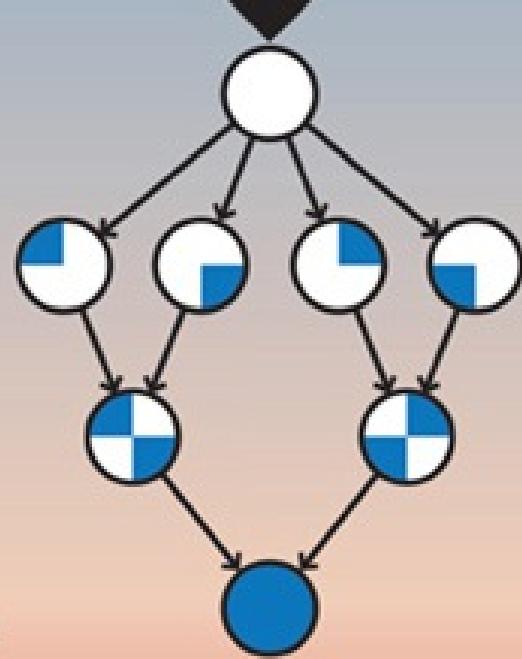


START

An Introduction to
Problem Solving in Java
with a Focus on Concurrency

CONCURRENT

Barry Wittman
Aditya Mathur
Tim Korb



2014 EDITION

© 2014 by Barry Wittman, Aditya Mathur, Tim Korb.

Distributed by Purdue University Press, <http://www.thepress.purdue.edu>.

Wittman *To the set of all people that I do not dedicate this book to*
Mathur *To my mother*
Korb *To my family*

Contents

Contents

Preface

Acknowledgements

1 Computer Basics

1.1 Problem: Buying a computer

1.2 Concepts: Hardware and software

 1.2.1 Hardware

 1.2.2 Software

 1.2.3 Examples

1.3 Syntax: Data representation

 1.3.1 Compilers and interpreters

 1.3.2 Numbers

 1.3.3 Conversion across number systems

 1.3.4 Integer representation in a computer

 1.3.5 Binary arithmetic

 1.3.6 Negative integers in a computer

 1.3.7 Overflow and underflow

 1.3.8 Bitwise operators

 1.3.9 Rational numbers

1.4 Solution: Buying a computer

1.5 Concurrency: Multicore processors

 1.5.1 The Good

 1.5.2 The Bad

 1.5.3 The Ugly

1.6 Summary

Exercises

 Conceptual Problems

2 Problem Solving and Programming

2.1 Problem: How to solve problems

 2.1.1 What is a program?

 2.1.2 What is a programming language?

 2.1.3 An example problem

2.2 Concepts: Developing software

 2.2.1 Software development lifecycle

 2.2.2 Acquiring a Java compiler

2.3 Syntax: Java basics

 2.3.1 Java program structure

 2.3.2 Command line input and output

 2.3.3 GUI input and output

2.3.4 A few operations

2.3.5 Java formatting

2.4 Solution: How to solve problems

2.4.1 Bouncing ball solution (command line version)

2.4.2 Bouncing ball solution (GUI version)

2.4.3 Testing and maintenance

2.5 Concurrency: Solving problems in parallel

2.5.1 Parallelism and concurrency

2.5.2 Sequential versus concurrent programming

2.5.3 Kinds of concurrency

2.6 Summary

Exercises

Conceptual Problems

Programming Practice

3 Primitive Types and Strings

3.1 Problem: College cost calculator

3.2 Concepts: Types

3.2.1 Types as sets and operations

3.2.2 Primitive and reference types in Java

3.2.3 Type safety

3.3 Syntax: Types in Java

3.3.1 Variables and literals

3.3.2 Primitive types

3.3.3 Reference types

3.3.4 Assignment and comparison

3.3.5 Constants

3.4 Syntax: Useful libraries

3.4.1 The Math library

3.4.2 Random numbers

3.4.3 Wrapper classes

3.5 Solution: College cost calculator

3.6 Concurrency: Expressions

3.6.1 Splitting expressions

3.6.2 Care in splitting expressions

3.7 Summary

Exercises

Conceptual Problems

Programming Practice

4 Selection

4.1 Problem: Monty Hall simulation

4.2 Concepts: Choosing between options

- 4.2.1 Simple choices
- 4.2.2 Boolean operations
- 4.2.3 Nested choices

4.3 Syntax: Selection in Java

- 4.3.1 if statements
- 4.3.2 The boolean type and its operations
- 4.3.3 switch statements

4.4 Solution: Monty Hall

4.5 Concurrency: Selection

Exercises

- Conceptual Problems
- Programming Practice

5 Repetition

5.1 Problem: DNA searching

5.2 Concepts: Repetition

5.3 Syntax: Loops in Java

- 5.3.1 while loops
- 5.3.2 for loops
- 5.3.3 do-while loops
- 5.3.4 Nested loops
- 5.3.5 Common pitfalls

5.4 Solution: DNA searching

5.5 Concurrency: Loops

Exercises

- Conceptual Problems
- Programming Practice
- Experiments

6 Arrays

6.1 Introduction

6.2 Problem: Conway's Game of Life

- 6.2.1 Terminal limitations

6.3 Concepts: Lists of data

- 6.3.1 Data structure attributes
- 6.3.2 Characteristics of an array

6.4 Syntax: Arrays in Java

- 6.4.1 Array declaration and instantiation
- 6.4.2 Indexing into arrays
- 6.4.3 Using loops with arrays
- 6.4.4 Redirecting input

6.5 Examples: Array usage

6.6 Concepts: Multidimensional lists

6.7 Syntax: Advanced arrays in Java

- 6.7.1 Multidimensional arrays
- 6.7.2 Reference types
- 6.7.3 Ragged arrays
- 6.7.4 Common pitfalls

6.8 Examples: Two-dimensional arrays

6.9 Advanced Syntax: Special array tools in Java

- 6.9.1 The for-each loop
- 6.9.2 The Arrays class

6.10 Solution: Conway's Game of Life

6.11 Concurrency: Arrays

Exercises

- Conceptual Problems
- Programming Practice
- Experiments

7 Simple Graphical User Interfaces

7.1 Problem: Codon extractor

7.2 Concepts: User interaction

7.3 Syntax: Dialogs and the JOptionPane class

- 7.3.1 Generating an information dialog
- 7.3.2 Generating a Yes-No confirm dialog
- 7.3.3 Generating a dialog with a list of options
- 7.3.4 Generating a dialog with a custom icon
- 7.3.5 Generating an input dialog

7.4 Solution: Codon extractor

7.5 Concurrency: Simple GUIs

7.6 Summary

Exercises

- Conceptual Problems
- Programming Practice

8 Methods

8.1 Problem: Three card poker

8.2 Concepts: Dividing work into segments

- 8.2.1 Reasons for static methods
- 8.2.2 Parallel to mathematical functions
- 8.2.3 Control flow

8.3 Syntax: Methods

- 8.3.1 Defining methods
- 8.3.2 Calling methods
- 8.3.3 Class variables

8.4 Examples: Defining methods

8.5 Solution: Three card poker

8.6 Concurrency: Methods

Exercises

 Conceptual Problems

 Programming Practice

 Experiments

9 Classes

9.1 Problem: Nested expressions

9.2 Concepts: Object-oriented programming

 9.2.1 Objects

 9.2.2 Encapsulation

 9.2.3 Instance methods

9.3 Syntax: Classes in Java

 9.3.1 Fields

 9.3.2 Constructors

 9.3.3 Methods

 9.3.4 Access modifiers

9.4 Examples: Classes

9.5 Advanced Syntax: Nested classes

 9.5.1 Static nested classes

 9.5.2 Inner classes

9.6 Solution: Nested expressions

9.7 Concurrency: Objects

Exercises

 Conceptual Problems

 Programming Practice

 Experiments

10 Interfaces

10.1 Problem: Sort it out

10.2 Concepts: Making a promise

10.3 Syntax: Interfaces

 10.3.1 Interfaces and static

10.4 Advanced Syntax: Local and anonymous classes

 10.4.1 Local classes

 10.4.2 Anonymous classes

10.5 Solution: Sort it out

10.6 Concurrency: Interfaces

Exercises

 Conceptual Problems

 Programming Practice

 Experiments

11 Inheritance

11.1 Problem: Boolean circuits

11.2 Concepts: Refining classes

 11.2.1 Basic inheritance

 11.2.2 Adding functionality

 11.2.3 Code reuse

11.3 Syntax: Inheritance in Java

 11.3.1 The extends keyword

 11.3.2 Access restriction and visibility

 11.3.3 Constructors

 11.3.4 Overriding methods and hiding fields

 11.3.5 The Object class

11.4 Examples: Problem solving with inheritance

11.5 Solution: Boolean circuits

11.6 Concurrency: Inheritance

Exercises

 Conceptual Problems

 Programming Practice

 Experiments

12 Exceptions

12.1 Problem: Bank burglary

12.2 Concepts: Error handling

 12.2.1 Error codes

 12.2.2 Exceptions

12.3 Syntax: Exceptions in Java

 12.3.1 Throwing exceptions

 12.3.2 Handling exceptions

 12.3.3 Catch or specify

 12.3.4 The finally keyword

 12.3.5 Customized exceptions

12.4 Solution: Bank burglary

12.5 Concurrency: Exceptions

 12.5.1 InterruptedException

Exercises

 Conceptual Problems

 Programming Practice

 Experiments

13 Concurrent Programming

13.1 Introduction

13.2 Problem: Deadly virus

13.3 Concepts: Splitting up work

- 13.3.1 Task decomposition
- 13.3.2 Domain decomposition
- 13.3.3 Tasks and threads
- 13.3.4 Memory architectures and concurrency

13.4 Syntax: Threads in Java

- 13.4.1 The Thread class
- 13.4.2 Creating a thread object
- 13.4.3 Starting a thread
- 13.4.4 Waiting for a thread
- 13.4.5 The Runnable interface

13.5 Examples: Concurrency and speedup

13.6 Concepts: Thread scheduling

- 13.6.1 Nondeterminism
- 13.6.2 Polling

13.7 Syntax: Thread states

13.8 Solution: Deadly virus

13.9 Summary

Exercises

[Conceptual Problems](#)

[Programming Practice](#)

[Experiments](#)

14 Synchronization

14.1 Introduction

14.2 Problem: Dining philosophers

14.3 Concepts: Thread interaction

14.4 Syntax: Thread synchronization

- 14.4.1 The synchronized keyword
- 14.4.2 The wait() and notify() methods

14.5 Pitfalls: Synchronization challenges

- 14.5.1 Deadlock

- 14.5.2 Starvation and livelock

- 14.5.3 Sequential execution

- 14.5.4 Priority inversion

14.6 Solution: Dining philosophers

Exercises

[Conceptual Problems](#)

[Programming Practice](#)

[Experiments](#)

15 Constructing Graphical User Interfaces

15.1 Problem: Math applet

15.2 Concepts: Graphical user interfaces

15.2.1 Swing and AWT

15.3 Syntax: GUIs in Java

15.3.1 Creating a frame

15.3.2 Widgets

15.3.3 Adding actions to widgets

15.3.4 Adding sounds and images

15.3.5 Layout managers

15.3.6 Menus

15.3.7 Applets

15.4 Solution: Math applet

15.5 Concurrency: GUIs

15.5.1 Worker threads

15.6 Summary

Exercises

Conceptual Problems

Programming Practice

Experiments

16 Testing and Debugging

16.1 Fixing bugs

16.1.1 Common sequential bugs

16.2 Concepts: Approaches to debugging

16.2.1 Assertions

16.2.2 Print statements

16.2.3 Step-through execution

16.2.4 Breakpoints

16.3 Syntax: Java debugging tools

16.3.1 Assertions

16.3.2 Print statements

16.3.3 Step-through debugging in Java

16.3.4 Array errors

16.3.5 Scope errors

16.3.6 Null pointer errors

16.4 Concurrency: Parallel bugs

16.4.1 Race conditions

16.4.2 Deadlocks and livelocks

16.4.3 Sequential execution

16.5 Finding and avoiding bugs

16.6 Concepts: Design, implementation, and testing

16.6.1 Design

16.6.2 Implementation

16.6.3 Testing

16.7 Syntax: Java testing tools

16.7.1 JUnit testing

16.8 Concurrency: Testing tools

16.8.1 ConTest

16.8.2 Concutest

16.8.3 Intel tools

16.9 Examples: Testing a class

Exercises

Conceptual Problems

Programming Practice

Experiments

17 Polymorphism

17.1 Problem: Banking account with a vengeance

17.2 Concepts: Polymorphism

17.2.1 The is-a relationship

17.2.2 Dynamic binding

17.2.3 General vs. specific

17.3 Syntax: Inheritance tools in Java

17.3.1 Abstract classes and methods

17.3.2 Final classes and methods

17.3.3 Casting

17.3.4 Inheritance and exceptions

17.4 Solution: Banking account with a vengeance

17.5 Concurrency: Atomic libraries

Exercises

Conceptual Problems

Programming Practice

Experiments

18 Dynamic Data Structures

18.1 Problem: Infix conversion

18.2 Concepts: Dynamic data structures

18.2.1 Dynamic arrays

18.2.2 Linked lists

18.2.3 Abstract data types

18.3 Syntax: Dynamic arrays and linked lists

18.3.1 Dynamic arrays

18.3.2 Linked lists

18.4 Syntax: Abstract data types (ADT)

18.4.1 Stacks

18.4.2 Abstract Data Type: Operations on a stack

18.4.3 Queues

18.4.4 Abstract Data Type: Operations on a queue

18.5 Advanced Syntax: Generic data structures

- 18.5.1 Generics in Java
- 18.5.2 Using a Generic Class
- 18.5.3 Using Java Libraries

18.6 Solution: Infix conversion

18.7 Concurrency: Linked lists and thread safety

18.8 Concurrency: Thread-safe libraries

Exercises

 Conceptual Problems

 Programming Practice

19 Recursion

19.1 Problem: Maze of doom

19.2 Concepts: Recursive problem solving

- 19.2.1 What is recursion?
- 19.2.2 Recursive definitions
- 19.2.3 Iteration vs. recursion
- 19.2.4 Call stack

19.3 Syntax: Recursive methods

19.4 Syntax: Recursive data structures

- 19.4.1 Trees

 19.4.2 Generic dynamic data structures and recursion

19.5 Solution: Maze of doom

19.6 Concurrency: Futures

Exercises

 Conceptual Problems

 Programming Practice

 Experiments

20 File I/O

20.1 Problem: A picture is worth 1,000 bytes

20.2 Concepts: File I/O

- 20.2.1 Non-volatile storage
- 20.2.2 Stream model
- 20.2.3 Text files and binary files

20.3 Syntax: File operations in Java

- 20.3.1 The File class
- 20.3.2 Reading and writing text files
- 20.3.3 Reading and writing binary files

20.4 Examples: File examples

20.5 Solution: A picture is worth 1,000 bytes

20.6 Concurrency: File I/O

Exercises

[Conceptual Problems](#)
[Programming Practice](#)
[Experiments](#)

21 Network Communication

[21.1 Problem: Web server](#)

[21.1.1 HTTP requests](#)

[21.1.2 HTTP responses](#)

[21.2 Concepts: TCP/IP communication](#)

[21.3 Syntax: Networking in Java](#)

[21.3.1 Addresses](#)

[21.3.2 Sockets](#)

[21.3.3 Receiving and sending data](#)

[21.4 Solution: Web server](#)

[21.5 Concurrency: Networking](#)

[Exercises](#)

[Conceptual Problems](#)

[Programming Practice](#)

[Experiments](#)

Index

Preface to Draft 6.0

Welcome to *Start Concurrent!* This book is intended as an entry point into the multicore revolution that is now in full swing. It is designed to introduce students to concurrent programming at the same time they are learning the basics of sequential programming, early in their college days. After mastering the concepts covered here, students should be prepared when they encounter more complex forms of concurrency in advanced courses and in the workplace. A generation of students who learn concurrency from their first course will be ready to exploit the full power of multicore chips by the time they join the workforce.

Multicore processors are omnipresent. Whether you use a desktop or a laptop, chances are that your computer has a multicore chip at its heart. Desktop parallel computers have been prophesied for years. That time has come. Parallel computers sit on our desks and our laps. This progress in microprocessor technology has thrown a challenge to educators: *How can we teach concurrent programming?*

Computer programming has been taught in academia for decades. However, the unwritten goal in nearly every beginning programming class has been teaching students to write, compile, test, and debug **sequential** programs. Material related to concurrent programming is often left to courses about operating systems and programming languages or courses in high performance computing. Now that parallel computers are on our desks, should we consider introducing the fundamentals of concurrent programming in beginner classes in programming? Of course, there are many opinions about this question.

For our part, we believe that concurrent programming can be, and should be, taught to first year students. This book aims at introducing concurrent programming from almost the first day. The rationale for our belief stems from another belief that procedural thinking, sequential as well as concurrent, is natural. People knew how to solve problems in a sequential manner, long before the study of algorithms became a formal subject and computer science a formal discipline. And this rationale applies to problem solving using a collection of sequential solutions applied concurrently. Watch a cook in the kitchen and you will see concurrency in action. Watch a movie and you will see concurrency in action as various subplots, scenes, and flashbacks weave the plot together. Parents use concurrent solutions to solve day-to-day problems as they juggle caring for their children, a career, and a social life.

If people naturally solve problems sequentially and concurrently, why do we need to teach them programming? Programming is a way to map an algorithmic solution of a problem to an artificial language such as Java. It is an activity that requires formal analysis, specialized vocabulary, and razor sharp logic. The real intellectual substance of programming lies in this mapping process. What is the best way to transform a sequential solution to an artificial language? How can a sequential solution be broken into concurrent parts that run faster than the original? How can a large problem be divided into small, manageable chunks that can be programmed separately and then integrated into a whole? In addition, there are issues of testing, debugging, documentation, and management of the software development process, which combine to make programming a limitless field for intellectual curiosity.

Target audience

This book is intended to teach college level students with no programming experience over a period of two semesters. Although we start with concurrency concepts from the very beginning, it is difficult for students with no prior programming experience to write useful multithreaded programs by the end of their first semester. By the end of the second semester, however, this book can lead a student from a blank slate to a capable programmer of complex parallel programs that exploit the power of multicore processors.

