(Scanner in, Connection conn)
throws SQLException
{
    int customerNumber = newCustomer(conn, in);
    int id = getNewId(conn, "Invoice");
    try (PreparedStatement stat = conn.prepareStatement(
        "INSERT INTO Invoice VALUES (?, ?, 0)"))
    {
        stat.setInt(1, id);
        stat.setInt(2, customerNumber);
        stat.executeUpdate();
    }

    boolean done = false;
    while (!done)
    {
        String productCode = nextLine(in, "Product code (D=Done, L=List)");
        if (productCode.equals("D")) { done = true; }
        else if (productCode.equals("L")) { listProducts(conn); }
        else if (findProduct(conn, productCode))
        {
            int quantity = nextInt(in, "Quantity");
            addLineItem(conn, id, productCode, quantity);
        }
        else { System.out.println("Invalid product code."); }
    }
    showInvoice(conn, id);
}

public static void addInvoice(Scanner in, Connection conn)
throws SQLException
{
}
try (PreparedStatement stat = conn.prepareStatement(
    "INSERT INTO Customer VALUES (?, ?, ?, ?, ?, ?)")
) {
    stat.setInt(1, id);
    stat.setString(2, name);
    stat.setString(3, address);
    stat.setString(4, city);
    stat.setString(5, state);
    stat.setString(6, zip);
    stat.executeUpdate();
}
return id;
}
/**
Finds a product in the database.
@param conn the database connection
@param code the product code to search
@return true if there is a product with the given code
*/
private static boolean findProduct(Connection conn, String code)
    throws SQLException
{
    try (PreparedStatement stat = conn.prepareStatement(
        "SELECT * FROM Product WHERE Product_Code = ?")
    ) {
        stat.setString(1, code);
        ResultSet result = stat.executeQuery();
        boolean found = result.next();
    }
    return found;
}
/**
Adds a line item to the database.
@param conn the database connection
@param id the invoice ID
@param code the product code
@param quantity the quantity to order
*/
private static void addLineItem(Connection conn, int id, String code, int quantity)
    throws SQLException
{
    try (PreparedStatement stat = conn.prepareStatement(
        "INSERT INTO LineItem VALUES (?, ?, ?)")
    ) {
        stat.setInt(1, id);
        stat.setString(2, code);
        stat.setInt(3, quantity);
        stat.executeUpdate();
    }
}
/**
Lists all products in the database.
@param conn the database connection
*/
private static void listProducts(Connection conn) throws SQLException
{
try (Statement stat = conn.createStatement()) {
    ResultSet result = stat.executeQuery("SELECT Product_Code, Description FROM Product");
    while (result.next()) {
        String code = result.getString(1);
        String description = result.getString(2);
        System.out.println(code + " " + description);
    }
}

/**
   Gets a new ID for a table. This method should be called from inside a transaction that also creates the new row with this ID. The ID field should have name table_Number and type INTEGER.
   @param table  the table name
   @return  a new ID that has not yet been used.
*/
private static int getNewId(Connection conn, String table) throws SQLException {
    try (Statement stat = conn.createStatement()) {
        ResultSet result = stat.executeQuery("SELECT max(" + table + ".Number) FROM " + table);
        result.next();
        int max = result.getInt(1);
        return max + 1;
    }
}

/**
   Shows an invoice.
   @param conn  the database connection
   @param id    the invoice ID
*/
private static void showInvoice(Connection conn, int id) throws SQLException {
    try (PreparedStatement stat = conn.prepareStatement("SELECT Customer.Name, Customer.Address, "
        + "FROM Customer, Invoice "
        + "WHERE Customer.Customer_Number = Invoice.Customer_Number "
        + "AND Invoice.Invoice_Number = ?")
    ) {
        stat.setInt(1, id);
        ResultSet result = stat.executeQuery();
        result.next();
        System.out.println(result.getString(1));
        System.out.println(result.getString(2));
        System.out.println(result.getString(3).trim() + " , "
            + result.getString(4) + " " + result.getString(5));
    }
        + "FROM Product, LineItem "
    )
24.5 Application: Entering an Invoice

W1047

220 + "AND LineItem.Invoice_Number = ?")
221 {
222    stat.setInt(1, id);
223    result = stat.executeQuery();
224    while (result.next())
225    {
226        String code = result.getString(1);
227        String description = result.getString(2).trim();
228        int qty = result.getInt(3);
229        System.out.println(qty + " x " + code + " " + description);
230    }
231 }
232 }
233 }
234 }
235 }

/**
 * Prompts the user and reads a line from a scanner.
 * @param in the scanner
 * @param prompt the prompt
 * @return the string that the user entered
 */
private static String nextLine(Scanner in, String prompt)
{
    System.out.print(prompt + " ");
    return in.nextLine();
}

/**
 * Prompts the user and reads an integer from a scanner.
 * @param in the scanner
 * @param prompt the prompt
 * @return the integer that the user entered
 */
private static int nextInt(Scanner in, String prompt)
{
    System.out.print(prompt + " ");
    int result = in.nextInt();
    in.nextLine(); // Consume newline
    return result;
}

This example completes this introduction to Java database programming. You have seen how you can use SQL to query and update data in a database and how the JDBC library makes it easy for you to issue SQL commands in a Java program.

9. Why do the InvoiceEntry methods throw a SQLException instead of catching it?
10. How could one simplify the first query of the showInvoice method?
11. This program assumes that the customer does not pay at the time of order entry. How can the program be modified to handle payment?
12. This program does not display the amount due. How could we display it?

Practice It
Now you can try these exercises at the end of the chapter: P24.1, P24.2.
Transactions

An important part of database processing is transaction handling. A transaction is a set of database updates that should either succeed in its entirety or not happen at all. For example, consider a banking application that transfers money from one account to another. This operation involves two steps: reducing the balance of one account and increasing the balance of another account. No software system is perfect, and there is always the possibility of an error. The banking application, the database program, or the network connection between them could exhibit an error right after the first part—then the money would be withdrawn from the first account but never deposited to the second account. Clearly, this would be very bad. There are many other similar situations. For example, if you change an airline reservation, you don’t want to give up your old seat until the new one is confirmed.

What all these situations have in common is that there is a set of database operations that are grouped together to carry out the transaction. All operations in the group must be carried out together—a partial completion cannot be tolerated. In SQL, you use the `COMMIT` and `ROLLBACK` commands to manage transactions.

For example, to transfer money from one account to another, you issue the commands:

```sql
UPDATE Account SET Balance = Balance - 1000
WHERE Account_Number = '95667-2574'
UPDATE Account SET Balance = Balance + 1000
WHERE Account_Number = '82041-1196'
COMMIT
```

The `COMMIT` command makes the updates permanent. Conversely, the `ROLLBACK` command undoes all changes up to the last `COMMIT`.

When you program with JDBC, by default the JDBC library automatically commits all database updates. That is convenient for simple programs, but it is not what you want for transaction processing. Thus, you should first turn the autocommit mode off:

```java
Connection conn = . . .;
conn.setAutoCommit(false);
```

Then issue the updates that form the transaction and call the `commit` method of the `Connection` class:

```java
Statement stat = conn.createStatement();
stat.executeUpdate("UPDATE Account SET Balance = Balance - 
  + amount + " WHERE Account_Number = " + fromAccount);
stat.executeUpdate("UPDATE Account SET Balance = Balance + 
  + amount + " WHERE Account_Number = " + toAccount);
conn.commit();
```

Conversely, if you encounter an error, then call the `rollback` method. This typically happens in an exception handler:

```java
try
{
  . . .
} catch (Exception ex)
{
  conn.rollback();
}
```

You may wonder how a database can undo updates when a transaction is rolled back. The database actually stores your changes in a set of temporary tables. If you make queries within a transaction, the information in the temporary tables is merged with the permanent data for the purpose of computing the query result, giving you the illusion that the updates have already
taken place. When you commit the transaction, the temporary data are made permanent. When you execute a rollback, the temporary tables are simply discarded.

### Object-Relational Mapping

Database tables store rows that contain strings, numbers, and other fundamental data types, but not arbitrary objects. In Sections 24.1.2 and 24.1.3, you learned how to translate object references into database relationships. An object-relational mapper automates this process. The Java Enterprise Edition contains such a mapper. You add annotations to the Java classes that describe the relationships. The rules are simple:

- Add `@Entity` to every class that should be stored in the database.
- Each entity class needs an ID that is annotated with `@Id`.
- Relationships between classes are expressed with `@OneToOne`, `@OneToMany`, `@ManyToOne`, and `@ManyToMany`.

Here are the annotations for the invoice classes. A customer can have many invoices, but each invoice has exactly one customer. This is expressed by the `@ManyToOne` annotation. Conversely, each line item is contained in exactly one invoice, but each invoice can have many line items. This is expressed by the `@OneToMany` relationship.

```java
@Entity public class Invoice
{
    @Id private int id;
    @ManyToOne private Customer theCustomer;
    @OneToMany private List<LineItem> items;
    private double payment;
    . . .
}

@Entity public class LineItem
{
    @Id private int id;
    @ManyToOne private Product theProduct;
    private int quantity
    . . .
}

@Entity public class Product
{
    @Id private int id;
    private String description;
    private double price;
    . . .
}

@Entity public class Customer
{
    @Id private int id;
    private String name;
    private String address;
    private String city;
    private String state;
    private String zip;
    . . .
}
```

© Eric Isselé/iStockphoto.
The object-relational mapper processes the annotations and produces a database table layout. You don’t have to worry exactly how the data are stored in the database. For example, to store a new invoice, simply build up the Java object and call

```java
toDoManager.persist(invoice);
```

As a result of this call, the data for the invoice and line items are automatically stored in the various database tables.

To read data from the database, you do not use SQL—after all, you do not know the exact table layout. Instead, you formulate a query in an object-oriented query language. A typical query looks like this:

```sql
SELECT x FROM Invoice x WHERE x.id = 11731
```

The result is a Java object of type `Invoice`. The references to the customer and line item objects have been automatically populated with the proper data from various tables.

Object-relational mapping technology is powerful and convenient. However, you still need to understand the underlying principles of relational databases in order to specify efficient mappings and queries.

**WORKED EXAMPLE 24.1 Programming a Bank Database**

In this Worked Example, we will develop a complete database program. We will reimplement the ATM simulation of Chapter 12, storing the customer and account data in a database. Recall that in the simulation, every customer has a customer number, a PIN, and two bank accounts: a checking account and a savings account.

We’ll store the information in two tables:

| BankCustomer | | |
|--------------|--------------|
| Customer_Number | PIN | Checking_Account_Number | Savings_Account_Number |
| INTEGER | INTEGER | INTEGER | INTEGER |

| Account | | |
|---------|--------------|
| Account_Number | Balance |
| INTEGER | DECIMAL(10, 2) |

The Bank class now needs to connect to the database whenever it is asked to find a customer. The method that finds a customer makes a query

```sql
SELECT * FROM BankCustomer WHERE Customer_Number = . . .
```

It then checks that the PIN matches, and it constructs a `Customer` object. This method also turns the row-and-column information of the database into object-oriented data:

```java
public Customer findCustomer(int customerNumber, int pin)
    throws SQLException
{
    try (Connection conn = SimpleDataSource.getConnection())
    {
        Customer c = null;
        PreparedStatement stat = conn.prepareStatement(
            "SELECT * FROM BankCustomer WHERE Customer_Number = ?");
```

© Tom Horyn/iStockphoto.
stat.setInt(1, customerNumber);

ResultSet result = stat.executeQuery();
if (result.next() & pin == result.getInt("PIN"))
{
    c = new Customer(customerNumber,
        result.getInt("Checking_Account_Number"),
        result.getInt("Savings_Account_Number"));
}
return c;

Note that the method throws a SQLException. Why don’t we catch that exception and return null if an exception occurs? There are many potential reasons for a SQL exception, and the Bank class doesn’t want to hide the exception details. But the Bank class also doesn’t know anything about the user interface of the application, so it can’t display information about the exception to the user. By throwing the exception to the caller, the information can reach the part of the program that interacts with the user.

The BankAccount class in this program is quite different from the implementation you have seen throughout the book. Now we do not store the balance of the bank account in the object; instead, we look it up from the database:

```java
public double getBalance() throws SQLException
{
    try (Connection conn = SimpleDataSource.getConnection())
    {
        double balance = 0
        PreparedStatement stat = conn.prepareStatement(
            "SELECT Balance FROM Account WHERE Account_Number = ?" );
        stat.setInt(1, accountNumber);
        ResultSet result = stat.executeQuery();
        if (result.next())
        {
            balance = result.getDouble(1);
        }
    return balance;
    }
}
```

The deposit and withdraw operations immediately update the database as well:

```java
public void deposit(double amount)
    throws SQLException
{
    try (Connection conn = SimpleDataSource.getConnection())
    {
        PreparedStatement stat = conn.prepareStatement(
            "UPDATE Account"
            + " SET Balance = Balance + ?"
            + " WHERE Account_Number = ?");
        stat.setDouble(1, amount);
        stat.setInt(2, accountNumber);
        stat.executeUpdate();
    }
}
```

It seems somewhat inefficient to connect to the database whenever the bank balance is accessed, but it is much safer than storing it in an object. Suppose you have two instances of the ATM program running at the same time. Then it is possible that both programs modify the same bank account. If each of them copied the bank balances from the database into objects, then the modifications made by one user would not be seen by the other.
You can try out this simultaneous access yourself, simply by running two instances of the ATM simulation. Alternatively, you can modify the main method of the ATMViewer class to pop up two ATM frames.

The source code for the modified ATM application follows. The source code for the ATM and ATMSimulator/ATMFrame classes is only changed minimally, by adding code to deal with the SQLException. The Customer class is unchanged. We do not list those classes, but you will find them in the ch24/worked_example_1 folder of this chapter’s companion code.

**worked_example_1/Bank.java**

```java
import java.sql.Connection;
import java.sql.ResultSet;
import java.sql.PreparedStatement;
import java.sql.SQLException;

/**
 * A bank consisting of multiple bank accounts.
 */
public class Bank {
    /**
     * Finds a customer with a given number and PIN.
     * @param customerNumber the customer number
     * @param pin the personal identification number
     * @return the matching customer, or null if none found
     */
    public Customer findCustomer(int customerNumber, int pin)
            throws SQLException {
            try (Connection conn = SimpleDataSource.getConnection()) {
                Customer c = null;
                PreparedStatement stat = conn.prepareStatement(
                        "SELECT * FROM BankCustomer WHERE Customer_Number = ?");
                stat.setInt(1, customerNumber);
                ResultSet result = stat.executeQuery();
                if (result.next() && pin == result.getInt("PIN")) {
                    c = new Customer(customerNumber,
                            result.getInt("Checking_Account_Number"),
                            result.getInt("Savings_Account_Number"));
                }
                return c;
            }
    }
}
```

**worked_example_1/BankAccount.java**

```java
import java.sql.Connection;
import java.sql.ResultSet;
import java.sql.PreparedStatement;
import java.sql.SQLException;

/**
 * A bank account has a balance that can be changed by deposits and withdrawals.
 */
```
public class BankAccount {
    private int accountNumber;

    /**
     * Constructs a bank account with a given balance.
     * @param anAccountNumber the account number
     */
    public BankAccount(int anAccountNumber) {
        accountNumber = anAccountNumber;
    }

    /**
     * Deposits money into a bank account.
     * @param amount the amount to deposit
     */
    public void deposit(double amount) throws SQLException {
        try (Connection conn = SimpleDataSource.getConnection()) {
            PreparedStatement stat = conn.prepareStatement("UPDATE Account
                + " SET Balance = Balance + ?
                + " WHERE Account_Number = ?");
            stat.setDouble(1, amount);
            stat.setInt(2, accountNumber);
            stat.executeUpdate();
        }
    }

    /**
     * Withdraws money from a bank account.
     * @param amount the amount to withdraw
     */
    public void withdraw(double amount) throws SQLException {
        try (Connection conn = SimpleDataSource.getConnection()) {
            PreparedStatement stat = conn.prepareStatement("UPDATE Account
                + " SET Balance = Balance - ?
                + " WHERE Account_Number = ?");
            stat.setDouble(1, amount);
            stat.setInt(2, accountNumber);
            stat.executeUpdate();
        }
    }

    /**
     * Gets the balance of a bank account.
     * @return the account balance
     */
    public double getBalance() throws SQLException {
        try (Connection conn = SimpleDataSource.getConnection()) {
            // Code for getting balance
        }
    }
}
Develop strategies for storing data in a database.

- A relational database stores information in tables. Each table column has a name and a data type.
- SQL (Structured Query Language) is a command language for interacting with a database.
- Use the SQL commands `CREATE TABLE` and `INSERT INTO` to add data to a database.
- You should avoid rows with replicated data. Instead, distribute the data over multiple tables.
- A primary key is a column (or set of columns) whose value uniquely specifies a table record.
- A foreign key is a reference to a primary key in a linked table.
- Implement one-to-many relationships with linked tables, not replicated columns.

Use SQL to query and update a database.

- Use the SQL `SELECT` command to query a database.
- The `WHERE` clause selects data that fulfill a condition.
- A join is a query that involves multiple tables.
- The `UPDATE` and `DELETE` SQL commands modify the data in a database.

Install a database system and test that you can connect to it from a Java program.

- You need a JDBC (Java Database Connectivity) driver to access a database from a Java program.
- Make sure the JDBC driver is on the class path when you launch the Java program.
- To connect to the database, you need to specify a database URL, user name, and password.

Write Java programs that access and update database records.

- Use a `Connection` object to access a database from a Java program.
- A `Connection` object can create `Statement` objects that are used to execute SQL commands.
The result of a SQL query is returned in a ResultSet object.
Metadata are data about an object. Result set metadata describe the properties of a result set.

**Develop programs that use the JDBC library for accessing a database.**

### Standard Library Items Introduced in This Chapter

<table>
<thead>
<tr>
<th>Java Class/Library Item</th>
<th>Java Class/Library Item</th>
<th>Java Class/Library Item</th>
</tr>
</thead>
<tbody>
<tr>
<td>java.io.File</td>
<td>java.sql.PreparedStatement</td>
<td>java.sql.ResultSet</td>
</tr>
<tr>
<td>pathSeparator</td>
<td>execute</td>
<td>execute</td>
</tr>
<tr>
<td>java.lang.Class</td>
<td>executeQuery</td>
<td>executeUpdate</td>
</tr>
<tr>
<td>forName</td>
<td>setDouble</td>
<td>setInt</td>
</tr>
<tr>
<td>java.sql.Connection</td>
<td>setString</td>
<td>close</td>
</tr>
<tr>
<td>close</td>
<td>java.sql.ResultSet</td>
<td>java.sql.ResultSetMeta</td>
</tr>
<tr>
<td>commit</td>
<td>close</td>
<td>getColumnCount</td>
</tr>
<tr>
<td>createStatement</td>
<td>getDouble</td>
<td>getColumnDisplaySize</td>
</tr>
<tr>
<td>prepareStatement</td>
<td>getInt</td>
<td>getColumnLabel</td>
</tr>
<tr>
<td>rollback</td>
<td>java.sql.ResultSetMeta</td>
<td>java.sql.SQLException</td>
</tr>
<tr>
<td>setAutoCommit</td>
<td>getMetaData</td>
<td>java.sql.Statement</td>
</tr>
<tr>
<td>java.sql.DriverManager</td>
<td>getString</td>
<td>close</td>
</tr>
<tr>
<td>getConnection</td>
<td>next</td>
<td>execute</td>
</tr>
<tr>
<td>java.sql.PreparedStatement</td>
<td>executeQuery</td>
<td>executeUpdate</td>
</tr>
<tr>
<td>java.sql.ResultSet</td>
<td>getResultSet</td>
<td>getResultSetCount</td>
</tr>
<tr>
<td>java.sql.ResultSetMeta</td>
<td>java.util.Properties</td>
<td>getUpdateCount</td>
</tr>
<tr>
<td>java.sql.SQLException</td>
<td>getProperty</td>
<td>load</td>
</tr>
<tr>
<td>java.sql.Statement</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Review Exercises

- **R24.1** Design a set of database tables to store people and cars. A person has a name, a unique driver license number, and an address. Every car has a unique vehicle identification number, manufacturer, type, and year. Every car has one owner, but one person can own multiple cars.

- **R24.2** Design a set of database tables to store library books and patrons. A book has an ISBN (International Standard Book Number), an author, and a title. The library may have multiple copies of each book, each with a different book ID. A patron has a name, a unique ID, and an address. A book may be checked out by at most one patron, but one patron can check out multiple books.

- **R24.3** Design a set of database tables to store sets of coins in purses. Each purse has an owner name and a unique ID. Each coin type has a unique name and a value. Each purse contains some quantity of coins of a given type.

- **R24.4** Design a set of database tables to store students, classes, professors, and classrooms. Each student takes zero or more classes. Each class has one professor, but a professor can teach multiple classes. Each class has one classroom.

- **R24.5** Give SQL commands to create a Book table, with columns for the ISBN, author, and title, and to insert all textbooks that you are using this semester.

- **R24.6** Give SQL commands to create a Car table, with columns for the vehicle identification number, manufacturer, model, and year of each car, and to insert all cars that your family members own.
Exercises R24.7–R24.17 refer to the invoice database of Section 24.2 on page W1022.

- **R24.7** Give a SQL query that lists all products in the invoice database of Section 24.2.
- **R24.8** Give a SQL query that lists all customers in California.
- **R24.9** Give a SQL query that lists all customers in California or Nevada.
- **R24.10** Give a SQL query that lists all customers not in Hawaii.
- **R24.11** Give a SQL query that lists all customers who have an unpaid invoice.
- **R24.12** Give a SQL query that lists all products that have been purchased by a customer in California.
- **R24.13** Give a SQL query that lists all line items that are part of invoice number 11731.
- **R24.14** Give a SQL query that computes the sum of all quantities that are part of invoice number 11731.
- **R24.15** Give a SQL query that computes the total cost of all line items in invoice number 11731.
- **R24.16** Give a SQL update statement that raises all prices by ten percent.
- **R24.17** Give a SQL statement that deletes all customers in California.

- **R24.18** Pick a database system (such as DB2, Oracle, Postgres, or SQL Server) and determine from the web documentation:
  - What JDBC driver do you need? Is it automatically discovered?
  - What is the database URL?

- **R24.19** Where is the file derby.jar located in your installation of the Java development kit?

- **R24.20** Suppose you run the command
  ```
  java -classpath derby.jar;. TestDB database.properties
  ```
as described in Section 24.3. Match the following five error messages to the error conditions they correspond to.

<table>
<thead>
<tr>
<th>Error Message</th>
<th>Error Condition</th>
</tr>
</thead>
<tbody>
<tr>
<td>Usage: java [-options] class [args...]</td>
<td>The suffix ; create=true was missing from the jdbc.url entry in database.properties.</td>
</tr>
<tr>
<td>Exception in thread “main” java.sql.SQLException: No suitable driver found for jdbc:derby:BigJavaDB;create=true</td>
<td>The database.properties file was not present in the current directory.</td>
</tr>
<tr>
<td>Exception in thread “main” java.lang.NoClassDefFoundError: TestDB</td>
<td>derby.jar and . should have been separated by a colon on this operating system.</td>
</tr>
<tr>
<td>Exception in thread “main” java.io.FileNotFoundException: database.properties</td>
<td>The TestDB.class file was not present in the current directory.</td>
</tr>
<tr>
<td>Exception in thread “main” java.sql.SQLException: Database 'BigJavaDB' not found.</td>
<td>The derby.jar file was not present in the current directory.</td>
</tr>
</tbody>
</table>
What is the difference between a Connection and a Statement?

Of the SQL commands introduced in this chapter, which yield result sets, which yield an update count, and which yield neither?

How is a ResultSet different from an Iterator?

- E24.1 Write a Java program that creates a Coin table with coin names and values; inserts coin types penny, nickel, dime, quarter, half dollar, and dollar; and prints out the sum of the coin values. Use SQL commands CREATE TABLE, INSERT, and SELECT SUM.

- E24.2 Write a Java program that creates a Car table with car manufacturers, models, model years, and fuel efficiency ratings. Insert several cars. Print out the average fuel efficiency. Use SQL commands CREATE TABLE, INSERT, and SELECT AVG.

- E24.3 Improve the ExecSQL program and make the columns of the output line up. Hint: Use the getColumnDisplaySize method of the ResultSetMetaData class.

- E24.4 Modify the program in Section 24.5 so that the user has the choice of selecting an existing customer. Provide an option to search for the customer by name or zip code.

- E24.5 Write a Java program that uses the database tables from the invoice database in Section 24.2. Prompt the user for an invoice number and print out the invoice, formatted as in Chapter 12.

- E24.6 Write a Java program that uses the database tables from the invoice database in Section 24.2. Produce a report that lists all customers, their invoices, the amounts paid, and the unpaid balances.

- P24.1 Write a Java program that uses a library database of books and patron data, as described in Exercise R24.2. Patrons should be able to check out and return books. Supply commands to print the books that a patron has checked out and to find who has checked out a particular book. Create and populate Patron and Book tables before running the program.

- P24.2 Write a Java program that creates a grade book for a class. Create and populate Student and Grade tables before running the program. The program should be able to display all grades for a given student. It should allow the instructor to add a new grade (such as “Homework 4: 100”) or modify an existing grade.

- P24.3 Write a program that assigns seats on an airplane as described in Exercise P12.6. Keep the seating information in a database.

- P24.4 Write a program that keeps an appointment calendar in a database. An appointment includes a description, a date, the starting time, and the ending time; for example,

  Dentist 2016/10/3 17:30 18:30
  CS1 class 2016/10/4 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.
P24.5 Modify the ATM simulation program of Worked Example 24.1 so that the program pops up two ATM frames. Verify that the database can be accessed simultaneously by two users.

P24.6 Write a program that uses a database of quizzes. Each quiz has a description and one or more multiple-choice questions. Each question has one or more choices, one of which is the correct one. Use tables Quiz, Question, and Choice. The program should show the descriptions of all quizzes, allow the user to choose one, and present all questions in the chosen quiz. When the user has provided responses to all questions, show the score.

P24.7 Enhance the program of Exercise P24.6 so that it stores the user’s responses in the database. Add Student and Response tables. (User is a reserved word in SQL.)

P24.8 Write a program that uses the database of Exercise P24.7 and prints a report showing how all students performed on all quizzes.

ANSWERS TO SELF-CHECK QUESTIONS

1. The telephone number for each customer may not be unique—the same number might be shared by roommates. Even if the number were unique, however, it can change when a customer moves. In that situation, both the primary and all foreign keys would need to be updated. Therefore, a customer ID is a better choice.

2. Customer 3176 ordered ten toasters.

3. SELECT Name
   FROM Customer
   WHERE State <> 'AK' AND State <> 'HI'

4. SELECT Invoice.Invoice_Number
   FROM Invoice, Customer
   WHERE Invoice.Customer_Number = Customer.Customer_Number
   AND Customer.State = 'HI'

5. Connect to the database with a program that lets you execute SQL instructions. Try creating a small database table, adding a record, and selecting all records. Then drop the table again.

6. You didn’t set the class path correctly. The JAR file containing the JDBC driver must be on the class path.

7. result.next(). If there is at least one result, then next returns true.

8. ResultSet result = stat.executeQuery(
      "SELECT COUNT(*) FROM Customer "
      + "WHERE State = 'HI'";
    result.next();
    int count = result.getInt(1);

Note that the following alternative is significantly slower if there are many such customers.

9. In this program, error reporting is the responsibility of the main method.

10. By passing the customer number as an argument. Then the query would only involve the customer table.

11. The most convenient approach is to ask the user about the payment before entering the items. Then the payment can be added to the statement in the addInvoice method.

12. We can either compute it in the loop that displays the line items, or we can issue a query

   SELECT SUM(Product.Price * LineItem.Quantity)
   FROM Product, LineItem
     AND LineItem.Invoice_Number = ?
CHAPTER 25

XML

CHAPTER GOALS

To learn to use XML elements and attributes
To understand the concept of an XML parser
To read and write XML documents
To design Document Type Definitions for XML documents

CHAPTER CONTENTS

25.1 XML TAGS AND DOCUMENTS W1060

HT1 Designing an XML Document Format W1064
PT1 Prefer XML Elements over Attributes W1065
PT2 Avoid Children with Mixed Elements and Text W1066

25.2 PARSING XML DOCUMENTS W1067

CE1 XML Elements Describe Objects, Not Classes W1071

25.3 CREATING XML DOCUMENTS W1072

HT2 Writing an XML Document W1077
ST1 Grammars, Parsers, and Compilers W1079

25.4 VALIDATING XML DOCUMENTS W1081

HT3 Writing a DTD W1088
ST2 Schema Languages W1089
ST3 Other XML Technologies W1090
The Extensible Markup Language (XML) is a popular mechanism for encoding data. Independent of any programming language, XML allows you to encode complex data in a form that the recipient can easily parse. It is simple enough that a wide variety of programs can generate XML data. XML data has a nested structure, so you can use it to describe hierarchical data sets—for example, an invoice that contains many items, each of which consists of a product and a quantity. Because the XML format is standardized, libraries for parsing the data are widely available and—as you will see in this chapter—easy to use for a programmer.

### 25.1 XML Tags and Documents

The XML format uses a mixture of text and tags to describe data. Tags are enclosed in angle brackets `<...>`. An element is a unit of information that is delimited by a start-tag and a matching end-tag. An element can contain text and other elements. For example, `<city>Sunnyvale</city>` is an element with a text child, and

```
<address>
    <street>1195 W. Fairfield Rd.</street>
    <city>Sunnyvale</city>
    <state>CA</state>
</address>
```

is an element with three child elements. In the following sections, you will see why XML is more useful than a plain text format, how it is related to HTML, and which rules you need to follow when producing an XML document.

#### 25.1.1 Advantages of XML

To understand the advantages of using XML for encoding data, let’s look at a typical example. We will encode product descriptions, so that they can be transferred to another computer. Your first attempt might be a naïve encoding like this:

```
Toaster
29.95
```

In contrast, here is an XML encoding of the same data:

```
<product>
    <description>Toaster</description>
    <price>29.95</price>
</product>
```

The advantage of the XML version is clear: You can look at the data and understand what they mean. Of course, this is a benefit for the programmer, not for a computer program. A computer program has no understanding of what a “price” is. As a programmer, you still need to write code to extract the price as the content of the price element. Nevertheless, the fact that an XML document is comprehensible by humans is a huge advantage for program development.
A second advantage of the XML version is that it is resilient to change. Suppose the product data change, and an additional data item is introduced to denote the manufacturer. In the naïve format, the manufacturer might be added after the price, like this:

<table>
<thead>
<tr>
<th>Toaster</th>
<th>29.95</th>
</tr>
</thead>
<tbody>
<tr>
<td>General Appliances</td>
<td></td>
</tr>
</tbody>
</table>

A program that can process the old format might get confused when reading a sequence of products in the new format. The program would think that the price is followed by the name of the next product. Thus, the program needs to be updated to work with both the old and new data formats. As data get more complex, programming for multiple versions of a data format can be difficult and time-consuming.

When using XML, on the other hand, it is easy to add new elements:

```xml
<product>
  <description>Toaster</description>
  <price>29.95</price>
  <manufacturer>General Appliances</manufacturer>
</product>
```

Now a program that processes the new data can still extract the old information in the same way—as the contents of the `description` and `price` elements. The program need not be updated, and it can tolerate different versions of the data format.

### 25.1.2 Differences Between XML and HTML

If you know HTML, you may have noticed that the XML format of the product data looked somewhat like HTML code. However, there are some differences that we will discuss in this section.

Let's start with the similarities. The XML tag pairs, such as `<price>` and `</price>` look just like HTML tag pairs, for example `<li>` and `</li>`. Both in XML and in HTML, tags are enclosed in angle brackets `< >`, and a start-tag is paired with an end-tag that starts with a slash `/` character.

However, web browsers are quite permissive about HTML. For example, you can omit an end-tag `</li>` and the browser will try to figure out what you mean. In XML, this is not permissible. When writing XML, pay attention to the following rules:

- You must pay attention to the letter case of the tags; for example, in XML `<li>` and `<LI>` are different tags that bear no relation to each other.
- Every start-tag must have a matching end-tag. You cannot omit tags, such as `</li>`. A tag that ends in `/>` is both a start- and end-tag:

```xml
<img src="hamster.jpeg"/>
```

When the parser sees the `/>`, it knows not to look for a matching end-tag.

- Finally, attribute values must be enclosed in quotes. For example,

```xml
<img src="hamster.jpeg" width=400 height=300/>
```

is not acceptable. You must use

```xml
<img src="hamster.jpeg" width="400" height="300"/>
```
Moreover, there is an important conceptual difference between HTML and XML. HTML has one specific purpose: to describe web documents. In contrast, XML is an extensible syntax that can be used to specify many different kinds of data. For example, the VRML language uses the XML syntax to describe virtual reality scenes. The MathML language uses the XML syntax to describe mathematical formulas. You can use the XML syntax to describe your own data, such as product records or invoices.

Most people who first see XML wonder how an XML document looks inside a browser. However, that is not generally a useful question to ask. Most data that are encoded in XML have nothing to do with browsers. For example, it would probably not be exciting to display an XML document with nothing but product records (such as the ones in the previous section) in a browser. Instead, you will learn in this chapter how to write programs that analyze XML data. XML does not tell you how to display data; it is merely a convenient format for representing data.

25.1.3 The Structure of an XML Document

In this section, you will see the rules for properly formatted XML. In XML, text and tags are combined into a document. The XML standard recommends that every XML document start with a declaration

```xml
<?xml version="1.0"?>
```

Next, the XML document contains the actual data. The data are contained in a root element. For example,

```xml
<?xml version="1.0"?>
<invoice>
  more data
</invoice>
```

The `invoice` root element is an example of an XML element. An element has one of two forms:

```xml
<elementName> content </elementName>
```

or

```xml
<elementName/>
```

In the first case, the element has content—elements, text, or a mixture of both. A good example is a paragraph in an HTML document:

```html
<p>Use XML for <strong>robust</strong> data formats.</p>
```

The `p` element contains

1. The text: “Use XML for ”
2. A `strong` child element

For XML files that contain documents in the traditional sense of the term, the mixture of text and elements is useful. The XML specification calls this type of content mixed content. But for files that describe data sets—such as our product data—it is better to stick with elements that contain either other elements or text. Content that consists only of elements is called element content.
An element can have *attributes*. For example, the `a` element of HTML has an `href` attribute that specifies the URL of a hyperlink:

```html
<a href="http://horstmann.com"> . . . </a>
```

An attribute has a name (such as `href`) and a value. In XML, the value must be enclosed in single or double quotes.

An element can have multiple attributes, for example

```html
<img src="hamster.jpeg" width="400" height="300"/>
```

And, as you have already seen, an element can have both attributes and content.

```html
<a href="http://horstmann.com">Cay Horstmann's web site</a>
```

Programmers often wonder whether it is better to use attributes or child elements. For example, should a product be described as

```html
<product description="Toaster" price="29.95"/>
```

or

```html
<product>
  <description>Toaster</description>
  <price>29.95</price>
</product>
```

The former is shorter. However, it violates the spirit of attributes. Attributes are intended to provide information *about* the element content. For example, the `price` element might have an attribute `currency` that helps interpret the element content. The content 29.95 has a different interpretation in the element

```html
<price currency="USD">29.95</price>
```

than it does in the element

```html
<price currency="EUR">29.95</price>
```

You have now seen the components of an XML document that are needed to use XML for encoding data. There are other XML constructs for more specialized situations—see [http://www.xml.com/axml/axml.html](http://www.xml.com/axml/axml.html) for more information. In the next section, you will see how to use Java to parse XML documents.

**SELF CHECK**

1. Write XML code with a `student` element and child elements `name` and `id` that describe you.

2. What does your browser do when you load an XML file, such as the `section_2/items.xml` file that is contained in the companion code for this book?

3. Why does HTML use the `src` attribute to specify the source of an image instead of `<img>hamster.jpeg</img>`?

**Practice It**

Now you can try these exercises at the end of the chapter: R25.1, R25.2, R25.3.
Designing an XML Document Format

This How To walks you through the process of designing an XML document format. You will see in Section 25.4 how to formally describe the format with a document type definition. Right now, we focus on an informal definition of the document content. The “output” of this activity is a sample document.

**Step 1** Gather the data that you must include in the XML document. Write them on a sheet of paper. If at all possible, work from some real-life examples. For example, suppose you need to design an XML document for an invoice. A typical invoice has

- An invoice number
- A shipping address
- A billing address
- A list of items ordered

If possible, gather some actual invoices. Decide which features of the actual invoices you need to include in your XML document.

**Step 2** Analyze which data elements need to be refined. Continue refinement until you reach data values that can be described by single strings or numbers. Make a note of all data items that you discovered during the refinement process. When done, you should have a list of data elements, some of which can be broken down further and some of which are simple enough to be described by a single string or number.

For example, the “shipping address” actually contains the customer name, street, city, state, and ZIP code.

The “list of items ordered” contains items. Each item contains a product and the quantity ordered. Each product contains the product name and price.

Thus, our list now contains

- Address
- Name
- Street
- City
- State
- ZIP code
- List of items ordered
- Product
- Description
- Price
- Quantity

Keep breaking the data items down until each of them can be described by a single string or number. For example, an address cannot be described by a single string, but a city can be described by a single string.

**Step 3** Come up with a suitable element name that describes the entire XML document. This element becomes the root element. For example, the invoice data would be contained in an element named invoice.