The content in this book could also be used for single semester courses. [Chapters 1 through 12](#) are intended for the absolute beginner. If you do not want to introduce concurrent programming in a first course, these chapters should prove adequate. The concurrency material and exercises in these chapters are well-marked and can be ignored without negatively impacting the other material. For a second course in programming, [Chapters 1 through 12](#) should be used as review material as well as an introduction to concurrent programming. Most material from [Chapters 13](#) onward could then be covered in a single semester.

Nature of the material covered

Java is a complex language. Its long list of features makes it difficult for an instructor to decide what to cover and what to leave out. Often there is a tendency to cover more material than less. We have noticed that today's student uses not only a textbook but also the large volume of material available on the web to learn any subject, including programming. Our focus is consequently more on fundamental elements of programming and less on giving a complete description of Java. Where appropriate we direct the student to websites where relevant reference material can be found.

Classes and objects are an essential part of Java. Some educators have adopted an “objects early” approach that focuses heavily on object oriented principles from the very beginning. Although we see many merits in this approach, we feel compelled to start with logic, arithmetic, and control flow so that students have a firm foundation of what to put inside their objects. A full treatment of classes and objects unfolds throughout the book, moving naturally from monolithic programs to decomposition into methods to full object orientation.

Organization

The material covered can be divided up in different ways depending on the needs of the instructor or the student. [Chapters 1 through 12](#), with the exception of [Chapter 7](#), are designed to introduce the student to Java and programming in general. [Chapters 7 and 15](#) cover material related to graphical user interfaces and can be skipped if these topics are not of interest. [Chapters 13 and 14](#) give an in-depth treatment of the concurrency features of Java. Although we make an effort to mark concurrency material and keep it independent from the rest of the content, those chapters numbered 15 and higher will assume some knowledge and interest in concurrency. [Chapter 15](#) itself covers debugging and testing, which is even more crucial in a concurrent environment. The rest of the book covers advanced material relating to OO design, data structures, and I/O.

Chapter layout

One feature of this book that separates it from many Java textbooks is its problem-driven approach. Most chapters are divided into the following parts.

Problem

A motivating problem is given at the beginning of almost all chapters. This problem is intended to show the value of the material covered in the chapter as well as sketching a practical application.

Concepts

One or more short sections devoted to concepts is given in each chapter. The concepts described in these sections are the fundamental topics covered in the chapter, as well as main ideas needed to solve the chapter's motivating problem. These concepts are intended to be broad and language neutral. Java syntax is kept to an absolute minimum in these sections.

Syntax

Each chapter has one or more sections describing the Java syntax needed to implement the concepts already described in the Concepts sections. These sections are typically longer and have numbered examples in Java code sprinkled throughout.

Solution

After the appropriate concepts and Java syntax needed to solve the motivating problem have been given, a solution to the motivating problem is provided near the end of the chapter. In this way, students are given plenty of time to think about the approach needed to solve the problem before the answer is given.

Concurrency

For all of the chapters except for [Chapters 13](#) and [14](#), the dedicated concurrency chapters, additional relevant concurrency concepts and syntax are introduced in these specially marked sections, spreading concurrency throughout the book.

Exercises

Each chapter ends with exercises, which are divided into three sections: Conceptual Problems, Programming Practice, and Experiments. Most Conceptual Problems are simple and are intended as a quick test of the student's understanding. Problems in Programming Practice require students to implement a short program in Java and can be used as homework assignments. Experiments are a special feature of this textbook and are especially appropriate in the context of concurrency. Experiments focus on the performance of a program, usually in terms of speed or memory usage. Students will need to run short programs and measure their running time or other features, gaining practical insight into speedup and other advantages and challenges of concurrency. References to exercises are given throughout the chapter text.

We hope that structuring chapters in this way can be useful for many different kinds of readers. Novice programmers may wish to read each chapter from start to end. Experienced programmers who have never programmed Java may focus primarily on the **Syntax** sections to learn the appropriate Java syntax and semantics. Rusty Java programmers may prefer to focus on the clearly numbered examples and exercises. Of course, instructors are encouraged to use the motivating problems to motivate their lectures as well.

In addition, specially marked **Pitfall** sections throughout the book highlight common programming errors and mistakes.

Exercises involving concurrency or GUIs are clearly marked so that students and instructors who are not interested in these topics can avoid them.

What topics does this book not cover?

This book is not intended to be a comprehensive guide to Java. Instead, it is intended to teach how to use computers to solve problems, especially concurrently. Java has a marvelous wealth of packages and libraries that we do not have the space to cover. For example, the Swing package for building user interfaces is discussed, but not in its entirety. For material not found in this book, we expect students to refer to the material available on the Oracle Java tutorial website (<http://download.oracle.com/javase/tutorial/>) and other reference books and websites.

Suggestions

Quest for the ideal: For many of us, teaching programming is a constant challenge. While some students, inspired by their interactions with computers, immediately love the subject, others find it tedious and uninteresting. Perhaps no combination of pedagogy and teaching excellence will get all students in a large freshman programming class to like programming. Nevertheless, some of us keep trying to excite and motivate every single student in a class full of curious minds. Perhaps this is the ideal that keeps many of us engaged in, and never bored of, teaching first year programming.

Java IDE: It is important that the students be introduced to a Java IDE very early in the course. We recommend that students use a simpler rather than a more complex IDE. We have successfully used DrJava though other simple IDEs might work just as well. For instructors hoping to give their students experience with an industry-level IDE, we give examples using Eclipse as well as DrJava in the chapter on testing and debugging and a few other times when relevant.

Concurrency at the start: It is important to begin the course with a lecture or two on the relationship between problem-solving and computers. [Chapter 2](#) covers this topic. It is during these very early lectures instructors can introduce the notions of both sequential and concurrent solutions. One could use simple problems from areas such as mathematics or physics or even day to day life that lead to sequential and concurrent solutions. Early exposure to solutions these problems, programmed in Java, can be beneficial students even if they do not understand all the syntax.

Input and output: The issue of what input and output material to cover can be dealt with in several ways. While programming attractive GUIs may be exciting, some instructors prefer to postpone detailed treatment of GUI-related material until late in a course. In this book we have decided to follow a flexible approach. We begin by discussing the use of `System.out.print()` and `Scanner` and the `JOptionPane` class as alternatives for basic input and output. Our assumption is that most instructors will prefer the simplicity of command line I/O; however, those who favor a more GUI-heavy approach can start early in [Chapter 7](#) for GUI basics and eventually move onto [Chapter 15](#) for more depth in GUIs and Swing. Instructors who wish to concentrate only on command line I/O are free to ignore these chapters.

Sample courses in programming

Portions of this material were used in an introductory course called “Introduction to Programming with Concurrency.” More details on this course, including the topics covered each week, are available at <http://multicore.cs.purdue.edu/>.

Barry Wittman, Elizabethtown College

Aditya P. Mathur and Tim Korb, Purdue University

September 21, 2012

Acknowledgements

A number of people have played a significant role in motivating the authors to undertake the task of writing this book and in the choice of topics. First, during the spring of 2008, several faculty from the Department of Computer Science and a scientist from Purdue’s ITaP, participated in early discussions related to the teaching of concurrent programming in freshman classes. Despite the multitude of issues raised, all participants seemed to agree on one point: that we ought to introduce concurrency early in the Computer Science undergraduate curriculum. Thanks to all the participants, namely, Buster Dunsmore, Ananth Grama, Suresh Jagannathan, Sunil Prabhakar, Faisal Saied, and Ja Vitek. We benefited from advice, encouragement, and support from a number of alumni and corporate partners; special thanks to Kevin Kahn, Andrew Chien, and Carl Murray.

Thanks to the many anonymous reviewers who carefully read through the Draft 3.0 of this manuscript and made valuable suggestions. This current draft has resulted from the many suggestions and critiques of these reviewers.

In the fall of 2008, we offered an experimental freshman class entitled “Introduction to Programming with Concurrency.” This class was certainly one of the best we have taught to freshmen—they retained their enthusiasm throughout the semester to learn the subject. Thanks to Alexander Bartol, Alexander Coe, Eric Fisher, Benjamin Gilliland-Sauer, John Graff, Tyler Holzer, Kelly, Jordan Kelly, Azfar Khandoker, Zackary Naas, Ravi Pareek, Carl Rhodes, Robert Schwalm, Andrew Wirtz, and Christopher Womble.

Special thanks to Azfar Khandoker who not only attended CS190M, but also worked on an independent study project to create exercises using the Lego robot to help students learn programming. Azfar’s work has led to the use of robots in two freshman programming classes taught at Purdue.

We appreciate the support and cooperation of the faculty, and their students, who are our first “test users”: Prof. David John of Wake Forest University and Prof. Sunil Prabhakar of Purdue University.

Thanks to our able assistants Karla Cotter and Nicole Piegza for assisting us with time management. This project required us to steal several precious weekend hours from members of our respective families. Finally, we thank members of our families for their constant support throughout this project.

Chapter 1

Computer Basics

Computers are useless. They can only give you answers.

—Pablo Picasso

1.1 Problem: Buying a computer

We begin almost every chapter of this book with a motivating problem. Why? Sometimes it helps to see how tools can be applied in order to see why they’re useful. As we move through each chapter, we cover background concepts needed to solve the problem in the **Concepts** section, the specific technical details (usually in the form of Java syntax) required in the **Syntax and Semantics** section, and eventually the solution to the problem in the **Solution** sections. If you’re not interested in the problem, that’s fine! Feel free to skip ahead to the **Concepts** section or directly to the **Syntax and Semantics** section, especially if you already have some programming experience. Then, if you’d like to see another detailed example, the solution to each chapter’s problem is still available as a reference.

We’ll start with a problem that is not about programming and may be familiar to you. Imagine you are just about to start college and need a computer. Should you buy a Mac or PC? What kind of computer is going to run programs faster? Do some kinds of computers crash more than others? Which features are worth paying more for? Why are there so many buzzwords and so much impenetrable jargon associated with buying a computer?

When many people hear “computer science,” these are often the first questions that come to mind. Most of this book is about programming computers in a language called Java and not about the computers themselves. We try to present all the material so that almost any kind of computer can be used when programming the problems and examples. Nevertheless, both the hardware that makes up your computer and the other software running on it affect the way the programs you write work.

1.2 Concepts: Hardware and software

Computers are ubiquitous. We see them nearly everywhere. They are found in most homes, shops, cars, aircraft, phones, and inside many other devices. Sometimes they are obvious, like a laptop sitting on a desk. But, most computers are hidden inside other devices such as a watch or a flat panel television. Computers can be complex or relatively simple machines. Despite their diversity, we can think of all computers in terms of their *hardware* and the *software* that runs on it.

1.2.1 Hardware

Hardware consists of the physical components that make up a computer but not the programs or data stored on it. Hardware components can be seen and touched, if you are willing to open the computer

case. One way to organize hardware is to break it down into three categories: the processor, the memory, and input and output (I/O) devices.

This view of a computer is a simplified version of what is called the von Neumann architecture or a stored-program computer. It is a good (but imperfect) model of most modern computers. In this model, a program (a list of instructions) is stored in memory. The processor loads the program and performs the instructions, some of which require the processor to do a lot of number-crunching. Sometimes the processor reads data out of memory or writes new data into it. Periodically, the processor may send output to one of the output devices or receive input from one of the input devices.

In other words, the processor thinks, the memory stores, and the I/O devices interact with the outside world. The processor sits between the memory and the I/O devices. Let's examine these categories further.

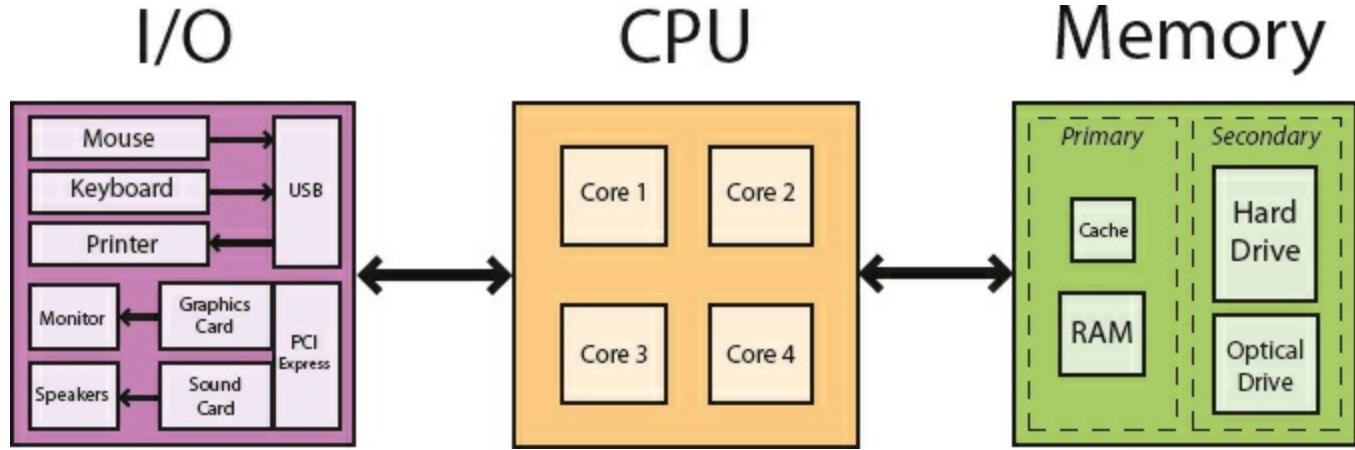


Figure 1.1: Hardware components in a typical desktop computer categorized into CPU, memory, and I/O devices.

CPU

The processor, or central processing unit (CPU), is the “brain” of a computer. It fetches instructions, decodes them, and executes them. It may send data to or from memory or I/O devices. The CPU of virtually all modern computers is a *microprocessor*, meaning that all the computation is done by a single integrated circuit fabricated out of silicon. What are the important features of CPUs? How do we measure their speed and power?

Frequency: The speed of a CPU (and indeed a computer as a whole) is often quoted in gigahertz (GHz). Hertz (Hz) is a measurement of frequency. If something happens once per second, it has a frequency of exactly 1 Hz. Perhaps the second hand on your watch moves with a frequency of 1 Hz. In North America, the current in electrical outlets alternates with a frequency of approximately 60 Hz. Sound can also be measured by frequency. The lowest-pitched sound the human ear can hear is around 20 Hz. The highest-pitched sound is around 20,000 Hz. Such a sound pulses against your eardrum 20,000 times per second. That sounds like a lot, but many modern computers operate at a frequency of 1 to 4 gigahertz. The prefix “giga” means “billion.” So, we are talking about computers doing **something** more than a billion (1,000,000,000) times per second.

But **what** are they doing? This frequency is the *clock rate*, which marks how often a regular electrical signal passes through the CPU. On each tick, the CPU does some computation. How much? It depends. On some systems, simple instructions (like adding two numbers) can be computed in a single clock cycle. Other instructions can take ten or more clock cycles. Different processor designs can take different numbers of cycles to execute the same instructions. Instructions are also *pipelined*, meaning that one instruction is being executed while another one is being fetched or decoded. Different processors can have different ways of optimizing this process. Because of these differences, the frequency of a processor as measured in gigahertz is not a good way to compare the effective speed of one processor to another, unless the two processors are very closely related. Even though it doesn't really make sense, clock rate is commonly advertised as the speed of a computer.

Word size: Perhaps you have heard of a 32-bit or 64-bit computer. As we discuss in the subsection about memory, a bit is a 0 or a 1, the smallest amount of information you can record. Most new laptop and desktop computers are 64-bit machines, meaning that they operate on 64 bits at a time and can use 64-bit values as memory addresses. The instructions that it executes work on 64-bit quantities, i.e., numbers made up of 64 0s and 1s. The size of data that a computer can operate on with a single instruction is known as its *word size*.

In day to day operations, word size is not important to most users. Certain programs that interact directly with the hardware, such as the operating system, may be affected by the word size. For example, most modern 32-bit operating systems are designed to run on a 64-bit processor, but most 64-bit operating systems do not run on a 32-bit processor.

Programs often run faster on machines with a larger word size, but they typically take up more memory. A 32-bit processor (or operating system) cannot use more than 4 gigabytes (defined below) of memory. Thus, a 64-bit computer is needed to take advantage of the larger amounts of memory that are now available.

Cache: Human brains both perform computations and store information. A computer CPU performs computations, but, for the most part, does not store information. The CPU cache is the exception. Most modern CPUs have a small, very fast section of memory built right onto the chip. By guessing about what information the CPU is going to use next, it can preload it into the cache and avoid waiting around for the slower regular memory.

Over time, caches have become more complicated and often have multiple levels. The first level is very small but incredibly fast. The second level is larger and slower. And so on. It would be preferable to have a large, first-level cache, but fast memory is expensive memory. Each level is larger, slower, and cheaper than the last.

Cache size is not a heavily advertised CPU feature, but it makes a huge difference in performance. A processor with a larger cache can often outperform a processor that is faster in terms of clock rate.

Cores: Most laptops and desktops available today have *multicore* processors. These processors contain two, four, six, or even more cores. Each core is a processor capable of independently executing instructions, and they can all communicate with the same memory.

In theory, having six cores could allow your computer to run six times as fast. In practice, this speedup is rarely the case. Learning how to get more performance out of multicore systems is one of the major themes of this book. [Chapters 13](#) and [14](#) as well as sections marked **Concurrency** in other chapters are specifically tailored for students interested in programming these multicore systems to work effectively. If you aren't interested in concurrent programming, you can skip these chapters and sections and use this book as a traditional introductory Java programming textbook. On the other hand, if you are interested in the increasingly important area of concurrent programming, [Section 1.5](#) near the end of this chapter is the first **Concurrency** section of the book and discusses multicore processors more deeply.

Memory

Memory is where all the programs and data on a computer are stored. The memory in a computer is usually not a single piece of hardware. Instead, the storage requirements of a computer are met by many different technologies.

At the top of the pyramid of memory is primary storage, memory that the CPU can access and control directly. On desktop and laptop computers, primary storage usually takes the form of random access memory (RAM). It is called random access memory because it takes the same amount of time to access any part of RAM. Traditional RAM is volatile, meaning that its contents are lost when it's unpowered. All programs and data must be loaded into RAM to be used by the CPU.

After primary storage comes secondary storage, which is dominated by hard drives that store data on spinning magnetic platters. Optical drives (such as CD, DVD, and Blu-ray), flash drives, and the now virtually obsolete floppy drives fall into the category of secondary storage as well. Secondary storage is slower than primary storage, but it is non-volatile. Some forms of secondary storage such as CD-ROM and DVD-ROM are read only, but most are capable of reading and writing.

Before we can compare these kinds of storage effectively, we need to have a system for measuring how much they store. In modern digital computers, all data is stored as a sequence of 0s and 1s. In memory, the space that can hold either a single 0 or a single 1 is called a *bit*, which is short for “binary digit.”

A bit is a tiny amount of information. For organizational purposes, we call a sequence of eight bits a *byte*. The word size of a CPU is two or more bytes, but memory capacity is usually listed in bytes not words. Both primary and secondary storage capacities have become so large that it is inconvenient to describe them in bytes. Computer scientists have borrowed prefixes from physical scientists to create suitable units.

Common units for measuring memory are bytes, kilobytes, megabytes, gigabytes, and terabytes. Each unit is 1,024 times the size of the previous unit. Notice that 2^{10} (1,024) is almost the same as 10^3 (1,000). Sometimes it is not clear which value is meant. Disk drive manufacturers always use powers of 10 when they quote the size of their disks. Thus, a 1 TB hard disk can hold 10^{12} (1,000,000,000,000) bytes, not 2^{40} (1,099,511,627,776) bytes. Standards organizations have advocated that the terms kibibyte (KiB), mebibyte (MiB), gibibyte (GiB), and tebibyte (TiB) be used to refer to the units based on powers of 2 while the traditional names be used to refer only to the units based on powers of 10, but the new terms have not yet become popular.

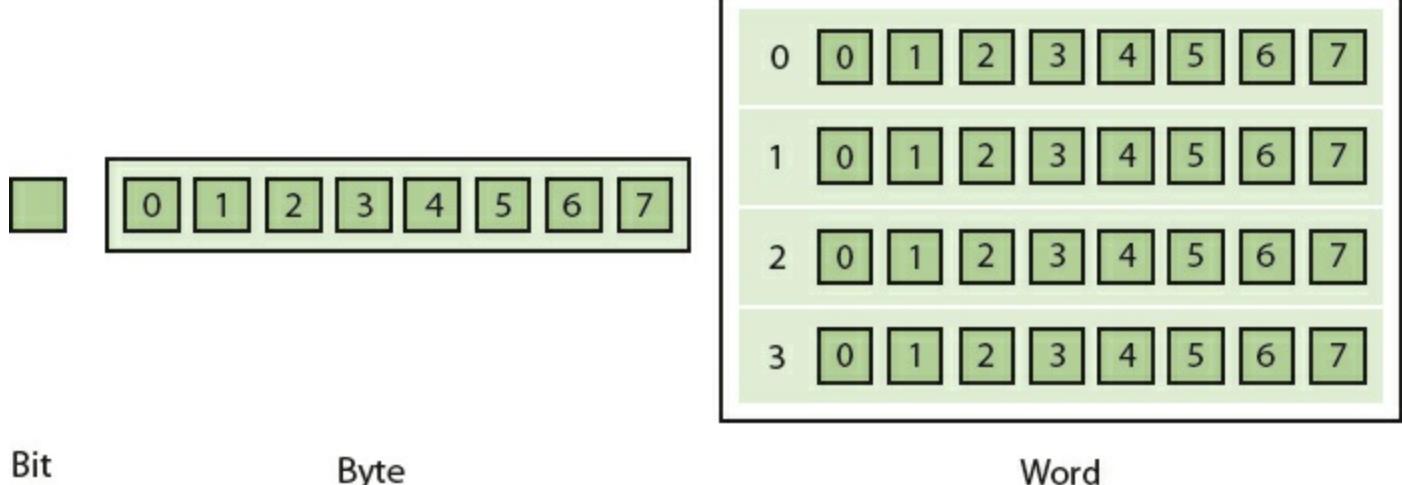


Figure 1.2: A computer's memory contains bits organized as bytes and words. One bit contains either a 0 or a 1. A byte contains eight bits. A word may contain two or more bytes. Shown here is a word containing four bytes, or 32 bits. Computer scientists often number items starting at zero, as we discuss in [Chapter 6](#).

Unit	Size	Bytes	Practical Measure
byte	8 bits	$2^0 = 10^0$	a single character
kilobyte (KB)	1,024 bytes	$2^{10} \approx 10^3$	a paragraph of text
megabyte (MB)	1,024 kilobytes	$2^{20} \approx 10^6$	a minute of MP3 music
gigabyte (GB)	1,024 megabytes	$2^{30} \approx 10^9$	half an hour of DVD video
terabyte (TB)	1,024 gigabytes	$2^{40} \approx 10^{12}$	80% of a human's memory capacity, estimated by Raymond Kurzweil

We called memory a pyramid earlier in this section. At the top there is a small but very fast amount of memory. As we work down the pyramid, the storage capacity grows but the speed slows down. Of course, the pyramid for every computer is different. Below is a table that shows many kinds of memory moving from the fastest and smallest to the slowest and largest. Effective speed is hard to measure (and is changing as technology progresses), but note that each layer in the pyramid tends to be 10-100 times slower than the previous layer.

Memory	Typical Capacity	Use
Cache	kilobytes or megabytes	Cache is fast, temporary storage for the CPU itself. Modern CPUs have two or three levels of cache that get progressively bigger and slower.
RAM	gigabytes	The bulk of primary memory is RAM. RAM comes on sticks that can be swapped out to upgrade a computer.
Flash drives	gigabytes or tens of gigabytes	Flash drives mark the beginning of secondary storage. Flash drives come as USB keychain drives but also as drives that sit inside the computer (sometimes called <i>solid state drives</i> or SSDs). As the price of flash drives drops, they are expected to replace hard drives entirely. (Some expensive SSDs already have capacities in the terabyte range.)
Hard drives	hundreds of gigabytes or terabytes	Hard drives are still the most common secondary storage for desktops, laptops, and servers. They are limited in speed partly because of their moving parts.
Tape backup	terabytes and beyond	Some large companies still store huge quantities of information on magnetic tape. Tape performs well for long sequential accesses.
Network storage	terabytes and beyond	Storage that is accessed through a network is limited by the speed of the network. Many companies use networked computers for backup and redundancy as well as distributed computation. Microsoft, Amazon, Google, and others rent their network storage systems at rates based on storage size and total data throughput. These services are part of what is called <i>cloud computing</i> .

I/O devices

I/O devices have much more variety than CPUs or memory. Some I/O devices, such as USB ports, are permanently connected by a printed circuit board to the CPU. Other devices called *peripherals* are connected to a computer as needed. Their types and features are many and varied, and in this book, we do not go deeply into how to interact with I/O devices.

Common input devices include mice, keyboards, touch pads, microphones, game pads, and drawing tablets. Common output devices include monitors, speakers, and printers. Some devices perform both input and output, such as a network card.

Remember that our view of computer hardware as CPU, memory, and I/O devices is only a model. A PCI Express socket can be considered an I/O device, but the graphics card that fits into the socket

can be considered one as well. And the monitor that connects to the graphics card is yet another one. Although the graphics card is an I/O device, it has its own processor and memory, too. It's pointless to get bogged down in too many details. One of the most important skills in computer science is finding the right level of detail and abstraction to view a given problem.

1.2.2 Software

Without hardware computers would not exist, but software is equally important. Software is the programs and data that are executed and stored by the computer. The focus of this book is learning to write software.

Software includes the infinite variety of computer programs. With the right tools (many of which are free), anyone can write a program that runs on a Windows, Mac, or Linux machine. Although it would be nearly impossible to list all the different kinds of software, a few categories are worth mentioning.

Operating Systems: The *operating system* (OS) is the software that manages the interaction between the hardware and the rest of the software. Programs called *drivers* are added to the OS for each hardware device. For example, when an application wants to print a document, it communicates with the printer via a printer driver that is customized for the specific printer, the OS, and the computer hardware. The OS also schedules, runs, and manages memory for all other programs. The three most common OSes for desktop machines are Microsoft Windows, Mac OS, and Linux. At the present time, all three run on similar hardware based on the Intel x86 and x64 architectures.

Microsoft does not sell desktop computers, but many desktop and laptop computers come bundled with Windows. For individuals and businesses who assemble their own computer hardware, it is also possible to purchase Windows separately. In contrast, most computers running Mac OS are sold by Apple, and Mac OS is usually bundled with the computer. Linux is *open-source software*, meaning that all the source code used to create it is freely available. In spite of Linux being free, many consumers prefer Windows or Mac OS because of ease of use, compatibility with specific software, and technical support. Many consumers are also unaware that hardware can be purchased separately from an OS or that Linux is a free alternative to the other two.

Other computers have OSes as well. The Motorola Xoom and many kinds of mobile telephones use the Google Android OS. The Apple iPad and iPhone use the competing Apple iOS. Phones, microwave ovens, automobiles, and countless other devices have computers in them that use some kind of embedded OS.

Example 1.1: CPU scheduling

Consider two applications running on a mobile phone with a single core CPU. One application is a web browser and the other is a music player. The user may start listening to music and then start the browser. In order to function, both applications need to access the CPU at the same time. Since the CPU only has a single core, it can execute only one instruction at a time.

Rather than forcing the user to finish listening to the song before using the web browser, the OS

switches the CPU between the two applications very quickly. This switching allows the user to continue browsing while the music plays in the background. The user perceives an illusion that both applications are using the CPU at the same time. ■

Compilers: A *compiler* is a kind of program that is particularly important to programmers. Computer programs are written in special languages, such as Java, that are human readable. A compiler takes this human-readable program and turns it into instructions (often machine code) that a computer can understand.

To compile the programs in this book, you use the Java compiler javac, either directly by typing its name as a command or indirectly as Eclipse, DrJava, or some other tool that runs the compiler for you.

Business Applications: Many different kinds of programs fall under the umbrella of business or productivity software. Perhaps the most famous is the Microsoft Office suite of tools, which includes the word-processing software Word, the spreadsheet software Excel, and the presentation software PowerPoint.

Programs in this category are often the first to come to mind when people think of software, and this category has had tremendous historical impact. The popularity of Microsoft Office led to the widespread adoption of Microsoft Windows in the 1990s. A single application that is so desirable that a consumer is willing to buy the hardware and the OS just to be able to run it is sometimes called a *killer app*.

Video Games: Video games are software like other programs, but they deserve special attention because they represent an enormous, multi-billion dollar industry. They are usually challenging to program, and the video game development industry is highly competitive.

The intense 3D graphics required by modern video games have pushed hardware manufacturers such as Nvidia, AMD, and Intel to develop high-performance graphics cards for desktop and laptop computers. At the same time, companies like Nintendo, Sony, and Microsoft have developed computers such as the Wii, PS3, and Xbox 360 that specialize in video games but are not designed for general computing tasks.

Web Browsers: Web browsers are programs that can connect to the Internet and download and display web pages and other files. Early web browsers could only display relatively simple pages containing text and images. Because of the growing importance of communication over the Internet, web browsers have evolved to include plug-ins that can play sounds, display video, and allow for sophisticated real-time communication.

Popular web browsers include Microsoft Internet Explorer, Mozilla Firefox, Apple Safari, and Google Chrome. Each has advantages and disadvantages in terms of compatibility, standards compliance, security, speed, and customer support. The Opera web browser is not well known on desktop computers, but it is commonly used on mobile telephones.

1.2.3 Examples

Here are a few examples of modern computers with a brief description of their hardware and some of

the software that runs on them.

Example 1.2: Desktop computer

The Inspiron 560 Desktop is a modestly priced computer manufactured and sold by Dell, Inc. It can be configured with different hardware options, but one choice uses a 64-bit Intel Pentium E6700 CPU that runs at a clock rate of 3.2 GHz with a 2 MB cache and two cores. In terms of memory, you can choose between 4 and 6 GB worth of RAM. You can also choose to have a 500 GB or 1 TB hard drive. The computer comes with a DVD±RW optical drive.

For I/O, the computer has various ports for connecting USB devices, monitors, speakers, microphones, and network cables. By default, it includes a keyboard, a mouse, a network card, a graphics card, and an audio card. For an additional charge, a monitor, speakers, and other peripherals can be purchased.

A 64-bit edition of Microsoft Windows 7 is included. (Software often uses version numbers to mark changes in features and support, but Microsoft has adopted some very confusing numbering schemes. Windows 7 is the successor to Windows Vista, which is the successor to Windows XP. Windows 7 is **not** the seventh version of the Windows OS, but Windows 8 is the successor to Windows 7.) Depending on the price, different configurations of Microsoft Office 2010 with more or fewer features can be included. [Figure 1.3\(a\)](#) shows a picture of the Dell Inspiron 560. ■



Figure 1.3: (a) Dell Inspiron 560. (b) Apple iPhone 4. (c) Motorola Xoom.

Example 1.3: Smartphone

All mobile phones contain a computer, but a phone that has features like a media player, calendar, GPS, or camera is often called a *smartphone*. Such phones often have sophisticated software that is comparable to a desktop computer. One example is the Apple iPhone 4.

This phone uses a CPU called the A4, which has a single core, a cache of 512 KB, and a maximum

clock rate of 1 GHz, though the clock rate used in the iPhone 4 is not publicly known. The phone has 512 MB of RAM and uses either a 16 GB or 32 GB flash drive for secondary storage.

In terms of I/O, the iPhone 4 has a built-in liquid crystal display (LCD) that is also a touch screen for input. It has two cameras, an LED flash, a microphone, a speaker, a headphone jack, a docking connector, buttons, gyroscopes, accelerometers, and the capability to communicate on several kinds of wireless networks.

In addition to the Apple iOS 4 operating system, the iPhone runs a variety of applications just like a desktop computer. These applications are available from the iTunes App Store. [Figure 1.3\(b\)](#) shows a picture of the iPhone 4. ■

Example 1.4: Tablet

The Motorola Xoom is a *tablet computer*. A tablet computer has a touch screen and is generally lighter than a laptop. Some tablets have keyboards, but many newer models use the touch screen instead.

The Xoom uses the Nvidia Tegra 2 CPU, which runs at 1 GHz and has 1 MB of cache and two cores. It has 1 GB of RAM and a 32 GB flash drive for storage. It has a built-in LCD that is also touch screen for input, with a connector for a monitor. It has two cameras, an LED flash, a microphone, a speaker, a headset jack, buttons, gyroscopes, accelerometers, a barometer, and the capability to communicate on several kinds of wireless networks.

It uses the Google Android 3 operating system, which can run applications available from the Android Market. [Figure 1.3\(c\)](#) shows a picture of the Motorola Xoom. ■

1.3 Syntax: Data representation

After each **Concepts** section, this book usually has a **Syntax** section. Syntax is the rules for a language. These **Syntax** sections generally focus on concrete Java language features and technical specifics that are related to the concepts described in the chapter.

In this chapter, we are still trying to describe computers at a general level. Consequently, the technical details we cover in this section will not be Java syntax. Although everything we say applies to Java, it also applies to many other programming languages.

1.3.1 Compilers and interpreters

This book is primarily about solving problems with computer programs. From now on, we only mention hardware when it has an impact on programming. The first step to writing a computer program is deciding what language to use.

Most humans communicate via natural languages such as Chinese, English, French, Russian, or Tamil. However, computers are poor at understanding natural languages. As a compromise, programmers write programs (instructions for a computer to follow) in a language more similar to a natural language than it is to the language understood by the CPU. These languages are called *high-level languages* because they are closer to natural language (the highest level) than they are to *machine language* (the lowest level). We may also refer to machine language as *machine code* or

native code.

Thousands of programming languages have been created over the years, but some of the most popular high level languages of all time include Fortran, Cobol, Visual Basic, C, C++, Python, Java and C#.

As we mentioned in the previous section, a compiler is a program that translates one language into another. In many cases, a compiler translates a high-level language into a low level language that the CPU can understand and execute. Because all the work is done ahead of time, this kind of compilation is known as static or ahead-of-time compilation. In other cases, the output of the compiler is an intermediate language that is easier for the computer to understand than the high-level language but still takes some translation before the computer can follow the instructions.

An *interpreter* is a program that is similar to a compiler. However, an interpreter takes code in one language as input and, on the fly, runs each instruction on the CPU as it translates it. Interpreters generally execute the code more slowly than if it had been translated to machine language before execution.

Note that both compilers and interpreters are normal programs. They are usually written in high-level languages and compiled into machine language before execution. This raises a philosophical question: If you need a compiler to create a program, where did the first compiler come from?