**Step 4** Come up with suitable element names for the top-level decomposition that you found in Step 1. These become the children of the root element. For example, the invoice element has children

- Address
- Items

**Step 5** Repeat this process to give names to the other elements that you discovered in Step 2.

As you do this, make a comprehensive example that shows all elements at work. For the invoice problem, here is an example:
Step 6  Check that the document doesn’t have mixed content.
That is, make sure each element has as its children either additional elements or text, but not both. If necessary, add more child elements to wrap any text.
For example, suppose the product element looked like this:

```
<product>
  <description>Ink Jet Refill Kit</description>
  29.95
</product>
```
Perhaps someone thought it was “obvious” that the last entry was the price. However, following Programming Tip 25.2, it is best to wrap the price inside a price element, like this:

```
<product>
  <description>Ink Jet Refill Kit</description>
  <price>29.95</price>
</product>
```

Prefer XML Elements over Attributes
Attributes are shorter than elements. For example,

```
<product description="Toaster" price="29.95"/>
```
seems simpler than

```
<product>
  <description>Toaster</description>
  <price>29.95</price>
</product>
```
There is the temptation to use attributes because they are “easier to type”. But of course, you don’t type XML documents, except for testing purposes. In real-world situations, XML documents are generated by programs.
Attributes are less flexible than elements. Suppose we want to add a currency indication to the value. With elements, that’s easy to do:

```xml
<price currency="USD">29.95</price>
```
or even

```xml
<price>
  <currency>USD</currency>
  <amount>29.95</amount>
</price>
```

With attributes, you are stuck—you can’t refine the structure. Of course, you could use

```xml
<product description="Toaster" price="USD 29.95"/>
```

But then your program has to parse the string USD 29.95 and manually take it apart. That’s just the kind of tedious and error-prone coding that XML is designed to avoid.

In HTML, there is a simple rule when using attributes. All strings that are not part of the displayed text are attributes. For example, consider a link.

```html
<a href="http://horstmann.com">Cay Horstmann's web site</a>
```
The text inside the a element, Cay Horstmann’s web site, is part of what the user sees on the web page, but the href attribute value http://horstmann.com is not displayed on the page.

Of course, HTML is a little different from the XML documents that you construct to describe data, such as product lists, but the same basic rule applies. Anything that’s a part of your data should not be an attribute. An attribute is appropriate only if it tells something about the data but isn’t a part of the data itself. If you find yourself engaged in metaphysical discussions to determine whether an item is part of the data or tells something about the data, make the item an element, not an attribute.

### Avoid Children with Mixed Elements and Text

The children of an element can be

1. Elements
2. Text
3. A mixture of both

In HTML, it is common to mix elements and text, for example

```html
<p>Use XML for <strong>robust</strong> data formats.</p>
```

But when describing data sets, you should not mix elements and text. For example, you should not do the following:

```xml
<price>
  <currency>USD</currency>
  29.95
</price>
```

Instead, the children of an element should be either text

```xml
<price>29.95</price>
```
or elements

```xml
<price>
  <currency>USD</currency>
  <amount>29.95</amount>
</price>
```

There is an important reason for this design rule. As you will see later in this chapter, you can specify much stricter rules for elements that have only child elements than for elements whose children can contain mixed content.
To read and analyze the contents of an XML document, you need an XML parser. A parser is a program that reads a document, checks whether it is syntactically correct, and takes some action as it processes the document.

Two kinds of XML parsers are in common use. Streaming parsers read the XML input one token at a time and report what they encounter: a start-tag, text, an end-tag, and so on. In contrast, a tree-based parser builds a tree that represents the parsed document. Once the parser is done, you can analyze the tree.

Streaming parsers are more efficient for handling large XML documents whose tree structure would require large amounts of memory. Tree-based parsers, however, are easier to use for most applications—the parse tree gives you a complete overview of the data, whereas a streaming parser gives you the information in bits and pieces.

In this section, you will learn how to use a tree-based parser that produces a tree structure according to the DOM (Document Object Model) standard. The DOM standard defines interfaces and methods to analyze and modify the tree structure that represents an XML document.

In order to parse an XML document into a DOM tree, use the DocumentBuilder class from the java.xml package. To get a DocumentBuilder object, first call the static newInstance method of the DocumentBuilderFactory class, then call the newDocumentBuilder method on the factory object:

```java
DocumentBuilderFactory factory = DocumentBuilderFactory.newInstance();
DocumentBuilder builder = factory.newDocumentBuilder();
```

Once you have a DocumentBuilder, you can read a document. To read a document from a file, first construct a File object from the file name, then call the parse method of the DocumentBuilder class:

```java
String fileName = "";
File f = new File(fileName);
Document doc = builder.parse(f);
```

If the document is located on the Internet, use a URL:

```java
String urlName = "";
URL u = new URL(urlName);
Document doc = builder.parse(u);
```

You can also read a document from an arbitrary input stream:

```java
InputStream in = "";
Document doc = builder.parse(in);
```

Once you have created a new document or read a document from a file, you can inspect and modify it.

The easiest method for inspecting a document is the XPath syntax. In the DOM standard, a node is the common superclass for all components that make up an XML document. In particular, text sequences and elements are nodes. An XPath describes a node or set of nodes, using a syntax that is similar to directory paths. For example, consider the following XPath, applied to the document in Figures 1 and 2:

```
/items/item[1]/quantity
```

This XPath selects the quantity of the first item, that is, the value 8. (In XPath, array positions start with 1. Accessing /items/item[0] would be an error.)
Similarly, you can get the price of the second product as

```
/items/item[2]/product/price
```

To get the number of items, use the XPath expression

```
count(/items/item)
```

In our example, the result is 2.

The total number of children can be obtained as

```
count(/items/*)
```

In our example, the result is again 2 because the `items` element has exactly two children.

To select attributes, use an `@` followed by the name of the attribute. For example,

```
/items/item[2]/product/price/@currency
```

would select the `currency` attribute of the price element if it had one.

Finally, if you have a document with variable or unknown structure, you can find out the name of a child with an expression such as the following:

```
name(/items/item[1]/*[1])
```

The result is the name of the first child of the first item, or `product`.

---

**Figure 1**
An XML Document

```xml
<?xml version="1.0"?>
<items>
  <item>
    <product>
      <description>Ink Jet Refill Kit</description>
      <price>29.95</price>
    </product>
    <quantity>8</quantity>
  </item>
  <item>
    <product>
      <description>4-port Mini Hub</description>
      <price>19.95</price>
    </product>
    <quantity>4</quantity>
  </item>
</items>
```

---

**Figure 2**
The Tree View of the Document
25.2

Parsing XML Documents

W1069

That is all you need to know about the XPath syntax to analyze simple documents.
(See Table 1 for a summary.) There are many more options in the XPath syntax that
we do not cover here. If you are inter­ested, look up the specification (http://www.
w3.org/TR/xpath) or work through the online tutorial (http://www.zvon.org/xxl/XPathTutorial/General/examples.html).
To evaluate an XPath expression in Java, first create an XPath object:
XPathFactory xpfactory = XPathFactory.newInstance();
XPath path = xpfactory.newXPath();

Then call the evaluate method, like this:
String result = path.evaluate(expression, doc)

Here, expression is an XPath expression and doc is the Document object that represents
the XML document. For example, the statement
String result = path.evaluate("/items/item[2]/product/price", doc)

sets result to the string "19.95".
Now you have all the tools that you need to read and analyze an XML document.
The example pro­gram at the end of this section puts these techniques to work. (The
program uses the LineItem and Product classes from Chapter 12.) The class ItemListParser can parse an XML document that contains a list of product descriptions. Its
parse method takes the file name and returns an array list of LineItem objects:
ItemListParser parser = new ItemListParser();
ArrayList<LineItem> items = parser.parse("items.xml");

The ItemListParser class translates each XML element into an object of the corresponding Java class. We first get the number of items:
int itemCount = Integer.parseInt(path.evaluate("count(/items/item)", doc));

For each item element, we gather the product data and construct a Product object:
String description = path.evaluate(
"/items/item[" + i + "]/product/description", doc);
double price = Double.parseDouble(path.evaluate(
"/items/item[" + i + "]/product/price", doc));
Product pr = new Product(description, price);

Then we construct a LineItem object in the same way, and add it to the items array list.

Table 1 XPath Syntax Summary
Syntax Element

Purpose

Example

name

Matches an element

item

/

Separates elements

/items/item

[n]

Selects a value from a set

/items/item[1]

@name

Matches an attribute

price/@currency

*

Matches anything

/items/*[1]

count

Counts matches

count(/items/item)

name

The name of a match

name(/items/*[1])


Here is the complete source code:

```
import java.io.File;
import java.io.IOException;
import java.util.ArrayList;
import javax.xml.parsers.DocumentBuilder;
import javax.xml.parsers.DocumentBuilderFactory;
import javax.xml.parsers.ParserConfigurationException;
import javax.xml.xpath.XPath;
import javax.xml.xpath.XPathExpressionException;
import javax.xml.xpath.XPathFactory;
import org.w3c.dom.Document;
import org.xml.sax.SAXException;

/**
   * An XML parser for item lists.
   */
public class ItemListParser
{
  private DocumentBuilder builder;
  private XPath path;

  /**
   * Constructs a parser that can parse item lists.
   */
  public ItemListParser()
  throws ParserConfigurationException
  {
    DocumentBuilderFactory dbfactory = DocumentBuilderFactory.newInstance();
    builder = dbfactory.newDocumentBuilder();
    XPathFactory xpfactory = XPathFactory.newInstance();
    path = xpfactory.newXPath();
  }

  /**
   * Parses an XML file containing an item list.
   * @param fileName the name of the file
   * @return an array list containing all items in the XML file
   */
  public ArrayList<LineItem> parse(String fileName)
  throws SAXException, IOException, XPathExpressionException
  {
    File f = new File(fileName);
    Document doc = builder.parse(f);
    ArrayList<LineItem> items = new ArrayList<>();
    int itemCount = Integer.parseInt(path.evaluate("count(/items/item)", doc));
    for (int i = 1; i <= itemCount; i++)
    {
      String description = path.evaluate("/items/item[" + i + "]/product/description", doc);
      double price = Double.parseDouble(path.evaluate("/items/item[" + i + "]/product/price", doc));
      Product pr = new Product(description, price);
      int quantity = Integer.parseInt(path.evaluate("/items/item[" + i + "]/quantity", doc));
      ItemListParser.java:119
```
import java.util.ArrayList;

/**
   * This program parses an XML file containing an item list.
   * It prints out the items that are described in the XML file.
   */
public class ItemListParserDemo {
    public static void main(String[] args) throws Exception {
        ItemListParser parser = new ItemListParser();
        ArrayList<LineItem> items = parser.parse("items.xml");
        for (LineItem anItem : items) {
            System.out.println(anItem.format());
        }
    }
}

Program Run

<table>
<thead>
<tr>
<th>Item</th>
<th>Price</th>
<th>Quantity</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ink Jet Refill Kit</td>
<td>29.95</td>
<td>8</td>
<td>239.6</td>
</tr>
<tr>
<td>4-port Mini Hub</td>
<td>19.95</td>
<td>4</td>
<td>79.8</td>
</tr>
</tbody>
</table>

4. What is the result of evaluating the XPath statement /items/item[1]/product/price in the XML document of Figure 2?

5. Which XPath statement yields the name of the root element of any XML document?

**Practice It** Now you can try these exercises at the end of the chapter: R25.10, E25.1, E25.4.

---

**XML Elements Describe Objects, Not Classes**

When you convert XML documents to Java classes, you need to determine a class for each element type. A common mistake is to make a separate class for each XML element. For example, consider a slightly different invoice description, with separate shipping and billing addresses:

```
<invoice>
  <shipto>
    <name>AOME Computer Supplies Inc.</name>
    <street>1195 W. Fairfield Rd.</street>
    <city>Sunnyvale</city>
    <state>CA</state>
    <zip>94085</zip>
  </shipto>
  <items>
    <item>
      <name>Ink Jet Refill Kit</name>
      <price>29.95</price>
      <quantity>8</quantity>
      <total>239.6</total>
    </item>
    <item>
      <name>4-port Mini Hub</name>
      <price>19.95</price>
      <quantity>4</quantity>
      <total>79.8</total>
    </item>
  </items>
</invoice>
```
Should you have a class Shipto to match the shipto element and another class Billto to match the billto element? That makes no sense, because both of them have the same contents: elements that describe an address.

Instead, you should think of the XML element as the value of an instance variable and then determine an appropriate class. For example, an invoice object has instance variables

- billto, of type Address
- shipto, also of type Address

Note that you don’t see the classes in the XML document. There is no notion of a class Address in the XML document describing an invoice. To make element classes explicit, you use an XML schema—see Special Topic 25.2 for more information.

25.3 Creating XML Documents

In the preceding section, you saw how to read an XML file into a Document object and analyze the contents of that object. In this section, you will see how to do the opposite—build up a Document object and then save it as an XML file. Of course, you can also generate an XML file simply as a sequence of print statements. However, that is not a good idea—it is easy to build an illegal XML document in this way, as when data contain special characters such as < or &.

Recall that you needed a DocumentBuilder object to read in an XML document. You also need such an object to create a new, empty document. Thus, to create a new document, first make a DocumentBuilderFactory, then a DocumentBuilder, and finally the empty document:

```java
DocumentBuilderFactory factory = DocumentBuilderFactory.newInstance();
DocumentBuilder builder = factory.newDocumentBuilder();
Document doc = builder.newDocument(); // An empty document
```

A document contains two kind of nodes, elements and text nodes. The DOM standard provides interfaces for these node types, as well as a common superinterface Node (see Figure 3). You use the createElement method of the Document interface to create the elements that you need:

```java
Element priceElement = doc.createElement("price");
```

You set element attributes with the setAttribute method. For example,

```java
priceElement.setAttribute("currency", "USD");
```

You have to work a bit harder for inserting text. First create a text node:

```java
Text textNode = doc.createTextNode("29.95");
```
Then add the text node to the element:

```java
priceElement.appendChild(textNode);
```

To construct the tree structure of a document, it is a good idea to use a set of helper methods. We start out with a helper method that creates an element with text:

```java
private Element createTextElement(String name, String text) {
    Text t = doc.createTextNode(text);
    Element e = doc.createElement(name);
    e.appendChild(t);
    return e;
}
```

Using this helper method, we can construct a price element like this:

```java
Element priceElement = createTextElement("price", "29.95");
```

Next, we write a helper method to create a product element from a Product object:

```java
private Element createProduct(Product p) {
    Element e = doc.createElement("product");
    e.appendChild(createTextElement("description", p.getDescription()));
    e.appendChild(createTextElement("price", " + p.getPrice()));
    return e;
}
```

These helper methods are called from the createItem helper method:

```java
private Element createItem(LineItem anItem) {
    Element e = doc.createElement("item");
    e.appendChild(createProduct(anItem.getProduct()));
    e.appendChild(createTextElement("quantity", " + anItem.getQuantity()));
    return e;
}
```
A helper method

```java
private Element createItems(ArrayList<LineItem> items)
```

for the items element is implemented in the same way—see the program listing at the end of this section.

Now you build the document as follows:

```java
ArrayList<LineItem> items = . . . ;
doc = builder.newDocument();
Element root = createItems(items);
doc.appendChild(root);
```

Once you have built the document, you will want to write it to a file. The DOM standard provides the LSSerializer interface for this purpose. Unfortunately, the DOM standard uses very generic methods, which makes the code that is required to obtain a serializer object look like a “magic incantation”:

```java
DOMImplementation impl = doc.getImplementation();
DOMImplementationLS implLS = (DOMImplementationLS) impl.getFeature("LS", "3.0");
LSSerializer ser = implLS.createLSSerializer();
```

Once you have the serializer object, you simply use the `writeToString` method:

```java
String str = ser.writeToString(doc);
```

By default, the LSSerializer produces an XML document without spaces or line breaks. As a result, the output looks less pretty, but it is actually more suitable for parsing by another program because it is free from unnecessary white space.

If you want white space, you use yet another magic incantation after creating the serializer:

```java
ser.getDomConfig().setParameter("format-pretty-print", true);
```

Here is an example program that shows how to build and print an XML document:

```java
section_3/ItemListBuilder.java
```

```java
import java.util.ArrayList;
import javax.xml.parsers.DocumentBuilder;
import javax.xml.parsers.DocumentBuilderFactory;
import javax.xml.parsers.ParserConfigurationException;
import org.w3c.dom.Document;
import org.w3c.dom.Element;
import org.w3c.dom.Text;

/**
 * Builds a DOM document for an array list of items.
 */
public class ItemListBuilder {
    private DocumentBuilder builder;
    private Document doc;

    /**
     * Constructs an item list builder.
     */
    public ItemListBuilder() {
    }
```
```java
DocumentBuilderFactory factory = DocumentBuilderFactory.newInstance();
builder = factory.newDocumentBuilder();

/**
   * Builds a DOM document for an array list of items.
   * @param items the items
   * @return a DOM document describing the items
   */
public Document build(ArrayList<LineItem> items) {
    doc = builder.newDocument();
    doc.appendChild(createItems(items));
    return doc;
}

/**
   * Builds a DOM element for an array list of items.
   * @param items the items
   * @return a DOM element describing the items
   */
private Element createItems(ArrayList<LineItem> items) {
    Element e = doc.createElement("items");
    for (LineItem anItem : items) {
        e.appendChild(createItem(anItem));
    }
    return e;
}

/**
   * Builds a DOM element for an item.
   * @param anItem the item
   * @return a DOM element describing the item
   */
private Element createItem(LineItem anItem) {
    Element e = doc.createElement("item");
    e.appendChild(createProduct(anItem.getProduct()));
    e.appendChild(createTextElement("quantity", "" + anItem.getQuantity()));
    return e;
}

/**
   * Builds a DOM element for a product.
   * @param p the product
   * @return a DOM element describing the product
   */
private Element createProduct(Product p) {
    Element e = doc.createElement("product");
```
Section 3/ItemListBuilderDemo.java

```java
import java.util.ArrayList;
import org.w3c.dom.DOMImplementation;
import org.w3c.dom.Document;
import org.w3c.dom.ls.DOMImplementationLS;
import org.w3c.dom.ls.LSSerializer;

/**
 * This program demonstrates the item list builder. It prints the XML file corresponding to a DOM document containing a list of items.
 */
public class ItemListBuilderDemo {
    public static void main(String[] args) throws Exception {
        ArrayList<LineItem> items = new ArrayList<>();
        items.add(new LineItem(new Product("Toaster", 29.95), 3));
        items.add(new LineItem(new Product("Hair dryer", 24.95), 1));

        ItemListBuilder builder = new ItemListBuilder();
        Document doc = builder.build(items);
        DOMImplementation impl = doc.getImplementation();
        DOMImplementationLS implLS = (DOMImplementationLS) impl.getFeature("LS", "3.0");
        LSSerializer ser = implLS.createLSSerializer();
        String out = ser.writeToString(doc);
        System.out.println(out);
    }
}
```

This program uses the Product and LineItem classes from Chapter 12. The LineItem class has been modified by adding getProduct and getQuantity methods.

Program Run

```xml
<?xml version="1.0" encoding="UTF-8"?><items><item><product>
<description>Toaster</description><price>29.95</price></product>
<quantity>3</quantity></item><item><product><description>Hair dryer</description><price>24.95</price></product><quantity>1</quantity>
</item></items>
```
6. Suppose you need to construct a `Document` object that represents an XML document other than an item list. Which methods from the `ItemListBuilder` class can you reuse?

7. How would you write a document to the file `output.xml`?

**Practice It**

Now you can try these exercises at the end of the chapter: R25.12, P25.4, P25.5.

---

**HOW TO 25.2 Writing an XML Document**

What is the best way to write an XML document? This How To shows you how to produce a `Document` object and generate an XML document from it.

---

**Step 1**

Provide the outline of a document builder class.

To construct the `Document` object from an object of some class, you should implement a class such as this one:

```java
public class MyBuilder {
  private DocumentBuilder builder;
  private Document doc;

  public Document build(SomeClass x) { . . . }

  private Element createTextElement(String name, String text) {
    Text t = doc.createTextNode(text);
    Element e = doc.createElement(name);
    e.appendChild(t);
    return e;
  }
}
```

**Step 2**

Look at the format of the XML document that you want to create.

Consider all elements, except those that have only text content. Find the matching Java classes. In the `ItemListBuilder` example, we ignore `quantity`, `description`, and `price` because they have text content. The remaining elements and their Java classes are:

- `product` - `Product`
- `item` - `LineItem`
- `items` - `ArrayList<LineItem>`

**Step 3**

For each element in Step 2, add a helper method to your builder class.

Each helper method has the form

```java
private Element createElementName(ClassForElement x)
```

For example,

```java
public class MyBuilder {
  private DocumentBuilder builder;
  private Document doc;

  public Document build(SomeClass x) {
    // Construction code...
  }

  private Element createTextElement(String name, String text) {
    Text t = doc.createTextNode(text);
    Element e = doc.createElement(name);
    e.appendChild(t);
    return e;
  }
}
```

---

**Step 1**

Provide the outline of a document builder class.

To construct the `Document` object from an object of some class, you should implement a class such as this one:

```java
public class MyBuilder {
  private DocumentBuilder builder;
  private Document doc;

  public Document build(SomeClass x) { . . . }

  private Element createTextElement(String name, String text) {
    Text t = doc.createTextNode(text);
    Element e = doc.createElement(name);
    e.appendChild(t);
    return e;
  }
}
```

**Step 2**

Look at the format of the XML document that you want to create.

Consider all elements, except those that have only text content. Find the matching Java classes. In the `ItemListBuilder` example, we ignore `quantity`, `description`, and `price` because they have text content. The remaining elements and their Java classes are:

- `product` - `Product`
- `item` - `LineItem`
- `items` - `ArrayList<LineItem>`

**Step 3**

For each element in Step 2, add a helper method to your builder class.

Each helper method has the form

```java
private Element createElementName(ClassForElement x)
```

For example,

```java
public class MyBuilder {
  private DocumentBuilder builder;
  private Document doc;

  public Document build(SomeClass x) {
    // Construction code...
  }

  private Element createTextElement(String name, String text) {
    Text t = doc.createTextNode(text);
    Element e = doc.createElement(name);
    e.appendChild(t);
    return e;
  }
}
```
Step 4 Implement the helper methods.

For each element, call the helper methods of its children. However, if a child has text content, call createTextElement instead.

For example, the item element has two children: product and quantity. The former has a helper method, and the latter has text content. Therefore, the createItem method calls createProduct and createTextElement:

```java
private Element createItem(LineItem anItem) {
    Element e = doc.createElement("item");
    e.appendChild(createProduct(anItem.getProduct()));
    e.appendChild(createTextElement("quantity", "" + anItem.getQuantity()));
    return e;
}
```

You may find it helpful to implement the helper methods “bottom up”, starting with the simplest method (such as createProduct) and finishing with the method for the root element (createItems).

Step 5 Finish off your builder by writing a constructor and the build method.

```java
public class MyBuilder {
    public MyBuilder() throws ParserConfigurationException {
        DocumentBuilderFactory factory = DocumentBuilderFactory.newInstance();
        builder = factory.newDocumentBuilder();
    }
    public Document build(ClassForRootElement x) {
        doc = builder.newDocument();
        doc.appendChild(createRootElementName(x));
        return doc;
    }
    ...
}
```

Step 6 Use a class, such as the LSSerializer, to convert the Document to a string.

For example,

```java
Invoice x = ...;
InvoiceBuilder builder = new InvoiceBuilder();
Document doc = builder.build(x);
LSSerializer ser = ...;
String str = ser.writeToString(doc);
```
Grammars, Parsers, and Compilers

Grammars are very important in many areas of computer science to describe the structure of computer programs or data formats. To introduce the concept of a grammar, consider this set of rules for a set of simple English language sentences:

1. A sentence has a noun phrase followed by a verb and another noun phrase.
2. A noun phrase consists of an article followed by an adjective list followed by a noun.
3. An adjective list consists of an adjective or an adjective followed by an adjective list.
4. Articles are “a” and “the”.
5. Adjectives are “quick”, “brown”, “lazy”, and “hungry”.
6. Nouns are “fox”, “dog”, and “hamster”.
7. Verbs are “jumps over” and “eats”.

Here are two sentences that follow these rules:
- The quick brown fox jumps over the lazy dog.
- The hungry hamster eats a quick brown fox.

Symbolically, these rules can be expressed by a formal grammar:

\[
\begin{align*}
\text{<sentence> } &::= \text{<noun-phrase> <verb> <noun-phrase>} \\
\text{<noun-phrase> } &::= \text{<article> <adjective-list> <noun>} \\
\text{<adjective-list> } &::= \text{<adjective> | <adjective> <adjective-list>} \\
\text{<article> } &::= \text{a | the} \\
\text{<adjective> } &::= \text{quick | brown | lazy | hungry} \\
\text{<noun> } &::= \text{fox | dog | hamster} \\
\text{<verb> } &::= \text{jumps over | eats}
\end{align*}
\]

Here the symbol ::= means “can be replaced with” and | separates alternate choices. For example, <article> can be replaced with “a” or “the”.

The grammar symbols, such as <noun>, happen to be enclosed in angle brackets just like XML tags, but they are different from tags. One purpose of a grammar is to produce strings that are valid according to the grammar by starting with the start symbol (<sentence> in this example) and applying replacement rules until the resulting string is free from symbols. See Table 2 for an example of the replacement process.

If you have a grammar and a string, such as “the hungry hamster eats a quick brown fox” or “a brown jumps over hamster quick lazy”, you can parse the sentence: that is, check whether the sentence is described by the grammar rules and, if it is, show how it can be derived from the start symbol (see Table 2). Another way to show the derivation is to construct a parse tree (see Figure 4).

| Table 2 Deriving a Sentence from a Grammar |
|---|---|
| String                  | Rule   |
| <sentence>              | Start  |
| <noun-phrase> <verb> <noun-phrase> | 1 |
| <noun-phrase> eats <noun-phrase> | 7 |
| <article> <adjective-list> <noun> eats <noun-phrase> | 2 |
| the <adjective-list> <noun> eats <noun-phrase> | 4 |
Table 2  Deriving a Sentence from a Grammar

<table>
<thead>
<tr>
<th>String</th>
<th>Rule</th>
</tr>
</thead>
<tbody>
<tr>
<td>the &lt;adjective&gt; &lt;noun&gt; eats &lt;noun-phrase&gt;</td>
<td>3</td>
</tr>
<tr>
<td>the hungry &lt;noun&gt; eats &lt;noun-phrase&gt;</td>
<td>5</td>
</tr>
<tr>
<td>the hungry hamster eats &lt;noun-phrase&gt;</td>
<td>6</td>
</tr>
<tr>
<td>the hungry hamster eats &lt;article&gt; &lt;adjective-list&gt; &lt;noun&gt;</td>
<td>2</td>
</tr>
<tr>
<td>the hungry hamster eats a &lt;adjective-list&gt; &lt;noun&gt;</td>
<td>4</td>
</tr>
<tr>
<td>the hungry hamster eats a &lt;adjective&gt; &lt;adjective-list&gt; &lt;noun&gt;</td>
<td>3</td>
</tr>
<tr>
<td>the hungry hamster eats a quick &lt;adjective-list&gt; &lt;noun&gt;</td>
<td>5</td>
</tr>
<tr>
<td>the hungry hamster eats a quick &lt;adjective&gt; &lt;noun&gt;</td>
<td>3</td>
</tr>
<tr>
<td>the hungry hamster eats a quick brown &lt;noun&gt;</td>
<td>5</td>
</tr>
<tr>
<td>the hungry hamster eats a quick brown fox</td>
<td>6</td>
</tr>
</tbody>
</table>

A parser is a program that reads strings and decides whether the input conforms to the rules of a certain grammar. Some parsers—such as the DOM XML parser—build a parse tree in the process or report an error message when a parse tree cannot be constructed. Other parsers—such as the SAX XML parser—call user-specified methods whenever a part of the input was successfully parsed.

The most important use for parsers is inside compilers for programming languages. Just as our grammar can describe (some) simple English language sentences, the valid “sentences” in a programming language can be described by a grammar. The actual grammar for the Java programming language occupies about fifteen pages in *The Java Language Specification* (http://docs.oracle.com/javase/specs/jls/se8/html/index.html). To give a flavor of a small subset of such a grammar, here is a grammar that describes arithmetic expressions.

```
<expression> ::= <term> | <expression> <additive-operator> <term>
<additive-operator> ::= + | -
<term> ::= <factor> | <term> <multiplicative-operator> <factor>
<multiplicative-operator> ::= * | /
<integer> ::= <digits> | - <digits>
<digits> ::= <digit> | <digit> <digits>
<digit> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
```

An example of a valid expression in this grammar is

\[-2 \times (3 + 10)\]

Try deriving this expression from the <expression> start symbol, as was done in Table 2 or Figure 4!

In a compiler, parsing the program source is the first step toward generating code that the target processor (the Java virtual machine in the case of Java) can execute. Writing a parser is a challenging and interesting task. You may at one point in your studies take a course in compiler construction, in which you learn how to write a parser and how to generate code from the parsed input. Fortunately, to use XML you don’t have to know how the parser does its job.
You simply ask the XML parser to read the XML input and then process the resulting Document tree.

In this section you will learn how to specify rules for XML documents of a particular type. There are several mechanisms for this purpose. The oldest and simplest mechanism is a Document Type Definition (DTD), the topic of this section. We discuss other mechanisms in Special Topic 25.2.

### 25.4.1 Document Type Definitions

Consider a document of type items. Intuitively, items denotes a sequence of item elements. Each item element contains a product and a quantity. A product contains a description and a price. Each of these elements contains text describing the product’s description, price, and quantity. The purpose of a DTD is to formalize this description.

A DTD is a sequence of rules that describes

- The valid attributes for each element type.
- The valid child elements for each element type.

Let us first turn to child elements. The valid child elements of an element are described by an ELEMENT rule:

```xml
<!ELEMENT items (item*)>
```

This means that an item list must contain a sequence of 0 or more item elements.

As you can see, the rule is delimited by `<!...>`, and it contains the name of the element whose children are to be constrained (items), followed by a description of what children are allowed.

**Figure 4** A Parse Tree for a Simple Sentence

A DTD is a sequence of rules that describes the valid child elements and attributes for each element type.
Next, let us turn to the definition of an item element:

```
<!ELEMENT item (product, quantity)>
```

This means that the children of an item element must be a product element, followed by a quantity element.

The definition for a product is similar:

```
<!ELEMENT product (description, price)>
```

Finally, here are the definitions of the three remaining elements:

```
<!ELEMENT quantity (#PCDATA)>
<!ELEMENT description (#PCDATA)>
<!ELEMENT price (#PCDATA)>
```

The symbol #PCDATA refers to text, called “parsed character data” in XML terminology. The character data can contain any characters. However, certain characters, such as `<` and `&`, have special meaning in XML and need to be replaced if they occur in character data. Table 3 shows the replacements for special characters.

<table>
<thead>
<tr>
<th>Character</th>
<th>Encoding</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>&lt;</code></td>
<td>&lt;</td>
<td>Less than (left angle bracket)</td>
</tr>
<tr>
<td><code>&gt;</code></td>
<td>&gt;</td>
<td>Greater than (right angle bracket)</td>
</tr>
<tr>
<td><code>&amp;</code></td>
<td>&amp;</td>
<td>Ampersand</td>
</tr>
<tr>
<td><code>'</code></td>
<td>'</td>
<td>Apostrophe</td>
</tr>
<tr>
<td><code>&quot;</code></td>
<td>&quot;</td>
<td>Quotation mark</td>
</tr>
</tbody>
</table>

The complete DTD for an item list has six rules, one for each element type:

```
<!ELEMENT items (item*)>
<!ELEMENT item (product, quantity)>
<!ELEMENT product (description, price)>
<!ELEMENT quantity (#PCDATA)>
<!ELEMENT description (#PCDATA)>
<!ELEMENT price (#PCDATA)>
```

Let us have a closer look at the descriptions of the allowed children. Table 4 shows the expressions used to describe the children of an element. The EMPTY reserved word is self-explanatory: an element that is declared as EMPTY may not have any children. For example, the HTML DTD defines the `img` element to be EMPTY—an image has only attributes, specifying the image source, size, and placement, and no children.

More interesting child rules can be formed with the regular expression operations (`*` `+` `?` `|` `.`). (See Table 4 and Figure 5. Also see Special Topic 11.4 for more information on regular expressions.) You have already seen the `*` (“0 or more”) and `, (sequence) operations. The children of an items element are 0 or more item elements, and the children of an item are a sequence of product and quantity elements.
<table>
<thead>
<tr>
<th>Rule Description</th>
<th>Element Content</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>EMPTY</strong></td>
<td>No children allowed</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>$(E^*)$</td>
<td>Any sequence of 0 or more elements $E$</td>
</tr>
<tr>
<td>$(E^+)$</td>
<td>Any sequence of 1 or more elements $E$</td>
</tr>
<tr>
<td>$(E?)$</td>
<td>Optional element $E$ (0 or 1 occurrences allowed)</td>
</tr>
<tr>
<td>$(E_1, E_2, \ldots)$</td>
<td>Element $E_1$, followed by $E_2, \ldots$</td>
</tr>
<tr>
<td>$(E_1</td>
<td>E_2</td>
</tr>
<tr>
<td>$(\texttt{PCDATA})$</td>
<td>Text only</td>
</tr>
<tr>
<td>$(\texttt{PCDATA}</td>
<td>E_1</td>
</tr>
<tr>
<td><strong>ANY</strong></td>
<td>Any children allowed</td>
</tr>
</tbody>
</table>

You can also combine these operations to form more complex expressions:

```xml
<!ELEMENT section (title, (paragraph | (image, title?))+)
```

defines an element `section` whose children are:

1. A title element
2. A sequence of one or more of the following:
   - paragraph elements
   - image elements followed by optional title elements

---

**Figure 5**

**DTD Regular Expression Operations**
Thus,

```xml
<section>
    <title/>
    <paragraph/>
    <image/>
</section>
```

is valid, but

```xml
<section>
    <paragraph/>
    <paragraph/>
    <title/>
</section>
```

is not—there is no starting title, and the title at the end doesn’t follow an image.

You already saw the (#PCDATA) rule. It means that the children can consist of any character data. For example, in our product list DTD, the description element can have any character data inside.

You can also allow mixed content—any sequence of character data and specified elements. However, in mixed content, you have no control over the order in which the elements appear. As explained in Programming Tip 25.2, you should avoid mixed content for DTDs that describe data sets. This feature is intended for documents that contain both text and markup instructions, such as HTML pages.

Finally, you can allow an element to have children of any type—you should avoid that for DTDs that describe data sets.

You now know how to specify what children an element may have. A DTD also gives you control over the allowed attributes of an element. An attribute description looks like this:

```xml
<!ATTLIST Element Attribute Type Default>
```

The most useful attribute type descriptions are listed in Table 5. The CDATA type describes any sequence of character data. As with #PCDATA, certain characters, such as `<` and `&`, need to be encoded (as `&lt;`, `&amp;` and so on). There is no practical difference between the CDATA and #PCDATA types. Simply use CDATA in attribute declarations and #PCDATA in element declarations.

Rather than allowing arbitrary attribute values, you can specify a finite number of choices. For example, you may want to restrict a currency attribute to U.S. dollar, euro, and Japanese yen. Then use the following declaration:

```xml
<!ATTLIST price currency (USD | EUR | JPY) #REQUIRED>
```

You can use letters, numbers, and the hyphen (-) and underscore (_) characters for the attribute values.

<table>
<thead>
<tr>
<th>Table 5 Common Attribute Types</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type Description</td>
</tr>
<tr>
<td>CDATA</td>
</tr>
<tr>
<td>($V_1 \mid V_2 \mid \ldots$)</td>
</tr>
</tbody>
</table>
Table 6  Attribute Defaults

<table>
<thead>
<tr>
<th>Default Declaration</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>#REQUIRED</td>
<td>Attribute is required</td>
</tr>
<tr>
<td>#IMPLIED</td>
<td>Attribute is optional</td>
</tr>
<tr>
<td>V</td>
<td>Default attribute, to be used if attribute is not specified</td>
</tr>
<tr>
<td>#FIXED V</td>
<td>Attribute must either be unspecified or contain this value</td>
</tr>
</tbody>
</table>

There are other type descriptions that are less common in practice. You can find them in the XML reference (http://www.xml.com/axml/axml.html).

The attribute type description is followed by a “default” declaration. The reserved words that can appear in a “default” declaration are listed in Table 6.

For example, this attribute declaration specifies that each price element must have a currency attribute whose value is any character data:

```xml
<!ATTLIST price currency CDATA #REQUIRED>
```

To fulfill this declaration, each price element must have a currency attribute, such as `<price currency="USD">`. A price without a currency would not be valid.

For an optional attribute, you use the #IMPLIED reserved word instead:

```xml
<!ATTLIST price currency CDATA #IMPLIED>
```

That means that you can supply a currency attribute in a price element, or you can omit it. If you omit it, then the application that processes the XML data implicitly assumes some default currency.

A better choice would be to supply the default value explicitly:

```xml
<!ATTLIST price currency CDATA "USD">
```

That means that the currency attribute is understood to mean USD if the attribute is not specified. An XML parser will then report the value of currency as USD if the attribute was not specified.

Finally, you can state that an attribute can only be identical to a particular value. For example, the rule

```xml
<!ATTLIST price currency CDATA #FIXED "USD">
```

means that a price element must either not have a currency attribute at all (in which case the XML parser will report its value as USD), or specify the currency attribute as USD. Naturally, this kind of rule is not very common.

You have now seen the most common constructs for DTDs. Using these constructs, you can define your own DTDs for XML documents that describe data sets. In the next section, you will see how to specify which DTD an XML document should use, and how to have the XML parser check that a document conforms to its DTD.