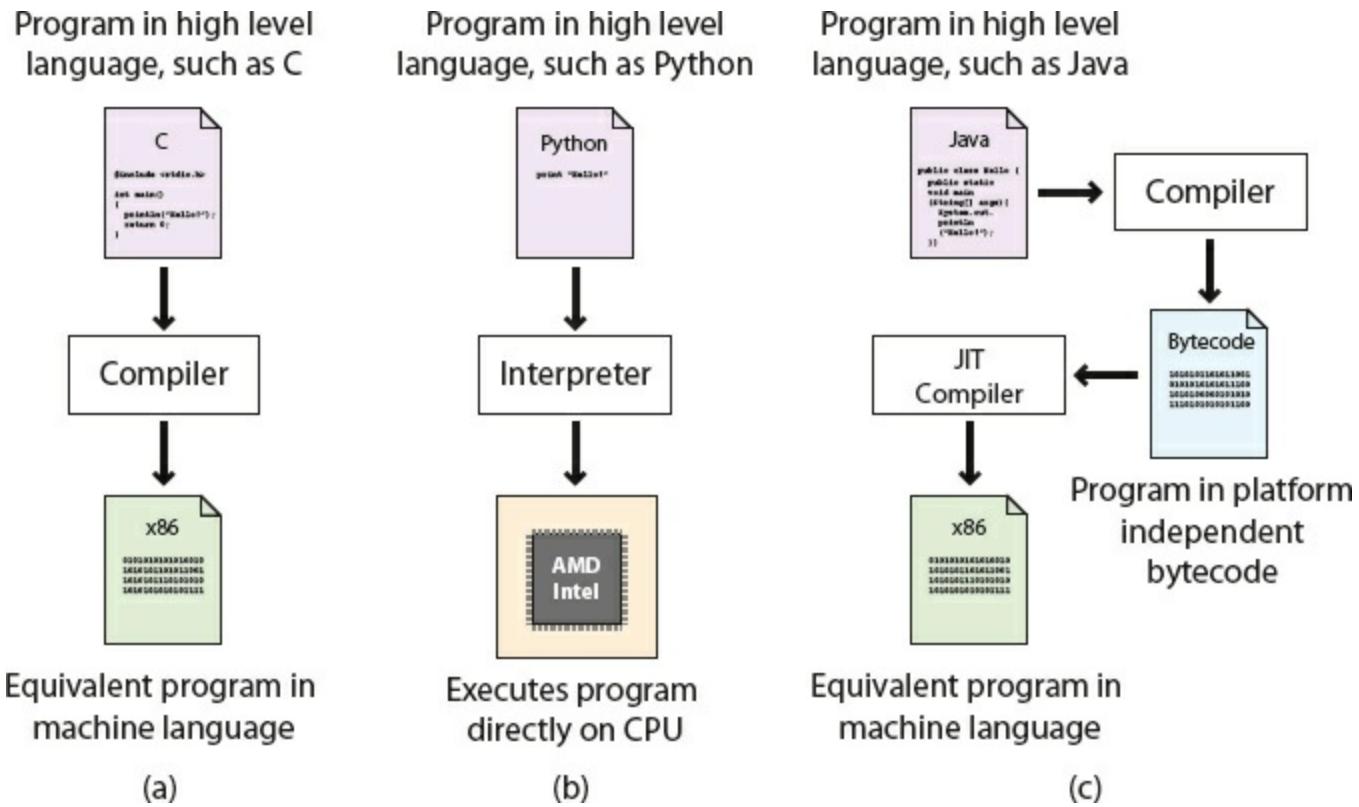


Figure 1.4: (a) Static compilation. (b) Interpreted execution. (c) Compilation into bytecode with late just-in-time compilation.

Example 1.5: Java compilation

Java is the popular high-level programming language we will focus on in this book. The standard way to run a Java program has an extra step that most compiled languages do not. Most compilers for Java, though not all, translate a program written in Java to an intermediate language known as

bytecode. This intermediate version of the high-level program is used as input for another program called the Java Virtual Machine (JVM). Most popular JVMs translate the bytecode into machine code that is executed directly by the CPU. This conversion from bytecode into machine code is done with a just-in-time (JIT) compiler. It's called "just-in-time" because sections of bytecode are not compiled until the moment they are needed. Because the output is going to be used for this specific execution of the program, the JIT can do optimizations to make the final machine code run particularly well in the current environment.

Why does Java use the intermediate step of bytecode? One of Java's design goals is to be platform independent, meaning that it can be executed on any kind of computer. This is a difficult goal because every combination of OS and CPU will need different low level instructions. Java attacks the problem by keeping its bytecode platform independent. You can compile a program into bytecode on a Windows machine and then run the bytecode on a JVM in a Mac OS X environment. Part of the work is platform independent, and part is not. Each JVM must be tailored to the combination of OS and hardware that it runs on.

The Java language and original JVM were developed by Sun Microsystems, Inc., which was bought by Oracle Corporation in 2009. Oracle continues to produce HotSpot, the standard JVM, but many other JVMs exist, including Apache Harmony and Dalvik, the Google Android JVM. ■

1.3.2 Numbers

All data inside of a computer is represented with numbers. Although humans use numbers in our daily lives, the representation and manipulation of numbers by computers work differently. In this subsection we introduce the notions of number systems, bases, conversion from one base to another, and arithmetic in number systems.

A few number systems

A number system is a way to represent numbers. It is easy to confuse the *numeral* that represents the number with the number itself. You might think of the number ten as "10," a numeral made of two symbols, but the number itself is the concept of **ten-ness**. You could express that quantity by holding up all your fingers, with the symbol "X," or by knocking ten times.

Representing ten with "10" is an example of a *positional number system*, namely base 10. In a positional number system, the position of the digits determines the magnitude they represent. For example, the numeral 3,432 contains the digit 3 twice. The first time, it represents three groups of one thousand. The second time, it represents three groups of ten. (The Roman numeral system is an example of a number system that is **not** positional.)

The numeral 3,432 and possibly every other normally written number you have seen is expressed in the base 10 or *decimal* system. It is called base 10 because, as you move from the rightmost digit leftward, the value of each position goes up by a factor of 10. Also, in base 10, ten is the smallest positive integer that requires two digits for representation. Each smaller number has its own digit: 0, 1, 2, 3, 4, 5, 6, 7, 8, and 9. Representing ten requires two existing digits to be combined. Every base has the property that the number it is named after takes two digits to write, namely "1" and "0." (An exception is base 1, which does not behave like the other bases and is not a normal positional number system.)

Example 1.6: Decimal numbers

The number 723 can be written as $723 = 7 \times 10^2 + 2 \times 10^1 + 3 \times 10^0$.

Note that the rightmost digit is the ones place, which is equivalent to 10^0 . Be sure to start with b^0 and not b^1 when considering the value of a number written in base b , no matter what b is. The second digit from the right is multiplied by 10^1 , and so on. The product of a digit and the corresponding power of 10 tells us how much a digit contributes to the number. In the above expansion, digit 7 contributes 700 to the number 723. Similarly, digits 2 and 3 contribute, respectively, 20 and 3 to 723.

As we move to the right, the power of 10 goes down by one, and this pattern works even for negative powers of 10. If we expand the fractional value 0.324, we get $0.324 = 3 \times 10^{-1} + 2 \times 10^{-2} + 4 \times 10^{-3}$.

We can combine the above two numbers to get $723.324 = 7 \times 10^2 + 2 \times 10^1 + 3 \times 10^0 + 3 \times 10^{-1} + 2 \times 10^{-2} + 4 \times 10^{-3}$. ■

We can expand these ideas to any base, checking our logic against the familiar base 10. Suppose that a numeral consists of n symbols $s_{n-1}, s_{n-2}, \dots, s_1, s_0$. Furthermore, suppose that this numeral belongs to the base b number system. We can expand the value of this numeral to:

$$s_{n-1}s_{n-2}\dots s_1s_0 = s_{n-1} \times b^{n-1} + s_{n-2} \times b^{n-2} + \dots + s_1 \times b^1 + s_0 \times b^0$$

The leftmost symbol in the numeral is the *highest order digit* and the rightmost symbol is the *lowest order digit*. For example, in the decimal numeral 492, 4 is the highest order digit and 2 the lowest order digit.

Fractions can also be expanded in a similar manner. For example, a fraction with n symbols $s_1, s_2, \dots, s_{n-1}, s_n$ in a number system with base b , can be expanded to:

$$0.s_1s_2\dots s_{n-2}s_{n-1} = s_1 \times b^{-1} + s_2 \times b^{-2} + \dots + s_{n-1} \times b^{n-1} + s_n \times b^{-n}$$

To avoid confusion, the base number is always written in base 10. As computer scientists, we are interested in base 2 because that's the base used to express numbers inside of a computer. Base 2 is also called *binary*. The only symbols allowed to represent numbers in binary are “0” and “1,” the binary digits or *bits*.

In the binary numeral 100110, the leftmost 1 is the highest order bit and the rightmost 0 is the lowest order bit. By the rules of positional number systems, the highest order bit represents $1 \times 2^5 = 32$.

Example 1.7: Binary numbers

Examples of numbers written in binary are 100, 111, 0111, and 10101. Recall that the base of the binary number system is 2. Thus, we can write a number in binary as the sum of products of powers of 2. For example, the numeral 10011 can be expanded to:

$$10011 = 1 \times 2^4 + 0 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 1 \times 2^0 = 16 + 0 + 0 + 2 + 1 = 19$$

By expanding the number, we have also shown how to convert a binary numeral into a decimal

numeral. Remember that both 10011 and 19 represent the same value, namely nineteen. The conversion between bases changes only the way the number is written. As before, the rightmost bit is multiplied by 2^0 to determine its contribution to the binary number. The bit to its left is multiplied by 2^1 to determine its contribution, and so on. In this case, the leftmost 1 contributes $1 \times 2^4 = 16$ to the value. ■

Exercise 1.4

Exercise 1.5

Another useful number system is *base 16*, also known as *hexadecimal*. Hexadecimal is surprising because it requires more than the familiar 10 digits. Numerals in this system are written with 16 hexadecimal digits that include the ten digits 0 through 9 and the six letters A, B, C, D, E, and F. The six letters, starting from A, correspond to the values 10, 11, 12, 13, 14, and 15.

Hexadecimal is used as a compact representation of binary. Binary numbers can get very long, but four binary digits can be represented with a single hexadecimal digit.

Example 1.8: Hexadecimal numbers

39A, 32, and AFBC12 are examples of numbers written in hexadecimal. A hexadecimal numeral can be expressed as the sum of products of powers of 16. For example, the hexadecimal numeral A0BF can be expanded to:

$$A \times 16^3 + 0 \times 16^2 + B \times 16^1 + F \times 16^0$$

To convert a hexadecimal numeral to decimal, we must substitute the values 10 through 15 for the digits A through F. Now we can rewrite the sum of products from above as:

$$10 \times 16^3 + 0 \times 16^2 + 11 \times 16^1 + 15 \times 16^0 = 4096 + 0 + 176 + 15 = 4463$$

Thus, we get A0BF = 4463. ■

The base 8 number system is also called *octal*. Like hexadecimal, octal is used as a shorthand for binary. A numeral in octal uses the octal digits 0, 1, 2, 3, 4, 5, 6, and 7. Otherwise the same rules apply. For example, the octal numeral 377 can be expanded to:

$$377 = 3 \times 8^2 + 7 \times 8^1 + 7 \times 8^0 = 255$$

You may have noticed that it is not always clear which base a numeral is written in. The digit sequence 337 is a legal numeral in octal, decimal, and hexadecimal, but it represents different numbers in each system. Mathematicians use a subscript to denote the base in which a numeral is written.

Thus, $337_8 = 255_{10}$, $377_{10} = 377_{10}$, and $377_{16} = 887_{10}$. Base numbers are always written in base 10. A number without a subscript is assumed to be in base 10. In Java, there is no way to mark subscripts and so prefixes are used. A prefix of 0 is used for octal, no prefix is used for decimal, and a prefix of 0x is used for hexadecimal. A numeral cannot be marked as binary in Java. The corresponding numerals in Java code would thus be written 0377, 377, and 0x377. Be careful not to

pad numbers with zeroes in Java. Remember that the value 056 is **not** the same as the value 56 in Java.

The following table lists a few characteristics of the four number systems we have discussed with representations of the numbers 7 and 29.

Number System	Base	Digits	Math Numerals	Java Numerals
Binary	2	0, 1	$111_2, 11101_2$	—
Octal	8	0, 1, 2, 3, 4, 5, 6, 7	$7_8, 35_8$	07, 035
Decimal	10	0, 1, 2, 3, 4, 5, 6, 7, 8, 9	7, 29	7, 29
Hexadecimal	16	0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F	$7_{16}, 1D_{16}$	0x7, 0x1D

1.3.3 Conversion across number systems

It is often useful to know how to convert a number represented in one base to the equivalent representation in another base. The examples have shown how to convert a numeral in any base to decimal by expanding the numeral in the sum-of-product form and then adding the different terms together. But how do you convert a decimal numeral to another base?

Decimal to binary conversion

There are at least two different ways to convert a decimal numeral to binary. One way is to write the decimal number as a sum of powers of two as in the following conversion of the number 23.

$$23 = 16 + 0 + 4 + 2 + 1 = 1 \times 2^4 + 0 \times 2^3 + 1 \times 2^2 + 2 + 1 \times 2^0 = 10111_2$$

First, find the largest power of two that is greater than or equal to the number. In this case, 16 fits the bill because 32 is too large. Subtract that value from the number, leaving 7 in this case. Then repeat the process. The last step is to collect the coefficients of the powers of two into a sequence to get the binary equivalent. We used 16, 4, 2, and 1 but skipped 8. If we write a 1 for every place we used and a 0 for every place we skipped, we get $23 = 10111_2$. While this is a straightforward procedure for decimal to binary conversion, it can be cumbersome for larger numbers.

Exercise 1.6

An alternate way to convert a decimal numeral to an equivalent binary numeral is to divide the given number by 2 until the quotient is 0 (keeping only the integer part of the quotient). At each step, record the remainder found when dividing by 2. Collect these remainders (which will always be either 0 or 1) to form the binary equivalent. The least significant bit is the remainder obtained after the first division, and the most significant bit is the remainder obtained after the last division.

Example 1.9: Decimal to binary with remainders

Let's use this method to convert 23 to its binary equivalent. The following table shows the steps need

for the conversion. The leftmost column lists the step number. The second column contains the number to be divided by 2 at each step. The third column contains the quotient for each step, and the last column contains the current remainder.

Step	Number	Quotient	Remainder
1	23	11	1
2	11	5	1
3	5	2	1
4	2	1	0
5	1	0	1

We begin by dividing 23 by 2, yielding 11 as the quotient and 1 as the remainder. The quotient 11 is then divided by 2, yielding 5 as the quotient and 1 as the remainder. This process continues until we get a quotient of 0 and a remainder of 1 in Step 5. We now collect the remainders and get the same result as before, $23 = 10111_2$. ■

Other conversions

A decimal number can be converted to its hexadecimal equivalent by using either of the two procedures described above. Instead of writing a decimal number as a sum of powers of 2, one writes it as a sum of powers of 16. Similarly, when using the division method, instead of dividing by 2, one divides by 16. Octal conversion is similar.

Exercise 1.8

We use hexadecimal because it is straightforward to convert from it to binary or back. The following table lists binary equivalents for the 16 hexadecimal digits.

Hexadecimal digit	Binary equivalent	Hexadecimal digit	Binary equivalent
0	0000	8	1000
1	0001	9	1001
2	0010	A	1010
3	0011	B	1011
4	0100	C	1100
5	0101	D	1101
6	0110	E	1110
7	0111	F	1111

With the help of the table above, let's convert $3FA_{16}$ to binary. By simple substitution, $3FA_{16} = 0011\ 1111\ 1010_2$. Note that we have grouped the binary digits into clusters of 4 bits each. Of course, the leftmost zeroes in the binary equivalent are useless as they do not contribute to the value of the

1.3.4 Integer representation in a computer

In mathematics, binary numerals can represent arbitrarily big numbers. Inside of a computer, the size of a number is constrained by the number of bits used to represent it. For general purpose computation, 32- and 64-bit integers are the most commonly used. The largest integer that Java represents with 32 bits is 2,147,483,647, which is good enough for most tasks. For larger numbers, Java can represent up to 9,223,372,036,854,775,807 with 64 bits. Java also provides representations for integers using 8 and 16 bits.

These representations are easy to determine for positive numbers: Find the binary equivalent of the number and then pad the left side with zeroes to fill the remaining space. For example, $19 = 10011_2$. If stored using 8 bits, 19 would be represented as 0001 0011. If stored using 16 bits, 19 would be represented as 0000 0000 0001 0011. (We separate groups of 4 bits for easier reading.)

1.3.5 Binary arithmetic

Recall that computers deal with numbers in their binary representation, meaning that all arithmetic is done on binary numbers. Sometimes it is useful to understand how this process works and how it is similar and different from decimal arithmetic. The table below lists rules for binary addition.

+	0	1
0	0	1
1	1	10

As indicated above, the addition of two 1s leads to a 0 with a carry of 1 into the next position to the left. Addition for numbers composed of more than one bit use the same rules as any addition, carrying values that are too large into the next position. In decimal addition, values over 9 must be carried. In binary addition, values over 1 must be carried. The next example shows a sample binary addition. To simplify its presentation, we assume that integers are represented with 8 bits.

Example 1.10: Binary addition

Let's add the numbers 60 and 6 in binary. Using the conversion techniques described above, we can find that $60 = 111100_2$ and $6 = 110_2$. Inside the computer, these numbers would already be in binary and padded to fill 8 bits.

Binary	Decimal
0011 1100	60
+ 0000 0110	6
<hr/>	
0100 0010	66

The result is no surprise, but note that the addition can proceed in binary without conversion to decimal at any point. ■

Subtraction in binary is also similar to subtraction in decimal. The rules are given in the following table.

-	0	1
0	0	1
1	(1)1	0

When subtracting a 1 from a 0, a 1 is borrowed from the next left position. The next example illustrates binary subtraction.

Example 1.11: Binary subtraction

Again, we'll use 60 and 6 and their binary equivalents given above.

Binary	Decimal
0011 1100	60
- 0000 0110	6
0011 0110	54

■ Exercise 1.14

1.3.6 Negative integers in a computer

Negative integers are also represented in computer memory as binary numbers, using a system called *two's complement*. When looking at the binary representation of a signed integer in a computer, the leftmost (most significant) bit will be 1 if the number is negative and 0 if it is positive. Unfortunately, there's more to finding the representation of a negative number than flipping this bit.

Suppose that we need to find the binary equivalent of the decimal number -12 using 8-bits in two's complement form. The first step is to convert 12 to its 8-bit binary equivalent. Doing so we get $12 = 0000\ 1100$. Note that the leftmost bit of the representation is a 0, indicating that the number is positive. Next we take the two's complement of the 8-bit representation in two steps. In the first step, we flip every bit, i.e., change every 0 to 1 and every 1 to 0. This gives us the *one's complement* of the number, $1111\ 0011$. In the second step, we add 1 to the one's complement to get the two's complement. The result is $1111\ 0011 + 1 = 1111\ 0100$.

Thus, the 8-bit, two's complement binary equivalent of -12 is $1111\ 0100$. Note that the leftmost bit is a 1, indicating that this is a negative number.

Exercise 1.7

Example 1.12: Decimal to two's complement

Let us convert -29 to its binary equivalent assuming that the number is to be stored in 8-bit, two's complement form. First we convert positive 29 to its 8-bit binary equivalent, $29 = 0001\ 1101$.

Next we obtain the one's complement of the binary representation by flipping 0s to 1s and 1s to 0s. This gives us 1110 0010. Finally, we add 1 to the one's complement representation to get 1110 0010 + 1 = 1110 0011, which is the desired binary equivalent of -29. ■

Exercise 1.11

Example 1.13: Two's complement to decimal

Let us now convert the 8-bit, two's complement value 1000 1100 to decimal. We note that the leftmost bit of this number is 1, making it a negative number. Therefore, we reverse the process of making a two's complement. First, we subtract 1 from the representation, yielding $1000\ 1100 - 1 = 1000\ 1011$. Next, we flip all the bits in this one's complement form, yielding 0111 0100.

Now we convert this binary representation to its decimal equivalent, yielding 116. Thus, the decimal equivalent of 1000 1100 is -116. ■

Exercise 1.12

Why do we use two's complement? First of all, we needed a system that could represent both positive and negative numbers. We could have simply used the leftmost bit as a sign bit and represented the rest of the number as a positive binary number. Doing so would require a check on the bit and some conversion for negative numbers every time a computer wanted to perform an addition or subtraction.

Because of the way it's designed, positive and negative integers stored in two's complement can be added or subtracted **without** any special conversions. The leftmost bit is added or subtracted just like any other bit, and values that carry past the leftmost bit are ignored. Two's complement has an advantage over one's complement in that there is only one representation for zero. The next example shows two's complement in action.

Example 1.14: Two's complement arithmetic

We'll add -126 and 126. If you perform the needed conversions, their 8-bit, two's complement representations are 1000 0010 and 0111 1110.

Binary	Decimal
1000 0010	-126
+ 0111 1110	126
0000 0000	0

As expected, the sum is 0.

Now, let's add the two negative integers -126 and -2, whose 8-bit, two's complement representations are 1000 0010 and 1111 1110.

Binary	Decimal
1000 0010	-126
+ 1111 1110	-2
1000 0000	-128

The result is -128, which is the smallest negative integer that can be represented in 8-bit two's complement. ■

1.3.7 Overflow and underflow

When performing an arithmetic or other operation on numbers, an overflow is said to occur when the result of the operation is larger than the largest value that can be stored in that representation. An underflow is said to occur when the result of the operation is smaller than the smallest possible value.

Both overflows and underflows lead to wrapped around values. For example, adding two positive numbers together can result in a negative number or adding two negative numbers together can result in a positive number.

Example 1.15: Binary addition with overflow

Let's add the numbers 124 and 6. Their 8-bit, two's complement representations are 0111 1100 and 0000 0110.

Binary	Decimal
0111 1100	124
+ 0000 0110	6
1000 0010	-126

This surprising result happens because the largest 8-bit two's complement integer is 127. Adding 124 and 6 yields 130, a value larger than this maximum, resulting in overflow with a negative answer. ■

The smallest (most negative) number that can be represented in 8-bit two's complement is -128. A result smaller than this will result in underflow. For example, $-115 - 31 = 110$. Try out the conversions needed to test this result.

Exercise 1.13

1.3.8 Bitwise operators

Although we will most commonly manipulate numbers using traditional mathematical operations such as addition, subtraction, multiplication, and division, there are also operations that work directly on the binary representations of the numbers. Some of these operators are equivalent to mathematical operations, and some are not.

Operator Name	Description	
&	Bitwise AND	Combines two binary representations into a new representation which has 1s in every position that both the original representations have a 1
	Bitwise OR	Combines two binary representations into a new representation which has 1s in every position that either of the original representations have a 1
-	Bitwise XOR	Combines two binary representations into a new representation which has 1s in every position that the original representations have different values
-	Bitwise NOT	Takes a representation and creates a new representation in which every bit is

complement flipped from 0 to 1 and 1 to 0

<< Signed left Moves all the bits the specified number of positions to the left, leaving the shift sign bit unchanged

>> Signed right Moves all the bits the specified number of positions to the right, padding the shift left with copies of the sign bit

>>> Unsigned right shift Moves all the bits the specified number of positions to the right, padding with 0s

Bitwise AND, bitwise OR, and bitwise XOR take two integer representations and combine them to make a new representation. In bitwise AND, each bit in the result will be a 1 if **both** of the original integer representations in that position are 1 and 0 otherwise. In bitwise OR, each bit in the result will be a 1 if **either** of the original integer representations in that position are 1 and 0 otherwise. In bitwise XOR, each bit in the result will be a 1 if the two bits of the original integer representations in that position are not the same and 0 otherwise.

Bitwise complement is a unary operator like the negation operator (-). Instead of just changing the sign of a value (which it will also do), its result has every 1 in the original representation changed to 0 and every 0 to 1.

The signed left shift, signed right shift, and unsigned right shift operators all create a new binary representation by shifting the bits in the original representation a certain number of places to the left or the right. The signed left shift moves the bits to the left, padding with 0s, but does not change the sign bit. If you do a signed left shift by n positions, it is equivalent to multiplying the number by 2^n . The signed right shift moves the bits to the right, padding with whatever the sign bit is. If you do a signed right shift by n positions, it is equivalent to dividing the number by 2^n (with integer division). The unsigned right shift moves the bits to the right, including the sign bit, filling the left side with 0s. And unsigned right shift will always make a value positive but is otherwise similar to a signed right shift. A few examples follow.

Example 1.16: Bitwise operators

Here are a few examples of the result of bitwise operations. We will assume that the values are represented using 32-bit two's complement, instead of using 8-bit values as before. In Java, bitwise operators automatically convert smaller values to 32-bit representations before proceeding.

Let's consider the result of $21 \& 27$.

	Binary	Decimal
	0000 0000 0000 0000 0000 0000 0001 0101	21
&	0000 0000 0000 0000 0000 0000 0001 1011	27
	<hr/>	<hr/>
	0000 0000 0000 0000 0000 0000 0001 0001	17

Note how this result is different from 21|27.

	Binary	Decimal
	0000 0000 0000 0000 0000 0000 0001 0101	21
	0000 0000 0000 0000 0000 0000 0001 1011	27
	<hr/>	<hr/>
	0001 1111	31

And also from 21^27.

	Binary	Decimal
	0000 0000 0000 0000 0000 0000 0001 0101	21
^	0000 0000 0000 0000 0000 0000 0001 1011	27
	<hr/>	<hr/>
	0000 1110	14

Ignoring overflows, signed left shifting is equivalent to repeated multiplications by 2. Consider $11 \ll 3$. The representation 0000 0000 0000 0000 0000 0000 0001 1011 is shifted to the left to make 0000 0000 0000 0000 0000 0000 0101 1000 = 88 = 11 $\times 2^3$.

Signed right shifting is equivalent to repeated integer divisions by 2. Consider $-104 \gg 2$. The representation 1111 1111 1111 1111 1111 1111 1001 1000 is shifted to the right to make 1111 1111 1111 1111 1111 1111 1110 0110 = -26 = -104 $\div 2^2$.

Unsigned right shifting is the same as signed right shifting except when it is done on negative numbers. Since their sign bit is replaced by 0, an unsigned right shift produces a (generally large) positive number. Consider $-104 \ggg 2$. The representation 1111 1111 1111 1111 1111 1111 1001 1000 is shifted to the right to make 0011 1111 1111 1111 1111 1111 1110 0110 = 1,073,741,798.

Because of the way two's complement is designed, bitwise complement is equivalent to negating the sign of the number and then subtracting 1. Consider $\sim(-104)$. The representation 1111 1111 1111 1111 1111 1111 1001 1000 is complemented to 0000 0000 0000 0000 0000 0000 0110 0111 = 103.

1.3.9 Rational numbers

We have seen how to represent positive and negative integers in computer memory. In this section we see how rational numbers, such as 12.33, -149.89, and 3.14159, can be converted into binary and represented.

Scientific notation

Scientific notation is closely related to the way a computer represents a rational number in memory. Scientific notation is a tool for representing very large or very small numbers without writing a lot of

zeroes. A decimal number in scientific notation is written $a \times 10^b$ where a is called the *mantissa* and b is called the *exponent*.

For example, the number 3.14159 can be written in scientific notation as 0.314159×10^1 . In this case, 0.314159 is the mantissa, and 1 is the exponent. Here a few more examples of writing numbers in scientific notation.

$$3.14159 = 3.14159 \times 10^0$$

$$3.14159 = 314159 \times 10^{-5}$$

$$-141.324 = -0.141324 \times 10^3$$

$$30,000 = .3 \times 10^5$$

There are many ways of writing a number in scientific notation. A more standardized way of writing real numbers is *normalized scientific notation*. In this notation, the mantissa is always written as a number whose absolute value is less than 10 but greater than or equal to 1. Following are a few examples of decimal numbers in normalized scientific notation.

$$3.14159 = 3.14159 \times 10^0$$

$$-141.324 = -1.41324 \times 10^3$$

$$30,000 = 3.0 \times 10^4$$

A shorthand for scientific notation is E notation, which is written with the mantissa followed by the letter ‘E’ followed by the exponent. For example, 39.2 in E notation can be written 3.92E1 or 0.392E2. The letter ‘E’ should be read “multiplied by 10 to the power.” E notation can be used to represent numbers in scientific notation in Java. Instead of writing the number 345000000 in Java code, 345E6 or .345E9 could be used instead.

Fractions

A rational number can be broken into an integer part and a fractional part. In the number 3.14, 3 is the integer part, and .14 is the fractional part. We have already seen how to convert the integer part to binary. Now we will see how to convert the fractional part into binary. We can then combine the binary equivalents of the integer and fractional parts to find the binary equivalent of a decimal real number.

A decimal fraction f is converted to its binary equivalent by successively multiplying it by 2. At the end of each multiplication step, either a 0 or a 1 is obtained as an integer part and is recorded separately. The remaining fraction is again multiplied by 2 and the resulting integer part recorded. This process continues until the fraction reduces to zero or enough binary digits for the desired precision have been found. The binary equivalent of f then consists of the bits in the order they have been recorded, as shown in the next example.

Example 1.17: Fraction conversion to binary

Let's convert 0.8125 to binary. The table below shows the steps to do so.

Step	f	$f \times 2$	Integer part	Remainder
1	0.8125	1.625	1	0.625
2	0.625	1.25	1	0.25
3	0.25	0.5	0	0.5
4	0.5	1.0	1	0

We then collect all the integer parts and get 0.1101 as the binary equivalent of 0.8125. We can convert this binary fraction back into decimal to verify that it is correct.

$$0.1101 = 1 \times 2^{-1} + 1 \times 2^{-2} + 0 \times 2^{-3} + 1 \times 2^{-4} = 0.5 + 0.25 + 0 + 0.0625 = 0.8125$$

In some cases, the process described above will never have a remainder of 0. In such cases we can only find an approximate representation of the given fraction as demonstrated in the next example.

Example 1.18: Non-terminating fraction

Let us convert 0.3 to binary assuming that we have only five bits in which to represent the fraction. The following table shows the five steps in the conversion process.

Step	f	$f \times 2$	Integer part	Remainder
1	0.3	0.6	0	0.6
2	0.6	1.2	1	0.2
3	0.2	0.4	0	0.4
4	0.4	0.8	0	0.8
5	0.8	1.6	1	0.6

Collecting the integer parts we get 0.01001 as the binary representation of 0.3. Let's convert this back to decimal to see how accurate it is.

$$0.01001 = 0 \times 2^{-1} + 1 \times 2^{-2} + 0 \times 2^{-3} + 0 \times 2^{-4} + 1 \times 2^{-5} = 0.25 + 0.03125 = 0.28125$$

Five bits are not enough to represent 0.3 fully. In this case, we have an error of $0.3 - 0.28125 = 0.01875$. Most computers use many more bits to represent fractions and obtain much better accuracy in their representation. ■

Exercise 1.15

Exercise 1.16

Now that we understand how integers as well as fractions can be converted from one number base to another, we can convert any rational number from one base to another. The next example demonstrates one such conversion.

Example 1.19: Rational number converted to binary

Let's convert 14.3 to binary assuming that we will only use six bits to represent the fractional part. First we convert 14 to binary using the technique described earlier. This gives us $14 = 1110_2$. Taking the method outlined in [Example 1.18](#) one step further, our six bit representation of 0.3 is 0.010011. Combining the two representations gives $14.3_{10} = 1110.010011_2$. ■

Floating point notation

Floating point notation is a system used to represent rational numbers in computer memory. In this notation a number is represented as $a \times b^e$, where a gives the *significant digits* (mantissa) of the number and e is the exponent. The system is very similar to scientific notation, but computers usually have base $b = 2$ instead of 10.

For example, we could write the binary number 1010.1 in floating point notation as 10.101×2^2 or as 101.01×2^1 . In any case, this number is equivalent to 10.5 in decimal.

In standardized floating point notation, a is written so that only the most significant non-zero digit is to the left of the decimal point. Most computers use the IEEE 754 floating point notation to represent rational numbers. In this notation, the memory to store the number is divided into three segments: one bit used to mark the sign of the number, m bits to represent the mantissa (also known as the *significand*), and e bits to represent the exponent.

In IEEE floating point notation, numbers are commonly represented using 32 bits (known as *single precision*) or using 64 bits (known as *double precision*). In single precision, $m = 23$ and $e = 8$. In double precision, $m = 52$ and $e = 11$. To represent positive and negative exponents, the exponent has a *bias* added to it so that the result is never negative. This bias is 127 for single precision and 1,023 for double precision. The packing of the sign bit, the exponent, and the mantissa is shown in [Figure 1.5\(a\)](#) and (b).

Example 1.20: Single precision IEEE format

The following is a step-by-step demonstration of how to construct the single precision binary representation in IEEE format of the number 10.5.

1. Convert 10.5 to its binary equivalent using methods described earlier, yielding $10.5_{10} = 1010.1_2$. Unlike the case of integers, the sign of the number is taken care of separately for floating point. Thus, we would use 1010.1_2 for -10.5 as well.
2. Write this binary number in standardized floating point notation, yielding 1.0101×2^3 .
3. Remove the leading bit (always a 1 for non-zero numbers), leaving .0101.
4. Pad the fraction with zeroes on the right to fill the 23-bit mantissa, yielding 0101 0000 0000 0000 0000 000. Note that the decimal point is ignored in this step.
5. Add 127 to the exponent. This gives us an exponent of $3 + 127 = 130$.
6. Convert the exponent to its 8-bit unsigned binary equivalent. Doing so gives us $130_{10} =$

10000011₂.

7. Set the sign bit to 0 if the number is positive and to 1 otherwise. Since 10.5 is positive, we set the sign bit to 0.

We now have the three components of 10.5 in binary. The memory representation of 10.5 is shown in [Figure 1.5](#). Note in the figure how the sign bit, the exponent, and the mantissa are packed into 32 bits. ■

Exercise 1.17

Largest and smallest numbers

Fixing the number of bits used for representing a real number limits the numbers that can be represented in computer memory using the floating point notation. The largest rational number that can be represented in single precision has an exponent of 127 (254 after bias) with a mantissa consisting of all 1s:

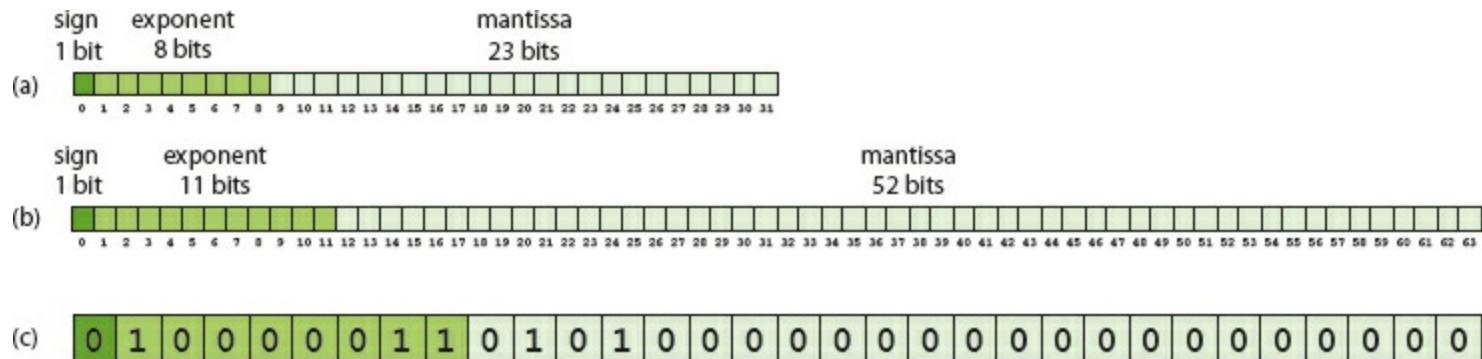


Figure 1.5: Layouts for floating point representation (a) in single precision, (b) in double precision, and (c) of 10.5_{10} in single precision.

0 1 1 1 1 1 1 1 0 1

This number is approximately 3.402×10^{38} . To represent the smallest (closest to zero) non-zero number, we need to examine one more complication in the IEEE format. An exponent of 0 implies that the number is unnormalized. In this case, we no longer assume that there is a 1 bit to the left of the mantissa. Thus, the smallest non-zero single precision number has its exponent set to 0 and its mantissa set to all zeros with a 1 in its 23rd bit:

0 0000 0000 0000 0000 0000 0000 0000 001

Unnormalized single precision values are considered to have an exponent of -126. Thus, the value of this number is $2^{-23} \times 2^{-126} = 2^{-149} \approx 1.4 \times 10^{-45}$. Now that we know the rules for storing both integers and floating point numbers, we can list the largest and smallest values possible in 32- and 64-bit representations in Java as shown in the following table. Note that **largest** means the largest positive number for both integers and floating point values, but **smallest** means the most negative

number for integers and the smallest positive non-zero value for floating point values.

Format	Largest number	Smallest number
32-bit integer	2,147,483,647	-2,147,483,648
64-bit integer	9,223,372,036,854,775,807	-9,223,372,036,854,775,808
32-bit floating point	3.4028235×10^{38}	1.4×10^{-45}
64-bit floating point	$1.7976931348623157 \times 10^{308}$	4.9^{-324}

Using the same number of bits, floating point representation can store much larger numbers than integer representation. However, floating point numbers are not always exact, resulting in approximate results when performing arithmetic. Always use integer formats when fractional parts are not needed.