### 25.4.2 Specifying a DTD in an XML Document

When you reference a DTD with an XML document, you can instruct the parser to check that the document follows the rules of the DTD. That way, the parser can check for errors in the document.
In the preceding section you saw how to develop a DTD for a class of XML documents. The DTD specifies the permitted elements and attributes in the document. An XML document has two ways of referencing a DTD:

1. The document may contain the DTD.
2. The document may refer to a DTD that is stored elsewhere.

A DTD is introduced with the `DOCTYPE` declaration. If the document contains its DTD, then the declaration looks like this:

```xml
<!DOCTYPE rootElement [ rules ]>
```

For example, an item list can include its DTD like this:

```xml
<?xml version="1.0"?>
<!DOCTYPE items [ 
  <!ELEMENT items (item*)> 
  <!ELEMENT item (product, quantity)> 
  <!ELEMENT product (description, price)> 
  <!ELEMENT quantity (#PCDATA)> 
  <!ELEMENT description (#PCDATA)> 
  <!ELEMENT price (#PCDATA)> 
]

<items>
  <item>
    <product>
      <description>Ink Jet Refill Kit</description>
      <price>29.95</price>
    </product>
    <quantity>8</quantity>
  </item>
  <item>
    <product>
      <description>4-port Mini Hub</description>
      <price>19.95</price>
    </product>
    <quantity>4</quantity>
  </item>
</items>
```

However, if the DTD is more complex, then it is better to store it outside the XML document. In that case, you use the `SYSTEM` reserved word inside the `DOCTYPE` declaration to indicate that the system that hosts the XML processor must locate the DTD. The `SYSTEM` reserved word is followed by the location of the DTD. For example, a `DOCTYPE` declaration might point to a local file:

```xml
<!DOCTYPE items SYSTEM "items.dtd">
```

Alternatively, the resource might be a URL anywhere on the Web:

```xml
<!DOCTYPE items SYSTEM "http://www.mycompany.com/dtds/items.dtd">
```

For commonly used DTDs, the `DOCTYPE` declaration can contain a `PUBLIC` reserved word. For example,

```xml
<!DOCTYPE faces-config PUBLIC "-//Sun Microsystems, Inc.//DTD Java Server Faces Config 1.0//EN" "http://java.sun.com/dtd/web-facesconfig_1_0.dtd">
```
A program parsing the DTD can look at the public identifier. If it is a familiar identifier, then it need not spend time retrieving the DTD from the URL.

### 25.4.3 Parsing and Validation

When you include a DTD with an XML document, then you can tell the parser to validate the document. That means that the parser will check that all child elements and attributes of an element conform to the ELEMENT and ATTLIST rules in the DTD. If a document is invalid, then the parser reports an error. To turn on validation, you use the setValidating method of the DocumentBuilderFactory class before calling the newDocumentBuilder method:

```java
DocumentBuilderFactory factory = DocumentBuilderFactory.newInstance();
factory.setValidating(true);
DocumentBuilder builder = factory.newDocumentBuilder();
Document doc = builder.parse(...);
```

Validation can simplify your code for processing XML documents. For example, if the DTD specifies that the child elements of each item element are product and quantity elements in that order, then you can rely on that fact and don’t need to put tedious checks in your code.

If the parser has access to the DTD, it can make another useful improvement. By default, the parser converts all spaces in the input document to text, even if the spaces are only used to logically line up elements. As a result, the document contains text nodes that are wasteful and can be confusing when you analyze the document tree.

To make the parser ignore white space, call the setIgnoringElementContentWhitespace method of the DocumentBuilderFactory class.

```java
factory.setValidating(true);
factory.setIgnoringElementContentWhitespace(true);
```

Finally, if the parser has access to the DTD, it can fill in default values for attributes. For example, suppose a DTD defines a currency attribute for a price element:

```xml
<!ATTLIST price currency CDATA "USD">
```

If a document contains a price element without a currency attribute, then the parser can supply the default:

```java
String attributeValue = priceElement.getAttribute("currency");
// Gets "USD" if no currency specified
```

This concludes our discussion of XML. You now know enough XML to put it to work for describing data formats. Whenever you are tempted to use a “quick and dirty” file format, you should consider using XML instead. By using XML for data interchange, your programs become more professional, robust, and flexible.

---

8. How can a DTD specify that the quantity element in an item is optional?

9. How can a DTD specify that a product element can contain a description and a price element, in any order?

10. How can a DTD specify that the description element has an optional attribute language?

**Practice It** Now you can try these exercises at the end of the chapter: R25.13, E25.3, E25.5.
Writing a DTD

You write a DTD to describe a set of XML documents of the same type. The DTD specifies which elements contain child elements (and the order in which they may appear) and which elements contain text. It also specifies which elements may have attributes, which attributes are required, and which defaults are used for missing attributes.

These rules are for DTDs that describe program data. DTDs that describe narrative text generally have a much more complex structure.

Step 1  Get or write a couple of sample XML documents.

For example, if you wanted to make a DTD for XML documents that describe an invoice, you could study samples such as the one in How To 25.1.

Step 2  Make a list of all elements that can occur in the XML document.

In the invoice example, they are

- invoice
- address
- name
- street
- city
- state
- zip
- items
- item
- product
- description
- quantity

Step 3  For each of the elements, decide whether its children are elements or text.

It is best to avoid elements whose children are a mixture of both.

In the invoice example, the following elements have element content:

- invoice
- address
- items
- item
- product

The remainder contain text.

Step 4  For elements that contain text, the DTD rule is

```xml
<!ELEMENT elementName (#PCDATA)>
```

Thus, we have the following simple rules for the invoice elements that contain text:

```xml
<!ELEMENT name (#PCDATA)>
<!ELEMENT street (#PCDATA)>
<!ELEMENT city (#PCDATA)>
<!ELEMENT state (#PCDATA)>
<!ELEMENT zip (#PCDATA)>
<!ELEMENT quantity (#PCDATA)>
<!ELEMENT description (#PCDATA)>
```

Step 5  For each element that contains other elements, make a list of the possible child elements.

Here are the lists in the invoice example:

- invoice
  - address
  - items
- address
  - name
  - street
  - city
  - state
  - zip
- items
  - item
- product
  - description
  - price
25.4 Validating XML Documents

Step 6 For each of those elements, decide in which order the child elements should occur and how often they should occur.

Then form the rule

```xml
<!ELEMENT elementName child1 count1, child2 count2, ... >
```

where each `count` is one of the following:

<table>
<thead>
<tr>
<th>Quantity</th>
<th>Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 or 1</td>
<td>?</td>
</tr>
<tr>
<td>1</td>
<td>omit</td>
</tr>
<tr>
<td>0 or more</td>
<td>*</td>
</tr>
<tr>
<td>1 or more</td>
<td>+</td>
</tr>
</tbody>
</table>

In the invoice example, the `items` element can contain any number of items, so the rule is

```xml
<!ELEMENT items (item*)>
```

In the remaining cases, each child element occurs exactly once. That leads to the rules

```xml
<!ELEMENT invoice (address, items)>
<!ELEMENT address (name, street, city, state, zip)>
<!ELEMENT item (product, quantity)>
<!ELEMENT product (description, price)>
```

Step 7 Decide whether any elements should have attributes.

Following Programming Tip 25.1, it is best to avoid attributes altogether or to minimize the use of attributes. Because we have no good reason to add attributes in the invoice example, our invoice is complete without attributes.

Schema Languages

Several mechanisms have been developed to deal with the limitations of DTDs. DTDs cannot express certain details about the structure of an XML document. For example, you can’t force an element to contain just a number or a date—any text string is allowed for a `(#PCDATA)` element.

The XML Schema specification is one mechanism for overcoming these limitations. An XML schema is like a DTD in that it is a set of rules that documents of a particular type need to follow, but a schema can contain far more precise rule descriptions.

Here is just a hint of how an XML schema is specified. For each element, you specify the element name and the type. For example, this definition restricts the contents of `quantity` to an integer.

```xml
<xsd:element name="quantity" type="xsd:integer"/>
```

Note that an XML schema is itself written in XML—unlike a DTD, which uses a completely different syntax. (The `xsd:` prefix is a `name space` prefix to denote that `xsd:element` and `xsd:integer` are part of the XML Schema Definition name space. See Special Topic 25.3 for more information about name spaces.)
In XML Schema, you can define complex types, much as you define classes in Java. Here is the definition of an Address type:

```xml
<xsd:complexType name="Address">
  <xsd:sequence>
    <xsd:element name="name" type="xsd:string"/>
    <xsd:element name="street" type="xsd:string"/>
    <xsd:element name="city" type="xsd:string"/>
    <xsd:element name="state" type="xsd:string"/>
    <xsd:element name="zip" type="xsd:string"/>
  </xsd:sequence>
</xsd:complexType>
```

Then you can specify that an invoice should have shipto and billto instance variables that are both of type Address:

```xml
<xsd:element name="shipto" type="Address"/>
<xsd:element name="billto" type="Address"/>
```

These examples show that an XML schema can be more precise than a DTD.

The XML Schema specification has many advanced features—see the W3C web site, www.w3.org/xml, for details. However, some programmers find that specification overly complex and instead use a competing standard called Relax NG—see www.relaxng.org. Relax NG is simpler than XML Schema, and it shares a feature with DTDs: a compact notation that is not XML.

For example, in Relax NG, you simply write

```xml
element quantity { xsd:integer }
```

to denote that quantity is an element containing an integer. The designers of Relax NG realized that XML, despite its many advantages, is not always the best notation for humans.

**Other XML Technologies**

This chapter covers the subset of the XML 1.0 specification that is most useful for common programming situations. Since version 1.0 of the XML specification was released, there has been a huge amount of interest in advanced XML technologies. A number of useful technologies have recently been standardized. Among them are:

- Schema Definitions
- Name Spaces
- XHTML
- XSL and Transformations

Special Topic 25.2 contains more information about schema definitions.

Name spaces were invented to ensure that many different people and organizations can develop XML documents without running into conflicts with element names. For example, if you look inside Special Topic 25.2, you will see that XML Schema definitions have element names that are prefixed with a tag `xsd:`; such as

```xml
<xsd:element name="city" type="xsd:string"/>
```

That way, the tag and attribute names, such as `element` and `string`, don’t conflict with other names. In that regard, name spaces are similar to Java packages. However, a name space prefix such as `xsd:` is just a shortcut for the actual name space identifier, which is a much longer, unique string. For example, the full name space for XML Schema definitions is `http://www.w3.org/2000/08/XMLSchema`. Each schema definition starts out with the statement

```xml
<xsd:schema xmlns:xsd="http://www.w3.org/2000/08/XMLSchema"
```

which binds the `xsd` prefix to the full name space.
XHTML is the most recent recommendation of the W3C for formatting web pages. Unlike HTML, XHTML is fully XML-compliant. Once web-editing tools switch to XHTML, it will become much easier to write programs that parse web pages. The XHTML standard has been carefully designed to be backward compatible with existing browsers.

While XHTML documents are intended to be viewed by browsers, general XML documents are not designed to be viewed at all. Nevertheless, it is often desirable to transform an XML document into a viewable form. XSL (Extensible Stylesheet Language) was created for this purpose. A style sheet indicates how to change an XML document into an HTML document, or even a completely different format, such as PDF.

For more information on these and other emerging technologies, see the W3C web site, http://www.w3.org/xml.

**CHAPTER SUMMARY**

**Describe the purpose of XML and the structure of an XML document.**
- XML allows you to encode complex data, independent of any programming language, in a form that the recipient can easily parse.
- XML files are readable by computer programs and by humans.
- XML-formatted data files are resilient to change.
- XML describes the meaning of data, not how to display them.
- An XML document starts out with an XML declaration and contains elements and text.
- An element can contain text, child elements, or both (mixed content). For data descriptions, avoid mixed content.
- Elements can have attributes. Use attributes to describe how to interpret the element content.

**Use a parser and the XPath language to process an XML document.**
- A parser is a program that reads a document, checks whether it is syntactically correct, and takes some action as it processes the document.
- A DocumentBuilder can read an XML document from a file, URL, or input stream. The result is a Document object, which contains a tree.
- An XPath describes a node or node set, using a notation similar to that for directory paths.

**Write Java programs that create XML documents.**
- The Document interface has methods to create elements and text nodes.
- Use an LSSerializer to write a DOM document.

**Explain the use of DTDs for validating XML documents.**
- A DTD is a sequence of rules that describes the valid child elements and attributes for each element type.
• An XML document can contain its DTD or refer to a DTD that is stored elsewhere.
• When referencing an external DTD, you must supply a URL for locating the DTD.
• When your XML document has a DTD, you can request validation when parsing.
• When you parse an XML file with a DTD, tell the parser to ignore white space.

STANDARD LIBRARY ITEMS INTRODUCED IN THIS CHAPTER

javax.xml.parsers.DocumentBuilder
  newDocument
  parse
javax.xml.parsers.DocumentBuilderFactory
  newDocumentBuilder
  newInstance
  setIgnoringElementContentWhitespace
  setValidating
javax.xml.xpath.XPath
  evaluate
javax.xml.xpath.XPathExpressionException
javax.xml.xpath.XPathFactory
  newInstance
  newXPath
org.w3c.dom.Document
  createElement
  createTextNode
  getImplementation

org.w3c.dom.DOMConfiguration
  setParameter
org.w3c.dom.DOMImplementation
  getFeature
org.w3c.dom.Element
  getAttribute
  setAttribute
org.w3c.dom.ls.DOMImplementationLS
  createLSSerializer
org.w3c.dom.ls.LSSerializer
  getDomConfig
  writeToString
org.xml.sax.SAXException

REVIEW EXERCISES

- **R25.1** Give some examples to show the differences between XML and HTML.
- **R25.2** Design an XML document that describes a bank account.
- **R25.3** Draw a tree view for the XML document you created in Exercise R25.2.
- **R25.4** Write the XML document that corresponds to the parse tree in Figure 2.
- **R25.5** Make an XML document describing a book, with child elements for the author name, the title, and the publication year.
- **R25.6** Add a description of the book’s language to the document of Exercise R25.5. Should you use an element or an attribute?
- **R25.7** What is mixed content? What problems does it cause?
- **R25.8** Design an XML document that describes a purse containing three quarters, a dime, and two nickels.
- **R25.9** Explain why a paint program, such as Microsoft Paint, is a WYSIWYG program that is also “what you see is all you’ve got”.

javax.xml.parsers.DocumentBuilder
  newDocument
  parse
javax.xml.parsers.DocumentBuilderFactory
  newDocumentBuilder
  newInstance
  setIgnoringElementContentWhitespace
  setValidating
javax.xml.xpath.XPath
  evaluate
javax.xml.xpath.XPathExpressionException
javax.xml.xpath.XPathFactory
  newInstance
  newXPath
org.w3c.dom.Document
  createElement
  createTextNode
  getImplementation

org.w3c.dom.DOMConfiguration
  setParameter
org.w3c.dom.DOMImplementation
  getFeature
org.w3c.dom.Element
  getAttribute
  setAttribute
org.w3c.dom.ls.DOMImplementationLS
  createLSSerializer
org.w3c.dom.ls.LSSerializer
  getDomConfig
  writeToString
org.xml.sax.SAXException
**R25.10** Consider the XML file

```xml
<purse>
  <coin>
    <value>0.5</value>
    <name lang="en">half dollar</name>
  </coin>
  <coin>
    <value>0.25</value>
    <name lang="en">quarter</name>
  </coin>
</purse>
```

What are the values of the following XPath expressions?

- a. `/purse/coin[1]/value`
- b. `/purse/coin[2]/name`
- c. `/purse/coin[2]/name/@lang`
- d. `name(/purse/coin[2]/*[1])`
- e. `count(/purse/coin)`
- f. `count(/purse/coin[2]/name)`

**R25.11** With the XML file of Exercise R25.10, give XPath expressions that yield

- a. the value of the first coin.
- b. the number of coins.
- c. the name of the first child element of the first coin element.
- d. the name of the first attribute of the first coin’s name element. (The expression `@*` selects the attributes of an element.)
- e. the value of the lang attribute of the second coin’s name element.

**R25.12** Harry Hopeless doesn’t want to build a DOM tree to produce an XML document. Instead, he uses the following code:

```java
System.out.println("<?xml version="1.0"?><items>
for (LineItem anItem: items) {
    Product p = anItem.getProduct();
    System.out.println("<item><product><description>" + p.getDescription()
    + "</description><prices>" + p.getPrice()
    + "</price></product><quantity>" + anItem.getQuantity()
    + "<quantity></item>";
}
System.out.println("</items>");
```

What can go wrong? How can one fix the problems?

**R25.13** Design a DTD that describes a bank with bank accounts.

**R25.14** Design a DTD that describes a library patron who has checked out a set of books. Each book has an ID number, an author, and a title. The patron has a name and telephone number.

**R25.15** Write the DTD file for the following XML document

```xml
<?xml version="1.0"?>
<productlist>
  <product>
    <name>Comtrade Tornado</name>
  </product>
</productlist>
```
<price currency="USD">2495</price>
<score>60</score>
</product>

<product>
  <name>AMAX Powerstation 75</name>
  <price>2999</price>
  <score>62</score>
</product>
</productlist>

**R25.16** Design a DTD for invoices, as described in How To 25.3.

**R25.17** Design a DTD for simple English sentences, as described in Special Topic 25.1.

**R25.18** Design a DTD for arithmetic expressions, as described in Special Topic 25.1.

---

**E25.1** Write a program that can read XML files, such as

```
<purse>
  <coin>
    <value>0.5</value>
    <name>half dollar</name>
  </coin>
  ...
</purse>
```

Your program should construct a Purse object and print the total value of the coins in the purse.

**E25.2** Building on Exercise E25.1, make the program read an XML file as described in that exercise. Then print an XML file of the form

```
<purse>
  <coins>
    <coin>
      <value>0.5</value>
      <name>half dollar</name>
    </coin>
    <quantity>3</quantity>
  </coins>
  <coins>
    <coin>
      <value>0.25</value>
      <name>quarter</name>
    </coin>
    <quantity>2</quantity>
  </coins>
</purse>
```

**E25.3** Repeat Exercise E25.1, using a DTD for validation.

**E25.4** Write a program that can read XML files, such as

```
<bank>
  <account>
    <number>3</number>
    <balance>1295.32</balance>
  </account>
</bank>
```
Your program should construct a Bank object and print the total value of the balances in the accounts.

**E25.5** Repeat Exercise E25.4, using a DTD for validation.

**E25.6** Enhance Exercise E25.4 as follows: First read the XML file in, then add ten percent interest to all accounts, and write an XML file that contains the increased account balances.

**P25.1** Write a DTD file that describes documents that contain information about countries: name of the country, its population, and its area. Create an XML file that has five different countries. The DTD and XML should be in different files. Write a program that uses the XML file you wrote 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 per square mile).

**P25.2** Write a parser to parse invoices using the invoice structure described in How To 25.1. The parser should parse the XML file into an Invoice object and print out the invoice in the format used in Chapter 12.

**P25.3** Modify Exercise P25.2 to support separate shipping and billing addresses. Supply a modified DTD with your solution.

**P25.4** Write a document builder that turns an invoice object, as defined in Chapter 12, into an XML file of the format described in How To 25.2.

**P25.5** Modify Exercise P25.4 to support separate shipping and billing addresses.

**Graphics P25.6** Write a program that can read an XML document of the form

```xml
<rectangle>
  <x>5</x>
  <y>10</y>
  <width>20</width>
  <height>30</height>
</rectangle>
```

and draw the shape in a window.

**Graphics P25.7** Write a program that can read an XML document of the form

```xml
<ellipse>
  <x>5</x>
  <y>10</y>
  <width>20</width>
  <height>30</height>
</ellipse>
```

and draw the shape in a window.
**Graphics P25.8** Write a program that can read an XML document of the form

```xml
<rectangularshape shape="ellipse">
  <x>5</x>
  <y>10</y>
  <width>20</width>
  <height>30</height>
</rectangularshape>
```

Support shape attributes "rectangle", "roundrectangle", and "ellipse".
Draw the shape in a window.

**Graphics P25.9** Write a program that can read an XML document of the form

```xml
<polygon>
  <point>
    <x>5</x>
    <y>10</y>
  </point>
  ...
</polygon>
```

and draw the shape in a window.

**Graphics P25.10** Write a program that can read an XML document of the form

```xml
<drawing>
  <rectangle>
    <x>5</x>
    <y>10</y>
    <width>20</width>
    <height>30</height>
  </rectangle>
  <line>
    <x1>5</x1>
    <y1>10</y1>
    <x2>25</x2>
    <y2>40</y2>
  </line>
  <message>
    <text>Hello, World!</text>
    <x>20</x>
    <y>30</y>
  </message>
</drawing>
```

and show the drawing in a window.

**Graphics P25.11** Repeat Exercise P25.10, using a DTD for validation.

**P25.12** Following Exercise P12.2, design an XML format for the appointments in an appointment calendar. Write a program that first reads in a file with appointments, then another file of the format

```xml
<commands>
  <add>
    <appointment>
      ...
    </appointment>
  </add>
  ...
  <remove>
    ...
  </remove>
</commands>
```
Your program should process the commands and then produce an XML file that consists of the updated appointments.

ANSWERS TO SELF-CHECK QUESTIONS

1. Your answer should look similar to this:
   
   <student>
       <name>James Bond</name>
       <id>007</id>
   </student>

2. Most browsers display a tree structure that indicates the nesting of the tags. Some browsers display nothing at all because they can’t find any HTML tags.

3. The text hamster.jpg is never displayed, so it should not be a part of the document. Instead, the src attribute tells the browser where to find the image that should be displayed.

4. 29.95.

5. name(/*[1]).

6. The createTextElement method is useful for creating other documents.

7. First construct a string, as described, and then use a PrintWriter to save the string to a file.

8. <!ELEMENT item (product, quantity?)>

9. <!ELEMENT product ((description, price) | (price, description))>

10. <!ATTLIST description language CDATA #IMPLIED>
WEB APPLICATIONS

CHAPTER GOALS

To understand the web application concept
To learn the syntactical elements of the JavaServer Faces web application framework
To manage navigation in web applications
To build three-tier web applications

CHAPTER CONTENTS

26.1 THE ARCHITECTURE OF A WEB APPLICATION  W1100
26.2 THE ARCHITECTURE OF A JSF APPLICATION  W1102
   ST1 Session State and Cookies  W1107
26.3 JAVABEANS COMPONENTS  W1108
26.4 NAVIGATION BETWEEN PAGES  W1109
   HT1 Designing a Managed Bean  W1115
26.5 JSF COMPONENTS  W1116
26.6 A THREE-TIER APPLICATION  W1118
   ST2 AJAX  W1125
Web applications for a wide variety of purposes, such as e-mail, banking, shopping, and playing games, run on servers and interact with users through a web browser. Developing web-based user interfaces is more complex and challenging than writing graphical user interfaces. Fortunately, frameworks for web programming have emerged that are roughly analogous to Java’s Swing framework for user-interface programming. In this chapter, you will learn how to write web applications using the JavaServer Faces (JSF) framework.

### 26.1 The Architecture of a Web Application

A **web application** is an application whose user interface is displayed in a web browser. The application program resides on the web server. The user fills out form elements and clicks on buttons and links. The user inputs are transmitted over the Internet to the server, and the server program updates the web page that the user sees (see Figure 1).

The browser sends a request to the server using a protocol called HTTP (Hyper-text transfer protocol). When a user clicks on a link, the request is very simple. The browser simply asks the server for the page with a given address, for example:

```
GET /index.html HTTP/1.1
Host: horstmann.com
```

When the user fills data (such as a user name and password) into a form and then clicks on a button, the HTTP request includes the data that the user provided. Such a request has a slightly different format, like this:

```
POST /login.xhtml HTTP/1.1
Host: horstmann.com
Content-Type: application/x-www-form-urlencoded
Content-Length: 46

username=jqpublic&passwd=secret&login=Log%20in
```

The exact syntax of the request is not important; what matters is that HTTP simply tells the server what the user requested. As a result of the request, the server sends a web page in a format called HTML (Hyper-text markup language). An HTML page contains tags that describe the structure of the page: headings, bullets, links, images, input elements, and so on.

![Internet Architecture Diagram](image)

**Figure 1** The Architecture of a Web Application
For example, here is the HTML code for a simple form that prompts for a user name and password:

```html
<html>
  <head>
    <title>A Simple Form</title>
  </head>
  <body>
    <form action="login.xhtml" method="POST">
      <p>
        User name:  
        <input type="text" name="username" />
        Password: 
        <input type="password" name="passwd" />
        <input type="submit" name="login" value="Log in"/>
      </p>
    </form>
  </body>
</html>
```

Figure 2 shows the form. Note that there are three input elements: a text field, a password field, and a submit button. (The HTML tags are summarized in Appendix J.)

When a submit button is pressed, the form data is submitted to the server. The web server analyzes the request and sends a new HTML page to the browser. The new page might tell the user that the login was successful and ask the user to specify another action. Alternatively, the new page might tell the user that the login failed.

This simple example illustrates why it is difficult to implement a web application. Imagine what the server program has to do. At any time, it might receive a request with form data. At that point, the server program has to remember which form it has last sent to the client. It then needs to analyze the submitted data, decide what form to show next, and produce the HTML tags for that form.

There are multiple challenges. As described in Special Topic 26.1, the HTTP protocol is stateless — there is no memory of which form was last sent when a new request is received. Generating the HTML tags for a form is tedious. Perhaps most importantly, an application that consists of response strategies for a large number of request types is very hard to comprehend without additional structure.

In order to overcome these challenges, various web application frameworks have been developed. A web application framework hides the low-level details of analyzing HTTP and generating HTML from the application programmer. In this chapter, you will learn about the JavaServer Faces (JSF) framework, the web framework that is a part of the Java Enterprise Edition. You can think of JSF as “Swing for the Web”. Both Swing and JSF handle the tedious details of capturing user input and painting...
text fields and buttons. Swing captures mouse and keyboard events and paints pixels in a frame. JSF handles form-posting events and paints by emitting HTML code. This chapter describes JSF 2.0, an improved version of the original JSF framework, that became available in 2009.

1. Why are two different protocols (HTML and HTTP) required by a web application?
2. How can a web application know which user is trying to log in when the information of the sample login screen is submitted?

Practice It Now you can try these exercises at the end of the chapter: R26.1, R26.2.

26.2 The Architecture of a JSF Application

In the following sections, we give an overview of the architecture of a JSF application and show a very simple sample application.

26.2.1 JSF Pages

The user interface of a JSF application is described by a set of JSF pages. Each JSF page has the following structure:

```xml
<?xml version="1.0" encoding="UTF-8"?>
<html xmlns="http://www.w3.org/1999/xhtml"
     xmlns:h="http://java.sun.com/jsf/html">
    <h:head>
      <title>Page title</title>
    </h:head>
    <h:body>
      <h:form>
        Page contents
      </h:form>
    </h:body>
</html>
```

You can think of this as the required “plumbing”, similar to the public static void main incantation that is required for every Java program. If you compare this page with the HTML page from the preceding section, you will notice that the main elements are very similar to a regular HTML page, but several elements (head, body, and form) are JSF tags with an h: prefix.

Here is a complete example of a JSF page:

```xml
<?xml version="1.0" encoding="UTF-8"?>
<html xmlns="http://www.w3.org/1999/xhtml"
     xmlns:h="http://java.sun.com/jsf/html">
    <h:head>
      <title>The time application</title>
    </h:head>
    <h:body>
      <h:form>
        <h:outputText value="The current time is..."/>
      </h:form>
    </h:body>
</html>
```
26.2 The Architecture of a JSF Application

The current time is #{timeBean.time}

The purpose of a JSF page is to generate an HTML page. The basic process is as follows:

- The HTML tags that are present in the JSF page (such as title and p) are retained. These are the static part of the page: the formatting instructions that do not change.
- The JSF tags are translated into HTML. This translation is dynamic: it depends on the state of Java objects that are associated with the tags. In our example, the expression #{timeBean.time} has been replaced by dynamically generated text, namely the current time.

Figure 4 shows the basic process. The browser requests a JSF page. The page is processed by the JSF container, the server-side software that implements the JSF framework. The JSF container translates all JSF tags into text and HTML tags, yielding a pure HTML page. That page is transmitted to the client browser. The browser displays the page.
The expression #{timeBean.time} is called a **value expression**. Value expressions invoke method calls on Java objects, which are called *managed beans*. These objects are called “managed” because they are controlled by the JSF container. The container creates a managed bean when it is first used in a value expression. The scope of the managed bean determines which clients can access the object and how long the object stays alive.

In this chapter, we only consider managed beans with *session scope*. A session-scoped object can be accessed by all requests from the same browser. If multiple users are simultaneously accessing a JSF application, each of them is given a separate object. This is a good default for simple web applications.

Below is the code for the `TimeBean` class. Note the following:

- You declare a session-scoped managed bean with the annotations `@ManagedBean` and `@SessionScoped`.
- The name of the bean in a value expression is the class name with the first letter changed to lowercase, e.g., `timeBean`.
- The value expression `timeBean.time` calls the `getTime` method. You will see the reason in the next section.
- The `getTime` method uses the `DateFormat` class to format the current time, producing a string such as `9:00:00 AM`.
- When deploying the application, all class files must be placed inside the `WEB-INF/classes` directory. Because many application servers also require that classes be contained in a package, we place our classes inside the `bigjava` package. For that reason, the class is contained in the `WEB-INF/classes/bigjava` directory.

```java
package bigjava;

import java.text.DateFormat;
import java.util.Date;
import java.util.TimeZone;
import javax.faces.bean.ManagedBean;
import javax.faces.bean.SessionScoped;

@ManagedBean
@SessionScoped
public class TimeBean {
    private DateFormat timeFormatter;

    /**
     * Initializes the formatter.
     */
    public TimeBean() {
        timeFormatter = DateFormat.getTimeInstance();
    }

    /**
     * Read-only time property.
     */
    @return the formatted time
```
26.2.3 Separation of Presentation and Business Logic

We will look at value expressions and managed beans in more detail in the next section. The key observation is that every JSF application has two parts: presentation and business logic.

The term “presentation” refers to the user interface of the web application: the arrangement of the text, images, buttons, and so on. The business logic is the part of the application that is independent of the visual presentation. In commercial applications, it contains the rules that are used for business decisions: what products to offer, how much to charge, to whom to extend credit, and so on. In our example, we simulated the business logic with a TimeBean object.

JSF pages define the presentation logic. Managed beans define the business logic. Value expressions tie the two together.

The separation of presentation logic and business logic is very important when designing web applications. Some web technologies place the code for the business logic right into the web page. However, this quickly turns into a serious problem. Programmers are rarely skilled in web design (as you can see from the boring web pages in this chapter). Graphic designers don’t usually know much about programming and find it very challenging to improve web pages that contain a lot of code. JSF solves this problem. In JSF, the graphic designer only sees the elements that make up the presentation logic. It is easy to take a boring JSF page and make it pretty by adding banners, icons, and so on.

26.2.4 Deploying a JSF Application

To run a JSF application, you need a server with a JSF container. We suggest that you use the GlassFish application server, http://glassfish.java.net, which has, together with many other features that you can ignore, a JSF container and a convenient administration interface.

To deploy a JSF application, follow these steps:

1. Make a separate directory tree for each web application.
2. Place JSF pages (such as index.xhtml) into the root directory of the application’s directory tree.
3. Create a WEB-INF subdirectory in your application directory.
4. Place all Java classes inside a classes subdirectory of the WEB-INF directory. Note that you should place your classes into a package. Compile with

   ```bash
cd WEB-INF/classes
   javac -classpath glassfish/modules/jsf-api.jar bigjava/*.java
   ```
5. Place the file `web.xml` (which is shown below) inside the `WEB-INF` subdirectory. Some servers need the `web.xml` file to configure the JSF container. We also turn on development mode, which gives better error messages.

6. Zip up all application files into a file with extension `.war` (Web Archive). This is easily achieved by running the `jar` command from the command line, after changing to the application directory. For example,
```
cd time
jar cvf time.war .
```
The period (.) denotes the current directory. The `jar` command creates an archive `time.war` consisting of all files in all subdirectories of the current directory.

7. Make sure the application server is started. The application server listens to web requests, typically on port 8080.

8. Deploy the application to the application server. With GlassFish, this can be achieved either through the administrative interface or simply by copying the WAR file into a special deployment directory. By default, this is the subdirectory `domains/domain1/autodeploy` inside the GlassFish installation directory.

9. Point your browser to a URL such as `http://localhost:8080/time/faces/index.xhtml`. Note the `faces` part in the URL. If you forget this part, the file will not be processed by the JSF container.

Figure 5 shows the directory structure for the application.

![Figure 5](image)

**Figure 5**
The Directory Structure of the time Application

```xml
<?xml version="1.0" encoding="UTF-8"?>
<web-app xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns="http://java.sun.com/xml/ns/javaee"
xmlns:web="http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd"
xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd"
version="2.5">
<servlet>
<servlet-name>Faces Servlet</servlet-name>
<servlet-class>javax.faces.webapp.FacesServlet</servlet-class>
</servlet>
<servlet-mapping>
<servlet-name>Faces Servlet</servlet-name>
<url-pattern>/faces/*</url-pattern>
</servlet-mapping>
<welcome-file-list>
<welcome-file>faces/index.xhtml</welcome-file>
</welcome-file-list>
<context-param>
<param-name>javax.faces.PROJECT_STAGE</param-name>
```
3. What steps are required to add the image of a clock to the time application? (The clock doesn’t have to show the correct time.)

4. Does a Swing program automatically separate presentation and business logic?

5. Why does the WAR file need to be deployed to the application server?

**Practice It** Now you can try these exercises at the end of the chapter: R26.3, R26.7, E26.1.

**Session State and Cookies**

Recall that HTTP is a stateless protocol. A browser sends a request to a web server. The web server sends the reply and then disconnects. This is different from other protocols, such as POP, where the mail client logs into the mail server and stays connected until it has retrieved all e-mail messages. In contrast, a browser makes a new connection to the web server for each web page, and the web server has no way of knowing that those connections originate from the same browser. This makes it difficult to implement web applications. For example, in a shopping application, it is essential to track which requests came from a particular shopper.

Cookies were invented to overcome this restriction. A cookie consists of a small string that the web server sends to a browser, and that the browser sends back to the same server with all further requests. That way, the server can tie the stream of requests together. The JSF container matches up the cookies with the beans that have session scope. When a browser request contains a cookie, the value expressions in the JSF page refer to the matching beans.

![Figure 6 Viewing the Cookies in a Browser](image-url)
You may have heard some privacy advocates complaining about cookies. Cookies are not inherently evil. When used to establish a session or to remember login information, they can make web applications more user-friendly. But when cookies are used to track your identity while you surf the Web, there can be privacy concerns. For example, Figure 6 shows some of the cookies that my browser held on a particular day. I have no recollection of visiting the advertising sites, so it is a bit disconcerting to see that my browser communicated with them.

Some people turn off cookies, and then web applications need to use another scheme to establish a session, typically by embedding a session identifier in the request URL or in a hidden field of a form. The JSF session mechanism automatically switches to URLs with session identifiers if the client browser doesn’t support cookies.

### 26.3 JavaBeans Components

A software component is an entity that encapsulates functionality and can be plugged into a software system without programming. A managed bean is an example of a software component. When we added the timeBean object to the web application, we did not write Java code to construct the object or to call its methods.

Some programming languages have explicit support for components, but Java does not. Instead, in Java, you use a programming convention to implement components. A JavaBean is a Java class that follows this convention. A JavaBean exposes properties—values of the component that can be accessed without programming.

Just about any Java class can be a JavaBean—there are only two requirements:

- A JavaBean must have a constructor with no arguments.
- A JavaBean must have methods for accessing the component properties that follow the get/set naming convention. For example, to get or set a property named city, the methods must be called getCity and setCity.

In general, if the name of the property is propertyName, and its type is Type, then the associated methods must be of the form

```java
public Type getPropertyName()
public void setPropertyName(Type newValue)
```

Note that the name of a property starts with a lowercase letter (such as city), but the corresponding methods have an uppercase letter (getCity). The only exception is that property names can be all capitals, such as ID or URL, with corresponding methods getID or setURL.

If a property has only a get method, then it is a read-only property. If it has only a set method, then it is a write-only property.

A JavaBean can have additional methods, but they are not connected with properties.

Here is a simple example of a bean class that formats the time for a given city, which we will develop further in the next section:

```java
public class TimeZoneBean
{
    // Instance variables
    .

    // Required constructor with no arguments
    public TimeZoneBean() { . . . }
```
This bean has two properties: city and time.

You should not make any assumptions about the internal representation of properties in the bean class. The getter and setter methods may simply read or write an instance variable. But they may also do other work. An example is the getTime method from the TimeBean in the preceding section; it formats the current time.

When a property name is used in a value expression that is included in the JSF page, then the get method is involved. For example, when the string

```
The current time is #{timeBean.time}
```

is rendered, the JSF container calls the getTime method of the session’s TimeBean instance.

When a property name is used in an h:inputText tag (that is, the equivalent of an HTML input field or a JTextField), the situation is more complex. Consider this example:

```
<h:inputText value="#{timeZoneBean.city}"/>
```

When the JSF page is first displayed, the getCity method is called, and the current value of the city property is displayed. But after the user submits the page, the setCity method is called. It sets the city property to the value that the user typed into the input field.

### Practice It

Now you can try these exercises at the end of the chapter: R26.5, R26.6, E26.2.

## 26.4 Navigation Between Pages

In most web applications, users will want to move between different pages. For example, a shopping application might have a login page, a page to show products for sale, and a checkout page that shows the shopping cart. In this section, you will learn how to enable users to navigate from one page to another.

Consider a sample time zone program that displays the current time. If the time computation uses the time zone at the server location, it will not be very useful when the user is in another time zone. Therefore, the program will prompt for the city in which the user is located. When the user clicks a submit button, the program moves to the page next.xhtml and displays the time in the user’s time zone (see Figure 7). However, if no time zone is available for the city, the program displays the page error.xhtml.
A button yields an outcome, a string that determines the next page. Unless specified otherwise, the next page is the outcome string with the .xhtml extension added. For example, if the outcome string is error, the next page is error.xhtml. (It is possible to specify a different mapping from outcomes to pages, but there is no need to do so for a simple application.)

In many situations, the next page depends on the result of some computation. In our example, we need different outcomes depending on the city that the user entered. To achieve this flexibility, you specify a method expression as the action attribute:

```html
<h:commandButton value="Submit" action="#{timeZoneBean.checkCity}"/>
```

A method expression consists of the name of a bean and the name of a method. When the form is submitted, the JSF container calls timeZoneBean.checkCity(). The checkCity method returns the outcome string:

```java
public class TimeZoneBean {
    ...
    public String checkCity() {
        zone = getTimeZone(city);
        if (zone == null) { return "error"; }
        return "next";
    }
}
```

Figure 7 The timezone Application
If the next page does not depend on a computation, then you set the `action` attribute of the button to a fixed outcome string, like this:

```xml
<h:commandButton value="Back" action="index"/>
```

If a button has no `action` attribute, or if the action outcome is `null`, then the current page is redisplayed.

We can now complete our time zone application. The Java library contains a convenient `TimeZone` class that knows about time zones across the world. A time zone is identified by a string such as "America/Los_Angeles" or "Asia/Tokyo". The static method `getAvailableIDs` returns a string array containing all IDs:

```java
String[] ids = TimeZone.getAvailableIDs();
```

There are several hundred time zone IDs. (We are using time zones in this example because the `TimeZone` class gives us an interesting data source with lots of data. Later in this chapter, you will see how to access data from a database, but of course that's more complex.)

The static `getTimeZone` method returns a `TimeZone` object for a given ID string:

```java
String id = "America/Los_Angeles";  
TimeZone zone = TimeZone.getTimeZone(id);
```

Once you have a `TimeZone` object, you can use it in conjunction with a `DateFormat` object to get a time string in that time zone.

```java
DateFormat timeFormatter = DateFormat.getTimeInstance();  
timeFormatter.setTimeZone(zone);  
Date now = new Date();  
// Suppose the server is in New York, and it's noon there  
System.out.println(timeFormatter.format(now));  
// Prints 9:00:00 AM
```

Of course, we don’t expect the user to know about time zone ID strings, such as "America/Los_Angeles". Instead, we assume that the user will simply enter the city name. The time zone bean will check whether that string, with spaces replaced by underscores, appears at the end of one of the valid time zone IDs.

Here is the code for the bean class:

```java
package bigjava;  
import java.text.DateFormat;  
import java.util.Date;  
import java.util.TimeZone;  
import javax.faces.bean.ManagedBean;  
import javax.faces.bean.SessionScoped;  
/**
  * This bean formats the local time of day for a given city.
  */  
@ManagedBean  
@SessionScoped  
public class TimeZoneBean  
{  
  private DateFormat timeFormatter;  
  private String city;  
  private TimeZone zone;
  /**
```
Initializes the formatter.

```java
public TimeZoneBean()
{
    timeFormatter = DateFormat.getTimeInstance();
}
```

Setter for `city` property.
@throws aCity the city for which to report the local time

```java
public void setCity(String aCity)
{
    city = aCity;
}
```

Getter for `city` property.
@returns the city for which to report the local time

```java
public String getCity()
{
    return city;
}
```

Read-only time property.
@return the formatted time

```java
public String getTime()
{
    if (zone == null) { return "not available"; }
    timeFormatter.setTimeZone(zone);
    Date time = new Date();
    String timeString = timeFormatter.format(time);
    return timeString;
}
```

Action for checking a city.
@return "next" if time zone information is available for the city,
"error" otherwise

```java
public String checkCity()
{
    zone = getTimeZone(city);
    if (zone == null) { return "error"; }
    return "next";
}
```

Looks up the time zone for a city.
@param aCity the city for which to find the time zone
@return the time zone or null if no match is found

```java
private static TimeZone getTimeZone(String aCity)
{
    String[] ids = TimeZone.getAvailableIDs();
    for (int i = 0; i < ids.length; i++)
    {
```
if (timeZoneIDmatch(ids[i], aCity))
{
    return TimeZone.getTimeZone(ids[i]);
}
return null;

/**
 * Checks whether a time zone ID matches a city.
 * @param id the time zone ID (e.g., "America/Los_Angeles")
 * @param aCity the city to match (e.g., "Los Angeles")
 * @return true if the ID and city match
 */
private static boolean timeZoneIDmatch(String id, String aCity)
{
    String idCity = id.substring(id.indexOf('/') + 1);
    return idCity.replace('_', ' ').equals(aCity);
}

Following is the JSF page for setting the city. The h:inputText tag produces an input field and the h:commandButton tag produces a button. (We discuss its action attribute in the next section.) When the user clicks the button, the browser sends the form values (that is, the contents of the input field) back to the web application. The web application calls the setCity method on the bean because the input field has a #{timeZoneBean.city} value expression.

section_4/timezone/index.xhtml

The next JSF page shows the result, using two value expressions that display the city and time properties. These expressions invoke the getCity and getTime methods of the bean class.

section_4/timezone/next.xhtml
Figure 8 shows the directory structure of the timezone application.

### Self Check

8. What tag would you need to add to `error.xhtml` so that the user can click on a button labeled “Help” and see `help.xhtml`?

9. Which page would be displayed if the `checkCity` method returned `null` instead of “error”?

### Practice It

Now you can try these exercises at the end of the chapter: R26.10, E26.4, E26.5.
HOW TO 26.1  Designing a Managed Bean

A managed bean is just a regular Java class, with these three special characteristics:

- The bean must have a constructor with no arguments.
- Methods of the form
  
  ```java
  Type getPropertyName()
  void setPropertyName(Type x)
  ```
  define properties that can be accessed from JSF pages.
- Methods of the form
  ```java
  String methodName()
  ```
  can be used to specify command actions.

This How To provides step-by-step instructions for designing a managed bean class.

Step 1  Decide on the responsibility of the bean.

When designing a JSF application, it is tempting to stuff all code into a single bean class. Some development environments even encourage this approach. However, from a software engineering perspective, it is best to come up with different beans for different responsibilities. For example, a shopping application might have a UserBean to describe the current user, a SiteBean to describe how the user visits the shopping site, and a ShoppingCartBean that holds the items that the user is purchasing.

Step 2  Discover the properties that the bean should expose.

A property is an entity that you want to access or modify from your JSF pages. For example, a UserBean might have properties firstName, lastName, and password.

Sometimes, you have to resort to a bit of trickery. For example, consider adding an item to the shopping cart. You could use a property items, but it would be cumbersome to access all items in a JSF page and then set items to a new collection that contains one additional element. Instead, you can design a property addedItem. When that property is set, the setAddedItem method of your bean adds its value to the collection of items.

Step 3  Settle on the type and access permissions for each property.

Properties that are only used to generate output can be read-only. Properties that are used in h:inputText and other input tags must have read-write access.

Step 4  Define action methods for navigation.

Your action methods can carry out arbitrary tasks in order to react to the user inputs. The only limitation is that they don’t have access to the form data. Everything that the user entered on the form must have already been set as a bean property.

The action method’s return value is the name of the next page to be displayed, or null if you want to redisplay the current page.

Step 5  Implement the constructor with no arguments.

The constructor initializes any instance variables that are reused whenever the bean’s computation is executed. Examples are formatters, random number generators, and so on.

Step 6  Implement the get and set methods for all properties.

Most get and set methods simply get or set an instance variable. However, you can carry out arbitrary computations in these methods if it is convenient. For example, a get method may retrieve information from a database instead of an instance variable.
Step 7 Supply any needed helper methods.

Your bean can have methods that are not property getters and setters. For example, the \texttt{TimeZoneBean} has helper methods to look up the time zone for a city.

## 26.5 JSF Components

There are JSF components for text input, choices, buttons, and images.

The value attribute of an input component denotes the value that the user supplies.

In this section, you will see the most useful user-interface components that you can place on a JSF form. Table 1 shows a summary. (For a comprehensive discussion of all JSF components, see \textit{Core JavaServer Faces}, 3rd ed., by David Geary and Cay Horstmann (Sun Microsystems Press/Prentice Hall, 2010)).

Each component has a value attribute that allows you to connect the component value with a bean property, for example

\begin{verbatim}
    <h:inputSecret value="#{user.password}"/>
\end{verbatim}

The \texttt{h:inputTextArea} component has attributes to specify the rows of text and columns of characters, such as

\begin{verbatim}
    <h:inputTextArea value="#{user.comment}" rows="10" cols="40"/>
\end{verbatim}

<table>
<thead>
<tr>
<th>Component</th>
<th>JSF Tag</th>
<th>Common Attributes</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>Text Field</td>
<td>\texttt{h:inputText}</td>
<td>value</td>
<td></td>
</tr>
<tr>
<td>Password Field</td>
<td>\texttt{h:inputSecret}</td>
<td>value</td>
<td></td>
</tr>
<tr>
<td>Text Area</td>
<td>\texttt{h:inputTextArea}</td>
<td>value, rows, cols</td>
<td></td>
</tr>
<tr>
<td>Radio Button Group</td>
<td>\texttt{h:selectOneRadio}</td>
<td>value, layout</td>
<td></td>
</tr>
<tr>
<td>Checkbox</td>
<td>\texttt{h:selectOneCheckbox}</td>
<td>value</td>
<td></td>
</tr>
<tr>
<td>Checkbox Group</td>
<td>\texttt{h:selectManyCheckbox}</td>
<td>value, layout</td>
<td></td>
</tr>
<tr>
<td>Menu</td>
<td>\texttt{h:selectOneMenu} \texttt{h:selectManyMenu}</td>
<td>value</td>
<td></td>
</tr>
<tr>
<td>Image</td>
<td>\texttt{h:graphicImage}</td>
<td>value</td>
<td></td>
</tr>
<tr>
<td>Submit Button</td>
<td>\texttt{h:commandButton}</td>
<td>value, action</td>
<td></td>
</tr>
</tbody>
</table>
The radio button and checkbox groups allow you to specify horizontal or vertical layout:

```html
<h:selectOneRadio value="#{burger.topping}" layout="lineDirection">
```

In European languages, `lineDirection` means horizontal and `pageDirection` means vertical. However, in some languages, lines are written top-to-bottom, and the meanings are reversed.

Button groups and menus are more complex than the other user-interface components. They require you to specify two properties:

- the collection of possible choices
- the actual choice

The `value` attribute of the component specifies the actual choice to be displayed. The collection of possible choices is defined by a nested `f:selectItems` tag, like this:

```html
<h:selectOneRadio value="#{creditCardBean.expirationMonth}" layout="pageDirection">
    <f:selectItems value="#{creditCardBean.monthChoices}"/>
</h:selectOneRadio>
```

When you use the `f:selectItems` tag, you need to add the namespace declaration

```html
xmlns:f="http://java.sun.com/jsf/core"
```

to the `html` tag at the top of your JSF page.

The value of the `f:selectItems` tag must have a type that can describe a list of choices. There are several types that you can use, but the easiest—and the only one that we will discuss—is a `Map`. The keys of the map are the `labels`—the strings that are displayed next to each choice. The corresponding map values are the `label values`—the values that correspond to the selection. For example, a choice map for months would map January to 1, February to 2, and so on:

```java
public class CreditCardBean
{
    private Map<String, Integer> monthChoices;

    public Map<String, Integer> getMonthChoices()
    {
        Map<String, Integer> choices = new LinkedHashMap<>();
        choices.put("January", 1);
        choices.put("February", 2);
        // ... return choices;
    }
}
```

Here, we use a `LinkedHashMap` because we want to visit entries in the order in which they are inserted. This is more useful than a `HashMap`, which would visit the labels in random order or a `TreeMap`, which would visit them in alphabetical order (starting with April!).

The type of the `value` property of the component enclosing the `f:selectItems` tag must match the type of the map value. For example, `creditCardBean.expirationMonth` must be an integer, not a string. If multiple selections are allowed, the type of the `value` property must be a list or array of matching types. For example, if one could choose multiple months, a `selectManyRadio` component would have a `value` property with a type such as `int[]` or `ArrayList<Integer>`.
10. Which JSF components can be used to give a user a choice between “AM/PM” and “military” time?

11. How would you supply a set of choices for a credit card expiration year to a `h:selectOneMenu` component?

**Practice It**
Now you can try these exercises at the end of the chapter: R26.11, E26.3, P26.2.

### 26.6 A Three-Tier Application

In this chapter’s final JSF example, you will see a web application with a very common structure. In this example, we will use a database for information storage. We will enhance the time zone example by storing additional cities that are not known to the `TimeZone` class in a database. Such an application is called a **three-tier application** because it consists of three separate layers or tiers (see Figure 9):

- The presentation tier: the web browser
- The “business logic” tier: the JSF container, the JSF pages, and the JavaBeans
- The storage tier: the database

Contrast the three-tier architecture with the more traditional **client-server** or **two-tier architecture** that you saw in the database programs of Chapter 24. In that architecture, one of the tiers is the database server, which is accessed by multiple client programs on desktops. Each client program has a presentation layer—usually with a specially programmed graphical user interface—and business logic code. (See Figure 10.) When the business logic changes, a new client program must be distributed over all desktops. In contrast, in a three-tier application, the business logic resides on a server. When the logic changes, the server code is updated, but the presentation tier—the browser—remains unchanged. That is much simpler to manage than updating multiple desktops.

In our example, we will have a single database table, `CityZone`, with city and time zone names (see Figure 11).

```
section_6/multizone/sql/CityZone.sql
```

```
1 CREATE TABLE CityZone (City VARCHAR(40), Zone VARCHAR(40))
2 INSERT INTO CityZone VALUES ('San Francisco', 'America/Los_Angeles')
3 INSERT INTO CityZone VALUES ('Hamburg', 'Europe/Rome')
4 SELECT * FROM CityZone
```

![Figure 9 Three-Tier Architecture](image)
If the `TimeZoneBean` can’t find the city among the standard time zone IDs, it makes a database query:

```sql
SELECT Zone FROM CityZone WHERE City = the requested city
```

If there is a matching entry in the database, that time zone is returned.

To query the database, the bean needs a `Connection` object. In Chapter 24, we used the static `getConnection` method of the `DriverManager` class to obtain a database connection. However, JSF containers have a better mechanism for configuring a database in one central location so that multiple web applications can access it.

The GlassFish application server includes the Derby database. It has a predefined data source with the resource name `jdbc/__default`. In your bean code, you declare an instance variable of type `DataSource` and tag it with a `@Resource` annotation, like this:

```java
@Resource(name="jdbc/__default")
private DataSource source;
```

You can use the administrative interface of GlassFish to define other data sources.

When the application server loads the web application, it automatically initializes this instance variable. Whenever you need a database connection, call

```java
try (Connection conn = source.getConnection()) {
  Use the connection.
}
```

The application server provides an additional service: it pools database connections. When a pooled connection is closed, it is not physically terminated but instead returned to a queue and given out again to another caller of the `getConnection` method. Pooling avoids the overhead of creating new database connections. In a web application, it would be particularly inefficient to connect to the database with every web request. Connection pooling is completely automatic.

### CityZone

<table>
<thead>
<tr>
<th>City</th>
<th>Zone</th>
</tr>
</thead>
<tbody>
<tr>
<td>San Francisco</td>
<td>America/Los_Angeles</td>
</tr>
<tr>
<td>Hamburg</td>
<td>Europe/Rome</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

**Figure 11** The CityZone Table
In order to make the application more interesting, we enhanced the TimeZoneBean so that it manages a list of cities. You can add cities to the list and remove a selected city (see Figure 12).

You will find the code for this web application below. Figure 13 shows the directory structure of the application.

You have now seen how to use the JavaServer Faces technology to build web applications. JSF takes care of low-level details so that you don’t have to think about HTML forms and the HTTP protocol. Instead, you can focus on the presentation and business logic of your application.

```
<?xml version="1.0" encoding="UTF-8"?>
<html xmlns="http://www.w3.org/1999/xhtml"
     xmlns:h="http://java.sun.com/jsf/html">
  <h:head>
    <title>The multizone application</title>
  </h:head>
  <h:body>
    <h:form>
      <p>Enter city:<br/>
        <h:inputText value="#{timeZoneBean.cityToAdd}"/>
      </p>
      <h:commandButton value="Submit"
                        action="#{timeZoneBean.addCity}"/>
    </h:form>
  </h:body>
</html>
```
A Three-Tier Application

section_6/multizone/next.xhtml

```xml
<?xml version="1.0" encoding="UTF-8"?>
<html xmlns="http://www.w3.org/1999/xhtml"
xmlns:f="http://java.sun.com/jsf/core"
xmlns:h="http://java.sun.com/jsf/html">
<h:head>
<title>The multizone application</title>
</h:head>
<h:body>
<h:form>
<p>
<h:selectOneRadio value="#{timeZoneBean.cityToRemove}"
layout="pageDirection">
<f:selectItems value="#{timeZoneBean.citiesAndTimes}"/>
</h:selectOneRadio>
</p>
<p>
<h:commandButton value="Remove selected"
action="#{timeZoneBean.removeCity}"/>
<h:commandButton value="Add another" action="index"/>
</p>
</h:form>
</h:body>
</html>
```

section_6/multizone/error.xhtml

```xml
<?xml version="1.0" encoding="UTF-8"?>
<html xmlns="http://www.w3.org/1999/xhtml"
xmlns:h="http://java.sun.com/jsf/html">
<h:head>
<title>The multizone application</title>
</h:head>
<h:body>
<h:form>
<p>
Sorry, no information is available for #{timeZoneBean.cityToAdd}.
</p>
<p>
<h:commandButton value="Back" action="index"/>
</p>
</h:form>
</h:body>
</html>
```

section_6/multizone/WEB-INF/classes/bigjava/TimeZoneBean.java

```java
package bigjava;

import java.sql.Connection;
import java.sql.PreparedStatement;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.text.DateFormat;
import java.util.ArrayList;
```
import java.util.Date;
import java.util.Map;
import java.util.TimeZone;
import java.util.TreeMap;
import java.util.logging.Logger;
import javax.annotation.Resource;
import javax.faces.bean.ManagedBean;
import javax.faces.bean.SessionScoped;
import javax.sql.DataSource;

/**
 * This bean formats the local time of day for a given city.
 */
@ManagedBean
@SessionScoped
public class TimeZoneBean {
    @Resource(name="jdbc/__default")
    private DataSource source;

    private DateFormat timeFormatter;
    private ArrayList<String> cities;
    private String cityToAdd;
    private String cityToRemove;

    /**
     * Initializes the formatter.
     */
    public TimeZoneBean()
    {
        timeFormatter = DateFormat.getTimeInstance();
        cities = new ArrayList<>();
    }

    /**
     * Setter for cityToAdd property.
     * @param city the city to add to the list of cities
     */
    public void setCityToAdd(String city)
    {
        cityToAdd = city;
    }

    /**
     * Getter for cityToAdd property.
     * @return the city to add to the list of cities
     */
    public String getCityToAdd()
    {
        return cityToAdd;
    }

    /**
     * Setter for cityToRemove property.
     * @param city the city to remove from the list of cities
     */
    public void setCityToRemove(String city)
    {
        cityToRemove = city;
    }
26.6 A Three-Tier Application
69
70
71
72
73
74
75
76
77
78
79
80
81
82
83
84
85
86
87
88
89
90
91
92
93
94
95
96
97
98
99
100
101
102
103
104
105
106
107
108
109
110
111
112
113
114
115
116
117
118
119
120
121
122
123
124
125
126
127
128

/**

Getter for the cityToRemove property.
@return the city to remove from the list of cities

*/
public String getCityToRemove()
{
return cityToRemove;
}
/**

Read-only citiesAndTimes property.
@return a map containing the cities and formatted times

*/
public Map<String, String> getCitiesAndTimes()
{
Date time = new Date();
Map<String, String> result = new TreeMap<>();
for (int i = 0; i < cities.size(); i++)
{
String city = cities.get(i);
String label = city + ": ";
TimeZone zone = getTimeZone(city);
if (zone != null)
{
timeFormatter.setTimeZone(zone);
String timeString = timeFormatter.format(time);
label = label + timeString;
}
else
{
label = label + "unavailable";
}
result.put(label, city);
}
return result;
}
/**

Action for adding a city.
@return "next" if time zone information is available for the city,
"error" otherwise

*/
public String addCity()
{
TimeZone zone = getTimeZone(cityToAdd);
if (zone == null) { return "error"; }
cities.add(cityToAdd);
cityToRemove = cityToAdd;
cityToAdd = "";
return "next";
}
/**

Action for removing a city.
@return null if there are more cities to remove, "index" otherwise

*/
public String removeCity()
{

W1123


cities.remove(cityToRemove);
if (cities.size() > 0) { return null; }
else { return "index"; }

/**
 * Looks up the time zone for a city.
 * @param city the city for which to find the time zone
 * @return the time zone or null if no match is found
 */
private TimeZone getTimeZone(String city)
{
    String[] ids = TimeZone.getAvailableIDs();
    for (int i = 0; i < ids.length; i++)
    {
        if (timeZoneIDmatch(ids[i], city))
        {
            return TimeZone.getTimeZone(ids[i]);
        }
    }
    try
    {
        String id = getZoneNameFromDB(city);
        if (id != null)
        {
            return TimeZone.getTimeZone(id);
        }
    } catch (Exception ex)
    {
        Logger.global.info("Caught in TimeZone.getTimeZone: " + ex);
    }
    return null;
}

private String getZoneNameFromDB(String city)
throws SQLException
{
    if (source == null)
    {
        Logger.global.info("No database connection");
        return null;
    }
    try (Connection conn = source.getConnection())
    {
        PreparedStatement stat = conn.prepareStatement("SELECT Zone FROM CityZone WHERE City=?");
        stat.setString(1, city);
        ResultSet result = stat.executeQuery();
        if (result.next()) { return result.getString(1); } else { return null; }
    }
}

/**
Checks whether a time zone ID matches a city.
 * @param id the time zone ID (e.g., "America/Los_Angeles")
 * @param city the city to match (e.g., "Los Angeles")
 * @return true if the ID and city match
private static boolean timeZoneIDmatch(String id, String city)
{
    String idCity = id.substring(id.indexOf('/') + 1);
    return idCity.replace('_', ' ').equals(city);
}

12. Why don’t we just keep a database connection as an instance variable in the TimeZoneBean?

13. Why does the removeCity method of the TimeZoneBean return null or "index", depending on the size of the cities instance variable?

Practice It Now you can try these exercises at the end of the chapter: R26.12, E26.6, E26.7.

AJAX

In Section 26.1, you learned that a web application receives an HTTP request from the browser and then sends back an HTML form. The cycle repeats when the user submits the next form data. Web application designers and users dislike the “page flip”—the visual discontinuity between pages that is often accompanied by a significant delay, as the browser waits for the new form tags.

The AJAX (Asynchronous JavaScript and XML) technology, invented in 2005, aims to solve this problem. In an AJAX application, the browser does not merely display an HTML page, but it also executes code written in the JavaScript language. The JavaScript code continuously communicates with the server program and updates parts of the HTML page.

Figure 14 A Map Image with Partially-Fetched Tiles
One example of an AJAX application is the Google Maps™ mapping service. In a traditional map application, the user might click on a “move North” button and then wait until the browser receives the new map image and displays it in a new page. Map applications such as Google Maps use AJAX to fetch only the needed tiles, then fluidly rearrange the tiles in the current page without the dreaded page flip (see Figure 14).

AJAX applications are much more difficult to program than regular web applications. Frameworks are being proposed to handle these additional challenges. JSF 2 supports AJAX, giving the web application programmer the benefit of producing a pleasant user experience without having to worry about the intricate details of the JavaScript communication channel. The book’s companion code contains a modification of the multizone application that uses AJAX. When you click one of the buttons, the page is updated without a page flip.

**CHAPTER SUMMARY**

Describe the architecture of a web application.

- The user interface of a web application is displayed in a web browser.
- When a form is submitted, the names and values of the form elements are sent to the web server.
- Upon receiving the form data, the web server sends a new web page to the browser.

Describe the architecture of a JSF application.

- A JavaServer Faces (JSF) page contains HTML and JSF tags.
- The JSF container converts a JSF page to an HTML page, replacing all JSF tags with text and HTML tags.
- A managed bean is an object that is controlled by the JSF container.
- A bean with session scope is available for multiple requests by the same browser.
- The JSF technology enables the separation of presentation and business logic.

Explain how properties are defined in managed beans and accessed in value expressions.

- Properties of a software component can be accessed without having to write Java code.
- A JavaBean is a class that exposes properties through its `get` and `set` methods.
- In the value expression of an output tag, only the property getter is called.
- In the value expression of an input tag, the property setter is called when the page is submitted.

Implement navigation between pages.

- The outcome string of an action determines the next page that the JSF container sends to the browser.
- A method expression specifies a bean and a method that should be invoked on the bean.

Use common JSF components for designing a user interface.

- There are JSF components for text input, choices, buttons, and images.
• The value attribute of an input component denotes the value that the user supplies.
• Use an f:selectItems tag to specify all choices for a component that allows selection from a list of choices.

Develop applications that use JSF and a database.

• A three-tier application has separate tiers for presentation, business logic, and data storage.
• You define data sources in the JSF container and use resource annotations to initialize them.

STANDARD LIBRARY ITEMS INTRODUCED IN THIS CHAPTER

java.text.SimpleDateFormat
format
getTimeInstance
setTimeZone
java.util.LinkedHashMap
java.util.TimeZone
getAvailableIDs
getTimeZone
javax.sql.DataSource
getConnection

REVIEW EXERCISES

• R26.1 Most web browsers have a command to “view the source” of a web page. Load the page http://horstmann.com into your browser and view the source. What is the “language” used for formatting the source? What images, links, bullets, and input elements can you find?

• R26.2 Have a closer look at the HTTP POST request on page W1100. Where is the data that the user provided? What does login=Log%20in mean? (The code %20 denotes a space in the “URL encoding” scheme.)

• R26.3 What is the difference between a JSF page and a JSF container?

• R26.4 What is a bean?

• R26.5 What is a bean property?

• R26.6 Is a JButton a bean? Why or why not?

• R26.7 What is the software engineering purpose of using beans in conjunction with JSF pages?

• R26.8 How are variables in the JSF expression language different from variables in Java programs?

• R26.9 When is a bean constructed in a JSF application? Can you have two different instances of a bean that are active at the same time?

• R26.10 How can you implement error checking in a JSF application? Explain, using a login page as an example.

• R26.11 What input elements can you place on a JSF form? What are their Swing equivalents?

• R26.12 What is the difference between a client-server application and a three-tier application?
E26.1 Write a JSF application that reports the values of the following system properties of the web server:
- The Java version (java.version)
- The operating system name (os.name)
- The operating system version (os.version)
Supply a bean that uses the getProperties method of the System class.

E26.2 Write a JSF application that simulates two rolls of a die, producing an output such as “Rolled a 4 and a 6”. When the user reloads the page, a new pair of values should be displayed. Provide a bean that yields random numbers.

E26.3 Enhance Exercise E26.2 by producing a page that shows images of the rolled dice. Find GIF images of dice with numbers 1 through 6 on the front, and generate an HTML page that references the appropriate images. Hint: Use the tag <h:graphicImage value=imageURL/>, and take advantage of the fact that you can embed a value expression into regular text, such as “/image#{expression}.gif”.

E26.4 Write a web application that allows a user to specify six lottery numbers. Generate your own combination on the server, and then print out the user’s and the server’s combinations together with a count of matches.

E26.5 Add error checking to Exercise E26.4. If the lottery numbers are not within the correct range, or if there are duplicates, show an appropriate message and allow the user to fix the error.

E26.6 Personalize the time zone application of Section 26.3. Prompt the user to log in and specify a city to be stored in a profile. The next time the user logs in, the time of their favorite city is displayed automatically. Store users, passwords, and favorite cities in a database. You need a logout button to switch users.

E26.7 Extend Exercise E26.6 so that a user can choose multiple cities and all cities chosen by the user are remembered on the next login.

P26.1 Write a web version of the execSQL utility of Chapter 24. Allow users to type arbitrary SQL queries into a text area. Then submit the query to the database and display the result.

P26.2 Produce a web front end for the ATM program in Worked Example 12.1.

P26.3 Produce a web front end for the appointment calendar application of Exercise P12.2.

P26.4 Produce a web front end for the airline reservation program of Exercise P12.6.

Business P26.5 Write a shopping cart application. A database contains items that can be purchased and their prices, descriptions, and available quantities. If the user wants to check out, ask for the user account. If the user does not yet have an account, create one. The user name and address should be stored with the account in the database. Display an invoice as the last step in the check out process. When the user has confirmed the purchase, update the quantities in the warehouse.
Write a web-based grade book application that your instructor might use to manage student grades in this course. Your application should have one account for the instructor, and one account for each student. Instructors can enter and view grades for all students. Students can only see their own grades and their ranking within the course. Implement the features that your instructor uses for determining the course grade (such as dropping the lowest quiz score, counting homework as 30 percent of the total grade, and so on.) All information should be stored in a database.

ANSWERS TO SELF-CHECK QUESTIONS

1. Each protocol has a specific purpose. HTML describes the appearance of a page; it would be useless for sending requests from a browser to a server. HTTP describes a request; it cannot describe the appearance of a page.

2. The data of the POST request contain a portion username=the name supplied by the user&password=the password supplied by the user.

3. Place an image file, say clock.gif, into the time directory, and add a tag <img src="clock.gif"/> to the index.xhtml file.

4. No—it is possible (and sadly common) for programmers to place the business logic into the frame and component classes of the user interface.

5. The application server knows nothing about the files on your computer. You need to hand it the WAR file with all the application’s pages, code, and configuration files so that it can execute the application when it receives a web request.

6. No. The Scanner class does not have a constructor with no arguments.

7. There is no way of knowing without looking at the source code. Perhaps it simply executes a statement city = newValue, setting an instance variable of the bean class. But the method may also do other work, for example checking whether the city name is valid or storing the name in a database.

8. Add the tag <h:commandButton value="Help" action="/help"/> to error.xhtml.

9. The current page would be redisplayed.

10. h:selectOneRadio, h:selectOneMenu, or h:selectOneCheckbox

11. You would need a bean with a property such as the following:

```java
public Map<String, Integer> getYearChoices()
{
    Map<String, Integer> choices = new TreeMap<>();
    choices.put("2003", 2003);
    choices.put("2004", 2004);
    ...
    return choices;
}
```

Then supply a tag `<f:selectItems value="#{creditCard.yearChoices}"/>`.

12. Then the database connection would be kept open for the entire session.

13. As long as there are cities, the same page (next.xhtml) page is redisplayed. If all cities are removed, it is pointless to display the next.xhtml page, so the application navigates to the index.xhtml page.
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              |                 |
### Table 2 The Basic Latin (ASCII) Subset of Unicode

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>@</td>
<td>64</td>
<td>`</td>
<td>96</td>
<td>`</td>
<td>96</td>
<td>`</td>
<td>96</td>
</tr>
<tr>
<td>!</td>
<td>\u0021</td>
<td>33</td>
<td>A</td>
<td>\u0041</td>
<td>65</td>
<td>a</td>
<td>\u0061</td>
</tr>
<tr>
<td>&quot;</td>
<td>\u0022</td>
<td>34</td>
<td>B</td>
<td>\u0042</td>
<td>66</td>
<td>b</td>
<td>\u0062</td>
</tr>
<tr>
<td>#</td>
<td>\u0023</td>
<td>35</td>
<td>C</td>
<td>\u0043</td>
<td>67</td>
<td>c</td>
<td>\u0063</td>
</tr>
<tr>
<td>$</td>
<td>\u0024</td>
<td>36</td>
<td>D</td>
<td>\u0044</td>
<td>68</td>
<td>d</td>
<td>\u0064</td>
</tr>
<tr>
<td>%</td>
<td>\u0025</td>
<td>37</td>
<td>E</td>
<td>\u0045</td>
<td>69</td>
<td>e</td>
<td>\u0065</td>
</tr>
<tr>
<td>&amp;</td>
<td>\u0026</td>
<td>38</td>
<td>F</td>
<td>\u0046</td>
<td>70</td>
<td>f</td>
<td>\u0066</td>
</tr>
<tr>
<td>'</td>
<td>\u0027</td>
<td>39</td>
<td>G</td>
<td>\u0047</td>
<td>71</td>
<td>g</td>
<td>\u0067</td>
</tr>
<tr>
<td>(</td>
<td>\u0028</td>
<td>40</td>
<td>H</td>
<td>\u0048</td>
<td>72</td>
<td>h</td>
<td>\u0068</td>
</tr>
<tr>
<td>)</td>
<td>\u0029</td>
<td>41</td>
<td>I</td>
<td>\u0049</td>
<td>73</td>
<td>i</td>
<td>\u0069</td>
</tr>
<tr>
<td>*</td>
<td>\u002A</td>
<td>42</td>
<td>J</td>
<td>\u004A</td>
<td>74</td>
<td>j</td>
<td>\u006A</td>
</tr>
<tr>
<td>+</td>
<td>\u002B</td>
<td>43</td>
<td>K</td>
<td>\u004B</td>
<td>75</td>
<td>k</td>
<td>\u006B</td>
</tr>
<tr>
<td>,</td>
<td>\u002C</td>
<td>44</td>
<td>L</td>
<td>\u004C</td>
<td>76</td>
<td>l</td>
<td>\u006C</td>
</tr>
<tr>
<td>-</td>
<td>\u002D</td>
<td>45</td>
<td>M</td>
<td>\u004D</td>
<td>77</td>
<td>m</td>
<td>\u006D</td>
</tr>
<tr>
<td>.</td>
<td>\u002E</td>
<td>46</td>
<td>N</td>
<td>\u004E</td>
<td>78</td>
<td>n</td>
<td>\u006E</td>
</tr>
<tr>
<td>/</td>
<td>\u002F</td>
<td>47</td>
<td>O</td>
<td>\u004F</td>
<td>79</td>
<td>o</td>
<td>\u006F</td>
</tr>
<tr>
<td>0</td>
<td>\u0030</td>
<td>48</td>
<td>P</td>
<td>\u0050</td>
<td>80</td>
<td>p</td>
<td>\u0070</td>
</tr>
<tr>
<td>1</td>
<td>\u0031</td>
<td>49</td>
<td>Q</td>
<td>\u0051</td>
<td>81</td>
<td>q</td>
<td>\u0071</td>
</tr>
<tr>
<td>2</td>
<td>\u0032</td>
<td>50</td>
<td>R</td>
<td>\u0052</td>
<td>82</td>
<td>r</td>
<td>\u0072</td>
</tr>
<tr>
<td>3</td>
<td>\u0033</td>
<td>51</td>
<td>S</td>
<td>\u0053</td>
<td>83</td>
<td>s</td>
<td>\u0073</td>
</tr>
<tr>
<td>4</td>
<td>\u0034</td>
<td>52</td>
<td>T</td>
<td>\u0054</td>
<td>84</td>
<td>t</td>
<td>\u0074</td>
</tr>
<tr>
<td>5</td>
<td>\u0035</td>
<td>53</td>
<td>U</td>
<td>\u0055</td>
<td>85</td>
<td>u</td>
<td>\u0075</td>
</tr>
<tr>
<td>6</td>
<td>\u0036</td>
<td>54</td>
<td>V</td>
<td>\u0056</td>
<td>86</td>
<td>v</td>
<td>\u0076</td>
</tr>
<tr>
<td>7</td>
<td>\u0037</td>
<td>55</td>
<td>W</td>
<td>\u0057</td>
<td>87</td>
<td>w</td>
<td>\u0077</td>
</tr>
<tr>
<td>8</td>
<td>\u0038</td>
<td>56</td>
<td>X</td>
<td>\u0058</td>
<td>88</td>
<td>x</td>
<td>\u0078</td>
</tr>
<tr>
<td>9</td>
<td>\u0039</td>
<td>57</td>
<td>Y</td>
<td>\u0059</td>
<td>89</td>
<td>y</td>
<td>\u0079</td>
</tr>
<tr>
<td>;</td>
<td>\u003A</td>
<td>58</td>
<td>Z</td>
<td>\u005A</td>
<td>90</td>
<td>z</td>
<td>\u007A</td>
</tr>
<tr>
<td>;</td>
<td>\u003B</td>
<td>59</td>
<td>[</td>
<td>\u005B</td>
<td>91</td>
<td>}</td>
<td>\u007B</td>
</tr>
<tr>
<td>&lt;</td>
<td>\u003C</td>
<td>60</td>
<td>\</td>
<td>\u005C</td>
<td>92</td>
<td></td>
<td></td>
</tr>
<tr>
<td>=</td>
<td>\u003D</td>
<td>61</td>
<td>]</td>
<td>\u005D</td>
<td>93</td>
<td>}</td>
<td>\u007D</td>
</tr>
<tr>
<td>&gt;</td>
<td>\u003E</td>
<td>62</td>
<td>^</td>
<td>\u005E</td>
<td>94</td>
<td>~</td>
<td>\u007E</td>
</tr>
<tr>
<td>?</td>
<td>\u003F</td>
<td>63</td>
<td>_</td>
<td>\u005F</td>
<td>95</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### Table 3 The Latin-1 Subset of Unicode