Special numbers

Several binary representations in the floating point notation correspond to special numbers. These numbers are set aside and not used for results in normal computation.

0.0 and -0.0: When the exponent as well as the mantissa is 0, the number is interpreted as a 0.0 or -0.0 depending on the sign bit. For example, in a Java program, dividing 0.0 by -1.0 results in -0.0. Similarly, -0.0 divided by -1.0 is 0.0. Positive and negative zeroes only exist for floating point values. -0 is the same as 0 for integers. Dividing the integer 0 by -1 in Java results in 0 and not in -0.

Positive and negative infinity: An overflow or an underflow might occur while performing arithmetic on floating point values. In the case of an overflow, the resulting number is the special value that Java recognizes as infinity. In the case of an underflow, it is a special negative infinity value. For example, dividing 1.0 by 0.0 in Java results in infinity and dividing -1.0 by 0.0 results in negative infinity. These values have well defined behavior. For example, adding 1.0 to infinity yields infinity.

Note that floating point values and integers do not behave in the same way. Dividing the integer 1 by the integer 0 creates an error that can crash a Java program.

Not-a-number (NaN): Some mathematical operations may result in an undefined number. For example, $\sqrt{-2}$ is an imaginary number. Java has a value set aside for results that are not rational numbers. When we discuss how to find the square root of a value in Java, this not-a-number value will be the answer for the square root of a negative number.

Errors in floating point arithmetic

As we have seen, many rational numbers can only be approximately represented in computer memory. Thus, arithmetic done on the approximate values yields approximate answers. For example, 1.3 cannot be represented exactly using a 32-bit value. In this case, the product 1.3×3.0 will be 3.8999999 instead of 3.9. This error will propagate as additional operations are performed on

previous results. The next example illustrates this propagation of errors when a sequence of floating point operations are performed.

Example 1.21: Error propagation

Suppose that the price of several products is to be added to determine the total price of a purchase at a cash register that uses floating point arithmetic. For simplicity, let's assume that all items have a price of \$1.99. We don't know how many items will be purchased ahead of time and simply add the price of each item until all items have been scanned at the register. The table below shows the value of the total cost for different number of items purchased.

Items	Correct Cost	Calculated Cost	Absolute Error	Relative Error
100	199.0	1.9900015E02	1.5258789E-04	7.6677333E-07
500	995.0	9.9499670E02	3.2958984E-03	3.3124606E-06
1000	1990.0	1.9899918E03	8.1787109E-03	4.1099051E-06
10000	19900.0	1.9901842E04	1.8417969E00	9.2552604E-05

The first column in the table above is the number of items. The second column is the correct cost of all items purchased. The third column is the cost calculated by adding each item using single precision floating point addition. The fourth and fifth columns give the absolute and relative errors, respectively, of the calculated value. Note how the error increases as the number of additions goes up. In the last row, the absolute error is almost two dollars. ■

While the above example may seem unrealistic, it does expose the inherent dangers of floating point calculations. Although the errors are much less when using double precision representations, they still exist.

1.4 Solution: Buying a computer

We pose a motivating problem in the **Problem** section near the beginning of most chapters. Whenever there is a **Problem** section, there is a **Solution** section near the end in which we give a solution to the problem given earlier.

After all the discussion of the hardware, software, and data representation inside of a computer, you might feel more confused about which computer to buy than before. As a programmer, it is important to understand how data is represented, but this information plays virtually no role in deciding which computer to buy. Unlike most problems in this book, there is no concrete answer we can give here. Because the development of technology progresses so rapidly, any advice about computer hardware or software has a short shelf-life.

Software is a huge consideration, beginning with the OS. Because the choice of OS usually affect choice of hardware, we'll start there. The three major choices for a desktop or laptop OS are Microsoft Windows, Mac OS X, and Linux.

Windows is the most commonly used and is also heavily marketed for business use. Windows

suffered from many stability and security issues, but Microsoft has worked hard to address these. Mac OS (and the computers it is installed on) are marketed to an artistic and counter-culture population. Linux is popular among tech savvy users. Putting marketing biases aside, the three operating systems have become more similar to each other over time, and most people could be productive using any of the three. The following table lists some pros and cons for each OS.

OS	Pros	Cons
Microsoft Windows	<ul style="list-style-type: none">• Compatible with the largest number of programs• Can be purchased separately from hardware• Can run on Mac hardware	<ul style="list-style-type: none">• Expensive• Security concerns
Mac OS X	<ul style="list-style-type: none">• Polished user interface• Bundled with many useful programs• Tested for use on the hardware it comes with	<ul style="list-style-type: none">• Most expensive• Many business applications and games are released late or not at all for OS X• Difficult to run on non-Mac hardware
Linux	<ul style="list-style-type: none">• Free• Runs on almost any hardware• Highly customizable• Serviced by a community that develops many free applications for it	<ul style="list-style-type: none">• Can be difficult to install or configure• Few commercial applications are available for it• Limited customer support

Once you have decided on an OS, you can pick hardware and other software that is compatible with it. For Mac OS X, most of your hardware choices will be computers sold by Apple. For Windows and Linux, you can either have a computer built for you or build your own. Although computer hardware changes quickly, let's examine some general guidelines.

CPU: Remember that the speed of a CPU is measured in GHz (billions of clock cycles per second). Higher GHz is generally better, but it's hard to compare performance across different designs of CPU. There is also a diminishing returns effect: The very fastest, very newest CPUs are often considerably more expensive even if they only provide slightly better performance. It's usually more cost effective to select a CPU in the middle of the performance spectrum.

Cache size also has a huge effect on performance. The larger the cache, the less often the CPU has to read data from much slower memory. Since most new CPUs available today are 64-bit, the question of word size is not significant.

Although some specialists may prefer one or the other, both Intel and AMD make powerful, competitive consumer CPUs.

Memory: Memory includes RAM, hard drives, optical drives, and any other storage. RAM is usually easy to upgrade for desktop machines and less easy (though often possible) for laptops. The price of RAM per gigabyte goes down over time. It may be reasonable to start with a modest amount of RAM and then upgrade after a year or two when it becomes cheaper to do so. It takes a little bit of research to get exactly the right kind of RAM for your CPU and motherboard. The amount of RAM is dependent on what you want to do with your system. The minimum amount of RAM to run Microsoft Windows 7 is 1 GB for 32-bit versions and 2 GB for 64-bit versions. The minimum amount of RAM to run Apple Mac OS X 10.7 “Lion” is 2 GB. One rule of thumb is to have at least twice the minimum required RAM.

Hard drive storage is heavily dependent on how you expect to use your computer. 500 GB and 1 TB drives are not very expensive, and this represents a huge amount of storage. Only if you plan to have enormous libraries of video or uncompressed audio data will you likely need more. Corporate level databases and web servers and some other business systems can also require huge amounts of space. Hard drive speed is greatly affected by the hard drive’s cache size. As always, a bigger cache means better performance. Using a solid state drive (SSD) instead of a traditional hard drive has much better performance but higher cost.

Installing optical drives and other storage devices depends on individual needs. Note that a DVD±RW drive is an inexpensive solution for backing up data or reinstalling an operating system.

I/O Devices: The subject of I/O devices is very personal. It is difficult to say what anyone should buy without considering his or her specific needs. A monitor is the essential visual output device while a keyboard and mouse are the essential input devices. Speakers are very important as well. Most laptops have all of these things integrated in some form or another.

Someone interested in video games might want to invest in a powerful graphics card. Newer cards with more video RAM are generally better than older cards with less, but which card is best at a given price point is the subject of continual discussion at sites like AnandTech (<http://www.anandtech.com/>) and Tom’s Hardware (<http://www.tomshardware.com/>).

Printers are still useful output devices. Graphics tablets can make it easier to create digital art on a computer. The number of potentially worthwhile I/O devices is limitless.

This section is just a jumping off point for purchasing a computer. As you learn more about computer hardware and software, it will become easier to know what combination of the two will serve your needs. Of course, there is always more to know, and technology changes quickly.

1.5 Concurrency: Multicore processors

In the last decade, the word “core” has been splattered all over CPU packaging. Intel in particular has marketed the idea heavily with its Core, Core 2, i3 Core, i5 Core, and i7 Core chips. What are all these cores?

Looking back into the past, most consumer processors had a single *core*, or brain. They could only execute one instruction at a time. (Even this definition is a little hazy, because pipelining kept more than one instruction in the process of being executed, but overall execution proceeded sequentially.)

The advent of multicore processors has changed this design significantly. Each processor has several independent cores, each of which can execute different instructions at the same time. Before the arrival of multicore processors, a few desktop computers and many supercomputers had multiple separate processors that could achieve a similar effect. However, since multicore processors have more than one effective processor on the same silicon die, the communication time between processors is much faster and the overall cost of a multiprocessor system is cheaper.

1.5.1 The Good

Multicore systems have impressive performance. The first multicore processors had two cores, but current designs have four, six, or eight, and much greater numbers are expected. A processor with eight cores can execute eight different programs at the same time. Or, when faced with a computationally intense problem like matrix math, code breaking, or scientific simulation, a processor with eight cores could solve the problem eight times as fast. A desktop processor with 100 cores that can solve a problem 100 times faster is not out of reach.

In fact, modern graphics cards are already blazing this trail. Consider the 1080p standard for high definition video, which has a resolution of $1,920 \times 1,080$ pixels. Each pixel (short for picture element) is a dot on the screen. A screen whose resolution is 1080p has 2,073,600 dots. To maintain the illusion of smooth movement, these dots should be updated around 30 times per second. Computing the color for more than 2 million dots based on 3D geometry, lighting, and physics effects 30 times a second is no easy feat. Some of the cards used to render computer games have hundreds or thousands of cores. These cores are not general purpose or completely independent. Instead, they’re specialized to do certain kinds of matrix transformations and floating point computations.

Exercise 1.20

1.5.2 The Bad

Although chip-makers have spent a lot of money marketing multicore technology, they have not spent much money explaining that one of the driving forces behind the “multicore revolution” is a simple failure to make processors faster in other ways. In 1965, Gordon Moore, one of the founders of Intel remarked that the density of silicon microprocessors had been doubling every year (though he later revised this to every two years), meaning that twice as many transistors (computational building blocks) could fit in the same physical space. This trend, often called Moore’s Law, has held up reasonably well. For years, clever designs relying on shorter communication times, pipelining, and other schemes succeeded in doubling the effective performance of processors every two years.

At some point, the tricks became less effective and exponential gains in processor clock rate could no longer be maintained. As clock frequency increases, the signal becomes more chaotic, and it becomes more difficult to tell the difference between the voltages that represent 0s and 1s. Another problem is heat. The energy that a processor uses is related to the **square** of the clock rate. This relationship means that increasing the clock rate of a processor by a factor of 4 will increase its energy consumption (and heat generation) by a factor of 16.

The legacy of Moore's Law lives on. We are still able to fit more and more transistors into tinier and tinier spaces. After decades of increasing clock rate, chip-makers began using the additional silicon density to make processors with more than one core instead. Since 2005 or so, increases in clock rate have stagnated.

1.5.3 The Ugly

Does a processor with eight cores solve problems eight times as fast as its single core equivalent? Unfortunately, the answer is, "Almost never." Most problems are not easy to break into eight independent pieces.

For example, if you want to build eight houses and you have eight construction teams, then you probably can get pretty close to completing all eight houses in the time it would have taken for one team to build a single house. But what if you have eight teams and only one house to build? You might be able to finish the house a little early, but some steps necessarily come after others: The concrete foundation must be poured and solid before framing can begin. Framing must be finished before the roof can be put on. And so on.

Like building a house, most problems you can solve on a computer are difficult to break into concurrent tasks. A few problems are like painting a house and can be completed much faster with lots of concurrent workers. Other tasks simply cannot be done faster with more than one team on the job. Worse, some jobs can actually interfere with each other. If a team is trying to frame the walls while another team is trying to put the roof onto unfinished walls, neither will succeed, the house might be ruined, and people could get hurt.

On a desktop computer, individual cores generally have their own level 1 cache but share level 2 cache and RAM. If the programmer isn't careful, he or she can give instructions to the cores that will make them fight with each other, overwriting the memory that other cores are using and potentially crashing the program or giving an incorrect answer. Imagine if different parts of your brain were completely independent and fought with one another. The words that came out of your mouth might be random and chaotic and make no sense to your listener.

To recap, the first problem with concurrent programming is finding ways to break down problems so that they can be solved faster with multiple cores. The second problem is making sure that the different cores cooperate so that the answer is correct and makes sense. These are not easy problems, and many researchers are still working on finding better ways to do both.

Some educators believe that beginners will be confused by concurrency and should wait until later courses to confront these problems. We disagree: Forewarned is forearmed. Concurrency is an integral part of modern computation, and the earlier you get introduced to it, the more familiar it will be.

1.6 Summary

This introductory chapter focused on the fundamentals of a computer. We began with a description of computer hardware, including the CPU, memory, and I/O devices. We also described the software of a computer, highlighting key programs such as the operating system and compilers as well as other useful programs like business applications, video games, and web browsers.

Then, we introduced the topic of how numbers are represented inside the computer. Various number systems and conversion from one system to another were explained. We discussed how floating point notation is used to represent rational numbers. A sound knowledge of data representation helps a programmer decide what kind of data to use (integer or floating point and how much precision) as well as what kind of errors to expect (overflow, underflow, and floating point precision errors).

The next chapter extends the idea of data representation into the specific types of data that Java uses and introduces representation systems for individual characters and text.

Exercises

Conceptual Problems

- 1.1 Name a few programming languages other than Java.
- 1.2 What is the difference between machine code and bytecode?
- 1.3 What are some advantages of JIT compilation over traditional, ahead-of-time compilation?
- 1.4 Without converting to decimal, how can one find out whether a given binary number is odd or even?
- 1.5 Convert the following positive binary numbers into decimal.
 - a. 100_2
 - b. 111_2
 - c. 100000_2
 - d. 111101_2
 - e. 10101_2
- 1.6 Convert the following positive decimal numbers into binary.
 - a. 1
 - b. 15
 - c. 100
 - d. 1025
 - e. 567,899
- 1.7 What is the process for converting the representation of a binary integer given in one's complement into two's complement?

Perform the conversion from one's complement to two's complement on the representation 1011 0111, which uses 8 bits for storage.
- 1.8 Convert the following decimal numbers to their hexadecimal and octal equivalents.
 - a. 29
 - b. 100
 - c. 255
 - d. 382
 - e. 4,096
- 1.9 Create a table similar to the one on page 16 that lists the binary equivalents of octal digits.

Hint: Each octal digit can be represented as a sequence of three binary digits.

Use this table to convert the following octal numbers to binary.

- a. 337_8
- b. 24_8
- c. 777_8

1.10 The ternary number system has a base of 3 and uses symbols 0, 1, and 2 to construct numbers.

Convert the following decimal numbers to their ternary equivalents.

- a. 23
- b. 333
- c. 729

1.11 Convert the following decimal numbers to 8-bit, two's complement binary representations.

- a. -15
- b. -101
- c. -120

1.12 Given the following 8-bit binary representations in two's complement, find their decimal equivalents.

- a. 1100 0000
- b. 1111 1111
- c. 1000 0001

1.13 Perform the following arithmetic operation on the following 8-bit, two's complement binary representations of integers. Check your answers by performing arithmetic on equivalent decimal numbers.

- a. $0000\ 0011 + 0111\ 1110 =$
- b. $1000\ 1110 + 0000\ 1111 =$
- c. $1111\ 1111 + 1000\ 0000 =$
- d. $0000\ 1111 - 0001\ 1110 =$
- e. $1000\ 0001 - 1111\ 1100 =$

1.14 Extrapolate the rules for decimal and binary addition to rules for the hexadecimal system. Then, use these rules to perform the following additions in hexadecimal. Check your answers by converting the values and their sums to decimal.

- a. $A2F_{16} + BB_{16} =$
- b. $32C_{16} + D11F_{16} =$

- 1.15 Expand [Example 1.18](#) assuming that you have ten bits to represent the fraction. Convert the representation back to base 10. How far off is this value from 0.3?
- 1.16 Will the process in [Example 1.18](#) ever terminate assuming that we can use as many bits as needed to represent 0.3 in binary?
- 1.17 Derive the binary representation of the following decimal numbers assuming 32-bit (single) precision representation using the IEEE floating point format.
- 0.0125
 - 7.7
 - 10.3
- 1.18 The IEEE 754 standard also defines a 16-bit (half) precision format. In this format, there is one sign bit, five bits for the exponent, and ten bits for the mantissa. This format is the same as single and double precision in that it assumes that a bit with a value of 1 precedes the ten bits in the mantissa. It also uses a bias of 15 for the exponent. What is the largest decimal number that can be stored in this format?
- 1.19 Let a, b and c denote three real numbers. With real numbers, each of the equations below is true. Now suppose that all arithmetic operations are performed using floating point representations of these numbers. Indicate which of the following expressions are still always true and which are sometimes false.
- $(a + b) + c = a + (b + c)$
 - $a + b = b + a$
 - $a \times b = b \times a$
 - $a + 0 = a$
 - $(a \times b) \times c = a \times (b \times c)$
 - $a \times (b + c) = (a \times b) + (a \times c)$
- 1.20 What is a multicore microprocessor? Why do you think a multicore chip might be better than a single core chip? Search on the Internet to find the names of a few common multicore chips. Which chip does your computer use?

Concurrency

Chapter 2

Problem Solving and Programming

If you can't solve a problem, then there is an easier problem you can solve: find it.

—George Pólya

2.1 Problem: How to solve problems

How do we solve problems in general? This question is the motivating problem (or meta-problem, even) for this chapter. In fact, this question is the motivating problem for this book. We want to understand the process of solving problems with computers.

As we mentioned in the previous chapter, many *computer programs* such as business applications and web browsers have already been created to help people solve problems, but we want to solve new problems by writing our own programs. The art of writing these programs is called *computer programming* or just programming.

Many people reading this book will be computer science students, and that's great. However, computers have found their way into every segment of commercial enterprise and personal life. Consequently, programming has become a general-purpose skill that can aid almost anyone in their career, whether it is in transportation, medicine, the military, commerce, or innumerable other areas.