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Å</td>
<td>\u00C0</td>
<td>192</td>
<td>à</td>
<td>\u00E0</td>
<td>224</td>
<td>£</td>
<td>\u00A3</td>
<td>195</td>
</tr>
<tr>
<td>À</td>
<td>\u00C1</td>
<td>193</td>
<td>á</td>
<td>\u00E1</td>
<td>225</td>
<td>¥</td>
<td>\u00A4</td>
<td>196</td>
</tr>
<tr>
<td>Á</td>
<td>\u00C2</td>
<td>194</td>
<td>â</td>
<td>\u00E2</td>
<td>226</td>
<td>¢</td>
<td>\u00A5</td>
<td>197</td>
</tr>
<tr>
<td>¡</td>
<td>\u00A1</td>
<td>161</td>
<td>Á</td>
<td>\u00C3</td>
<td>198</td>
<td>¤</td>
<td>\u00A6</td>
<td>162</td>
</tr>
<tr>
<td>¢</td>
<td>\u00A2</td>
<td>162</td>
<td>Â</td>
<td>\u00C4</td>
<td>199</td>
<td>¥</td>
<td>\u00A7</td>
<td>163</td>
</tr>
<tr>
<td>£</td>
<td>\u00A3</td>
<td>163</td>
<td>Æ</td>
<td>\u00C5</td>
<td>200</td>
<td>ª</td>
<td>\u00A8</td>
<td>164</td>
</tr>
<tr>
<td>¨</td>
<td>\u00A9</td>
<td>164</td>
<td>¨</td>
<td>\u00C6</td>
<td>201</td>
<td>©</td>
<td>\u00A9</td>
<td>165</td>
</tr>
<tr>
<td>©</td>
<td>\u00A0</td>
<td>165</td>
<td>Ë</td>
<td>\u00C7</td>
<td>202</td>
<td>&quot;</td>
<td>\u00A0</td>
<td>166</td>
</tr>
<tr>
<td>¬</td>
<td>\u00A1</td>
<td>166</td>
<td>Ë</td>
<td>\u00C8</td>
<td>203</td>
<td>&quot;</td>
<td>\u00A1</td>
<td>167</td>
</tr>
<tr>
<td>°</td>
<td>\u00A2</td>
<td>167</td>
<td>Í</td>
<td>\u00C9</td>
<td>204</td>
<td>«</td>
<td>\u00A2</td>
<td>168</td>
</tr>
<tr>
<td>±</td>
<td>\u00A3</td>
<td>168</td>
<td>Ï</td>
<td>\u00CD</td>
<td>205</td>
<td>¬</td>
<td>\u00A3</td>
<td>169</td>
</tr>
<tr>
<td>½</td>
<td>\u00A4</td>
<td>169</td>
<td>Ñ</td>
<td>\u00CA</td>
<td>206</td>
<td>¯</td>
<td>\u00A4</td>
<td>170</td>
</tr>
<tr>
<td>¼</td>
<td>\u00A5</td>
<td>170</td>
<td>Ò</td>
<td>\u00CB</td>
<td>207</td>
<td>°</td>
<td>\u00A5</td>
<td>171</td>
</tr>
<tr>
<td>¾</td>
<td>\u00A6</td>
<td>171</td>
<td>Ô</td>
<td>\u00CC</td>
<td>208</td>
<td>±</td>
<td>\u00A6</td>
<td>172</td>
</tr>
<tr>
<td>-</td>
<td>\u00A7</td>
<td>172</td>
<td>Ó</td>
<td>\u00CD</td>
<td>209</td>
<td>±</td>
<td>\u00A7</td>
<td>173</td>
</tr>
<tr>
<td>®</td>
<td>\u00A8</td>
<td>173</td>
<td>Ò</td>
<td>\u00CF</td>
<td>210</td>
<td>2</td>
<td>\u00A8</td>
<td>174</td>
</tr>
<tr>
<td>½</td>
<td>\u00A9</td>
<td>174</td>
<td>Ô</td>
<td>\u00CE</td>
<td>211</td>
<td>3</td>
<td>\u00A9</td>
<td>175</td>
</tr>
<tr>
<td>¼</td>
<td>\u00AA</td>
<td>175</td>
<td>Õ</td>
<td>\u00CF</td>
<td>212</td>
<td>³</td>
<td>\u00AA</td>
<td>176</td>
</tr>
<tr>
<td>µ</td>
<td>\u00AB</td>
<td>176</td>
<td>Õ</td>
<td>\u00D0</td>
<td>213</td>
<td>¶</td>
<td>\u00AB</td>
<td>177</td>
</tr>
<tr>
<td>¶</td>
<td>\u00AC</td>
<td>177</td>
<td>Õ</td>
<td>\u00D1</td>
<td>214</td>
<td>·</td>
<td>\u00AC</td>
<td>178</td>
</tr>
<tr>
<td>·</td>
<td>\u00AD</td>
<td>178</td>
<td>Õ</td>
<td>\u00D2</td>
<td>215</td>
<td>×</td>
<td>\u00AD</td>
<td>179</td>
</tr>
<tr>
<td>½</td>
<td>\u00AE</td>
<td>179</td>
<td>Õ</td>
<td>\u00D3</td>
<td>216</td>
<td>½</td>
<td>\u00AE</td>
<td>180</td>
</tr>
<tr>
<td>¼</td>
<td>\u00AF</td>
<td>180</td>
<td>Õ</td>
<td>\u00D4</td>
<td>217</td>
<td>½</td>
<td>\u00AF</td>
<td>181</td>
</tr>
<tr>
<td>ß</td>
<td>\u00B0</td>
<td>181</td>
<td>Õ</td>
<td>\u00D5</td>
<td>218</td>
<td>¼</td>
<td>\u00B0</td>
<td>182</td>
</tr>
<tr>
<td>ñ</td>
<td>\u00B1</td>
<td>182</td>
<td>Õ</td>
<td>\u00D6</td>
<td>219</td>
<td>ñ</td>
<td>\u00B1</td>
<td>183</td>
</tr>
<tr>
<td>ñ</td>
<td>\u00B2</td>
<td>183</td>
<td>Õ</td>
<td>\u00D7</td>
<td>220</td>
<td>ñ</td>
<td>\u00B2</td>
<td>184</td>
</tr>
<tr>
<td>ñ</td>
<td>\u00B3</td>
<td>184</td>
<td>Õ</td>
<td>\u00D8</td>
<td>221</td>
<td>ñ</td>
<td>\u00B3</td>
<td>185</td>
</tr>
<tr>
<td>ñ</td>
<td>\u00B4</td>
<td>185</td>
<td>Õ</td>
<td>\u00D9</td>
<td>222</td>
<td>ñ</td>
<td>\u00B4</td>
<td>186</td>
</tr>
<tr>
<td>ñ</td>
<td>\u00B5</td>
<td>186</td>
<td>Õ</td>
<td>\u00DA</td>
<td>223</td>
<td>ñ</td>
<td>\u00B5</td>
<td>187</td>
</tr>
<tr>
<td>ñ</td>
<td>\u00B6</td>
<td>187</td>
<td>Õ</td>
<td>\u00DB</td>
<td>224</td>
<td>ñ</td>
<td>\u00B6</td>
<td>188</td>
</tr>
<tr>
<td>ñ</td>
<td>\u00B7</td>
<td>188</td>
<td>Õ</td>
<td>\u00DC</td>
<td>225</td>
<td>ñ</td>
<td>\u00B7</td>
<td>189</td>
</tr>
<tr>
<td>ñ</td>
<td>\u00B8</td>
<td>189</td>
<td>Õ</td>
<td>\u00DD</td>
<td>226</td>
<td>ñ</td>
<td>\u00B8</td>
<td>190</td>
</tr>
<tr>
<td>ñ</td>
<td>\u00B9</td>
<td>190</td>
<td>Õ</td>
<td>\u00DE</td>
<td>227</td>
<td>ñ</td>
<td>\u00B9</td>
<td>191</td>
</tr>
</tbody>
</table>
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, \( x + y \times z \) means \( x + (y \times z) \) because the \( \times \) operator has higher precedence than the + operator. Looking at the table below, you can tell that \( x \&\& y \mid\mid z \) means \((x \&\& y) \mid\mid z\) because the \( \mid\mid \) 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 \( x = (y = z)\).

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
<th>Associativity</th>
</tr>
</thead>
<tbody>
<tr>
<td>.</td>
<td>Access class feature</td>
<td>Left to right</td>
</tr>
</tbody>
</table>
| []       | Array subscript | \n
| ()       | Function call | \n
| ++       | Increment | Right to left |
| --       | Decrement | \n
| !        | Boolean not | \n
| ~        | Bitwise not | \n
| + (unary) | (Has no effect) | \n
| - (unary) | Negative | \n
| (TypeName) | Cast | \n
| new       | Object allocation | \n
| *         | Multiplication | Left to right |
| /         | Division or integer division | \n
| %         | Integer remainder | \n
| +         | Addition, string concatenation | Left to right |
| -         | Subtraction | \n
| <<        | Shift left | \n
| >>        | Right shift with sign extension | \n
| >>>       | Right shift with zero extension | \n
## Java Operator Summary

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
<th>Associativity</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>&lt;</code></td>
<td>Less than</td>
<td>Left to right</td>
</tr>
<tr>
<td><code>&lt;=</code></td>
<td>Less than or equal</td>
<td></td>
</tr>
<tr>
<td><code>&gt;</code></td>
<td>Greater than</td>
<td></td>
</tr>
<tr>
<td><code>&gt;=</code></td>
<td>Greater than or equal</td>
<td></td>
</tr>
<tr>
<td><code>instanceof</code></td>
<td>Tests whether an object’s type is a given type or a subtype thereof</td>
<td></td>
</tr>
<tr>
<td><code>==</code></td>
<td>Equal</td>
<td>Left to right</td>
</tr>
<tr>
<td><code>!=</code></td>
<td>Not equal</td>
<td></td>
</tr>
<tr>
<td><code>&amp;</code></td>
<td>Bitwise <em>and</em></td>
<td>Left to right</td>
</tr>
<tr>
<td><code>^</code></td>
<td>Bitwise exclusive <em>or</em></td>
<td>Left to right</td>
</tr>
<tr>
<td>`</td>
<td>`</td>
<td>Bitwise <em>or</em></td>
</tr>
<tr>
<td><code>&amp;&amp;</code></td>
<td>Boolean “short circuit” <em>and</em></td>
<td>Left to right</td>
</tr>
<tr>
<td>`</td>
<td></td>
<td>`</td>
</tr>
<tr>
<td><code>? :</code></td>
<td>Conditional</td>
<td>Right to left</td>
</tr>
<tr>
<td><code>=</code></td>
<td>Assignment</td>
<td></td>
</tr>
<tr>
<td><code>op=</code></td>
<td>Assignment with binary operator (<em>op</em> is one of +, -, *, /,</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

*op* is one of +, -, *, /, |, ^, <<, >>)*
<table>
<thead>
<tr>
<th>Reserved Word</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>abstract</td>
<td>An abstract class or method</td>
</tr>
<tr>
<td>assert</td>
<td>An assertion that a condition is fulfilled</td>
</tr>
<tr>
<td>boolean</td>
<td>The Boolean type</td>
</tr>
<tr>
<td>break</td>
<td>Breaks out of the current loop or labeled statement</td>
</tr>
<tr>
<td>byte</td>
<td>The 8-bit signed integer type</td>
</tr>
<tr>
<td>case</td>
<td>A label in a switch statement</td>
</tr>
<tr>
<td>catch</td>
<td>The handler for an exception in a try block</td>
</tr>
<tr>
<td>char</td>
<td>The 16-bit Unicode character type</td>
</tr>
<tr>
<td>class</td>
<td>Defines a class</td>
</tr>
<tr>
<td>const</td>
<td>Not used</td>
</tr>
<tr>
<td>continue</td>
<td>Skips the remainder of a loop body</td>
</tr>
<tr>
<td>default</td>
<td>The default label in a switch statement</td>
</tr>
<tr>
<td>do</td>
<td>A loop whose body is executed at least once</td>
</tr>
<tr>
<td>double</td>
<td>The 64-bit double-precision floating-point type</td>
</tr>
<tr>
<td>else</td>
<td>The alternative clause in an if statement</td>
</tr>
<tr>
<td>enum</td>
<td>An enumeration type</td>
</tr>
<tr>
<td>extends</td>
<td>Indicates that a class is a subclass of another class</td>
</tr>
<tr>
<td>final</td>
<td>A value that cannot be changed after it has been initialized, a method that cannot be overridden, or a class that cannot be extended</td>
</tr>
<tr>
<td>finally</td>
<td>A clause of a try block that is always executed</td>
</tr>
<tr>
<td>float</td>
<td>The 32-bit single-precision floating-point type</td>
</tr>
<tr>
<td>for</td>
<td>A loop with initialization, condition, and update expressions</td>
</tr>
<tr>
<td>goto</td>
<td>Not used</td>
</tr>
<tr>
<td>if</td>
<td>A conditional branch statement</td>
</tr>
<tr>
<td>implements</td>
<td>Indicates that a class realizes an interface</td>
</tr>
<tr>
<td>Reserved Word</td>
<td>Description</td>
</tr>
<tr>
<td>---------------</td>
<td>-------------</td>
</tr>
<tr>
<td>import</td>
<td>Allows the use of class names without the package name</td>
</tr>
<tr>
<td>instanceof</td>
<td>Tests whether an object’s type is a given type or a subtype thereof</td>
</tr>
<tr>
<td>int</td>
<td>The 32-bit integer type</td>
</tr>
<tr>
<td>interface</td>
<td>An abstract type with only abstract or default methods and constants</td>
</tr>
<tr>
<td>long</td>
<td>The 64-bit integer type</td>
</tr>
<tr>
<td>native</td>
<td>A method implemented in non-Java code</td>
</tr>
<tr>
<td>new</td>
<td>Allocates an object</td>
</tr>
<tr>
<td>package</td>
<td>A collection of related classes</td>
</tr>
<tr>
<td>private</td>
<td>A feature that is accessible only by methods of the same class</td>
</tr>
<tr>
<td>protected</td>
<td>A feature that is accessible only by methods of the same class, a subclass, or another class in the same package</td>
</tr>
<tr>
<td>public</td>
<td>A feature that is accessible by all methods</td>
</tr>
<tr>
<td>return</td>
<td>Returns from a method</td>
</tr>
<tr>
<td>short</td>
<td>The 16-bit integer type</td>
</tr>
<tr>
<td>static</td>
<td>A feature that is defined for a class, not for individual instances</td>
</tr>
<tr>
<td>strictfp</td>
<td>Uses strict rules for floating-point computations</td>
</tr>
<tr>
<td>super</td>
<td>Invokes the superclass constructor or a superclass method</td>
</tr>
<tr>
<td>switch</td>
<td>A selection statement</td>
</tr>
<tr>
<td>synchronized</td>
<td>A block of code that is accessible to only one thread at a time</td>
</tr>
<tr>
<td>this</td>
<td>The implicit parameter of a method; or invocation of another constructor of the same class</td>
</tr>
<tr>
<td>throw</td>
<td>Throws an exception</td>
</tr>
<tr>
<td>throws</td>
<td>Indicates the exceptions that a method may throw</td>
</tr>
<tr>
<td>transient</td>
<td>Instance variables that should not be serialized</td>
</tr>
<tr>
<td>try</td>
<td>A block of code with exception handlers or a finally handler</td>
</tr>
<tr>
<td>void</td>
<td>Tags a method that doesn’t return a value</td>
</tr>
<tr>
<td>volatile</td>
<td>A variable that may be accessed by multiple threads without synchronization</td>
</tr>
<tr>
<td>while</td>
<td>A loop statement</td>
</tr>
</tbody>
</table>
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
java.awt.Shape
java.io.Serializable
java.lang.AutoCloseable
java.lang.Cloneable
java.lang.Object
java.awt.BorderLayout implements Serializable
java.awt.Color implements Serializable
java.awt.Component implements Serializable
java.awt.Container
   javax.swing.JComponent
      javax.swing.AbstractButton
         javax.swing.JButton
      javax.swing.JMenuItem
      javax.swing.JMenu
         javax.swing.JMenuItem
      javax.swing.JButton
      javax.swing.JMenuItem
      javax.swing.JMenu
      javax.swing.JMenuItem
      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 implements Cloneable, Serializable
java.awt.FlowLayout implements Serializable
java.awt.Font implements Serializable
java.awt.Graphics
   java.awt.Graphics2D;
java.awt.GridLayout implements Serializable
java.awt.event.MouseAdapter implements MouseListener
java.awt.geom.Line2D implements Cloneable, Shape
   java.awt.geom.Line2D.Double implements Serializable
java.awt.geom.Point2D implements Cloneable
   java.awt.geom.Point2D.Double implements Serializable
java.awt.geom.RectangularShape implements Cloneable, Shape
   java.awt.geom.Rectangle2D
      java.awt.Rectangle implements Serializable
   java.awt.geom.Ellipse2D
      java.awt.geom.Ellipse2D.Double implements Serializable
```
The Java Library

java.io.File implements Comparable<File>, Serializable
java.io.InputStream
  java.io.FileInputStream
  java.io.ObjectInputStream
java.io.OutputStream
  java.io.FileOutputStream
  java.io.FilterOutputStream
  java.io.PrintStream
  java.io.ObjectOutputStream
java.io.RandomAccessFile
java.io.Writer
  java.io.PrintWriter
java.lang.Boolean implements Comparable<Boolean>, Serializable
java.lang.Character implements Comparable<Character>, Serializable
java.lang.Class implements Serializable
java.lang.Math
  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>, Serializable
java.lang.System
java.lang.Thread implements Runnable
java.lang.Throwable
  java.lang.Error
  java.lang.Exception
    java.lang.CloneNotSupportedException
    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.lang.System
java.sql.SQLException
javax.xml.xpath.XPathException
javax.xml.xpath.XPathExpressionException
org.xml.sax.SAXException
java.net.ServerSocket
java.net.Socket
java.net.URL implements Serializable
java.net.URLConnection
  java.net.HttpURLConnection
java.nio.file.Files
java.nio.file.Paths
java.sql.DriverManager
java.text.Format implements Serializable
java.text.SimpleDateFormat
java.util.AbstractCollection<E>
  java.util.AbstractList<E>
    java.util.AbstractSequentialList<E>
    java.util.LinkedList<E> implements List<E>, Queue<E>, Serializable
    java.util.ArrayList<E> implements List<E>, Serializable
  java.util.AbstractSequentialList<E>
    java.util.AbstractSet<E>
      java.util.HashSet<E> implements Serializable, Set<E>
      java.util.TreeSet<E> implements Serializable, SortedSet<E>
  java.util.AbstractCollection<E>
    java.util.AbstractList<E>
      java.util.AbstractSet<E>
    java.util.AbstractMap<K, V>
      java.util.HashMap<K, V> implements Map<K, V>, Serializable
      java.util.LinkedHashMap<K, V>
      java.util.TreeMap<K, V> implements Serializable, Map<K, V>
java.util.Arrays
java.util.Collections
java.util.Calendar
  java.util.GregorianCalendar
java.util.Date
    implements Serializable
java.util.Dictionary<K, V>
java.util.Hashtable<K, V>
java.util.Properties
  implements Serializable
java.util.EventObject
    implements Serializable
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.Optional<T>
java.util.OptionalDouble
java.util.OptionalInt
java.util.OptionalLong
java.util.Random
  implements Serializable
java.util.Scanner
java.util.TimeZone
  implements Serializable
java.util.concurrent.locks.ReentrantLock
    implements Lock, Serializable
java.util.logging.Level
  implements Serializable
java.util.logging.Logger
java.util.stream.Collectors
javax.swing.ButtonGroup
  implements Serializable
javax.swing.ImageIcon
  implements Serializable
javax.swing.Keystroke
  implements Serializable
javax.swing.Timer
  implements Serializable
javax.swing.border.AbstractBorder
  implements Serializable
javax.swing.border.EtchedBorder
javax.swing.border.TitledBorder
javax.xml.parsers.DocumentBuilder
javax.xml.parsers.DocumentBuilderFactory
javax.xml.xpath.XPathFactory
java.lang.Comparable<T>
java.lang.Runnable
java.nio.file.Path
java.sql.Connection
java.sql.ResultSet
java.sql.ResultSetMetaData
java.sql.Statement
java.util.Collection<E>
  java.util.List<E>
  java.util.Set<E>
    java.util.SortedSet<E>
java.util.Comparator<T>
java.util.EventIterator
  java.awt.event.ActionListener
  java.awt.event.KeyListener
  java.awt.event.MouseAdapter
    java.awt.event.ChangeListener
java.util.Iterator<E>
java.util.ListIterator<E>
java.util.Map<K, V>
  java.util.concurrent.locks.Condition
  java.util.concurrent.locks.Lock
java.util.function.Predicate<T>
java.util.stream.DoubleStream
java.util.stream.IntStream
java.util.stream.LongStream
java.util.stream.Stream<T>
A-12  Appendix D  The Java Library

javax.sql.DataSource
javax.xml.xpath.XPath
org.w3c.dom.DOMConfiguration
org.w3c.dom.DOMImplementation
org.w3c.dom.Node
    org.w3c.dom.CharacterData
org.w3c.dom.Text
org.w3c.dom.Document
org.w3c.dom.Element
org.w3c.dom.ls.DOMImplementationLS
org.w3c.dom.ls.LSSerializer

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

• 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
Appendix D  The Java Library  A-13

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.awt.FlowLayout

- 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 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.Graphics2D

- void draw(Shape s)

This method draws the outline of the given shape. Many classes—among them Rectangle and Line2D.Double—implement the Shape interface.

Parameters:  
s The shape to be drawn

- void drawString(String s, int x, int y)
- void drawString(String s, float x, float y)

These methods draw a string in the current font.

Parameters:  
s The string to draw
x,y The basepoint of the first character in the string

- void fill(Shape s)

This method draws the given shape and fills it with the current color.

Parameters:  
s The shape to be filled

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.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.
  Parameters: `other` A rectangle
  Returns: The smallest rectangle containing both this and other

Interface java.awt.Shape
  The Shape interface describes shapes that can be drawn and filled by a Graphics2D object.

Package java.awt.event

Interface java.awt.event.ActionEvent
  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.

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.awt.geom

Class java.awt.geom.Ellipse2D.Double

• Ellipse2D.Double(double x, double y, double w, double h)
  This constructs an ellipse from the specified coordinates.
  Parameters:  x, y  The top-left corner of the bounding rectangle
               w  The width of the bounding rectangle
               h  The height of the bounding rectangle

Class java.awt.geom.Line2D

• double getX1()
• double getX2()
• double getY1()
• double getY2()
  These methods get the requested coordinate of an endpoint of this line.
  Returns:  The x- or y-coordinate of the first or second endpoint

• void setLine(double x1, double y1, double x2, double y2)
  This methods sets the endpoints of this line.
  Parameters:  x1, y1  A new endpoint of this line
               x2, y2  The other new endpoint

Class java.awt.geom.Line2D.Double

• Line2D.Double(double x1, double y1, double x2, double y2)
  This constructs a line from the specified coordinates.
  Parameters:  x1, y1  One endpoint of the line
               x2, y2  The other endpoint

• Line2D.Double(Point2D p1, Point2D p2)
  This constructs a line from the two endpoints.
  Parameters:  p1, p2  The endpoints of the line

Class java.awt.geom.Point2D

• double getX()
• double getY()
  These methods get the requested coordinates of this point.
  Returns:  The x- or y-coordinate of this point

• void setLocation(double x, double y)
  This method sets the x- and y-coordinates of this point.
  Parameters:  x, y  The new location of this point

Class java.awt.geom.Point2D.Double

• Point2D.Double(double x, double y)
  This constructs a point with the specified coordinates.
  Parameters:  x, y  The coordinates of the point

Class java.awt.geom.RectangularShape

• int getHeight()
• int getWidth()
  These methods get the height or width of the bounding rectangle of this rectangular shape.
  Returns:  The height or width, respectively

• double getCenterX()
• double getCenterY()
• double getMaxX()
• double getMaxY()
• double getMinX()
• double getMinY()
  These methods get the requested coordinate value of the corners or center of the bounding rectangle of this shape.
  Returns:  The center, maximum, or minimum x- and y-coordinates

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.

Class java.io.IOException

This type of exception is thrown when an input/output error is encountered.

Class java.io.ObjectInputStream

- ObjectInputStream(InputStream in)
  This constructs an object input stream.
  Parameters: in The input stream to read from

- Object readObject()
  This method reads the next object from this object input stream.
  Returns: The next object

Class java.io.ObjectOutputStream

- ObjectOutputStream(OutputStream out)
  This constructs an object output stream.
  Parameters: out The output stream to write to

- Object writeObject(Object obj)
  This method writes the next object to this object output stream.
  Parameters: obj The object to write

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

- void close()
  This method closes this stream or writer and releases any associated system resources.

- void printf(int x)
- void printf(double x)
- void printf(Object x)
- void printf(String x)
- void println()
- void println(int x)
- void println(double x)
Appendix D  The Java Library

• 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

Class java.io.RandomAccessFile

• RandomAccessFile(String name, String mode)
  This method opens a named random access file for reading or read/write access.
  Parameters:  name  The file name
  mode  "r" for reading or "rw" for read/write access
• long getFilePointer()
  This method gets the current position in this file.
  Returns:  The current position for reading and writing
• long length()
  This method gets the length of this file.
  Returns:  The file length
• char readChar()
  double readDouble()
  int readInt()
  These methods read a value from the current position in this file.
  Returns:  The value that was read from the file
• void seek(long position)
  This method sets the position for reading and writing in this file.
  Parameters:  position  The new position
• void writeChar(int x)
  void writeChars(String x)
  void writeDouble(double x)
  void writeInt(int x)
  These methods write a value to the current position in this file.
  Parameters:  x  The value to be written

Interface java.io.Serializable
  A class should implement this interface in order to enable serialization of objects.

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
• 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.Cloneable
A class implements this interface to indicate that the Object.clone method is allowed to make a shallow copy of its instance variables.

Class java.lang.CloneNotSupportedException
This exception is thrown when a program tries to use Object.clone to make a shallow copy of an object of a class that does not implement the Cloneable interface.

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
- 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 \( y \), 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 \( x \).
  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
• 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

  • void notify()
    This method notifies one of the threads that is currently on the wait list for the lock of this object.

  • void notifyAll()
    This method notifies all of the threads that are currently on the wait list for the lock of this object.

  • 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

  • void wait()
    This method blocks the currently executing thread and puts it on the wait list for the lock of this object.

Interface java.lang.Runnable
  • void run()
    This method should be overridden to define the tasks to be carried out when this runnable is executed.

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
• **IntStream codePoints()**
  This method yields a stream of all code points in this string.

• **boolean equals(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 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
  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)**
  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.Thread**

• **boolean interrupted()**
  This method tests whether another thread has called the interrupt method on the current thread.
  Returns: true if the thread has been interrupted

• **static void sleep(int millis)**
  This method puts the calling thread to sleep.
  Parameters: millis The number of milliseconds to sleep
void start()
This method starts the thread and executes its run method.

Class java.lang.Throwables
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 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.

Package java.net

Class java.net.HttpURLConnection

• int getResponseCode()
  This method gets the response status code from this connection. A value of HTTP_OK indicates success.
  Returns: The HTTP response code
• String getResponseMessage()
  This method gets the response message of this connection’s HTTP request.
  Returns: The message, such as "OK" or "File not found"
• static int HTTP_OK
  This response code indicates a successful fulfillment of the request.

Class java.net.ServerSocket

• ServerSocket(int port)
  This constructs a server socket that listens to the given port.
  Parameters: port The port number to listen to
• Socket accept()
  This method waits for a client to connect to the port to which this server socket listens. When a connection occurs, the method returns a socket through which the server can communicate with the client.
  Returns: The socket through which the server can communicate with the client
• void close()
  This method closes the server socket. Clients can no longer connect.

Class java.net.Socket

• Socket(String host, int port)
  This constructs a socket that connects to a server.
  Parameters: host The host name
  Parameters: port The port number to connect to
• void close()
  This method closes the connection with the server.
• InputStream getInputStream()
  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 socket
• **OutputStream** `getOutputStream()`  
  This method gets the output stream through which the client can send information to the server.  
  **Returns:** The output stream associated with this socket

**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

**Class java.net.URLConnection**

• **URLConnection**(`URL u`)  
  This constructs a URLConnection object from a URL object.  
  **Parameters:** `u` The resource to which you intend to connect

• **int** `getContentLength()`  
  This method gets the value of the content-length header of this URL connection.  
  **Returns:** The number of bytes in the content that the server is sending

• **String** `getContentType()`  
  This method gets the value of the content-type header of this URL connection.  
  **Returns:** The MIME type of the content that the server is sending, such as "text/plain" or "image/gif"

• **InputStream** `getInputStream()`  
  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

• **void** `setIfModifiedSince(Date d)`  
  This method instructs the connection to request that the server send data only if the content has been modified since a given date.  
  **Parameters:** `d` The modification date

**Package java.nio.file**

**Class java.nio.file.Files**

• **static Path** `copy(Path source, Path target)`  
• **static Path** `move(Path source, Path target)`  
  These methods copy or move a file to another. The target must not exist.  
  **Parameters:**  
  - `source` The path to the source file  
  - `target` The path to the target file  
  **Returns:**  
  - `target`

• **static Path** `createFile(Path path)`  
• **static Path** `createDirectory(Path path)`  
  This method creates a file or directory. The parent directory must exist.  
  **Parameters:**  
  - `path` The path to the file or directory to be created  
  **Returns:**  
  - `path`

• **static void** `delete(Path file)`  
  This method deletes a file. The file must exist.  
  **Parameters:**  
  - `file` The path to the file  

• **static boolean** `exists(Path path)`  
• **static boolean** `isDirectory(Path path)`  
• **static boolean** `isRegularFile(Path path)`  
  These methods check whether the given path exists, is a directory, or a regular file.  
  **Parameters:**  
  - `path` The path to check  
  **Returns:**  
  - `true` if the path exists, `false` otherwise

• **static Stream<Path>** `list(Path dir)`  
• **static Stream<Path>** `walk(Path dir)`  
  These methods yield streams to all children or descendants of a directory.  
  **Parameters:**  
  - `dir` The path to the directory  
  **Returns:** Streams of paths

• **static byte[]** `readAllBytes(Path file)`  
• **static List<String>** `readAllLines(Path file)`  
• **static Stream<String>** `lines(Path file)`  
  These methods read all bytes or lines in a file.  
  **Parameters:**  
  - `file` The path to the file  
  **Returns:**  
  - `byte[]` for `readAllBytes`, `List<String>` for `readAllLines`, and `Stream<String>` for `lines`

**Interface java.nio.file.Path**

• **Path** `getName()`  
• **Path** `getParent()`  
  These methods yield the last part, or all but the last part, of this path.  
  **Parameters:**  
  - `null` for `getName` and `getParent`  
  **Returns:**  
  - `null` for `getName`, and the path without the last part for `getParent`

• **Path** `resolve(Path other)`  
  This method yields a path that is obtained by first following this path, then the other if the other path is relative. If other is absolute, then it is returned.  
  **Parameters:**  
  - `other` The path to follow relative to this path  
  **Returns:** A new `Path` instance
Class java.nio.file.Paths

- static Path get(String... components)
  This method constructs a path with the given components.

Package java.sql

Interface java.sql.Connection

- void close()
  This method closes the connection with the database.
- void commit()
  This method commits all database changes since the last call to commit or rollback.
- Statement createStatement()
  This method creates a statement object, which can be used to issue database commands.
  Returns: A statement object
- PreparedStatement prepareStatement(String command)
  This method creates a prepared statement for a SQL command that is issued repeatedly.
  Parameters: command The SQL command
  Returns: The statement object for setting parameters and executing the call
- void rollback()
  This method abandons all database changes since the last call to commit or rollback.
- void setAutoCommit(boolean b)
  This method sets the auto commit mode. By default, it is true. If it is set to false, then transactions are indicated with calls to commit or rollback.
  Parameters: b The desired auto commit mode

Class java.sql.DriverManager

- static Connection getConnection(String url, String username, String password)
  This method obtains a connection to the database specified in the database URL.
  Parameters: url The database URL
  username The database user name
  password The password for the database user
  Returns: A connection to the database

Interface java.sql.ResultSet

- void close()
  This method closes the result set.
- double getDouble(int column)
  This method returns the floating-point value at the cursor row and the given column.
  Parameters: column The column index (starting with 1)
  Returns: The data value
- double getDouble(String columnName)
  This method returns the floating-point value at the cursor row and the given column name.
  Parameters: columnName The column name
  Returns: The data value
- int getInt(int column)
  This method returns the integer value at the cursor row and the given column.
  Parameters: column The column index (starting with 1)
  Returns: The data value
Appendix D  The Java Library  A-25

• int getInt(String columnName)
  This method returns the integer value at the cursor row and the given column name.
  Parameters: columnName  The column name
  Returns:  The data value

• ResultSetMetaData getMetaData()
  This method returns the metadata associated with this result set.
  Returns:  The metadata

• String getString(int column)
  This method returns the value at the cursor row and the given column.
  Parameters: column  The column index (starting with 1)
  Returns:  The data value, as a string

• String getString(String columnName)
  This method returns the value at the cursor row and the given column name.
  Parameters: columnName  The column name
  Returns:  The data value, as a string

• boolean next()
  This method positions the cursor to the next row. You must call next once to move the cursor to the first row before calling any of the get methods.
  Returns: true if the cursor has been positioned on a row, false at the end of the result set

Interface java.sql.ResultSetMetaData

• int getColumnCount()
  This method returns the number of columns of this result set.
  Returns:  The number of columns

• int getColumnDisplaySize(int column)
  This method returns the number of characters that should be used to display the specified column in this result set.
  Parameters: column  The column index (starting with 1)
  Returns:  The number of characters that should be used to display this column

• String getColumnLabel(int column)
  This method returns the label for a column in this result set.
  Parameters: column  The column index (starting with 1)
  Returns:  The column label

Class java.sql.SQLException
  This exception is thrown when a database error occurs.

Interface java.sql.Statement

• void close()
  This method closes this statement.

• boolean execute(String command)
  This method executes a SQL command.
  Parameters: command  The command to execute
  Returns: true if the execution yielded a result set

• ResultSet executeQuery(String command)
  This method executes a SQL query.
  Parameters: command  The query command to execute
  Returns:  The query result

• int executeUpdate(String command)
  This method executes a SQL update command.
  Parameters: command  The update command to execute
  Returns:  The number of records affected by the update

• ResultSet getResultSet()
  This method gets the result of the last command.
  Returns:  The query result from the last command

• int getUpdateCount()
  This method gets the update count of the last command.
  Returns:  The number of records affected by the last command

Package java.text

Class java.text.DateFormat

• String format(Date aDate)
  This method formats a date.
  Parameters: aDate  The date to format
  Returns:  A string containing the formatted date

• static DateFormat getTimeInstance()
  This method returns a formatter that formats only the time portion of a date.
  Returns:  The formatter object

• void setTimeZone(TimeZone zone)
  This method sets the time zone to be used when formatting dates.
  Parameters: zone  The time zone to use
## 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 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

• default Stream<E> stream()
  This method yields a stream of the 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

• 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.

• default <U extends Comparable<? super U>>
  Comparator<T> thenComparing(Function<? super T, ? extends U> keyExtractor)
  This method makes a comparator from this comparator and 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 first applies this comparator, using the key function to break ties. For example, Comparator.comparing(person -> person.getLastName()).thenComparing(person -> person.getFirstName()) compares people by their last name, then by their first name if the last names are the same.

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.
The Java Library

### Appendix D

**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.LinkedHashMap<K, V>**

**LinkedHashMap<K, V>()**
This constructs an empty linked hash map. The iterator of a linked hash map visits the entries in the order in which they were added to the map.

**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

**Class 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.Optional<T>**

• **static <T> Optional<T> empty()**
  These methods yield an Optional object representing no value.

• **static <T> Optional<T> of(T value)**
  These methods yield no value, or the given value.

• **T get()**

• **T orElse(T other)**
  These methods yield the value represented by this Optional. If it represents no value, the first method throws a NoSuchElementException, and the second method returns the value other.

• **void ifPresent(Consumer<? super T> f)**
  This method applies the function f to the value represented by this Optional, or does nothing if it represents no value.
  **Parameters:**
  - f: A function accepting a value of type T

• **boolean isPresent()**
  This method returns true if this Optional represents a value.

**Class java.util.OptionalDouble**

**Class java.util.OptionalInt**

**Class java.util.OptionalLong**

These classes represent an optional value of type double, int, or long. They have the same methods as Optional<T>, with T replaced with double, int, or long, except for the methods to yield the represented value.

• **doublegetAsDouble()**

• **intgetAsInt()**

• **longgetAsLong()**
  These methods yield the value represented by this Optional, throwing a NoSuchElementException if no value is present.

**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

**Class java.util.Properties**

• **String getProperty(String key)**
  This method gets the value associated with a key in this properties map.
  **Parameters:**
  - key: The key for which to find the associated value
  **Returns:**
  The value, or null if the key is not present in the map

• **void load(InputStream in)**
  This method loads a set of key/value pairs into this properties map from an input stream.
  **Parameters:**
  - in: The input stream from which to read the key/value pairs (it must be a sequence of lines of the form key=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

**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
    Returns: This scanner

• **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.concurrent.locks**

**Interface java.util.concurrent.locks.Condition**

• **void await()**
  This method blocks the current thread until it is signalled or interrupted.

• **void signal()**
  This method unblocks one thread that is waiting on this condition.

• **void signalAll()**
  This method unblocks all threads that are waiting on this condition.

**Interface java.util.concurrent.locks.Lock**

• **void lock()**
  This method causes the current thread to acquire this lock. The thread blocks if the lock is not available.
• **Condition** `newCondition()`  
  This method creates a new condition object for this lock.  
  Returns: The condition object
• **void** `unlock()`  
  This method causes the current thread to relinquish this lock.

**Class java.util.concurrent.locks.ReentrantLock**

• **ReentrantLock()**  
  This constructs a new reentrant lock.

**Package java.util.function**

**Interface java.util.function.Predicate<T>**

• **boolean** `test(T t)`  
  This method tests whether a value fulfills this predicate.  
  Parameters: t The value to test

**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 java.util.stream**

**Class java.util.stream.Collectors**

The static methods of this class yield `Collector` objects that you use with the `collect` method of the `java.util.stream.Stream` interface.

• **static <T> Collector<T, ?, Double>** `averagingDouble(ToDoubleFunction<? super T> f)`  
  These methods yield a collector that produces the average or sum of the results of applying `f` to its input elements.  
  Parameters: f A function mapping values of type T to double, int, or long

• **static <T> Collector<T, ?, Integer>** `summingInt(ToIntFunction<? super T> f)`
• **static <T> Collector<T, ?, Long>** `summingDouble(ToDoubleFunction<? super T> f)`
• **static <T> Collector<T, ?, Long>** `counting()`
• **static <T> Collector<T, ?, Optional<T>>** `maxBy(Comparator<? super T> comparator)`
• **static <T> Collector<T, ?, Optional<T>>** `minBy(Comparator<? super T> comparator)`

These methods yield a collector that produces the largest or smallest of its inputs.  
  Parameters: comparator The comparator to use for comparing inputs

• **static <T> Collector<T, ?, List<T>>** `toList()`
• **static <T> Collector<T, ?, Set<T>>** `toSet()`

These methods yield a collector that produces a list or set of its inputs.
Interface java.util.stream.DoubleStream
This interface has the methods of java.util.stream.
Stream<T>, with T replaced by double, as well as the following method:
• <U> Stream<U> mapToObj(
    DoubleFunction<? extends U> f)
This method yields a stream of objects containing f(x) for all elements x in this stream.
Parameters: f  A function mapping double values to a type U

Interface java.util.stream.IntStream
This interface has the methods of java.util.stream.
Stream<T>, with T replaced by int, as well as the following methods:
• <U> Stream<U> mapToObj(IntFunction<? extends U> f)
This method yields a stream of objects containing f(x) for all elements x in this stream.
Parameters: f  A function mapping int values to a type U
• static IntStream range(int from, int to)
This method yields an IntStream containing the integers between from (inclusive) and to (exclusive).

Interface java.util.stream.LongStream
This interface has the methods of java.util.stream.
Stream<T>, with T replaced by long, as well as the following methods:
• <U> Stream<U> mapToObj(LongFunction<? extends U> f)
This method yields a stream of objects containing f(x) for all elements x in this stream.
Parameters: f  A function mapping long values to a type U
• long count()
This terminal operation counts the number of elements of this stream.
• Stream<T> distinct()
This operation yields a stream of the elements of this stream, with duplicates removed.
• Stream<T> filter(Predicate<? super T> p)
This operation yields a stream of the elements of this stream that fulfill the predicate.
Parameters: p  A function mapping values of the type T to boolean

Optional<T> findAny()
Optional<T> findFirst()
These terminal operations yield any element, or the first element, of this stream (which usually was the result from a filter operation).
• void forEach(Consumer<? super T> f)
This terminal operation applies a function to each stream element.
Parameters: f  A function accepting values of type T
• static <T> Stream<T> generate(Supplier<T> f)
This method yields an infinite stream consisting of values returned by f.
Parameters: f  A function with no parameters yielding values of type T
• static <T> Stream<T> iterate(T seed, UnaryOperator<T> f)
This method yields an infinite stream consisting of seed, f(seed), f(f(seed)), and so on.
Parameters: seed  A value of type T
    f  A function that accepts and returns values of type T
• Stream<T> limit(long n)
This operation yields a stream consisting of up to n elements of this stream.
Parameters: n  The maximum length of the resulting stream
• <R> Stream<R> map(
    Function<? super T, ? extends R> f)
These operations yield a stream consisting of the results of applying f to the elements of this stream.
Parameters: f  A function mapping values of the type T to values of type R, double, int, or long
• DoubleStream mapToDouble(
    ToDoubleFunction<? super T> f)
• IntStream mapToInt(ToIntFunction<? super T> f)
• LongStream mapToLong(ToLongFunction<? super T> f)
These operations yield a stream consisting of the results of applying f to the elements of this stream.
Parameters: f  A function mapping values of type T to values of type R, double, int, or long
• static <T> Stream<T> of(T... values)
  This operation yields a stream consisting of the
given values.
  Parameters: values A sequence of values of type T
• Stream<T> parallel()
  This operation yields a stream with the same
  elements as this stream, whose operations are
  parallelized when possible.
• Stream<T> skip(n)
  This operation yields a stream with the same
  elements as this stream after discarding the first n
  elements.
  Parameters: n The number of elements to be
discarded
• Stream<T> sorted()
  This operation yields a stream with the same
  elements as this stream in sorted order.
• <A> A[] toArray(IntFunction<A[]> arrayConstructor)
  This terminal operation yields an array containing
  the elements of this stream.
  Parameters: arrayConstructor The constructor
  for the resulting array, usually a
  constructor expression A[]::new

Package javax.sql

Interface javax.sql.DataSource
• Connection getConnection()
  This method returns a connection to the data source
  represented by this object.

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.JComboBox

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.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

void setJMenuBar(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.JMenuItem

JMenuItem(String text)
This constructs a menu item.
Parameters: text The text to appear in the menu item

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.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.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.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.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.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

Package `javax.xml.parsers`

Class `javax.xml.parsers.DocumentBuilder`

- `Document newDocument()`
  This constructs a new document object.
  Returns: An empty document

- `Document parse(File in)`
  This method parses an XML document in a file.
  Parameters:  
  - `in` The file containing the document
  Returns: The parsed document

- `Document parse(InputStream in)`
  This method parses an XML document in a stream.
  Parameters:  
  - `in` The input stream containing the document
  Returns: The parsed document

Class `javax.xml.parsers.DocumentBuilderFactory`

- `DocumentBuilder newDocumentBuilder()`
  This method creates a new document builder object.
  Returns: The document builder

- `static DocumentBuilderFactory newInstance()`
  This method creates a new document builder factory object.
  Returns: The document builder factory object

- `void setIgnoringElementContentWhitespace(boolean b)`
  This method sets the parsing mode for ignoring white space in element content for all document builders that are generated from this factory.
  Parameters:  
  - `b` true if white space should be ignored

- `void setValidating(boolean b)`
  This method sets the validation mode for all document builders that are generated from this factory.
  Parameters:  
  - `b` true if documents should be validated during parsing

Package `javax.xml.xpath`

Interface `javax.xml.xpath.XPath`

- `String evaluate(String path, Object context)`
  This method evaluates the given path expression in the given context.
  Parameters:  
  - `path` An XPath expression
  - `context` The starting context for the evaluation, such as a document, node, or node list
  Returns: The result of the evaluation
Class `javax.xml.xpath.XPathExpressionException`
This exception is thrown when an XPath expression cannot be evaluated.

Class `javax.xml.xpath.XPathFactory`
- `static XPathFactory newInstance()`  
  This method returns a factory instance that can be used to construct XPath objects.  
  Returns: An `XPathFactory` instance
- `XPath newXPath()`  
  This method returns an `XPath` object that can be used to evaluate XPath expressions.  
  Returns: An `XPath` object

Package `org.w3c.dom`

Interface `org.w3c.dom.Document`
- `Element createElement(String tagName)`  
  This method creates a new document element with a given tag.  
  Parameters: `tagName` The name of the XML tag  
  Returns: The created element
- `Text createTextNode(String text)`  
  This method creates a text node with the given text.  
  Parameters: `text` The text for the text node  
  Returns: The created text node
- `DOMImplementation getImplementation()`  
  This method returns the `DOMImplementation` object associated with this document.

Interface `org.w3c.dom.DOMConfiguration`
- `void setParameter(String name, Object value)`  
  This method sets the value of a configuration parameter.  
  Parameters: `name` The name of the parameter to set  
  `value` The new value or `null` to unset the parameter

Interface `org.w3c.dom.DOMImplementation`
- `Object getFeature(String feature, String version)`  
  This method gets an object that implements a specialized API (such as loading and saving of DOM trees).  
  Parameters: `feature` The feature version (such as "LS")  
  `version` The version number (such as "3.0")  
  Returns: The feature object

Package `org.w3c.dom.ls`

Interface `org.w3c.dom.ls.DOMImplementationLS`
- `LSSerializer createLSSerializer()`  
  This method creates a serializer object that can be used to convert a DOM tree to a string or stream.  
  Returns: The serializer object
- `DOMConfiguration getDomConfig()`  
  This method gets the configuration object that allows customization of the serializer behavior.
- `String writeToString(Node root)`  
  This method converts the DOM tree starting at the given node to a string.  
  Parameters: `node` The root node of the tree  
  Returns: The string representation of the tree
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:

```java
/**
   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.

```java
/**
* Convert calendar date into Julian day.
* @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:

```java
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:

```java
{ double xold; // Don't
double xnew;
boolean done;
... }
```

Define each variable just before it is used for the first time:

```java
{ ...
  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:

```java
int dimes = 0, nickels = 0; // Don't
```
Instead, use two separate definitions:

```java
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:

```java
if (p.getX() < 300) // Don't
```

Use `final` variables instead:

```java
final double WINDOW_WIDTH = 300;
```

```java
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

```java
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.

```java
int[] values; // OK
int values[]; // Ugh—this is an ugly holdover from C
```

When using collections, use type parameters and not “raw” types.

```java
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,

```java
if (x < 0)
{
  x++;
}
```

and not

```java
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:

```java
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, cnt, 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:

```java
    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:

```java
    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.
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
```

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Greeting.java</td>
</tr>
<tr>
<td>2</td>
<td>GreetingTester.java</td>
</tr>
</tbody>
</table>

Then you can compile all files with the command

```
javac @Greeting.list
```

The Java compiler options are summarized in Table 1.

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>-classpath locations</code></td>
<td>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 <code>location</code> is a directory, JAR file, or ZIP file. Locations are separated by a platform-dependent separator (: on Unix, ; on Windows).</td>
</tr>
<tr>
<td><code>-sourcepath locations</code></td>
<td>The compiler is to look for source files on this path. If not specified, source files are searched in the class path.</td>
</tr>
<tr>
<td><code>-d directory</code></td>
<td>The compiler places files into the specified directory.</td>
</tr>
<tr>
<td><code>-g</code></td>
<td>Generate debugging information.</td>
</tr>
<tr>
<td><code>-verbose</code></td>
<td>Include information about all classes that are being compiled (useful for troubleshooting).</td>
</tr>
<tr>
<td><code>-deprecation</code></td>
<td>Give detailed information about the usage of deprecated messages.</td>
</tr>
<tr>
<td><code>-Xlint:errorType</code></td>
<td>Carry out additional error checking. If you get warnings about unchecked conversions, compile with the <code>-Xlint:unchecked</code> option.</td>
</tr>
</tbody>
</table>
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
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
java [options] -jar jarFileName [argument1 argument2 ...]
```

The Java virtual machine options are summarized in Table 2.

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-classpath locations or -cp locations</td>
<td>Look for classes on this path, overriding the CLASSPATH environment variable. If neither is specified, the current directory is used. Each location is a directory, JAR file, or ZIP file. Locations are separated by a platform-dependent separator (; on Unix, ; on Windows).</td>
</tr>
<tr>
<td>-verbose</td>
<td>Trace class loading</td>
</tr>
<tr>
<td>-D property=value</td>
<td>Set a system property that you can retrieve with the System.getProperties method.</td>
</tr>
</tbody>
</table>

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`

```
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:

```bash
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)

```bash
javadoc -link http://download.oracle.com/javase/8/docs/api -d docdir *.java
```

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-link URL</td>
<td>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>.</td>
</tr>
<tr>
<td>-d directory</td>
<td>Store the output in directory. This is a useful option, because it keeps your current directory from being cluttered up with javadoc files.</td>
</tr>
<tr>
<td>-classpath locations</td>
<td>Look for classes on the specified paths, overriding the CLASSPATH environment variable. If neither is specified, the current directory is used. Each location is a directory, JAR file, or ZIP file. Locations are separated by a platform-dependent separator (: Unix, ; Windows).</td>
</tr>
<tr>
<td>-sourcepath locations</td>
<td>Look for source files on the specified paths. If not specified, source files are searched in the class path.</td>
</tr>
<tr>
<td>-author, -version</td>
<td>Include author, version information in the documentation. This information is omitted by default.</td>
</tr>
</tbody>
</table>

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.

```java
/**
   * 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>
<thead>
<tr>
<th><strong>Table 4</strong> Common javadoc Tags</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Tag</strong></td>
</tr>
<tr>
<td><strong>Description</strong></td>
</tr>
<tr>
<td>@param <em>parameter explanation</em></td>
</tr>
<tr>
<td>A parameter of a method. Use a separate tag for each parameter.</td>
</tr>
<tr>
<td>@return <em>explanation</em></td>
</tr>
<tr>
<td>The return value of a method.</td>
</tr>
<tr>
<td>@throws <em>exceptionType explanation</em></td>
</tr>
<tr>
<td>An exception that a method may throw. Use a separate tag for each exception.</td>
</tr>
<tr>
<td>@deprecated</td>
</tr>
<tr>
<td>A feature that remains for compatibility but that should not be used for new code.</td>
</tr>
<tr>
<td>@see { <em>packageName</em>.ClassName }</td>
</tr>
<tr>
<td>A reference to a related documentation entry.</td>
</tr>
<tr>
<td>@see { <em>packageName</em>.ClassName }#methodName( <em>Type1</em>, <em>Type2</em>, ... )</td>
</tr>
<tr>
<td>@see { <em>packageName</em>.ClassName }#variableName</td>
</tr>
<tr>
<td>@author</td>
</tr>
<tr>
<td>The author of a class or interface. Use a separate tag for each author.</td>
</tr>
<tr>
<td>@version</td>
</tr>
<tr>
<td>The version of a class or interface.</td>
</tr>
</tbody>
</table>
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.

\[
1.101_{\text{binary}} = 1 \times 2^0 + 1 \times 2^{-1} + 0 \times 2^{-2} + 1 \times 2^{-3}
\]

\[
= 1 + \frac{1}{2} + \frac{1}{8}
\]

\[
= 1 + 0.5 + 0.125 = 1.625
\]

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,

\[
100 \div 2 = 50 \text{ remainder } 0
\]

\[
50 \div 2 = 25 \text{ remainder } 0
\]

\[
25 \div 2 = 12 \text{ remainder } 1
\]

\[
12 \div 2 = 6 \text{ remainder } 0
\]

\[
6 \div 2 = 3 \text{ remainder } 0
\]

\[
3 \div 2 = 1 \text{ remainder } 1
\]

\[
1 \div 2 = 0 \text{ remainder } 1
\]

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,

\[
\begin{align*}
0.35 \cdot 2 &= 0.7 \\
0.7 \cdot 2 &= 1.4 \\
0.4 \cdot 2 &= 0.8 \\
0.8 \cdot 2 &= 1.6 \\
0.6 \cdot 2 &= 1.2 \\
0.2 \cdot 2 &= 0.4
\end{align*}
\]

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>
<thead>
<tr>
<th>Power</th>
<th>Decimal Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>2^0</td>
<td>1</td>
</tr>
<tr>
<td>2^1</td>
<td>2</td>
</tr>
<tr>
<td>2^2</td>
<td>4</td>
</tr>
<tr>
<td>2^3</td>
<td>8</td>
</tr>
<tr>
<td>2^4</td>
<td>16</td>
</tr>
<tr>
<td>2^5</td>
<td>32</td>
</tr>
<tr>
<td>2^6</td>
<td>64</td>
</tr>
<tr>
<td>2^7</td>
<td>128</td>
</tr>
<tr>
<td>2^8</td>
<td>256</td>
</tr>
<tr>
<td>2^9</td>
<td>512</td>
</tr>
<tr>
<td>2^{10}</td>
<td>1,024</td>
</tr>
<tr>
<td>2^{11}</td>
<td>2,048</td>
</tr>
<tr>
<td>2^{12}</td>
<td>4,096</td>
</tr>
<tr>
<td>2^{13}</td>
<td>8,192</td>
</tr>
<tr>
<td>2^{14}</td>
<td>16,384</td>
</tr>
<tr>
<td>2^{15}</td>
<td>32,768</td>
</tr>
<tr>
<td>2^{16}</td>
<td>65,536</td>
</tr>
</tbody>
</table>
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,

```java
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 1 0 1 0 1 1 1 1 0 0 0 1 0 0 0 0 0 0
1 0 1 1 1 1 1 0 1 0 1 1 1 1 0 0 0 0 1 0 0 0 0 0 0
1 0 1 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 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 1 0 0 1 0 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:

```java
double price = 4.35;
double quantity = 100;
double total = price * quantity; // Should be 100 * 4.35 = 435
System.out.println(total); // Prints 434.99999999999999
```

To see why the error occurs, carry out the long multiplication:

```
1 1 0 0 1 0 0 * 1 0 0.0 1 0 1 1 0|0 1 1 0|0 1 1 0|0 1 1 0...
1 0 0.0 1|0 1 1 0|0 1 1 0|0 1 1 0...
1 0 0.0 1|0 1 1 0|0 1 1 0|0 1 1...
0
0
1 0 0.0 1|0 1 1 0|0 1 1 0...
0
0
--------------------------------
1 1 0 1 1 0 0 1 0.1 1 1 1 1 1 1 1 1 1
```

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 ..., 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,

<p>| +13 | 0000 1101 |</p>
<table>
<thead>
<tr>
<th>-13</th>
<th>1111 0011</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 0000 0000</td>
<td></td>
</tr>
</tbody>
</table>

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 . . . 1111, 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:

```java
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

\[ b_0.b_1b_2b_3 \ldots \times 2^e \]

In this representation, \( e \) is the exponent, and the digits \( b_0.b_1b_2b_3 \ldots \) form the mantissa. The *normalized* representation is the one where \( b_0 \neq 0 \). For example,

\[
100_{\text{decimal}} = 1100100_{\text{binary}} = 1.100100_{\text{binary}} \times 2^6
\]

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,

\[
100_{\text{decimal}} = 01000010110010000000000000000000_{\text{single-precision IEEE}}
\]

In addition, there are several special values. Among them are:

- **Zero**: biased exponent = 0, mantissa = 0.
- **Infinity**: biased exponent = 11...1, mantissa = ±0.
- **NaN (not a number)**: biased exponent = 11...1, mantissa ≠ ±0.

```
1 bit  8 bit  23 bit
| sign | biased exponent \( e + 127 \) | mantissa (without leading 1) |
--- | --- | ---

Single Precision

1 bit  11 bit  52 bit
| sign | biased exponent \( e + 1023 \) | mantissa (without leading 1) |

Double Precision

**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,

\[111010001\text{binary} = \text{3B1}\text{hexadecimal}\]

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>
<thead>
<tr>
<th>Table 2 Hexadecimal Digits</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Hexadecimal</strong></td>
</tr>
<tr>
<td>0</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>2</td>
</tr>
<tr>
<td>3</td>
</tr>
<tr>
<td>4</td>
</tr>
<tr>
<td>5</td>
</tr>
<tr>
<td>6</td>
</tr>
<tr>
<td>7</td>
</tr>
<tr>
<td>8</td>
</tr>
<tr>
<td>9</td>
</tr>
<tr>
<td>A</td>
</tr>
<tr>
<td>B</td>
</tr>
<tr>
<td>C</td>
</tr>
<tr>
<td>D</td>
</tr>
<tr>
<td>E</td>
</tr>
<tr>
<td>F</td>
</tr>
</tbody>
</table>
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_{decimal} = 101110_{binary} (actually 00000000000000000000000000101110 as a 32-bit integer), and 13_{decimal} = 1101_{binary}. Now combine corresponding bits:

\[
\begin{array}{c}
0 & \cdots & 01110 \\
& \cdots & 00110 \\
\end{array}
\]

The answer is 1100_{binary} = 12_{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:

\[
\begin{array}{c}
0 & \cdots & 00001 \\
| & 0 & \cdots & 00010 \\
& & 0 & \cdots & 00011 \\
\end{array}
\]

Don’t confuse the & and | bit operators with the && and || operators. The latter work only on boolean values, not on bits of numbers.

<table>
<thead>
<tr>
<th>Table 3 The Unary Negation Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
</tr>
<tr>
<td>---</td>
</tr>
<tr>
<td>0</td>
</tr>
<tr>
<td>1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Table 4 The Binary And, Or, and Xor Operations</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
</tr>
<tr>
<td>---</td>
</tr>
<tr>
<td>0</td>
</tr>
<tr>
<td>0</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>1</td>
</tr>
</tbody>
</table>

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, $1 << 35$ is the same as $1 << 3$. Actually shifting 1 by 35 bits to the left would make no sense—the result would be 0.

The expression

$$1 << n$$

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

$$x = x | 1 << n$$

To check whether the $n$th bit is set, execute the test

```c
if ((x & 1 << n) != 0) . . .
```

Note that the parentheses around the & are required—the & operator has a lower precedence than the relational operators.
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)
Table 1 shows the arrows used to indicate relationships between classes. Multiplicity can be indicated in a diagram, as in Figure 4.

<table>
<thead>
<tr>
<th>Relationship</th>
<th>Symbol</th>
<th>Line Style</th>
<th>Arrow Tip</th>
</tr>
</thead>
<tbody>
<tr>
<td>Inheritance</td>
<td></td>
<td>Solid</td>
<td>Triangle</td>
</tr>
<tr>
<td>Interface</td>
<td></td>
<td>Dotted</td>
<td>Triangle</td>
</tr>
<tr>
<td>Implementation</td>
<td></td>
<td>Solid</td>
<td>Diamond</td>
</tr>
<tr>
<td>Dependency</td>
<td></td>
<td>Dotted</td>
<td>Open</td>
</tr>
</tbody>
</table>

Dependencies between objects are described by a dependency diagram. Figure 5 is a typical example.
State diagrams are used when an object goes through a discrete set of states that affects its behavior (see Figure 6).


**Figure 6** UML State Diagram for the ATM Class
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/](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:

```java
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

```java
modifiers constructor|method|instance variable|class
```

or an initialization block

```java
[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

```java
Type variableName [= initializer];
```

A constructor has the form

```java
ClassName(parameter1, parameter2, . . .)
[throws ExceptionType1, ExceptionType2, . . .]
{
  body
}
```

A method has the form

```java
Type methodName(parameter1, parameter2, . . .)
[throws ExceptionType1, ExceptionType2, . . .]
{
  body
}
```

An abstract method has the form

```java
abstract Type methodName(parameter1, parameter2, . . .);
```

Here is an example:

```java
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(parameter1, parameter2, ...);
```

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((parameter1, parameter2, ...))
```
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:

```java
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, . . ., parameter_n)
    [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

```java
public static double sum(double... values)
{
    double s = 0;
    for (double v : values) { s = s + v; }
    return s;
}
```

can be called as

```java
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

\[
\text{implicitParameterValue.methodName(\text{parameterValue}_1, \text{parameterValue}_2, \ldots)}
\]

Example:

```java
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,

```java
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,

```java
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

```java
return [\text{value}];
```

For example,

```java
public double getSalary()
{
    return salary;
}
```

A method can call itself. Such a method is called *recursive*:

```java
public static int factorial(int n)
{
    if (n <= 1) { return 1; }
    return n * factorial(n - 1);
}
```
Constructors

A constructor definition has the form

\[
\text{modifiers \ ClassName}(\text{parameter}_1, \text{parameter}_2, \ldots ) \\[\text{throws \ ExceptionType}_1, \text{ExceptionType}_2, \ldots ]
\{ \\
\text{body} \\
\}
\]

You invoke a constructor to allocate and construct a new object with a \texttt{new} expression

\[
\text{new \ ClassName}(\text{parameterValue}_1, \text{parameterValue}_2, \ldots )
\]

A constructor can call the body of another constructor of the same class with the syntax

\[
\text{this}(\text{parameterValue}_1, \text{parameterValue}_2, \ldots )
\]

For example,

\[
\text{public Employee()}
\{ \\
\quad \text{this}(\text{""}, 0);
\}
\]

It can call a constructor of its superclass with the syntax

\[
\text{super}(\text{parameterValue}_1, \text{parameterValue}_2, \ldots )
\]

The call to \texttt{this} or \texttt{super} must be the first statement in the constructor.

Arrays are constructed with the syntax

\[
\text{new \ ArrayType \[ = \{ \text{initializer}_1, \text{initializer}_2, \ldots \} }
\]

For example,

\[
\text{new int[]} = \{ 1, 4, 9, 16, 25 \}
\]

When an object is constructed, the following actions take place:

- All instance variables are initialized with 0, \texttt{false}, or \texttt{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, \texttt{false}, or \texttt{null}.
- The initializers of static variables and static initialization blocks are executed in the order in which they are declared.

Statements

A \textit{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:

```java
for (i = 0; i < 10; i++)
{
    sum = sum + i;
}
```

```java
for (int i = 0, j = 9; i < 10; i++, j--)
{
    a[j] = b[i];
}
```

The enhanced `for` loop has the form

```java
for (Type variable : array|iterableObject)
    statement
```

When this loop traverses an array, it is equivalent to

```java
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

```java
Iterator i = iterableObject.iterator();
while (i.hasNext())
{
    Type variable = i.next();
    statement
}
```

The `do` loop has the form

```java
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:

```java
label: statement
```

The labeled break statement

```java
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

```java
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 catch-
Block 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

```java
import packageName.ClassName;
```

or

```java
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:

```java
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:

```java
modifiers <typeParameter1, typeParameter2, ...> returnType methodName
```

Each type parameter has the form

```java
typeParameterName [extends bound1 & bound2 & ...]
```

For example,

```java
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

```java
? [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:

```java
/* 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.
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 <i>object-oriented</i> 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 <i>object-oriented</i> 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>
<ul><li>object-oriented</li>
<li>safe</li>
<li>platform-independent</li></ul>

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>
<thead>
<tr>
<th>Tag</th>
<th>Meaning</th>
<th>Children</th>
<th>Commonly Used Attributes</th>
</tr>
</thead>
<tbody>
<tr>
<td>html</td>
<td>HTML document</td>
<td>head, body</td>
<td></td>
</tr>
<tr>
<td>head</td>
<td>Head of an HTML document</td>
<td></td>
<td>title</td>
</tr>
<tr>
<td>title</td>
<td>Title of an HTML document</td>
<td></td>
<td></td>
</tr>
<tr>
<td>body</td>
<td>Body of an HTML document</td>
<td></td>
<td></td>
</tr>
<tr>
<td>h1 ... h6</td>
<td>Heading level 1 ... 6</td>
<td></td>
<td></td>
</tr>
<tr>
<td>p</td>
<td>Paragraph</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ul</td>
<td>Unnumbered list</td>
<td>li</td>
<td></td>
</tr>
<tr>
<td>ol</td>
<td>Ordered list</td>
<td>li</td>
<td></td>
</tr>
<tr>
<td>dl</td>
<td>Definition list</td>
<td>dt, dd</td>
<td></td>
</tr>
<tr>
<td>li</td>
<td>List item</td>
<td></td>
<td></td>
</tr>
<tr>
<td>dt</td>
<td>Term to be defined</td>
<td></td>
<td></td>
</tr>
<tr>
<td>dd</td>
<td>Definition data</td>
<td></td>
<td></td>
</tr>
<tr>
<td>table</td>
<td>Table</td>
<td>tr</td>
<td></td>
</tr>
<tr>
<td>tr</td>
<td>Table row</td>
<td>th, td</td>
<td></td>
</tr>
<tr>
<td>th</td>
<td>Table header cell</td>
<td></td>
<td></td>
</tr>
<tr>
<td>td</td>
<td>Table cell data</td>
<td></td>
<td></td>
</tr>
<tr>
<td>a</td>
<td>Anchor</td>
<td></td>
<td>href, name</td>
</tr>
<tr>
<td>img</td>
<td>Image</td>
<td></td>
<td>src, width, height</td>
</tr>
<tr>
<td>pre</td>
<td>Preformatted text</td>
<td></td>
<td></td>
</tr>
<tr>
<td>hr</td>
<td>Horizontal rule</td>
<td></td>
<td></td>
</tr>
<tr>
<td>br</td>
<td>Line break</td>
<td></td>
<td></td>
</tr>
<tr>
<td>i or em</td>
<td>Italic</td>
<td></td>
<td></td>
</tr>
<tr>
<td>b or strong</td>
<td>Bold</td>
<td></td>
<td></td>
</tr>
<tr>
<td>tt or code</td>
<td>Typewriter or code font</td>
<td></td>
<td></td>
</tr>
<tr>
<td>s or strike</td>
<td>Strike through</td>
<td></td>
<td></td>
</tr>
<tr>
<td>u</td>
<td>Underline</td>
<td></td>
<td></td>
</tr>
<tr>
<td>super</td>
<td>Superscript</td>
<td></td>
<td></td>
</tr>
<tr>
<td>sub</td>
<td>Subscript</td>
<td></td>
<td></td>
</tr>
<tr>
<td>form</td>
<td>Form</td>
<td></td>
<td>action, method</td>
</tr>
</tbody>
</table>
### Table 1  Selected HTML Tags

<table>
<thead>
<tr>
<th>Tag</th>
<th>Meaning</th>
<th>Children</th>
<th>Commonly Used Attributes</th>
</tr>
</thead>
<tbody>
<tr>
<td>input</td>
<td>Input field</td>
<td></td>
<td>type, name, value, size, checked</td>
</tr>
<tr>
<td>select</td>
<td>Combo box style selector</td>
<td>option</td>
<td>name</td>
</tr>
<tr>
<td>option</td>
<td>Option for selection</td>
<td></td>
<td></td>
</tr>
<tr>
<td>textarea</td>
<td>Multiline text area</td>
<td></td>
<td>name, rows, cols</td>
</tr>
</tbody>
</table>

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:

```html
<p>Java is</p>
<ul>
<li>object-oriented</li>
<li>safe</li>
<li>platform-independent</li>
</ul>
```

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

```html
<img src="hamster.jpeg"/>
```

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

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
<th>Commonly Contained in</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>Name of form element or anchor</td>
<td>input, select, textarea, a</td>
</tr>
<tr>
<td>href</td>
<td>Hyperlink reference</td>
<td>a</td>
</tr>
<tr>
<td>src</td>
<td>Source (as of an image)</td>
<td>img</td>
</tr>
<tr>
<td>code</td>
<td>Applet code</td>
<td>applet</td>
</tr>
<tr>
<td>width, height</td>
<td>Width, height of image or applet</td>
<td>img, applet</td>
</tr>
<tr>
<td>rows, cols</td>
<td>Rows, columns of text area</td>
<td>textarea</td>
</tr>
<tr>
<td>type</td>
<td>Type of input field, such as text, password, checkbox, radio, submit, hidden</td>
<td>input</td>
</tr>
<tr>
<td>value</td>
<td>Value of input field, or label of submit button</td>
<td>input</td>
</tr>
</tbody>
</table>
Table 2  Selected HTML Attributes

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
<th>Commonly Contained in</th>
</tr>
</thead>
<tbody>
<tr>
<td>size</td>
<td>Size of text field</td>
<td><code>input</code></td>
</tr>
<tr>
<td>checked</td>
<td>Check radio button or checkbox</td>
<td><code>input</code></td>
</tr>
<tr>
<td>action</td>
<td>URL of form action</td>
<td><code>form</code></td>
</tr>
<tr>
<td>method</td>
<td>GET or POST</td>
<td><code>form</code></td>
</tr>
</tbody>
</table>

It is considered polite to use several additional attributes with the `img` tag, namely the `image size` and an `alternate description`:

```html
<img src="hamster.jpeg" width="640" height="480"
alt="A photo of Harry, the Horrible Hamster"/>
```

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` for more information on XHTML.)

The most important tag on a web page is the `<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:

```html
```

When the viewer of the web page clicks on the words Cay Horstmann, 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

<table>
<thead>
<tr>
<th>Entity</th>
<th>Description</th>
<th>Appearance</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>&amp;lt;</code></td>
<td>Less than</td>
<td><code>&lt;</code></td>
</tr>
<tr>
<td><code>&amp;gt;</code></td>
<td>Greater than</td>
<td><code>&gt;</code></td>
</tr>
<tr>
<td><code>&amp;amp;</code></td>
<td>Ampersand</td>
<td><code>&amp;</code></td>
</tr>
<tr>
<td><code>&amp;quot;</code></td>
<td>Quotation mark</td>
<td><code>&quot;</code></td>
</tr>
<tr>
<td><code>&amp;nbsp;</code></td>
<td>Nonbreaking space</td>
<td></td>
</tr>
<tr>
<td><code>&amp;copy;</code></td>
<td>Copyright symbol</td>
<td><code>©</code></td>
</tr>
</tbody>
</table>
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.
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.
Array list  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.
Balanced tree  A tree in which each subtree has the property that the number of descendants to the left is approximately the same as the number of descendants to the right.
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 file  A file in which values are stored in their binary representation and cannot be read as text.
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.
Binary search tree  A binary tree in which each subtree has the property that all left descendants are smaller than the value stored in the root, and all right descendants are larger.
Binary tree  A tree in which each node has at most two child nodes.
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 \{\}. 
Boolean  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.

**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 \((\text{int}) x\).

**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.

**Client**  A computer program or system that issues requests to a server and processes the server responses.

**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.

**Collector**  An object that collects elements, usually from a stream, combining them into a result.

**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.

**Condition object**  An object that manages threads that currently cannot proceed.

**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.
**Constructor reference**  An expression of the form `Class::new` or `Class[]::new` that, like a lambda expression, can be converted into an instance of a functional interface.

**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.

**Deadlock**  A state in which no thread can proceed because each thread is waiting for another to do some work first.

**Deadly embrace**  A set of blocked threads, each of which could only be unblocked by the action of other threads in the set.

**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.

**DTD (Document Type Definition)**  A sequence of rules that describes the legal child elements and attributes for each element type in an SGML or XML document.

**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.

**Embedded system**  The processor, software, and supporting circuitry that is included in a device other than a computer.

**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.

File pointer  The position within a random-access file of the next byte to be read or written. It can be moved so as to access any byte in the file.

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.

Flushing an output stream  Sending all characters that are still held in a buffer to their destination.

Folder  See Directory.

Font  A set of character shapes in a particular style and size.

Foreign key  A reference to a primary key in a linked table.

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.

Generic method  A method with one or more type parameters.

Generic programming  Providing program components that can be reused in a wide variety of situations.

Grammar  A set of rules that specifies which sequences of tokens are legal for a particular document set.

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 two files.

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.

Heap  A balanced binary tree that is used for implementing sorting algorithms and priority queues.

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.

Higher-order function  A function that has another function as a parameter, or that yields another function.

HTML (Hypertext Markup Language)  The language in which web pages are described.

HTTP (Hypertext Transfer Protocol)  The protocol that defines communication between web browsers and web servers.

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.

Infinite stream  A stream with an infinite number of elements.

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.

Internet  A worldwide collection of networks, routing equipment, and computers using a common set of protocols that define how participants interact with each other.

Iterator  An object that can inspect all elements in a container such as a linked list.

JavaBean  A class with a no-argument constructor that exposes properties through its get and set methods.

javadoc  The documentation generator in the Java SDK. It extracts documentation comments from Java source files and produces a set of linked HTML files.

JavaServer Faces (JSF)  A framework for developing web applications that aids in the separation of user interface and program logic.

JDBC (Java Database Connectivity)  The technology that enables a Java program to interact with relational databases.

JDK  The Java software development kit that contains the Java compiler and related development tools.

Join  A database query that involves multiple tables.

JSF container  A program that executes JSF applications.

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 \(-2, 3.14, 6.02214115E23, \text{"Harry"}, \text{"H"} \).

Local variable  A variable whose scope is a block.

Lock  A data structure to regulate the scheduling of multiple threads. Once a thread has acquired a lock, other threads that also wish to acquire it must wait until the first thread relinquishes it.

Lock object  An object that allows a single thread to execute a section of a program.

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.

**Managed bean**  A JavaBean that is managed by a JSF container.

**Map**  A data structure that keeps associations between key and value objects.

**Markup**  Information about data that is added as humanly readable instructions. An example is the tagging of HTML documents with elements such as `<h1>` or `<b>`.

**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.

**Metadata**  Data that describe properties of a data set.

**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.

**Method expression**  In JSF, an expression describing a bean and a method that is to be applied to the bean at a later time.

**Method reference**  An expression of the form `Class::method` or `object::method`, where `method` is a lambda expression, can be converted into an instance of a functional interface.

**Mixed content**  In XML, a markup element that contains both text and other elements.

**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.

**Navigation rule**  In JSF, a rule that describes when to move from one web page to another.

**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 \(a \lor b \land c\) is interpreted as \(a \lor (b \land c)\). (See Appendix B.)

**Optional value** An instance of the `Optional<T>` class, representing either a value of type `T` or the absence of such a value.

**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.

**Parallel arrays** Arrays of the same length, in which corresponding elements are logically related.

**Parallel stream** A stream that attempts to apply operations in parallel.

**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.

**Parse tree** A tree structure that shows how a string conforms to the rules of a grammar.

**Parser** A program that reads a document, checks whether it is syntactically correct, and takes some action as it processes the document.

**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.

**Postfix operator** A unary operator that is written after its argument.

**Prefix operator** A unary operator that is written before its argument.

**Prepared statement** A SQL statement with a precomputed query strategy.

**Primary key** A column (or combination of columns) whose value uniquely specifies a table record.

**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.

**Property** A named value that is managed by a component.

**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.

**Race condition** A condition in which the effect of multiple threads on shared data depends on the order in which the threads are scheduled.

**Radio button** A user-interface component that can be used for selecting one of several options.

**Random access** The ability to access any value directly without having to read the values preceding it.

**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.
Red-black tree  A kind of binary search tree that rebalances itself after each insertion and removal.

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 database  A data repository that stores information in tables and retrieves data as the result of queries that are formulated in terms of table relationships.

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 * 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.

Runnable thread  A thread that can proceed provided it is given a time slice to do work.

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 access  Accessing values one after another without skipping over any of them.

Sequential search  See Linear search.

Serialization  The process of saving an object, and all the objects that it references, to a stream.

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.

Sign bit  The bit of a binary number that indicates whether the number is positive or negative.

Socket  An object that encapsulates a TCP/IP connection. To communicate with the other endpoint of the connection, you use the input and output streams attached to the socket.

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.

SQL (Structured Query Language)  A command language for interacting with a database.

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.

**Stored procedure**  A database procedure that is executed in the database kernel.

**Stream**  A data structure representing a sequence of elements, each of which is visited at most once, that enables operations to be applied in an efficient way (for example, lazily, or in parallel).

**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.

**Synchronized block**  A block of code that is controlled by a lock. To start execution, a thread must acquire the lock. Upon completion, it relinquishes the lock.

**Synchronized method**  A method that is controlled by a lock. In order to execute the method, the calling thread must acquire the lock.