2.1.1 What is a program?

If you don't have a lot of experience with computer science, writing a computer program may seem daunting. The programs that run on your computer are complex and varied. Where would you start if you wanted to create something like Microsoft Word or Adobe Photoshop?

A computer program is a sequence of instructions that a computer follows. Even the most complex program is just a list of instructions. The list can get very long, and sometimes the computer will jump around the list when it makes decisions about what to do next.

Some people talk about how smart computers are becoming. A computer is neither smart nor stupid because a computer has no intelligence to measure. A computer is a machine like a sewing machine or an internal combustion engine. It follows the instructions we give it blindly. It doesn't have feelings or opinions about the instructions it receives. Computers do exactly what we tell them to and rarely make mistakes.

Once in a while, a computer will make a mistake due to faulty construction, a bad power source, or cosmic rays, but well over 99.999% of the things that go wrong with computers are because some human somewhere gave a bad instruction. This point is one of the most challenging aspects of programming a computer. How do you organize your thoughts so that you express to the computer exactly what you want it to do? If you give a person directions to a drug store, you might say, "Walk east for two blocks and then go into the third door on the right." The human will fill in all the necessary details: Stopping for traffic, watching out for construction, and so on. Given a robot body, a

computer would do exactly what you say. The instructions never mentioned **opening** the door, and so a computer might walk right through the closed door, shattering glass in the process.

2.1.2 What is a programming language?

What's the right level of detail for instructions for a computer? It depends on the computer and the application. But how do we give these instructions? Programs are composed of instructions written in a *programming language*. In this book, we will use the Java programming language.

Why can't the instructions be given in English or some other natural language? If the previous example of a robot walking through a glass door didn't convince you, consider the quote from Groucho Marx, "One morning I shot an elephant in my pajamas. How he got into my pajamas, I'll never know."

Natural languages are filled with idioms, metaphors, puns, and other ambiguities. Good writers use these to improve their poetry and prose, but our goal in this book is not to write poetry or prose. We want to write instructions for a computer that are crystal clear.

Learning Java is not like learning Spanish or Swahili. Like most programming languages, Java is highly structured. It has fewer than 100 reserved words and special symbols. There are no exceptions to its grammatical rules. Don't confuse the process of designing a solution to a problem with the process of implementing that solution in a programming language. Learning to organize your thoughts into a sequential list of instructions is different from learning how to translate that list into Java or another programming language, but there's a tight connection between the two.

Learning to program is patterning your mind to think like a machine, to break everything down into the simplest logical steps. At first, Java code will look like gobbledegook. Eventually, it will become so familiar that a glance will tell you volumes about how a program works. Learning to program is not easy, but the kind of logical analysis involved is valuable even if you never program afterward. If you do need to learn other programming languages in the future, it will be easy once you've mastered Java.

2.1.3 An example problem

This chapter takes a broad approach to solving problems with a computer, but we need an example to make some of the steps concrete. We will use the following problem from physics as an example throughout this chapter.

A rubber ball is dropped on a flat, hard surface from height h . What is the maximum height the ball will reach after the k^{th} bounce? We will discuss the steps needed to create a program to solve this problem in the next section.

2.2 Concepts: Developing software

2.2.1 Software development lifecycle

The engineers who built the first digital computers also wrote the programs for them. In those days, programming was closely tied to the hardware, and the programs were not very long. As the

capabilities of computers have increased and programming languages have evolved, computer programs have grown more and more complicated. Hundreds of developers, testers, and managers are needed to write a set of programs as complex as Microsoft Windows 7.

Organizing the creation of such complicated programs is challenging, but the industry uses a process called the *software development lifecycle* to help. This process makes large projects possible, but we can apply it to the simpler programs we will write in this book as well. There are many variations on the process, but we will use a straightforward version with the following five steps:

1. **Understand the problem:** It seems obvious, but, when you go to write a program, all kinds of details crop up. Consider a program that stores medical records. Should the program give an error if a patient's age is over 150 years? What if advances in long life or cryogenic storage make such a thing possible? What about negative ages? An unborn child could be considered to have a negative age (or more outlandishly, someone who had traveled back into the past using a time machine).

Even small details like these must be carefully considered to understand a problem fully. In industry, understanding the problem is often tied to a *requirements document*, in which a client lays out the features and functionality that the final program should have. The “client” could also be a manager or executive officer of the company you work for who wants your development team to create a certain kind of program. Sometimes the client does not have a strong technical background and creates requirements that are difficult to fulfill or vaguely specified. The creation of the requirements document can be a negotiation process in which the software development team and their client decide together what features are desirable and reasonable.

If you are taking a programming class, you can think of your instructor as your client. Then, you can view a programming assignment as a requirements document and your grade as your payment awarded based on how well the requirements were fulfilled.

2. **Design a solution:** Once you have a good grasp on the problem, you can begin to design a solution. For large scale projects, this may include decisions about the kinds of hardware and software packages that will be needed to solve the problem. For this book, we will only talk about problems that can be solved on standard desktop or laptop computers with Java installed.

We will be interested only in the steps that the computer will have to take to solve the problem. A finite list of steps taken to solve a problem is called an *algorithm*. If you have ever done long division (or any other kind of arithmetic by hand), you have executed an algorithm. The steps for long division will work for any real numbers, and most human beings can follow the algorithm without difficulty.

An algorithm is often expressed in *pseudocode*, a high level, generic, code-like system of notation. Pseudocode is similar to a programming language, but it doesn't have the same detail. Algorithms are often designed in pseudocode so that they aren't tied to any one programming language.

When attacking a problem, it is typical to break it into several smaller subproblems and then create algorithms for the subproblems. This approach is called *top-down* programming.

3. Implement the solution: Once you have your solution planned, you can *implement* it in a programming language. This step entails taking all the pseudocode, diagrams, and any other plans for your solution and translating them into code in a programming language. We will always use Java for our language in this book, but real developers use whichever language they feel is appropriate.

If you have designed your solution with several parts, you can implement each one separately and then integrate the solutions together. Professional developers often assign different parts of the solution to different programmers or even different teams of programmers.

Students are often tempted to jump into the implementation step, but never forget that this is the third step of the process. If you don't fully understand the problem and have a plan to attack it, the implementation process can become bogged down and riddled with mistakes. At first, the problems we introduce and the programs needed to solve them will be simple. As you move into more complicated problems in this book and in your career as a programmer, a good understanding of the problem and a good plan for a solution will become more and more important.

4. Test the solution: Expressing your algorithm in a programming language is difficult. If your algorithm was wrong, your program will not always give the right answer. If your algorithm was right, but you made a mistake implementing it in code, your program will still be wrong. Programming is a very detail-oriented activity. Even experienced developers make mistakes constantly.

Good design practices help, but all code must be thoroughly tested after it has been implemented. It should be tested exhaustively with expected and unexpected input. Tiny mistakes in software called *bugs* can lie hidden for months or even years before they are discovered. Sometimes a software bug is a source of annoyance to the user, but other times, as in aviation, automotive, or medical software, people die because of bugs.

Most of this book is dedicated to designing solutions to problems and implementing them in Java, but [Chapter 16](#) is all about testing and debugging.

5. Maintenance: Imagine that you have gone through the previous four steps: You understood all the details of a problem, planned a solution to it, implemented that solution in a programming language, and tested it until it was perfect. What happens next?

Presumably your program was shipped to your customers and they happily use it. But what if a bug is discovered that slipped past your testing? What if new hardware comes out that is not compatible with your program? What if your customers demand that you change one little feature?

Particularly with complex programs that have a large number of consumers, a software development company must spend time on customer support. Responsible software developers are expected to fix bugs, close security vulnerabilities, and polish rough features. This process is called *maintenance*. Developers are often working on the next version of the product, which could be considered maintenance or a new project entirely.

Although we cannot stress the importance of the first four steps of the software development

lifecycle enough, maintenance is not something we talk about in depth.

The software development lifecycle we presented above is a good guide, but it does not go into details. Different projects require different amounts of time and energy for each step. It is also useful to focus on the steps because it is less expensive to fix a problem at an earlier stage in development. It is impossible to set the exact numbers, but some developers assume that it takes ten times as much effort to fix a problem at the current step than it would at the previous step.

Example 2.1: Rising costs of fixing an error

Imagine that your company works on computer-aided design (CAD) software. The requirements document for a new program lists the formula for the area of a triangle as $base \times height$ when the real formula is $\frac{1}{2} base \times height$. If that mistake were caught while understanding the problem, it would mean changing one line of text. Once the solution to the problem has been designed, there may be more references to the incorrect formula. Once the solution has been implemented, those references will have turned into code that is scattered throughout the program. If the project were poorly designed, several different pieces of code might independently calculate the area of a triangle incorrectly. Once the implementation has been tested, a change to the code will mean that everything has to be tested from the very beginning, since fixing one bug can cause other bugs to surface. Finally, once the maintenance stage has been reached, the requirements, plan, implementation, and testing would all need to be updated to fix the bug. Moreover, customers would already have the faulty program. Your company would have to create a patch to fix the bug and distribute it over the Internet or by mailing out CD-ROMs.

Most bugs are more complicated and harder to fix, but even this simple one causes greater and greater repercussions as it goes uncaught. A factor of ten for each level implies that it takes 10,000 times more effort to fix it in the maintenance phase than at the first phase. Since fixing it at the first phase would have required a few keystrokes and fixing it in the last phase would require additional development and testing with web servers distributing patches and e-mails and traditional letters apologizing for the mistake, a factor of 10,000 could be a reasonable estimate. ■

Now that we have a sense of the software development lifecycle, let's look at an example using the sample ball bouncing problem to walk through a few steps.

Example 2.2: Ball bouncing problem and plan

Recall the statement of the problem from the **Problem** section:

A rubber ball is dropped on a flat, hard surface from height h . What is the maximum height the ball will reach after the k^{th} bounce?

1. **Understand the problem:** This problem requires an understanding of some physics principles. When a ball is dropped, the height of its first bounce depends on a factor known as the *coefficient of restitution*.

If c is the coefficient of restitution, then the ball will bounce back the first time to a height of $h \times c$. Then, we can act as if the ball were being dropped from this new height when calculating the next bounce. Thus, it will bounce to a height of $h \times c^2$ the second time. By examining this pattern

for the third and fourth bounce, it becomes clear that the ball will bounce to a height of $h \times c^k$ on the k^{th} time. See [Figure 2.1](#) for a graphic description of this process.

We are assuming that $k > 0$ and that $c < 1$. Note that c depends on many factors, such as the elasticity of the ball and the properties of the floor on which the ball is dropped. However, if we know that we will be given c , we don't need to worry about any other details.

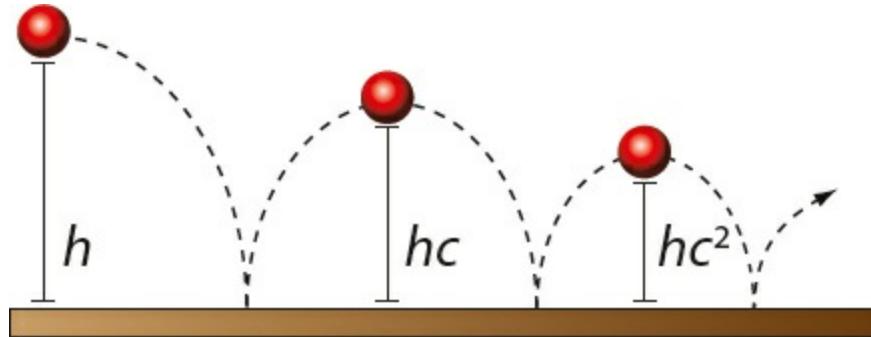


Figure 2.1: A ball is dropped from height h . The ball rises to height hc after the first bounce and to hc^2 after the second.

2. Design a solution: This problem is straightforward, but it's always useful to practice good design. Remember that you've got to plan your input and output as well as the computation in a program. As practice for more complicated problems, let's break this problem down into smaller subproblems.

Subproblem 1: Get the values of h , c , and k from the user.

Subproblem 2: Compute the height of the ball after the k^{th} bounce.

Subproblem 3: Display the calculated height.

The solution to each of the three subproblems requires input and generates an output. [Figure 2.2](#) shows how these solutions are connected. The first box in this figure represents the solution to subproblem 1. It asks a user to input values of parameters h , c , and k . It sends these values to the next box, which represents a solution to subproblem 2. This second box computes the height of the ball after k bounces and makes it available to the third box, which represents a solution to subproblem 3. This third box displays the calculated height.

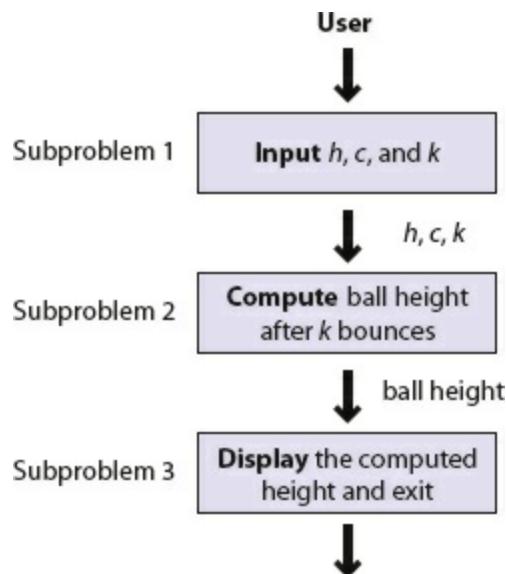


Figure 2.2: Connections between solutions to the three subproblems in the bouncing ball problem.

Before we can continue on to Step 3, we need to learn some Java. [Section 2.3](#) introduces you to the Java you'll need to solve this problem. ■

2.2.2 Acquiring a Java compiler

Before we introduce any Java syntax, you should make sure that you have a Java compiler set up so that you can follow along and test your solution. Programming is a hands-on skill. It is impossible to improve your programming abilities without practice. No amount of reading about programming is a substitute for the real thing.

Where can you get a Java compiler? Fortunately, there are free options for almost every platform. Non-Windows computers may already have the Java Runtime Environment (JRE) installed, allowing you to run Java programs; however, most Java development options require you to have the Java Development Kit (JDK). Oracle may change the website, but at the time of writing you can download the JDK from <http://www.oracle.com/technetwork/java/javase/downloads/>. Download the latest version (Java SE 7 at the time of writing) of the Java Platform, Standard Edition JDK and install it.

After having done so, you should be able to compile programs using the `javac` command, whose name is short for “Java compiler.” To do so, open a terminal window, also known as a command line interface or the console. To open the terminal in Windows, choose the Command Prompt option from the Start Menu or press Windows+R and type cmd into the Run box (or the Search box in the Start Menu in Windows 7). To open the terminal in Mac OS X, select Terminal from the Utilities folder. Linux users are usually familiar with their terminal.

Provided that you have your path set correctly, you should be able to open the terminal, navigate to a directory containing files that end in .java, and compile them using the `javac` command. For example, to compile a program called `Example.java` to bytecode, you would type:

```
javac Example.java
```

Compiling the program would create a .class file, in this case, `Example.class`. To run the program contained in `Example.class`, you would type:

```
java Example
```

Doing so fires up the JVM, which uses the JIT compiler to compile the bytecode inside `Example.class` into machine code and run it. Note that you would only type `Example` not `Example.class` when specifying the program to run. Using just these commands from the terminal, you can compile and run Java programs. The command line interface used to be the only way to interact with a computer, and, though it seems primitive at first, the command line has amazing power and versatility.

To use the command line interface to compile your own Java program, you must first create a text file with your Java code in it. The world of programming is filled with many text editor applications whose only purpose is to make writing code easier. These editors are not like Microsoft Word: They are not used to format the text into paragraphs or apply bold or italics. Their output is a raw (“plain”) text file, containing only unformatted characters, similar to the files created by Notepad. Many text editors, however, have advanced features useful for programmers, such as syntax highlighting (marking special words and symbols in the language with corresponding colors or fonts), language-appropriate auto-completion, and powerful find and replace tools. Two of the most popular text editors for command line use are vi-based editors, particularly Vim, and Emacs-based editors, particularly GNU Emacs.

Many computer users, however, are used to a *graphical user interface* (GUI), where a mouse can be used to interact with windows, buttons, text boxes, and other elements of a modern user interface. There are many Java programming environments that provide a GUI from which a user can write Java code, compile it, execute it, and even test and debug it. Because these tools are integrated into a single program, these applications are called *integrated development environments* (IDEs).

Two of the most popular Java IDEs are Eclipse and the NetBeans IDE, which are both open source, free, and available at <http://www.eclipse.org/> and <http://netbeans.org/>, respectively. At the time of writing, Eclipse is the most popular Java IDE for professional developers. Both Eclipse and NetBeans are powerful and complex tools. DrJava is a much simpler (and highly recommended) IDE designed for students and Java beginners. It is also free and is available from <http://www.drjava.org/>.

Which command line text editor or graphical IDE you use is up to you. Programming is a craft, and every craftsman has his or her favorite tools. Most of the content of this book is completely independent from the tools you use to write and compile your code. One exception is [Chapter 16](#), in which we briefly discuss the debugging tools in Eclipse and DrJava.

If you choose Eclipse, NetBeans, or another complex IDE, you may wish to read some online tutorials to get started. These IDEs often require the user to create a project and then put Java files inside. The idea of a project that contains many related source code documents is a useful one and is very common in software engineering, but it is not a part of Java itself.

2.3 Syntax: Java basics

In this section, we start with the simplest Java programs and work up to the solution to the bouncing

ball problem. Java was first released in 1995, a long time ago in the history of computer science. However, Java was based on many previous languages. Its *syntax* (the rules for constructing legal Java programs, just as English grammar is the rules for constructing legal English sentences) inherits ideas from C, C++, and other languages.

Some critics have complained about elements of the syntax or semantics of Java. *Semantics* are rules for what code means. Remember that Java is an arbitrary system, designed by fallible human beings. The rules for building Java programs are generally logical, but they are artificial. Learning a new programming language is a process of accepting a set of rules and coming up with ways to use those rules to achieve your own ends.