**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.

**TCP/IP (Transmission Control Protocol/Internet Protocol)**  The pair of communication protocols that is used to establish reliable transmission of data between two computers on the Internet.

**Terminal operation**  An operation on a stream that yields a value, causing any required pending stream operations to be executed.

**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.

**Thread**  A program unit that is executed independently of other parts of the program.

**Three-tier application**  An application that is composed of separate tiers for presentation logic, business logic, and data storage.

**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.

**Time slice**  A small amount of time used when scheduling threads. Each thread is given a small amount of time (a slice) in which to do its work, then control is given to another thread.

**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.

**Transaction**  A set of database operations that should either succeed in their entirety, or not happen at all.
Tree A data structure consisting of nodes, each of which has a list of child nodes, and one of which is distinguished as the root node.

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.

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.

Type variable A variable in the declaration of a generic type that can be instantiated with a 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.

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.

Value expression In JSF, an expression describing a bean and a property that is to be accessed at a later time.

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.


Web application An application that executes on a web server and whose user interface is displayed in a web browser.

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.

XML (Extensible Markup Language) A simple format for structured data in which the structure is indicated by markup instructions.
Index

Page references followed by t indicate material in tables.

Symbols

&& (ampersands), and operator, 209–210, 212, 213

* (asterisk), multiplication operator, 36, 137

\ (backslash)
  escape sequence, 156, 523
  in file names, 523

{ } (braces)
  enclosing blocks of code, 201–202
  lining up code, 181

< > (brackets), diamond syntax, 352

$ (dollar sign), in variable names, 37

. (dot), name syntax, 403

= (equal sign), assignment operator, 38–39

== (equal signs), relational operator
  comparing object references, 187
  comparing strings, 186, 189–190
  equality testing, 184–185
  versus equals method, 186
  syntax, 184
  testing for null, 187–189

! (exclamation point), not operator, 211

!= (exclamation point, equal), relational operator, 184

> (greater than), relational operator, 184

>= (greater than or equal), relational operator, 184

-> (hyphen, greater than), arrow, 485, 855

< (less than), relational operator, 184

<= (less than or equal), relational operator, 184

- (minus sign), subtraction operator, 36

-- (minus signs), decrement, 138

( ) (parentheses)
  in arithmetic operations, 137, 142
  in object construction, 47
  balancing, 142

% (percent sign), modulus
  avoiding negative remainders, 143
  description, 138–139

full code example, 140

+ (plus sign)
  addition operator, 36
  concatenation operator, 155

++ (plus signs), increment, 138

" (quotation marks), string delimiters, 12

/ (slash), division operator, 36, 137

// (slashes), comment delimiter, 38

/*...*/ (slash asterisk ...), comment delimiters, 38

/** (slash, asterisks), comment indicator, 88

; (semicolon)
  ending if statements, 182
  ending method statements, 11
  omitting, 13

_ (underscore), in variable names, 37

|| (vertical bars), or operator, 209–210, 212, 213

A

abs method, java.lang.Math class, 140t, A-18

abstract classes, 443–444

abstract data types, 741

abstract methods, 443–444

access specifiers, 81

accessor methods
  definition, 379–380
  description, 48–49
  full code example, 49

acos method, java.lang.Math class, A-19–20

actionPerformed method, java.awt.event.ActionListener interface creating buttons, 496–498, 499

description, A-14

reading text fields, 889–890

actor classes, 376

add method
  java.awt.Container class, A-13
  java.math.BigDecimal class, 136, A-22
  java.math.BigInteger class, 136, A-22

java.util.ArrayList class, 344, 697

interface, A-26

java.util.Collection interface, A-26

java.util.Iterator interface, 688


java.util.Queue interface, 699

javax.swing.ButtonGroup class, A-33

javax.swing.JMenu class, A-34

javax.swing.JMenuBar class, A-34


addChangeListener method, javax.swing.JSlider class, 914–916, A-35

addFirst method, java.util.LinkedList class, 723–724, A-28

addItem method, javax.swing.JComboBox class, A-33

addNode method, Node class, 778

Address class, 585–586

class, A-33

addKeyListener method, java.awt.Component class, A-12

addLast method, java.util.LinkedList class, 730–731, A-28

addMouseListener method, java.awt.Component class, 503, A-12

aggregation, 570–571, 574–575

algebraic expressions
  evaluating with stacks, 703–706
  full code example, 706

precedence, 704

algorithms. See also loops, common algorithms; streams, common algorithms.

class, 570–571, 574–575

class, 570–571, 574–575

development, 332–335

evolving into programs, 18–19

evolution, 18

equality testing, 184–185

equation, 138

evaluation, 138–139

evaluation, 138–139

evaluation, 138–139

evaluation, 138–139

evaluation, 138–139

evaluation, 138–139

evaluation, 138–139

evaluation, 138–139
algorithms (continued)
executable, 16
full code examples, 329, 335
pseudocode for, 19
reusing, 327–332
terminating, 16
for tiling a floor, 21–22
unambiguous, 16
allMatch method, java.util.stream.
Stream<T> interface, 862, A-32
Altair 8800, 407
American National Standards
Institute (ANSI), 686
ampersands (&), and operator,
209–210, 212, 213
Analytical Engine, 658
ancestors, 767
Andreesen, Marc, 456
anonymous objects, 488–489
anonymous classes, 488–489
ANSI (American National Standards
Institute), 686
anyMatch method, java.util.stream.
Stream<T> interface, 862, A-32
API (application programming
interface), 50–53
append method, javax.swing.JTextArea
class, 891–893, A-35
Apple II, 407
applets, definition, 6
application development, examples.
See “Hello, World” program;
printing an invoice (sample
program).
application programming interface
(API), 50–53
approximate equality, 185
approximate solutions, finding,
281–282
AreaMeasurer.java class, 484
arguments
definition, 12
method, 42–43
object construction, 47
Ariane rocket incident, 554
arithmetic operations. See also data
types; specific operations.
combining with assignment, 143
computation overflow, 131
hand tracing, 152–154
integer division, 138–139, 142
mathematical methods, 140
modulus, 138–139
powers, 139–140
remainders, 138–139
roots, 139–140
rounding, 131–132
roundoff errors, 132
unintended integer division, 142
arithmetic operators. See also specific
operators.
* (asterisk), multiplication, 36, 137
- (minus sign), subtraction, 36
-- (minus signs), decrement, 138
% (percent sign), modulus,
138–139
+ (plus sign), addition, 36
++ (plus signs), increment, 138
/ (slash), division, 36, 137
definition, 137
expressions, 137
parentheses, 137, 142
precedence, 137, A-5
ARPANET, 456
array lists. See also java.util.
ArrayList<E> class.
versus arrays, 350–351
auto-boxing, 347
constructor, 344
converting from array algorithms,
348
copying, 346
declaring, 344–345
definition, 343
diamond syntax, 352
enhanced for loop, 345–346
calling a method, 349
generic class, 344
inserting elements, 348
iterating through, 345–346
overview, 344–345
removing matches, 348–349
size, determining, 352
syntax, 343
type parameter, 344
wrapper classes, 347, 347f
array lists, implementing
adding/removing elements,
738–739
efficiency, 740f
full code example, 740
getting/setting elements, 737–738
writing a program, 739–741
array store exception, 833–834
ArrayList constructor, A-26
arrays. See also enumeration types.
versus array lists, 350–351
bounds errors, 310, 314
companion variable, 312
converting to streams, 847–848
creating from streams, 850
current size, 312
declaring, 308–311
definition, 308
filling, 311, 868–869
fixed length, 311
full code example, 313
How To example, 330–332
implementing stacks as, 743
initialization, 308–310
iterating through, 317–318
length, determining, 312–313
with methods, 312
multidimensional, 343
numbering elements of, 310
of objects, 314–315
overview, 308–310
parallel, 314–315
partially filled, 312–313
printing, 340–341
references, 311
resizing, 311, 312–313
sequences of related values, 314
size, determining, 352
sorting, 327
syntax, 309
underestimating data size, 327
unfilled, 314
uninitialized, 314
variable row lengths, 341–342
world population table (Worked
Example), 341
arrays, common algorithms
averaging elements, 319
binary search, 320
copying arrays, 323–324
element position, finding, 320
array separators, 319–320
filing, 318
growing arrays, 323–324
inserting elements, 321
linear search, 320
maximum value, finding, 319
minimum value, finding, 319
reading input, 324–326
removing elements, 320–321
binarystream, 321
binarysearch, 320
binarysearchimplementation, 773–774
BinarySearcher.java class, 656–657
BinarySearchTree.java class, 780–783
black boxes, 83
black-box testing, 206
blocks of code
definition, 201–202
classifying with braces, 201–202
Bluej environment, 54–55
Booch, Grady, 378
Boolean data type, A-17
Boolean constructor, A-17
Boolean operators
&& (ampersands), and operator, 209–210, 212, 213
! (exclamation point), not operator, 211
|| (vertical bars), or operator, 209–210, 212, 213
De Morgan’s Law, 209
full code example, 210
inverting conditions, 211
precedence, 210
short circuiting evaluation of, 213
Boolean variables, in loops, 261
booleanValue method, java.lang.Boolean class, A-17
border layout, 884
BorderLayout constructor, 884–885, A-12
borders, radio buttons, 894–895
boundary test cases, 206–207
bounds errors, arrays, 310, 314
bounds for loops, choosing, 314
averagingDouble method, java.util.stream.Collectors class, A-31
averagingInt method, java.util.stream.Collectors class, A-31
averagingLong method, java.util.stream.Collectors class, A-31
await method, java.util.concurrent.locks.Condition interface, A-30
Babbage, Charles, 658
baby name analysis (Worked Example), 538
backing up files, 10
backslash (\)
escape sequence, 156, 523
in file names, 523
backtracking
definition, 620
Eight Queens problem, 621–626
Four Queens problem, 622–623
full code example, 708
with stacks, 706–708
BadDataException.java class, 553–554
balanced trees, 772–773
balancing parentheses, 701
BankAccount.java class
full code example, 89
sample code, 94–95
BankAccountTester.java class, 101
batch files, 354
Berners-Lee, Tim, 456
BigDecimal type, 136, A-22
BigInteger type, 136, A-22
big-Oh notation
analyzing selection sort performance, 642–644
common growth rates, 645
definition, 644
measuring selection sort performance, 642–644
binary search, 320
binary search trees
binary search property, 775–776
code sample, 780–783
definition, 775
efficiency, 780, 780t
generic classes (Worked Example), 839
inserting nodes, 776–778
insertion points, 778
removing nodes, 778–780
self-organizing structure, 778
binary searching, 655–658
binary trees
balanced trees, 772–773
decision trees, 770
definition, 770
expression trees, 771
full code example, 774
Huffman trees, 771, (Worked Example) 774
implementing, 773–774
binarySearch method
java.util.Arrays class, 664–665
A-26
java.util.Collections class, 664–665, A-27
BinarySearcher.java class, 656–657
BinarySearchTree.java class, 780–783
black boxes, 83
black-box testing, 206
blocks of code
definition, 201–202
classifying with braces, 201–202
Bluej environment, 54–55
Booch, Grady, 378
Boolean data type, A-17
Boolean constructor, A-17
Boolean operators
&& (ampersands), and operator, 209–210, 212, 213
! (exclamation point), not operator, 211
|| (vertical bars), or operator, 209–210, 212, 213
De Morgan’s Law, 209
full code example, 210
inverting conditions, 211
precedence, 210
short circuiting evaluation of, 213
Boolean variables, in loops, 261
booleanValue method, java.lang.Boolean class, A-17
border layout, 884
BorderLayout constructor, 884–885, A-12
borders, radio buttons, 894–895
boundary test cases, 206–207
bounds errors, arrays, 310, 314
bounds for loops, choosing, 256
braces ({}), coding guidelines, A-44
closing blocks of code, 201–202
lining up code, 181
brackets (< >), diamond syntax, 352
breadth-first search, 787–789
break statements, 263–264
breakpoints, 283–285, 598–599
buckets, 748
buffer overrun attacks, 316
bugs, historical first, 287. See also debuggers; debugging.
button-press events, 496–498
buttons (user interface), 496–498
ButtonViewer.java class, 492–493
byte data type, 130
case, transforming, 852
case sensitivity
constants, 133
definition, 9
errors caused by, 15
misspelling words, 15
variable names, 37
cash
autonomous vehicles, 217
drawing, 110–113
CarViewer.java class, 113
case, transforming, 852
case classes
actors, 376
annotated, 488–489
candidates for, 567
collaborators, 568
conecrete, 443–444
consistent, 381–382
CRC (class-responsibility-collaborator) method, 567–569, 572–573
declaring, 11
definition, 11, 33
dependencies, 378–379, 381–382
discovering, 376–377, 566–567
for drawing shapes, 110–114
extending. See inheritance final, 444
identifying, 566–567
immutable, 379–380
importing from packages, 52
inner, 487–488
interfaces, converting from, 476
naming conventions, 37
nouns as, 566
public. See public interfaces,
classes
public interfaces, 41–42
testing, 54–55
utility, 376
classes, common patterns
collecting values, 387–388
counters, 387
full code example, 390
managing object properties, 388
modeling moving objects,
389–391
modeling objects with distinct
states, 388–389
summing totals, 386–387
classes, implementing
constructors, 92–93
How To, 96–99
instance variables, 91
for a menu (Worked Example), 99
methods, 93–95
class-responsibility-collaborator (CRC) method. See CRC
class-responsibility-collaborator) method.
click method, 82
ClickListener.java class, 492
close method, java.lang.Object class, 479–481, A-20
Cloneable interface, 479–481
close method
java.io.InputStream class, A-16
java.io.OutputStream class, A-16
java.io.ObjectInputStream class, A-16
java.io.ObjectOutputStream class, A-16
java.io.PrintStream class, A-16
java.io.PrintWriter class, 521–522, A-16
java.lang.AutoCloseable interface, A-17
java.net.ServerSocket class, A-22
java.net.Socket class, A-22
java.sql.Connection interface, A-24
java.sql.ResultSet interface, A-24
java.sql.Statement interface, A-25
java.util.Scanner class, A-30
code coverage, 206
coding guidelines
{} (braces), A-44
classes, A-40
constants, A-41–42
control flow, A-42–43
for statement, A-42
indentation, A-44
methods, A-41
naming conventions, A-44
overview, A-39
source files, A-40
unstable layout, A-45
variables, A-41–42
white space, A-44
codePoints method, java.lang.String class, 849, 864, 869, A-21
cohesion, classes and methods, 377–378, 381–382
collaborators, 568
collect method, java.util.stream.Stream<T> interface, 850, A-32
collecting values, common class patterns, 387–388
collections
creating from streams, 850
definition, 678
versus streams, 847
collections framework
choosing (How To), 694–695
definition, 678
full code example, 680
lists, 678–679
maps, 679–680, 692–694
priority queues, 679, 699–700
queues, 679, 699–700
sets, 679, 687–691
stacks, 679, 698
collectors, streams, 866
Collectors.joining method, 850
collisions, 696–697, 747
Color constructor, A-12
ColorFrame.java class, 915–916
colors, 66, 66r
ColorViewer.java class, 915
combining strings, 850–851
combo boxes
code samples, 897–900
definition, 896
overview, 896–897
command line arguments, 533–535
comments
// (slashes), delimiter, 38
/*...*/ (slash asterisk...), delimiters, 38
converting to documentation. See javadoc program
definition, 38
documentation, 88–90
public interfaces to classes, 87–90
on variables, 38
commit method, java.sql.Connection interface, A-24
companion variable, 312
Comparable interface, 477–479
Comparator.comparing function, 859
accepts expressions, 667
compare method
java.lang.Double class, A-17
java.lang.Integer class, A-17
java.util.Comparator<T> interface, 666–667, A-27
compareTo method
comparing objects, 477–478
comparing strings, 186
java.lang.Comparable<T> interface, 665, A-18
java.lang.String class, A-20
returned values, 666
comparing method, java.util.Comparator<T> interface, 667, A-27
comparisons
adjacent values, with loops, 271–272
adjacent values, with stream algorithms, 870
approximate equality, 185
floating-point numbers, 185
full code example, 187–189, 665
lexicographic ordering, 186
object contents, 187, 477–478. See also compareTo method; equals method.
object references, 187
objects, 665–667
rectangles, 187
relational operators, 184–185, 212
strings, 186, 189–190
syntax, 184
testing for null, 187–189
collection process, 9
compilers, 6
compile-time errors, 14
composition, 574–575
computation, limits of, 612–613
computation overflow, 131
computer programs, definition, 2. See also Java programs; programming; software.
computer viruses, 316
compilers
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, 155
concrete classes, 443–444
conditional operators, 182–183
consecutive integers, primitive streams, 863–864
consistency, classes and methods, 381–382
console input, 155
constants
case sensitivity, 133
declaring, 134
definition, 132
in interfaces, 473
naming conventions, 133
syntax, 134
constructing objects. See object construction.
constructor references, lambda expressions, 857–858
constructors
calling one from another, 110
declaring as void, 90
implementing classes, 92–93
invoking like methods, 48
versus methods, 85
naming, 86
no-argument, 86
specifying, 85–86
subclasses, 438
superclasses, 438

counters
counting in loops, 268–269
items in a list, 866–868
matches, with stream algorithms, 869
words, 846
counting method, java.util.stream.Collectors class, A-31
CountryValue.java class, 537–538
coupling, 569–570
CPU (central processing unit) definition, 3
simulating. See virtual machines.
CRC (class-responsibility-collaborator) method discovering classes, 576–578
in program design (How To), 572–573
createDirectory method, java.nio.files.Files class, A-23
createElement method, org.w3c.dom.Document interface, A-37
createFile method, java.nio.files.Files class, A-23
createLSSerializer method, org.w3c.dom.ls.DOMImplementationLS interface, A-37
createStatement method, java.sql.Connection interface, A-24
createTextNode method, org.w3c.dom.Document interface, A-37
credit card processing (Worked Example), 275
currentTimeMillis method, java.lang.System class, 640–641, A-21
Curry, Haskell, 856
customer queues, simulating (Worked Example), 708

D
dangling else problem, 201
data types. See also arithmetic operations; specific types.
abstract, 741
boolean, 130t
byte, 130t
char, 130t
double, 130t, 131–132
float, 130t
int, 36, 130t
long, 130t, 131
number literals, 131
numbers with fractions. See double data type; floating-point numbers.
numbers without fractions. See int type.
short, 130t
testing for, 453–455. See also instanceof operator.
DataAnalyzer.java class, 550–551
databases, privacy issues, 586
data class, 470, 484
dataSetReader.java class, 556–557
date constructor, A-27
de Morgan, Augustus, 213
de Morgan’s Law, 213
debuggers
breakpoints, 283–285
definition, 282
inspecting variables, 283–285
overview, 282–285
single stepping, 283–285
single-step command, 284
stepping into/over, 284–285
debugging code
the first bug, 287
How To, 285–286
tracing recursive methods, 598–599
Worked Example, 287
decision trees, 770
decisions. See also comparisons; if statements.
Boolean operations, 209–214
conditional operators, 182–183
declaring
array lists, 344–345
arrays, 308–311
classes, 11
constants, 134
interface variables, 80–81, 106
methods, 45
variables, 34–36, 40–41
decrement, -- (minus signs), 138
default methods
conflicting, 474–475
description, 473–474
full code example, 474
default package, 401
definite loops, 250
delete method, java.nio.files.Files
class, A-23
Denver's luggage handling system, 192
dependencies
classes, 378–379, 381–382, 569–570
methods, 378–379, 381–382
depth-first search, 787–789
descendants, 767
Deutsch, L. Peter, 609
dialog boxes
file choosing, 523–524
full code example, 160, 524
for input/output, 160–161
showInputDialog method, javax.
swing.JOptionPane
class, A-35
showMessageDialog method, javax.
swing.JOptionPane
class, A-35
showOpenDialog method, javax.
swing.JFileChooser
class, A-34
showSaveDialog method, javax.
swing.JFileChooser
class, A-34
diamond syntax, 352
dice. See rolling dice.
dice throws, simulating, 279–280
Die.java class, 279–280
DieSimulator.java class, 280
Difference Engine, 658
Dijkstra, Edsger, 206
dimension constructor, A-13
directories, 9–10. See also files.
discovering classes, 376–377, 566–567, 576–578
distinct method, java.util.stream.
Stream<T> interface, 853, A-32
divide method, java.math.BigInteger
class, A-22
“Division by zero” errors, 14
do loops, 258
documentation
API (application programming
interface), 50–53
comments, 88–90
clearing comments to. See
javadoc program
online help, 53
Swing, 911–918
documenting, methods, 579–581
dollar sign ($), in variable names, 37
dongles, 249
dot (.), name syntax, 403
double constructor, A-18
double data type
definition, 130
t floating-point numbers, 36
for financial calculations, 132
overflow, 131
precision, 131
storing in streams. See java.util.
stream.DoubleStream
interface.
double-black problem, 794–796
doubleValue method, java.lang.Double
class, A-18
doubly-linked lists, 684, (Worked
Example) 736
Dr. Java environment, 54
draw method, java.awt.Graphics2D
class, 62–63, A-13
drawing. See also graphical
applications; specific shapes.
a car, 110–113
circles, 64–65
colors, 66
on a component, 60–63
ellipses, 64–65
a face, 67–68
fills, 66
a flag (How To), 114–117
graphical shapes (How To),
114–117
lines, 65
rectangles, 59–63
shape classes, 110–114
drawLine method, java.awt.Graphics
class, A-13
drawString method, java.awt.
Graphics2D class, 65, A-13
dynamic method lookup, 440
ECMA (European Computer
Manufacturers Association), 686
editing pictures (Worked Example), 55
editors, definition, 8
Eight Queens problem, 621–626
EightQueens.java class, 625–626
electronic voting machines, 102
element separators
array algorithms, 319–320
stream algorithms, 869–870
ElevatorSimulation2.java class, 215–216
ElevatorSimulation.java class, 179
Ellipse2D.Double constructor, A-15
eLLipses, drawing, 64–65
e else statements, dangling else
problem, 201
empty method, java.util.Optional<T>
class, A-29
empty string versus null reference, 187–189
empty strings, 154
empty trees, 769
EmptyFrameViewer.java class, 60
encapsulation, instance variables, 82–84
encryption algorithms, 539
enhanced for loop. See also for loops.
array lists, 345–346
iterating through arrays, 317–318
ENIAC (electronic numerical
integrator and computer), 5
enumeration types, 203. See also
arrays.
EOFException, A-15
“equal exit cost” rule, 791–792
equal sign (==), assignment operator, 38–39
equal signs (==), relational operator
comparing object references, 187
comparing strings, 186, 189–190
equality testing, 184–185
versus equals method, 186
syntax, 184

 testing for null, 187–189
equality testing. See equal signs (==),
relational operator; equals
method.
equals method
comparing objects, 452–453
equals method (continued)
inheritance, 456
java.awt.Rectangle class, 187
java.lang.Object class, 452, A-20
java.lang.String class, 186–187, A-21
equalsIgnoreCase method, java.lang.
String class, A-21
erasing type parameters, 835–837
error handling, input errors, 549–554. See also exception handling.
error messages
logging, 208–209
reading exception reports, 160
stack trace, 160
trace messages, 208–209
errors. See also specific errors.
arrays, 314
compile-time, 14
dangling else problem, 201
declaring constructors as void, 90
declaring instance variables in local variables, 106
diagnosing with encapsulation, 83
“Division by zero,” 14
exceptions, 14
full code example, 14
ignoring parameter values, 96
logic, 14
in loops, 243–245
roundoff, 132, 185
run-time, 14
syntax, 14
unbalanced parentheses, 142
uninitialized object references, 107
unintended integer division, 142
unnecessary instance variables, 106–107
ESA (European Space Agency), 554
escape sequences, 156
EtchedBorder constructor, A-36
European Computer Manufacturers Association (ECMA), 686
evaluate method, javax.xml.xpath.
XPath interface, A-36
Evaluator .java class, 617–618
event handling, user-interface events, 491
event listeners
calling listener methods, 495
collectors as, 499
definition, 491
event adapters, 506–507
event source, 491
forgetting to attach, 498
full code example, 506
inner classes for, 493–495
keyboard events, 506
mouse events, 502–505
timer events, 499–502
event-controlled loops, 250
events
button press, 496–498
definition, 491
timer, 499–502
exception handlers, 542–543
exception handling. See also error handling.
application input errors, 549–554
catch clause, 542–543, 548
catching exceptions, 542–543
checked exceptions, 543–545
closing resources, 545–546
definition, 540
designing exception types, 546–548
finally statement, 549
internal errors, 543
NoSuchElementException error, 860, 861
NullPointerException error, 859
sample application, 549–554
squelching exceptions, 548
throw early, catch late, 548
throwing exceptions, 540–541, 548
throws clause, 544–545
try blocks, 542–543
try statement, 542–543
try/finally statement, 549
try-with-resources statement, 545–546
unchecked exceptions, 543
exception handling, full code examples
catching exceptions, 543
checked exceptions, 544
closing resources, 546
exception reports, 160
exceptions, definition, 14
exclamation point, equal (=), relational operator, 184
exclamation point (!), not operator, 211
executable algorithms, 16
execute method
java.sql.PreparedStatement interface, A-24
java.sql.Statement interface, A-25
executeQuery method
java.sql.PreparedStatement interface, A-24
java.sql.Statement interface, A-25
executeUpdate method
java.sql.PreparedStatement interface, A-24
java.sql.Statement interface, A-25
exists method
java.io.File class, A-15
java.nio.files.Files class, A-23
exit method, java.lang.System class, A-21
exp method, java.lang.Math class, 140, A-19–20
explicit parameters, 108
expression trees, 771
ExpressionCalculator .java class, 619–620
expressions, 137, 143
ExpressionTokenizer .java class, 618–619
extending classes. See inheritance.
extends reserved word, 429
extracting values from elements, 859

F

face, drawing, 67–68
FaceComponent .java class, 67–68
Fibonacci sequence, 604–609
FIFO (first in, first out), 699
Fifth-Generation Project, 217
file choosers, 523–525
File class, 520
File constructor, A-15
file dialog boxes
choosing files from a list, 523–524
full code example, 524
file names
as string literals, 523
to paths, 849
FileInputStream constructor, A-16
FileNotFoundException, 521–522, 543
FileOutputStream constructor, A-16
full code example, A-19
files. See also folders.
backing up, 10
definition, 9
buckets, 748
    collisions, 747
    efficiency, 751
    finding elements, 749
    hash codes, 747
    iterating over hash tables, 750–755
    linear probing, 755–756
    open addressing, 748, 755–756
    probing sequence, 755–756
    sample code, 751–755
    separate chaining, 748

hashCode method, java.util.Objects class, 687, A-29
HashMap constructor, java.util.HashMap<K, V> class, A-28
HashSet constructor, A-28
HashSetDemo.java class, 754–755
HashSet.java class, 751–754

hasNext method
    advancing an iterator, 726
    java.util.Iterator<E> interface, 688–689, A-28
    java.util.Scanner class, A-30
hasNextDouble method, java.util.Scanner class, 214–215, A-30
hasNextInt method, java.util.Scanner class, 214–215, A-30
hasNextLine method, java.util.Scanner class, A-30
hasPrevious method, java.util.ListIterator<E> interface, 684, A-28

HeapDemo.java class, 806–807
heaps
    adding nodes, 798–799
    definition, 797
    removing nodes, 800–801
    sample code, 802–807
    storing in arrays, 802
heapsort algorithm, 808–812
HeapSorter.java, 810–812
height of trees, 767
“Hello, World” program
    analyzing, 11–13
    source code, 11
    writing, 8–10
HelloPrinter.java class, 11
help, online, 53. See also documentation.
higher-order functions
    and comparators, 859
    lambda expressions, 858–859
high-level languages, 6
history of computers
    Altair 8800, 407
    Apple II, 407
    Babbage’s Difference Engine, 658
    corporate monopolies, 58
    first computer, 407
    first programmer, 658
    hardware evolution, 5
    IBM, 58
    microprocessors, 407
    Microsoft, 58
    personal computing, 407
    standardization, 686
    Turing machine, 612
    Univac Corporation, 58
    VisiCalc, 407
    Hoff, Marcian E., 407
    Houston, Frank, 355
Huffman trees, 771, (Worked Example) 774
IBM, history of computers, 58
IETF (Internet Engineering Task Force), 686
if statements. See also switch statements.
    dangling else problem, 201
    definition, 178
    duplicate code in branches, 183
    ending with a semicolon, 182
    flowchart for, 179
    implementing (How To), 190–192
    input validation, 214–216
    multiple alternatives, 193–196
    nesting, 196–199
    sample program, 179
    syntax, 180
ifPresent method
    java.util.Optional<T> class, A-29
    Optional type, 860–861
IllegalArgumentOutOfRangeException, java.lang.
    IllegalArgumentException, 540–541, A-18
ImageIcon constructor, A-33
immutable classes, 379–380
implementation, classes. See classes, implementing.
    implements reserved word, 469–471
    implicit parameters, 108
import directive, 395
importing
    classes from packages, 52
    packages, 401
in object, java.lang.System class, 145, A-21
income tax computation, 196–199
income tax rate schedule, 197t
increment, ++ (plus signs), 138
indefinite loops, 250
indenting code
    coding guidelines, A-44
    with tabs, 182
IndexOutOfBoundsException, 543–544
infinite loops, 244
infinite recursion, 598
infinite streams, 851–852
INFO message level, 209
info method, java.util.logging.
    Logger class, 209, A-31
information hiding. See encapsulation.
inheritance. See also polymorphism.
    class relationships, 571–572
    definition, 424
    equals method, 456
    full code example, 431, 449
    generic classes, 833
    generic methods, 833
    indicating, 429
    versus interfaces, 471–472
    purpose of, 428
    super reserved word, 437–438
inheritance hierarchy
    developing (How To), 445–449
    Worked Example, 450
initialization
    arrays, 308–310
    static methods, 395
initialization, variables
    assignment statements, 38
    definition, 35
    local, 105
    static, 395
Initials.java class, 158
inner classes
    definition, 488
    drawing an ellipse, 65
    as event listeners, 493–495
    instance variables, 494
    local variables, 494
    Node class, 722–723
    overview, 487–488
inorder traversal, 784–785
input. See also java.util.Scanner class; output.
from a console, 155
definition, 4
dialog boxes, 160–161
with dialog boxes (full code example), 160
from a keyboard, 145–146
prompting for, 145–146
reading, 145–146
redirecting, 262
strings, 155
input statements, syntax, 145
input validation, 214–216
InputStreamReader constructor, A-16
insertion sort
description, 645–646
full code example, 646
Worked Example, 667
instance methods, 143–144
instance variables
access specifiers, 81
declaring, 80–81
declaring in local variables, 106
definition, 80
encapsulation, 82–84
implementing classes, 91
return values, 82
syntax, 81
type, specifying, 81
unnecessary, 106–107
instanceof operator
description, 453–455
full code example, 454
instantiating, interface types, 472
int data type, 36, 130
int values, storing in streams. See IntStream interface.
integer division, 138–139, 142
Integer constructor, A–22
integer values, mixing with floating-point, 137
integers
computation overflow, 131
definition, 35
integrated development environment, 8
Intel Corporation, 144
interface types
constants, declaring, 473
declaring, 467–468
definition, 466
discovering, 466–467
generic, 486–487
implementing, 469–471
implements reserved word,
469–471
instantiating, 472
public methods, 472
interface variables
casting from interfaces to classes,
476–477
converting from classes, 475
full code example, 477
invoking methods on, 476
interfaces. See also specific interfaces.
for callbacks, 482–486
functional, 485–486
versus inheritance, 471
modifying method parameter
types, 495
static methods, 473
Worked Example: number sequences, 477
interior nodes, 767
international alphabets, 161
International Organization for Standardization (ISO), 686
Internet, history of, 456
Internet Engineering Task Force (IETF), standardization of computers, 686
Internet Explorer, 456
IntStream interface, 863–864
intValue method, java.lang.Integer class, A–22
InvestmentFrame2.java class, 889–890
InvestmentFrame3.java class, 892–893
Investment.java class, 241–242,
253–254
InvestmentRunner.java class, 242, 254
InvestmentViewer1.java class, 494–495
InvestmentViewer2.java class, 497–498
Invoice.java class, 582–584
InvoicePrinter.java class, 582
“is-a” relationship. See inheritance.
isDigit method, java.lang.Character class, A–17
isDirectory method, java.nio.files.Files class, A–23
isEditable method
javax.swing.JComboBox class, A–33
javax.swing.text.JTextComponent class, A–36
isLetter method, java.lang.Character class, A–17
isLowerCase method, java.lang.Character class, A–17
ISO (International Organization for Standardization), 686
isPresent method
java.util.Optional<T> class, A–29
Optional type, 861
isRegularFile method, java.nio.Files class, A–23
isSelected method, javax.swing.AbstractButton class, 894–895, 896, A–33
isUpperCase method, java.lang.Character class, A–17
iterate method, java.util.stream.Stream<T> interface, 851, A–32
Iterator class, 724–725
iterator method, java.util.Collection<E> interface, A–27
J
Jacobson, Ivar, 378
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, A–13–38. See also packages; specific elements.
inheritance hierarchy, A–9–12.
Java operators, summary of, A–5–6t
Java programs. See also specific elements.
class files, 9
compilation process, 9
compilers, 6
getting started. See “Hello, World” program.
machine code, 6
source code, 9
source files, 9
java.awt package, A–12–14
Index

java.lang.Double class
converting strings to numbers, 160
method summary, A-18. See also specific methods.
java.lang.Error class, summary, A-18
java.lang.IllegalArgumentException class
converting strings to numbers, 160
method summary, A-21. See also specific methods.
java.lang.Throwable class, method summary, A-22. See also specific methods.
java.math package, A-22
java.math.BigDecimal class
add method, 136
method summary, A-22
multiply method, 136
subtract method, 136
java.math.BigInteger class
add method, 136
method summary, A-22. See also specific methods.
multiply method, 136
subtract method, 136
java.net package, A-22–23
java.net.HttpURLConnection class,
method summary, A-23. See also specific methods.
java.net.Socket class, method summary, A-22. See also specific methods.
java.net.URL class
method summary, A-23. See also specific methods.
java.nio.file package, A-23–24
java.nio.file.Path interface, method summary, A-23–24. See also specific methods.
java.nio.file.Paths class, method summary, A-24. See also specific methods.
java.sql package, A-24–25
java.sql.Connection interface,
method summary, A-24. See also specific methods.
java.sql.DriverManager class, method summary, A-24. See also specific methods.
java.sql.ResultSet interface, method summary, A-24. See also specific methods.
java.sql.ResultSetMetaData interface, method summary, A-25. See also specific methods.
java.sql.SQLException class, A-25
java.sql.Statement interface, method summary, A-25. See also specific methods.
java.text package, A-25
java.text.SimpleDateFormat class, method summary, A-25. See also specific methods.
java.util package, A-26–30
java.util.ArrayList interface
add method, 344
get method, 344
method summary, A-26. See also specific methods.
java.util.Arrays class
copyOf method, 323–324
method summary, A-26. See also specific methods.
sort method, 327
toString method, 319–320
syntax, 343
java.util.Arrays class

java.util.Calendar class, method summary, A-26–27. See also specific methods.
java.util.Collection interface, 680, A-26–27
java.util.Collections class, method summary, A-27. See also specific methods.
java.util.Comparator class, method summary, A-30–31
java.util.Comparator class, method summary, A-30–31
java.util.Comparator&lt;E&gt; interface, 667, A-27
java.util.concurrent.ExecutorService interface, A-30–31
java.util.concurrent.locks.Condition interface, A-30–31
java.util.concurrent.locks.Lock interface, A-30–31
java.util.concurrent.locks.ReentrantLock class, A-31

java.util.Date class
get method, 225–226
method summary, A-27. See also specific methods.
java.util.DateFormat class
get method, 344
method summary, A-27. See also specific methods.
between method, 349–350
date value, 348
get year method, 354
method summary, A-27. See also specific methods.
java.util.File class
get method, 344
method summary, A-27. See also specific methods.
java.util.FileFilter interface
copyOf method, 323–324
method summary, A-27. See also specific methods.
java.util.Flushable interface
copyOf method, 323–324
method summary, A-27. See also specific methods.
java.util.Formatter class
format method, 131
get method, 344
method summary, A-27. See also specific methods.
java.util.HashMap interface
putAll method, 193
method summary, A-27. See also specific methods.
java.util.Hashtable class
putAll method, 193
method summary, A-27. See also specific methods.
java.util.List interface
add method, 180
addAll method, 180
method summary, A-27. See also specific methods.
java.util.List&lt;E&gt; interface
add method, 180
addAll method, 180
method summary, A-27. See also specific methods.
java.util.Map interface
add method, 180
addAll method, 180
method summary, A-27. See also specific methods.
java.util.Map&lt;K,V&gt; interface
add method, 180
addAll method, 180
method summary, A-27. See also specific methods.
java.util.NavigableMap&lt;K,V&gt; interface
add method, 180
addAll method, 180
method summary, A-27. See also specific methods.
java.util.NavigableSet&lt;E&gt; interface
add method, 180
addAll method, 180
method summary, A-27. See also specific methods.
java.util.NavigableSet interface
add method, 180
addAll method, 180
method summary, A-27. See also specific methods.
java.util.NavigableMap&lt;K,V&gt; interface
add method, 180
addAll method, 180
method summary, A-27. See also specific methods.
java.util.NavigableSet&lt;E&gt; interface
add method, 180
addAll method, 180
method summary, A-27. See also specific methods.
java.util.Optionals interface
stream method, 328
method summary, A-27. See also specific methods.
java.util.Optional interface
stream method, 328
method summary, A-27. See also specific methods.
java.util.OrderedMap&lt;K,V&gt; interface
add method, 180
addAll method, 180
method summary, A-27. See also specific methods.
java.util.Parameters interface
stream method, 328
method summary, A-27. See also specific methods.
java.util.Random interface
stream method, 328
method summary, A-27. See also specific methods.
java.util.Renumber interface
stream method, 328
method summary, A-27. See also specific methods.
java.util.SortedMap&lt;K,V&gt; interface
add method, 180
addAll method, 180
method summary, A-27. See also specific methods.
java.util.SortedSet&lt;E&gt; interface
add method, 180
addAll method, 180
method summary, A-27. See also specific methods.
java.util.SortedMap&lt;K,V&gt; interface
add method, 180
addAll method, 180
method summary, A-27. See also specific methods.
java.util.SortedSet interface
add method, 180
addAll method, 180
method summary, A-27. See also specific methods.
java.util.Stream interface
map method, 327
method summary, A-27. See also specific methods.
java.util.StringTokenizer class
copyOf method, 323–324
method summary, A-27. See also specific methods.
java.util.TimeZone class
compare method, 344
method summary, A-27. See also specific methods.
java.util.TreeSet&lt;E&gt; interface
add method, 180
addAll method, 180
method summary, A-27. See also specific methods.
java.util.TreeMap&lt;K,V&gt; interface
add method, 180
addAll method, 180
method summary, A-27. See also specific methods.
java.util.TreeSet interface
add method, 180
addAll method, 180
method summary, A-27. See also specific methods.
java.util.TreeMap interface
add method, 180
addAll method, 180
method summary, A-27. See also specific methods.
java.util.TreeMap&lt;K,V&gt; interface
add method, 180
addAll method, 180
method summary, A-27. See also specific methods.
java.util.TreeMap&lt;K,V&gt; interface
add method, 180
addAll method, 180
method summary, A-27. See also specific methods.
java.util.TreeMap&lt;K,V&gt; interface
add method, 180
addAll method, 180
method summary, A-27. See also specific methods.
java.util.TreeMap&lt;K,V&gt; interface
add method, 180
addAll method, 180
method summary, A-27. See also specific methods.
javax.swing.JLabel class
labeling buttons, 496–498
method summary, A-34. See also specific methods.
javax.swing.JMenu class, 905–910, A-34
javax.swing.JMenuBar class, method summary, A-34. See also specific methods.
javax.swing.JMenuItem class, 905–910, A-34
javax.swing.JOptionPane class
method summary, A-35. See also specific methods.
javax.swing.JPanel class
description, A-35
joining method, java.util.stream.Collectors class, A-31
javax.swing.JRadioButton class, 894, A-35
javax.swing.JScrollPane class, method summary, A-35. See also specific methods.
javax.swing.JSlider class, 912–916, A-35
javax.swing.JTextArea class, method summary, A-35. See also specific methods.
javax.swing.JTextField class, method summary, A-35. See also specific methods.
javax.swing.Keystroke class, method summary, A-35. See also specific methods.
javax.xml.parsers package, A-36
javax.xml.parsers.DocumentBuilder class, method summary, A-36. See also specific methods.
javax.xml.xpath package, A-36–37
javax.xml.xpath.XPath interface, method summary, A-36. See also specific methods.
javax.xml.xpath.XPathExpressionException exception, A-37
javax.xml.xpath.XPathFactory class, A-37
javax.swing.JButton class, 496–498, A-33
javax.swing.JCheckBox class, 896, A-33
javax.swing.JComboBox constructor, A-33
javax.swing.JFileChooser constructor, A-34
javax.swing.JLabel constructor, A-34
javax.swing.JMenu class, 905–910, A-34
javax.swing.JMenuBar class, 905–910, A-34
javax.swing.JMenuItem class, 905–910, A-34
javax.swing.JRadioButton class, 894, A-35
javax.swing.JScrollPane constructor, A-35
javax.swing.JSlider constructor, A-35
javax.swing.JTextArea class, 891–893, A-35
javax.swing.JTextField class, 888–890, A-35
JUnit testing framework, 405–406
Kahn, Bob, 456
keyboard input, 145–146
keyPressed method, 506, A-14
keyReleased method, 506, A-14
keySet method, java.util.Map<K, V> interface, 693, A-28–29
keyTyped method, 506, A-14
“knows-about” relationship. See dependencies, classes.
labels (user interface), 496
lambda expressions
comparators, 667
constructor references, 857–858
description, 485–486, 855–856
for event handling, 496
full code example, 856
functional interface, 856
higher-order functions, 858–859
history of, 856
keeping them short, 856–857
method references, 857–858
return statement, 856
sorted method, 855–856
sorting objects, 855–856
in streams, 846
syntax, 855
LargestInArray.java class, 325–326
LargestInArrayList.java class, 350–351
layout management
border layout, 884
complex layouts, 885–886
containers, 884
customizing frames with inheritance, 886–887
full code example, 886, 887
grid bag layout, 885
grid layout, 885
GUI builders, 904
How To, 901–903
layout managers, 884
nesting panels, 885–886
panels, 884
lazy processing, 847
leaf nodes, 767
Licklider, J.C.R., 456
length method, java.lang.String class, 41–42, 154, A-21
less than or equal (<=), relational operator, 184t
less than (<), relational operator, 184t
lexicographic ordering, 186
library. See Java library.
Licklider, J.C.R., 456
LIFO (last in, first out), 698
limit method, java.util.stream.Stream<T> interface, 853, A-32
limiting stream results, 852
line breaks, 156
Line2D.Double constructor, A-15
linear probing, 755–756
linear search, 320, 654–655, 870
LinearSearchDemo.java class, 655
LinearSearcher.java class, 654–655
LineItem.java class, 584
lines (graphic), drawing, 65
lines (of text), reading, 527–528
lines (text), reading, 527–528
lines method, java.nio.files.Files class, A-23
linked lists
definition, 681
doubly linked, 684
implementing queues as, 743–744
implementing stacks as, 741–743
list iterators, 683–685
nodes, 681–682
sample code, 685
structure of, 681–682

linked lists, implementing
adding/removing elements, 723–724, 726–729
advancing an iterator, 725–726
doubly-linked lists (Worked Example), 736
efficiency, 729–732
full code example, 735
inner classes, 722–723
Iterator class, 724–725
LinkedListIterator class, 724–725, 736
ListIterator class, 724–725
Node class, 722–723
sample code, 732–736
setting element values, 729
static classes, 736

LinkedHashMap constructor, java.util.
Iterator<E> interface, A-28
LinkedList class, 682–683
LinkedListIterator class, 724–725, 736
LinkedList.java class, 732–735
LinkedListStack.java class, 742–743
list iterators
methods for, 683–684
sample code, 685
List method, java.nio.files.Files class, A-23
ListDemo.java class, 685
listeners. See event listeners
ListIterator class, 724–725
ListIterator method
java.util.LinkedList<E> class, 683
class, A-32
LinkedList.java class, 735–736
lists, creating, 678–679
literals, string, 154
load method, java.util.Properties class, A-29
local variables
declaring instance variables in, 106
definition, 105
duplicate names, 202
full code example, 105
garbage collection, 105
initializing, 105
lock method, java.util.concurrent.
Locks.Lock interface, A-30
log method, java.lang.Math class,
a-19–20
log10 method, java.lang.Math class,
a-19–20
logging messages, 208–209
logic errors, 14
Loma Prieta earthquake, 193
long data type, 130, 131
long values, storing in streams. See LongStream interface.
LongStream interface, 863
loop and a half problem, 262–263
LoopFib.java class, 607–608
loops
asymmetric bounds, 256
Boolean variables, 261
bounds, choosing, 256
break statements, 263–264
common errors, 243–245
continue statements, 263–264
count-controlled, 250
counting iterations, 256
credit card processing (Worked Example), 275
definite, 250
definition, 238
do loop, 258
enhanced for loop, 317–318. See also for loops.
event-controlled, 250
flowcharting, 259
full code example, 258, 317–318
hand tracing, 245–249
improving recursion efficiency, 608–609
indefinite, 250
infinite, 244
loop and a half problem, 262–263
for loops, 250–255, 257. See also enhanced for loops.
manipulating image pixels (Worked Example), 278
nesting, 275–278
off-by-one errors, 244–245
post-test, 258
pre-test, 258
redirecting input/output, 262
sentinel values, 259–262
while, 238–243, 255
writing (How To), 272–275
loops, common algorithms
averages, computing, 268
comparing adjacent values, 271–272
counters, 268–269
finding first match, 269
full code example, 271
maximum/minimum values, 270
prompting for first match, 270
totals, computing, 268
luggage handling system, 192

M
machine code, 6
magic numbers, 137
main method
command line arguments, 533–535
definition, 11
managing object properties,
common class patterns, 388
map method, java.util.stream.
Stream<T> interface
description, A-32
as high-order function, 858
mapping a primitive-type stream, 864
transforming streams, 852
MapDemo.java class, 693–694
mapping primitive streams, 864
maps
definition, 679–680
description, 692–694
updating, 695
mapToDouble method, java.util.
stream.Stream<T> interface, A-32
mapToInt method, java.util.stream.
Stream<T> interface, A-32
mapToLong method, java.util.stream.
Stream<T> interface, A-32
mapToObj method
java.util.stream.DoubleStream interface, A-32
java.util.stream.IntStream interface, A-32
java.util.stream.LongStream interface, A-32
mathematical operations. See arithmetic operations.
method summary, java.lang.Math class, 140t
matrices. See arrays, two-dimensional.
max method, java.lang.Math class
description, A-19
Optional type, 861
primitive streams, 865
syntax, 140
as terminal operator, 862
maxBy method, java.util.stream.
Collectors class, A-31
maximum values
finding in arrays, 319
finding with loops, 270
getting, 130
stream algorithms for, 869
streams, 866–868
MAX_VALUE constant, java.lang.Integer
class, 130
MeasurableTester.java
class, 470–471
Measurer.java
class, 483–484
MeasurerTester.java
class, 484–485
Medals.java
class, 340–341
menu bars, 905
menu items, 905
menus
code sample, 907–910
definition, 905
implementing a class for, 99
javax.swing.JMenuItem
class, 905–910
javax.swing.JMenu
class, 905–910
overview, 905–907
submenus, 905
merge method, 694
MergeSortDemo.java
class, 649
MergeSorter java class, 648–649
messages. See error messages.
method references, lambda
expressions, 857–858
methods. See also specific methods.
abstract, 443–444
accessing data without modifying.
See accessor methods.
accessor, 48–49
arguments, 42–43, 315–316
call-by-reference, 382–386
call-by-value, 382–386
calling, 12, 32–33
versus constructors, 85
declaring, 45
default, 473–475
definition, 11, 33
documenting, 579–581
duplicate names. See overloading.
final, 444
functional interfaces, 485–486
versus functions, 858
implementing classes, 93–95
instance, 143–144
invoking on interface variables, 476
invoking on numbers, 143–144
measuring run time, 642
modifying data. See mutator
methods.
mutator, 48–49
naming conventions, 37
overloading, 45
passing information to. See
parameters; this reference.
private implementation, 42
functional interfaces, 485–486
versus functions, 858
implementing classes, 93–95
instance, 143–144
invoking on interface variables, 476
invoking on numbers, 143–144
measuring run time, 642
modifying data. See mutator
methods.
mutator, 48–49
naming conventions, 37
overloading, 45
passing information to. See
parameters; this reference.
private implementation, 42
public interfaces, classes, 84–85
public interfaces to classes, 41–42
return values, 43–44
static, 143–144
in UML diagrams, 573
verbs as, 566
with no results. See null, testing
for; Optional type.
methods, designing
accessors, 379–380
cohesion, 377–378, 381–382
consistency, 381–382
dependencies, 378–379, 381–382
mutators, 379–380
public interfaces, 377–378
side effects, 380–381
methods, full code examples
accessor methods, 49
method calls, 45
mutator methods, 49
methods, static
accessing instance variables, 394
definition, 391–393
full code example, 393
initialization, 395
min method, java.math.BigDecimal
class, 136, A-22
methods, designing
accessors, 379–380
cohesion, 377–378, 381–382
consistency, 381–382
dependencies, 378–379, 381–382
mutators, 379–380
public interfaces, 377–378
side effects, 380–381
methods, full code examples
accessor methods, 49
method calls, 45
mutator methods, 49
methods, static
accessing instance variables, 394
definition, 391–393
full code example, 393
initialization, 395
min method, java.lang.Math class
description, 140t, A-19
multiplier method
java.math.BigDecimal class, 136,
A-22
java.math.BigInteger class, 136,
A-22
Murphy, Walter, 586
mutations in parallel streams, 863
mu}
description, 48–49, 379–380
full code example, 48–49
Mycin program, 217

newline character, 156
name clashes, packages, 401–402
naming conventions
classes, 37
constants, 133
constructors, 86
local variables, 202
methods, 37
packages, 401–402
test classes, 406
type variables in generic classes, 826
variables, 37, 41
Naughton, Patrick, 6
negative-red problem, 794–796
nesting
if statements, 196–199
loops, 275–278
panels, 885–886
networks, definition, 4
new
operator, 47
newCondition method, java.util.
collectors.List
interface, A-31
newDocument method, javax.xml.
parsers.DocumentBuilder class, A-36
newInstance method, javax.xml.parsers.
DocumentBuilderFactory class, A-36
next
method
advancing an iterator, 725–726
java.sql.ResultSet interface, A-25
java.util.Iterator<T> interface, 683, 688–689, A-28
next method, java.util.Scanner class
consuming white space, 525–526, 529–530
description, A-30
data types, 130–132
with fractions. See floating-point numbers.
without fractions. See int type.
invoking methods on, 143–144
magic, 137
primitive types, 130

Object class. See also superclasses.
definition, 450
equals method, 452–453
instanceof operator, 453–454
toString method, 450–451, 455
object construction
arguments, 47
definition, 47
invoking a constructor like a method, 48
overview, 46–48
syntax, 47
object references
definition, 56
full code example, 57
number variables versus object variables, 56
overview, 55–58
uninitialized, 107
object variables versus number variables, 56
ObjectInputStream constructor, A-16
ObjectOutputStream constructor, A-16
objects
arrays of, 314–315
comparing, 665–667. See also
equals method.
copying, 477–481
definition, 32
hand tracing, 103–105
mock, 489–490
testing. See mock objects.
OFF message level, 209
off-by-one errors, 244–245
online help, 53. See also
documentation.
open addressing, 748, 755–756
openStream method, java.net.URL class, A-238
operators. See specific operators.
Optional type. See also null, testing for.
full code example, 860
get method, 861
Optional type (continued)

ifPresent method, 861
isPresent method, 861
obtaining the value of, 860
orElse method, 860–861

orElse method
java.util.Optional<T> class, A-29

org.w3c.dom
package, A-37
org.w3c.dom.Document interface,
method summary, A-37. See also specific methods.
org.w3c.dom.DOMConfiguration interface,
method summary, A-37. See also specific methods.
org.w3c.dom.DOMImplementation interface,
method summary, A-37. See also specific methods.
org.w3c.dom.Element interface,
method summary, A-37. See also specific methods.
org.w3c.dom.ls
package, A-37
org.w3c.dom.ls.DOMImplementationLS interface,
method summary, A-37. See also specific methods.
org.w3c.dom.ls.LSSerializer interface,
method summary, A-37. See also specific methods.
org.w3c.dom.Text interface, A-37
output object, System class, 12–13, 33

output. See also input.
definition, 4
dialog boxes, 160–161
with dialog boxes (full code example), 160
format specifiers, 146–148
formatting, 146–148
line breaks, 156
redirecting, 262

overflow, computation
definition, 131
doouble data type, 131
overloading methods, 45, 603
accidentally, 437
overriding methods
forcing, 443–444
preventing, 444
superclass, 429, 433, 443–444

P
packages. See also specific packages.
(dot), name syntax, 403
accessing, 403–404
API documentation, 52–53
default, 401
definition, 400
description, 7
importing, 401
importing classes from, 52
Java library, 6–7
name clashes, 401–402
naming conventions, 401–402
organizing classes in, 400–401
programming with (How To), 404–405
source files, 402–403
summary of, 400
syntax, 401

paintComponent method, javax.swing.
JComponent class, 61–63, A-34
PairDemo.java class, 828
Pair.java class, 827–828
palindromes, identifying (How To), 599–602
panels (user-interface)
creating, 496–498
definition, 884
nesting, 885–886
parallel arrays, 314–315
parallel method, java.util.stream.
Stream<T> interface, A-33
parallel streams, 851, 863
@param tag, 88
parameter passing, 44
parameter variables
ignoring, 96
javadoc program, 88
parameters
explicit, 108
implicit, 108
this reference, 107–109
parent nodes, 767
parentheses ( ), balancing, 142, 701
in arithmetic operations, 137, 142
in object construction, 47
parse method, javax.xml.parsers.
DocumentBuilder class, A-36
parseDouble method, java.lang.Double
class, A-18
converting strings to numbers, 160, 528–529

parseInt method, java.lang.Integer
class, A-18
converting strings to numbers, 160, 528–529
partially filled arrays, 312–313
PartialSolution.java class, 623–624
partitioning the range, 652–653
patent, definition, 539
paths, 767
paths from a filename, 849
PCs (personal computers),
schematic design of, 4. See also computers.
peek method
java.util.Queue<E> interface, 699, A-29
java.util.Stack<E> class, 698
Pentium floating-point bug, 144
percent sign (%), modulus, 138–139
performance estimation, array
algorithms
full code example, 663
linear time, 659–660
logarithmic time, 662–664
quadratic time, 660–661
triangle pattern, 661–662
performance estimation, big-Oh
notation, 643–645
peripheral devices, 4. See also specific
devices.
permutations of strings, 609–612
Permutations.java class, 610–611
personal computing, 407
PGP (Pretty Good Privacy) encryption, 539
PI constant, java.lang.Math class, 133, A-20
pictures, editing (Worked Example), 55. See also drawing; graphical
applications.
piracy, digital, 249
pivot, 652
pixel images, manipulating with
loops (Worked Example), 278
plus sign (+)
addition operator, 36
concatenation operator, 155
plus signs (++), increment, 138
Point2D.Double constructor, A-15
polymorphism. See also inheritance.
dynamic method lookup, 440, 442
overview, 439–440
range method, java.util.stream.
IntStream interface, A-32
read method, java.io.InputStream
class, A-16
readability, coding for, 851
readAllBytes method, java.nio.files.
Files class, A-23
readAllLines method, java.nio.files.
Files class, A-23
readChar method, RandomAccessFile
class, A-17
readDouble method, RandomAccessFile
class, A-17
reading input. See also text files,
reading and writing; writing output.
into arrays, 324–326
characters, 526
characters from a string, 528
classifying characters, 526–527
converting strings to numbers,
528–529
entire files, 533
error handling, 549–554
lines, 527–528
mixed input types, 529–530
from a prompt, 145–146
text files, 520–522
validating numbers, 529
white space, consuming, 525–526
words, 525–526
readInt method, RandomAccessFile
class, A-17
readObject method, java.io.Object-
InputStream class, A-16
Rectangle constructor, A-13
RectangleComponent2.java class, 504–504
RectangleComponent.java class, 62–63,
500–501
RectangleFrame2.java class, 504–505
RectangleFrame.java class, 501
rectangles
comparing, 187
drawing, 59–63
moving (full code example), 506
RectangleViewer2.java class, 505
RectangleViewer.java class, 62–63,
501–502
recursion
backtracking, 620–626
changing to a loop, 608–609
debugging, 598–599
definition, 594
finding files (Worked Example),
602
full code example, 602
identifying a palindrome (How
To), 599–602
infinite, 598
permutations, 609–612
setting breakpoints, 598–599
tracing, 598–599, 605–607
triangle numbers, 594–598
recursion, efficiency
changing to a loop, 608–609
Fibonacci sequence, 604–609
full code example, 609
trace messages, 605–607
recursion, mutual
description, 614–616
sample code, 602–604
syntax diagrams, 614
tokens, 615
recursive helper methods
description, 602–604
full code example, 603
overloading, 603
recursive methods, call pattern,
595–596
RecursiveFib.java class, 604–605
RecursiveFibTracer.java class,
605–607
red-black trees
basic properties, 790–792
definition, 790
double-black problem, 794–796
efficiency, 796
“equal exit cost” rule, 791–792
implementing (How To), 796
inserting nodes, 792–793
negative-red problem, 793–796
“no double red” rule, 791–792
removing nodes, 793–796
redirecting input/output, 262
ReentrantLock constructor, A-31
reflection, 838–839
regression testing, 352–354
regular expressions, 532
relational operators, 184–185, 212
remainers, calculating, 138–139
remove method
java.util.ArrayList<E> class, 345,
A-26
java.util.Collection<E> interface,
A-27
java.util.Iterator<E> interface,
684, 688, 726–727, A-28
java.util.Map<K, V> interface,
A-29
java.util.PriorityQueue<E> class,
A-29
java.util.Queue<E> interface, 726–727,
A-28
removeLast method, java.util.
LinkedList<E> class, A-28
remove method, java.awt.Component
class, 502, A-12.
repainting graphic components, 502
replace method, java.lang.String
class, 44, A-21
replaceAll method, java.lang.String
class, 532, A-21
requirements, gathering, 575–576
reserved words
summary of, A-7–8r
as illegal variable names, 37
resizing arrays, 311, 312–313
resolve method, java.nio.file.Path
interface, A-23
return statement, 82
lambda expressions, 856
@return tag, 88
return values, 43, 82
reusing algorithms, 327
reverse Polish notation, 702–703, 709
Richter scale, 193r
Rivest, Ron, 539
rocket incident, Ariane, 554
rollback method, java.sql.Connection
interface, A-24
rolling dice
simulating, 279–280
Worked Example, 332
root nodes, 767
roots, calculating, 139–140
round method, java.math.Math class
definition, 140r
description, A-20
rounding floating-point numbers,
140
rounding
description, 131–132
full code example, 140
roundoff errors, 132, 185
RSA encryption, 539
Rumbaugh, James, 378
run method, java.lang.Runnable
interface, A-20
run-time errors, 14
run-time stacks, 698

S
safety, Java, 7
sales tax calculation, full code
example, 104
Scanner class
character encoding, 525
constructing with a string, 523
reading text files, 520–522
Scanner constructor, A-30
scheduling time for programming,
208
ScoreTester.java class, 353
scroll bars, 892
searching
binary, 655–658
breadth-first, 787–789
depth-first, 787–789
linear, 654–655
secondary storage, 3–4
seek method, RandomAccessFile class,
A-17
SelectionSortDemo.java class, 638
SelectionSorter.java class, 637–638
SelectionSortTimer.java class, 641
semicolon (;)
ending if statements, 182
ending method statements, 11
omitting, 13
sentinel values, 259–262
SentinelDemo.java class, 260–261
separate chaining, 748
sequential search, 320
ServerSocket constructor, A-22
set method
java.util.ArrayList<E> class,
344–345, A-26
java.util.ListIterator<E>
interface, 684, 729, A-28
setAttribute method, org.w3c.dom.
Element interface, A-37
setAutoCommit method, java.sql.
Connection interface, A-24
setBorder method, java.awt.
Component class, 895, A-34
setColor method, java.awt.Graphics
class, 66, A-13
setDefaultCloseOperation method,
javax.swing.JFrame class, 59,
A-34
setDouble method, java.sql.
PreparedStatement interface,
A-24
setEditable method
javax.swing JComboBox class, A-33
javax.swing.text JTextComponent
class, 891, A-36
setFocusable method, java.awt.
Component class, 506, A-12
setFont method, java.awt.
Component class, A-34
setIfModifiedSince method, java.net.
URLConnection class, A-23
setIgnoredElementContentWhitespace
method, java.xml.parsers.
DocumentBuilderFactory class,
A-36
setInt method, java.sql.
PreparedStatement interface,
A-24
setJMenuBar method, java.awt.
JFrame class, 905–910, A-34
setLayout method, java.awt.Container
class, 884–885, A-13
setLevel method, java.util.logging.
Logger class, 209
setLine method, java.awt.geom.Line2D
class, A-15
setLocation method
java.awt.geom.Point2D class, A-15
java.awt.Rectangle class, A-14
setParameter method, org.w3c.dom.
DOMConfiguration interface,
A-37
setPreferredSize method, java.awt.
Component class, 887, A-12
sets
adding/removing elements, 688
binary search trees, 687
choosing an implementation,
687–688
definition, 679
description, 687–689
hash tables, 687
iterating over, 688–689
listing elements, 688–689
setSelected method, java.awt.
AbstractButton class, A-33
setSelectedItem method, java.awt.
JComboBox class, A-34
setSize method
java.awt.Component class, 59, A-12
java.awt.Rectangle class, 51–52,
A-14
setString method, java.sql.
PreparedStatement interface,
A-24
setText method, java.awt.
text JTextComponent class, 891, A-36
setTitle method, java.awt.
Frame class, 59, A-13
setValidating method, java.awt.
parsers.DocumentBuilderFactory
class, A-36
setVisible method, java.awt.
Component class, 59, A-12
Shamir, Adi, 539
shape classes, 110–114
shell scripts, 354
shipping costs, computing (full code
example), 204–205
short circuiting Boolean evaluation,
213
short data type, 130
showInputDialog method, java.awt.
JOptionPane class, 160, A-35
showMessageDialog method, java.awt.
JOptionPane class, 161, A-35
showOpenDialog method, java.awt.
JFileChooser class, 524, A-34
showSaveDialog method, java.awt.
JFileChooser class, 524, A-34
sibling nodes, 767
side effects, 380–381
signal method, java.util.concurrent.
locks.Condition interface, A-30
signalAll method, java.util.
concurrent.locks.Condition
interface, A-30
simulation programs, 279–282
sin method, java.lang.Math class,
140, A-20
single-step command, 284
size method
java.util.ArrayList<E> class, 344,
A-26
java.util.Collection<E> interface,
A-27
skip method, java.util.stream.
Stream<T> interface, A-33
slash, asterisks (**), comment indicator, 88
slash (/), division operator, 36, 137
slash asterisk... (**...*/), comment indicator, 88
slash (/), division operator, 36, 137
slashes (//), comment delimiter, 38
slashes (/*...*/), comment delimiters, 38
sleep method, java.lang.Thread class, A-21
Socket constructor, A-22
Software, definition, 2. See also programming.
Software piracy, 249
sort method
java.util.Arrays class, 327, 664–665, A-26
java.util.collections class, 664–665, A-27
sorted method
lambda expressions, 855–856
transforming streams, 853
sorting
arrays, 327
new streams, 853
objects, with lambda expressions, 855–856
sorting algorithms
definition, 636
full code example, 646, 653
insertion sort, 645–646
insertion sort (Worked Example), 667
partitioning the range, 652
pivot, 652
quicksort, 652–653
sorting algorithms, merge sort
description, 647
full code example, 652
measuring performance of, 650–652
sample programs, 648–649
sorting algorithms, selection sort
description, 636–637
measuring performance of, 642–646
sample programs, 637–639, 640–641
source code
“How Hello, World” program, 11
Java programs, 9
source files, 9
spaces in comparisons, 186
expressions, 143
input. See white space.
variable names, 37
spaghetti code, 204–205
spelling check, sample code, 690–691
spelling errors, 15
split method, java.lang.String class, 532, A-21
sqrt method, java.lang.Math class, 140, A-20
stack trace, 160
stacks
algebraic expressions, 703–706
backtracking, 706–708
balancing parentheses, 701
definition, 679
description, 698
full code example, 700, 708
LIFO (last in, first out), 698
reverse Polish notation, 702–703
run-time, 698
stacks, implementing
as arrays, 743
as linked lists, 741–743
sample code, 742–743
standardization of computers, 686
start method
java.lang.Thread class, A-22
javax.swing.Timer class, 500–501, A-40
stateChanged method, javax.swing.event ChangeListener interface, 914–916, A-36
states
definition, 11
punctuating, 11, 13
static classes, 736
static imports, 394–395
static methods
accessing instance variables, 394
definition, 392–393
full code example, 393
initialization, 395
in interfaces, 473
minimizing use of, 393–394
overview, 143–144
static imports, 395
stepping through code, 283–285
stop method, javax.swing.Timer class, A-35
StopWatch program, 639–642
StopWatch.java class, 640–641
storage devices. See also specific devices.
programmatically (full code example), 153

transforming streams. See streams, transforming.

transistors, 3

Travel time computation (Worked Example), 154

Trees of Hanoi (Worked Example), 626

trace messages, 208–209

treating. See hand-treating.

tracing recursive methods, 598–599, 605–607

transforming streams. See streams, transforming.

transistors, 3

translate method, java.awt.Rectangle

class, 52–54, A-15

throw statements, 540–541

Throwables constructor, A-22

throwing exceptions, 540–541, 548, 549

gthrows clause, 544–545

tiles for flooring, computing by hand, 152–154

termination algorithms, 16

test cases, selecting, 206–207

test method, java.util.function. 

Predicate <T> interface, A-31

text areas, 891–893

text fields, 888–890

text files, reading and writing. See also reading input; writing output. 

How To, 535–538

overview, 520–522

text I/O

code sample, 892–893

multiple lines, 891–893

scroll bars, 892

single line, 888–890

text areas, 891–893

text fields, 888–890

thenComparing method, java.util. 

Comparator<T> interface, 859, A-27

Therac-25 incidents, 355

this reference 
derivation, 107–109

full code example, 109

throw early, catch late, 548

throw statement, 540–541

Throwables constructor, A-22

throwing exceptions, 540–541, 548, 549

gthrows clause, 544–545

tiles for flooring, computing by hand, 152–154

terminated streams, producing new streams, 855

terminal operations (continued) 

min method, 862

noneMatch method, 862

terminating algorithms, 16

test cases, selecting, 206–207

test method, java.util.function. 

Predicate <T> interface, A-31

test suites, 352

tester classes, 100–101

testing 

black-box, 206

boundary cases, 206–207

classes, 54–55

code coverage, 206

for null, 187–189

mock objects, 489–490

programs, 53–54

regression, 352–354

unit, 100–101

white-box, 206

text areas, 891–893

text fields, 888–890

text files, reading and writing. See also reading input; writing output. 

How To, 535–538

overview, 520–522

text I/O

code sample, 892–893

multiple lines, 891–893

scroll bars, 892

single line, 888–890

text areas, 891–893

text fields, 888–890

thenComparing method, java.util. 

Comparator<T> interface, 859, A-27

Therac-25 incidents, 355

this reference 
derivation, 107–109

full code example, 109

throw early, catch late, 548

throw statement, 540–541

Throwables constructor, A-22

throwing exceptions, 540–541, 548, 549

gthrows clause, 544–545

tiles for flooring, computing by hand, 152–154

programmatically (full code example), 153

tiling a floor, algorithm for (Worked Example), 21–22

timer events, 499–502

Timer constructor, A-35

TitleBorder method, javax.swing. 

border.EtchedBorder class, A-36

toArray method, java.util.stream. 

Stream<T> interface, 850, A-33

toDegrees method, java.lang.Math 
class, 140r, A-20

tokens, 615

tolist method, java.util.stream. 

Collectors class, A-31

toLowerCase method, java.lang.String 
class, A-19

toRadians method, java.lang.Math 
class, 140r, A-20

toSet method, java.util.stream. 

Collectors class, 866, A-31

toString method 

full code example, 454

inheritance, 455

inserting element separators, 

319–320

java.lang.Integer class, A-19

java.lang.Object class, A-20

java.util.Arrays class, A-26

overriding, 450–451

Total.java class, 521–522

totals, computing with loops, 268

toUpperCase method, java.lang.String 
class, 41–42, A-21

Towers of Hanoi (Worked Example), 626

trace messages, 208–209

treating. See hand-treating.

tracing recursive methods, 598–599, 605–607

transforming streams. See streams, transforming.

transistors, 3

translate method, java.awt.Rectangle 
class, 52–54, A-15

throw statements, 540–541

try block, 542–543

try statement, 542–543

try/finally statement, 549

Turing, Alan, 612–613

Turing machine, 612–613

two-dimensional arrays. See arrays, 
two-dimensional.

type parameters 
erasing, 835–837

full code example, 837

in static context, 838

type parameters, constraining 

full code example, 835

generic classes, 831–832

generic methods, 831–832

wildcard types, 834–835

type variables 
definition, 824

naming, 826

types of data. See data types.
U

UML (Unified Modeling Language)
attributes, 573
definition, 378
methods, 573
in program design (How To), 572–573
relationship symbols, 572
tsample program, 578–579
The Unified Modeling Language
User Guide, 575
unambiguous algorithms, 16
unchecked exceptions, 543
undeclared variables, 40
underscore (_), in variable names, 37
unfilled arrays, 314
Unicode, 156–157, 161
Unicode encoding, 525, A-1–3
uninitialized arrays, 314
uninitialized variables, 38, 40
unintended integer division, 142
union method, java.awt.Rectangle
class, A-14
unit testing
annotating test methods, 405–406
definition, 100
JUnit framework, 405–406
naming test classes, 406
tester classes, 100–101
Univac Corporation, history of
computers, 58
unlock method, java.util.concurrent.
locks.Lock interface, A-31
URL constructor, A-23
URLConnection constructor, A-23
useDelimiter method, java.util.
Scanner class
description, A-30
reading characters, 526
user-interface components. See also
specific components.
business, 496–498
labels, 496
panels, 496–498
repainting, 502
UTF-8 encoding, 525
utility classes, 376

V
validating numeric input, 529
variables. See also instance variables;
local variables.
assigning values to, 38–39
comments, 38
declaration versus assignment, 40–41
declaring, 34–36
definition, 35
full code example, 39
initializing, 35, 38
inspecting during debugging, 283–285
in for loop headers, 257
naming conventions, 37, 41
object versus number, 56
scope, minimizing, 201–202
syntax, 35
undeclared, 40
uninitialized, 38, 40
variables, static
definition, 391–393
full code example, 393
initialization, 395
static imports, 395
verbs as methods, 566
vertical bars (|), or operator, 209–210, 212, 213
virtual machine, definition, 7
VisiCalc, 407
visitor pattern, 786–787
void reserved word
declaring constructors, 90
definition, 45
volume of a pyramid, computing
(Worked Example), 152
Volume.java class, 148
voting machines, 102

W
W3C (World Wide Web
Consortium), 686
wait method, java.lang.Object class,
A-20
walk method, java.nio.files.Files
class, A-23
web pages, reading (full code
example), 523
while loops, 238–243, 255
while statement, syntax, 239
white space
consuming, 525–526
trimming, 528
white-box testing, 206
wildcard types, type parameters,
834–835
wildcards, full code example, 835
Wilkes, Maurice, 287
word frequency, counting (Worked
Example), 695
words
counting, 846
picking out, 846
reading, 525–526
WorkOrder.java class, 806
world population table (Worked
Example), 341
World Wide Web, 456
wrapper classes, 347, 347
write method, java.io.OutputStream
class, A-16
writeChar method, RandomAccessFile
class, A-17
writeChars method, RandomAccessFile
class, A-17
writeDouble method, RandomAccessFile
class, A-17
writeInt method, RandomAccessFile
class, A-17
writeObject method, java.io.
ObjectOutputstream class,
A-16
writeToString method, org.w3c.dom.
ls.LSSerializer interface, A-37
writing output, 530–532. See also
reading input; text files,
reading and writing.
writing text files, 520–522

Z
Zimmerman, Phil, 539
# Illustration Credits

**Icons**

- Common Error icon (spider): © John Bell/iStockphoto.
- How To icon (compass): © Steve Simzer/iStockphoto.
- Java 8 icon (eightball): © subjug/iStockphoto.
- Papercrips: © 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 8

**Page 396–398, 416 (left to right, top to bottom):**

- Degas, Edgar *The Dance Lesson*: Collection of Mr. and Mrs. Paul Mellon 1999.47.6/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.

**Manet, Edouard Masked Ball at the Opera: Gift of Mrs. Horace Havemeyer in memory of her mother-in-law, Louisa 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 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.**

**Page 416, continued:**

- Pissarro, Camille *Peasant Girl with a Straw Hat*: Ailsa Mellon Bruce Collection/National Gallery of Art.
- Renoir, Auguste *Oarsmen at Chatou*: Gift of Sam A. Lewisohn 1951.5.2/National Gallery of Art.
System.out.println(i);
{  
    for (int i = 0; i < 10; i++)
    {
        balance = balance * (1 + rate / 100);
        year++;
    }
}

Selected Operators and Their Precedence

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>=</td>
<td>Assignment</td>
</tr>
<tr>
<td>==</td>
<td>Equal</td>
</tr>
<tr>
<td>!=</td>
<td>Not equal</td>
</tr>
<tr>
<td>&lt;</td>
<td>Less than</td>
</tr>
<tr>
<td>&lt;=</td>
<td>Less than or equal</td>
</tr>
<tr>
<td>&gt;</td>
<td>Greater than</td>
</tr>
<tr>
<td>&gt;=</td>
<td>Greater than or equal</td>
</tr>
<tr>
<td>+</td>
<td>Addition</td>
</tr>
<tr>
<td>-</td>
<td>Subtraction</td>
</tr>
<tr>
<td>*</td>
<td>Multiplication</td>
</tr>
<tr>
<td>/</td>
<td>Division</td>
</tr>
<tr>
<td>%</td>
<td>Remainder</td>
</tr>
<tr>
<td>++</td>
<td>Increment</td>
</tr>
<tr>
<td>--</td>
<td>Decrement</td>
</tr>
<tr>
<td>!</td>
<td>Boolean not</td>
</tr>
</tbody>
</table>

Mathematical Operations

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Math.pow(x, y)</td>
<td>Raising to a power x^y</td>
</tr>
<tr>
<td>Math.sqrt(x)</td>
<td>Square root sqrt(x)</td>
</tr>
<tr>
<td>Math.log(x)</td>
<td>Natural logarithm log(x)</td>
</tr>
<tr>
<td>Math.log10(x)</td>
<td>Base 10 logarithm log10(x)</td>
</tr>
<tr>
<td>Math.abs(x)</td>
<td>Absolute value</td>
</tr>
</tbody>
</table>
WILEY END USER LICENSE AGREEMENT

Go to www.wiley.com/go/eula to access Wiley’s ebook EULA.