There are reasons behind the rules, but we will not always be able to explain those reasons in this book. As you begin to learn Java, you will have to take it on faith that such-and-such a rule is necessary, even though it seems useless or mysterious. In time, these rules will become familiar and perhaps sensible. The mystery will fade away, and the rules will begin to look like an organic and logical (though perhaps imperfect) system.

2.3.1 Java program structure

The first rule of Java is that all code goes inside of a *class*. A class is a container for blocks of code called *methods* and it can also be used as a template to create *objects*. We'll talk a bit more about classes in this chapter and then focus on them heavily in [Chapter 9](#).

For now, you only need to remember that every Java program has at least one class. It is possible to put more than one class in a file, but you can only have one top-level *public* class per file. A public class is one that can be used by other classes. Almost every class in this book is public, and they should be clearly marked as such. To create a public class called Example, you would type the following:

```
public class Example {  
}
```

A few words in Java have a special meaning and cannot be used for anything else (like naming a class). These are called *keywords* or *reserved words*. The keyword *public* marks the class as public. The keyword *class* states that you are declaring a class. The name Example gives the name of the class. By convention, all class names start with a capital letter. The braces ({ and }) mark the start and end of the contents of the class. Right now, our class contains nothing.

Exercise 2.7

This text should be saved in a file called Example.java. It is required that the name of the public class matches the file that it's in, including capitalization. Once Example.java has been saved, you can compile it into bytecode. However, since there's nothing in class Example, you can't run it as a program.

A program is a list of instructions, and that list has to start somewhere. For a normal Java application, that place is the main() method. (Throughout this book, we always append parentheses () to mark the name of a method.) If we want to do something inside of Example, we'll have to add a main() method like so:

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

There are several new items now. As before, **public** means that other classes can use the `main()` method. The **static** keyword means that the method is associated with the class as a whole and not a particular object. The **void** keyword means that the method does not give back an answer. The word `main` is obviously the name of the method, but it has to be spelled exactly that way (including capitalization) to work. Perhaps the most confusing part is the expression `String[] args`, which is a list of text (strings) given as input to the `main()` method from the command line. As with the class, the braces mark the start and end of the contents of the `main()` method, which is currently empty.

Right now, you don't need to understand any of this. All you need to know is that, to start a program, you need a `main()` method and its syntax is always the same as the code listed above. If you save this code, you can compile `Example.java` and then run it, and... nothing happens! It is a perfectly legal Java program, but the list of instructions is empty.

2.3.2 Command line input and output

An important thing for a Java program to do is to communicate with the outside world (where we live). First, let's look at printing data to the command line and reading data in from the command line.

The `System.out.print()` method

Methods allow us to perform actions in Java. They are blocks of code packaged up with a name so that we can run the same piece of code repeatedly but with different inputs. We discuss them in much greater depth in [Chapter 8](#).

A common method for output is `System.out.print()`. The input (usually called *arguments*) to a method are given between its parentheses. Thus, if we want to print 42 to the terminal, we type:

```
System.out.print (42) ;
```

Note that the use of the method has a semicolon (`;`) after it. An executable line of code in Java generally ends with a semicolon to separate it from the next instruction. We can add this code to our `Example` class, yielding:

```
public class Example {  
    public static void main ( String [] args ) {  
        System.out.print (42) ;  
    }  
}
```

If we want to print out text, we give it as the argument to `System.out.print()`, surrounded by double quotes (`"`). It is necessary to surround text with quotes so that Java knows it is text and not the name of a class, method, or variable. Text surrounded by double quotes is called a *string*. The following program prints Forty two onto the terminal.

```
public class Example {  
    public static void main ( String [] args ) {  
        System.out.print (" Forty two ");  
    }  
}
```

Printing out one thing is great, but programs are usually composed of many instructions. Consider the following program:

```
public class Example {  
    public static void main ( String [] args ) {  
        System.out.print (2) ;  
        System.out.print (4) ;  
        System.out.print (6) ;  
        System.out.print (8) ;  
    }  
}
```

As you can see, each executable line ends with a semicolon, and they are executed in sequence. This code prints 2, 4, 6, and 8 onto the screen. However, we did not tell the program to move the cursor to a new line at any point. So, the output on the screen is 2468, which looks like a single number. If we want them to be on separate lines, we can achieve this with the `System.out.println()` method, which moves to a new line after it finishes output.

```
public class Example {  
    public static void main ( String [] args ) {  
        System.out.println (2) ;  
        System.out.println (4) ;  
        System.out.println (6) ;  
        System.out.println (8) ;  
    }  
}
```

This change makes the output into the following:

2
4
6
8

In Java, it is possible to insert some math almost anywhere. Consider the following program, which uses the `+` operator.

```
public class Example {  
    public static void main ( String [] args ) {
```

```
System.out.print (35 + 7);
```

```
}
```

```
}
```

This code prints out 42 to the terminal just like our earlier example, because it does the addition before giving the result to `System.out.print()` for output.

The Scanner class

We want to be able to read input from the user as well. For command line input, we need to create a `Scanner` object. This object is used to read data from the keyboard. The following program asks the user for an integer, reads in an integer from the keyboard, and then prints out the value multiplied by 2.

```
import java.util.Scanner ;  
  
public class Example {  
    public static void main ( String [] args ) {  
        Scanner in;  
        in = new Scanner ( System.in );  
        System.out.print (" Enter an integer : ");  
        int value ;  
        value = in. nextInt ();  
        System.out.print (" That number doubled is: ");  
        System.out.println ( value * 2 );  
    }  
}
```

This program introduces several new elements. First, note that it begins with `import java.util.Scanner;`. This line of code tells the compiler to use the `Scanner` class that is in the `java.util` package. A package is a way of organizing a group of related classes. Someone else wrote the `Scanner` class and all the other classes in the `java.util` package, but, by importing the package, we are able to use their code in our program.

Then, skip down to the first line in the `main()` method. The code `Scanner in;` declares a variable called `in` with type `Scanner`. A variable can hold a value. The variable has a specific type, meaning the kind of data that the value can be. In this case, the type is `Scanner`, meaning that the variable `in` holds a `Scanner` object. Declaring a variable creates a box that can hold things of the specified type. To declare a variable, first put the type you want it to have, then put its identifier or name, and then put a semicolon. We chose to call the variable `in`, but we could have called it `input` or even `marmalade` if we wanted. It is always good practice to name your variable so that it is clear what it contains.

The next line assigns a value to `in`. The assignment operator (`=`) looks like an equal sign from math, but think of it as an arrow that points left. Whatever is on the right side of the assignment operator will be stored into the variable on the left. So, the variable `in` was an initially empty box that could hold a `Scanner` object. The code `new Scanner(System.in)` creates a brand new `Scanner` object based

on System.in, which means that the input will be from the keyboard. The assignment stored this object into the variable in. The fact that System.in was used has **nothing** to do with the fact that our variable was named in. Again, don't expect to understand all the details at first. Any time you need to read data from the keyboard, you'll need these two lines of code, which you should be able to copy verbatim. It is possible to both declare a variable and assign its value in one line. Instead of the two line version, most programmers would type:

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

Similarly, the line int value; declares a variable for holding integer types. The next line uses the object in to read an integer from the user by calling the nextInt() method. If we wanted to read a floating point value, we would have called the nextDouble() method. If we wanted to read some text, we would have called the next() method. Unfortunately, these differences means that we have to know what type of data the user is going to enter. If the user enters an unexpected type, our program could have an error. As before, we could combine the declaration and the assignment into a single line:

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

The final two lines give output for our program. The former prints That number doubled is: to the terminal. The latter prints out a value that is twice whatever the user entered. The next two examples illustrate how Scanner can be used to read input while solving problems. The first example shows how these elements can be applied to subproblem 1 of the bouncing ball problem, and the second example introduces and solves a new problem.

Exercise 2.13

Example 2.3: Command line input

Subproblem 1 requires us to get the height, coefficient of restitution, and number of bounces from the user. [Program 2.1](#) shows a Java program to solve this subproblem.

Program 2.1: A Java program to get the height, coefficient of restitution, and number of bounces from the command line. (GetInputCLI.java)

```
1 import java.util.*;
2
3 public class GetInputCLI {
4     public static void main ( String [] args ) {
5         // Create an object named in for input
6         Scanner in = new Scanner ( System.in);
7
8         // Declare variables to hold input data
9         double height , coefficient ;
10        int bounces ;
11
12        // Declare user prompt strings
13        String enterHeight = " Enter the height :";
```

```

14     String enterCoefficient =
15         " Enter restitution coefficient : ";
16     String enterBounces = " Enter the number of bounces : ";
17
18     // Prompt the user and read data from the keyboard
19     System.out.println (" Bouncing Ball : Subproblem 1");
20     System.out.print ( enterHeight );
21     height = in.nextDouble ();
22     System.out.print ( enterCoefficient );
23     coefficient = in.nextDouble ();
24     System.out.print ( enterBounces );
25     bounces = in.nextInt ();
26 }
27 }
```

Unlike our earlier example, the first line of GetInputCLI.java is `import java.util.*;`. Instead of just importing the Scanner class, this line imports all the classes in the `java.util` package. The asterisk (*) is known as a *wildcard*. The wildcard notation is convenient if you need to import several classes from a package or if you don't know in advance the names of all the classes you'll need.

Exercise 2.8

After the import comes the class declaration, creating a class called `GetInputCLI`. We put a CLI at the end of this class name to mark that it uses the command line interface, contrasting with the GUI version that we're going to show next. Inside the class declaration is the definition of the `main()` method, showing where the program starts. The text that comes after double slashes (//) are called *comments*. Comments allow us to make our code more readable to humans, but the compiler ignores them.

If you follow the comments, you'll see that we declare two `double` variables (for holding double precision floating-point numbers) and an `int` variable (for holding an integer value). Next we declare three `String` variables and assign them three strings (segments of text).

The last section of code first prints out the name of the problem. Then, it prints out the value stored into `enterHeight`, which is “Enter the height: ”. After this prompt, the line `height = in.nextDouble();` tries to read in the height from the user. It waits until the user hits enter before reading the value and moving on to the next line. The last four lines of the program prompt and read in the coefficient of restitution and then the number of bounces. If you compile and run this program, the execution should match the steps described. Note that it only reads in the values needed to solve the problem. We have not added the code to compute the answer or display it. ■

Example 2.4: Input for distance computation

Let's write a program that takes as input the speed of a moving object and the time it has been moving. The goal is to compute and display the total distance it travels. We can divide this problem into the following three subproblems.

Subproblem 1: Input speed and duration.

Subproblem 2: Compute distance traveled.

Subproblem 3: Display the computed distance.

Program 2.2 solves each of these subproblems in order, using the command-line input and output tools we have just discussed.

Program 2.2: Program to compute the distance a moving object travels. (Distance.java)

```
1 import java.util.*;
2
3 public class Distance {
4     public static void main ( String [] args ) {
5         // Create an object named in for input
6         Scanner in = new Scanner ( System.in );
7         double speed , time ;
8         double distance ; // Distance to be computed
9
10        // Solution to subproblem 1: Read input
11        // Prompt the user and get speed and time
12        System.out.print (" Enter the speed : ");
13        speed = in.nextDouble ();
14        System.out.print (" Enter the time : ");
15        time = in. nextDouble ();
16
17        // Solution to subproblem 2: Compute distance
18        distance = speed * time ;
19
20        // Solution to subproblem 3: Display output
21        System.out.print (" Distance traveled : ");
22        System.out.print ( distance );
23        System.out.println (" miles. ");
24    }
25 }
```

The program starts with import statements, the class definition, and the definition of the main() method. At the beginning of the main() method, we have code to declare and initialize a variable of type Scanner named in. We also declare variables of type **double** to hold the input speed and time and the resulting distance.

Starting at line 12, we solve subproblem 1, prompting the user for the speed and the time and using our Scanner object to read them in. Because they are both floating-point values with type **double**, we use the nextDouble() method for input.

At line 18, we compute the distance traveled by multiplying speed by time and storing the result in distance. The last three lines of the main() method solve subproblem 3 by outputting “Distance traveled: ”, the computed distance, and “miles.”. If you run the program, all three items are printed on the same line on the terminal. ■

2.3.3 GUI input and output

If you are used to GUI-based programs, you might wonder how to do input and output with a GUI instead of on the command line. GUIs can become very complex, but in this chapter we introduce a relatively simple way to do GUI input and output and expand on it further in [Chapter 7](#). Then, we go into GUIs in much more depth in [Chapter 15](#).

A limited tool for displaying output and reading input with a GUI is the JOptionPane class. This class has a complicated method called showMessageDialog() that opens a small *dialog box* to display a message to the user. If we want to create the equivalent of the command-line program that displays the number 42, the code would be as follows.

```
import javax.swing.JOptionPane ;  
  
public class Example {  
    public static void main ( String [] args ) {  
        JOptionPane.showMessageDialog ( null , "42" , "Output Example" ,  
            JOptionPane.INFORMATION_MESSAGE ); }  
}
```

Like Scanner, we need to import JOptionPane as shown above in order to use it. The showMessageDialog() method takes several arguments to tell it what to do. For our purposes, the first one is always the special Java literal `null`, which represents an empty value. The next is the message you want to display, but it has to be text. That’s why “42” appears with quotation marks. The third argument is the title that appears at the top of the dialog. The final argument gives information about how the dialog should be presented. `JOptionPane.INFORMATION_MESSAGE` is a flag value that specifies that the dialog is giving information (instead of a warning or a question), causing an appropriate, system-specific icon to be displayed on the dialog.

If we wanted to make the call to showMessageDialog() a little easier to read, we could store the arguments into variables with short, easy to understand names. The following program does so and should behave exactly like the previous program. [Figure 2.3](#) shows what the resulting GUI might look like.

```
import javax.swing.JOptionPane ;  
  
public class Example {  
    public static void main ( String [] args ) {  
        String message = "42";  
        String title = "Output Example";  
        JOptionPane.showMessageDialog ( null , message , title ,  
            JOptionPane.INFORMATION_MESSAGE ); }
```

```
}
```

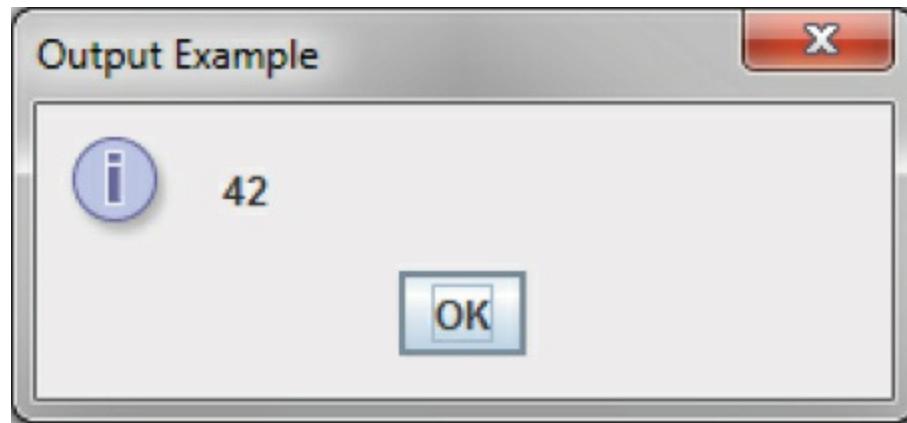


Figure 2.3: Example of showMessageDialog() used for output.

One way to do input with a GUI uses the showInputDialog() method, which is also inside the JOptionPane class. The showInputDialog() method *returns* a value. This means that it gives back an answer, which you can store into a variable by putting the method call on the right hand side of an assignment statement. Otherwise, it is nearly the same as showMessageDialog(). The following program prompts the user for his or her favorite word with a showInputDialog() method and then displays it again using a showMessageDialog() method.

```
import javax.swing.JOptionPane ;  
  
public class Example {  
    public static void main ( String [] args ) {  
        String message = " What is your favorite word ?";  
        String title = " Input Example ";  
        String word =  
        JOptionPane.showInputDialog (null , message , title ,  
            JOptionPane.QUESTION_MESSAGE );  
        JOptionPane.showMessageDialog (null , word , title ,  
            JOptionPane.INFORMATION_MESSAGE );  
    }  
}
```

Note that whatever the user typed in will be stored in word. Finally, the last line of the program displays whatever word was entered with showMessageDialog(). [Figure 2.4](#) shows the resulting GUI as the user is entering input.

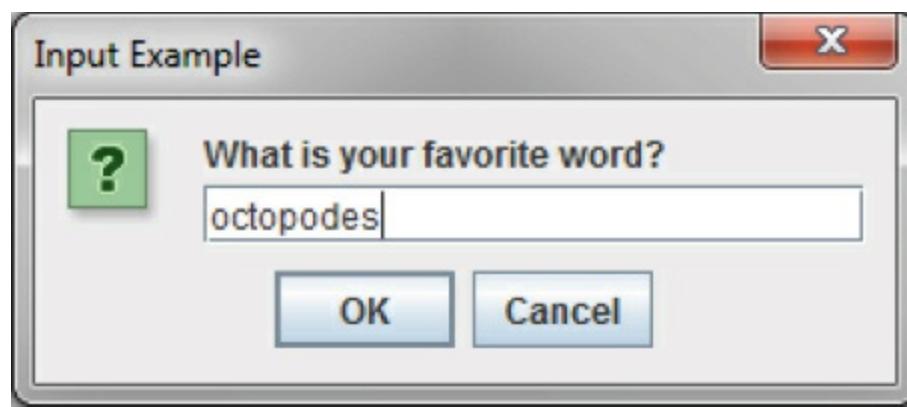


Figure 2.4: Example of showInputDialog() used for input.

Remember that the value returned from the showInputDialog() method is always text, that is, it always has type String. Although there are lots of great things you can do with a String value, you cannot do normal arithmetic like you can with an integer or a floating-point number. However, there are ways to convert a String representation of a number to the number itself. If you have a String that represents an integer, you use the Integer.parseInt() method to convert it. If you have a String that represents a floating-point number, you use the Double.parseDouble() method to convert it. The following segment of code shows a few illustrations of the issues involved.

```
int x = "41" * 3;  
// Text cannot be multiplied by an integer  
  
int y = Integer.parseInt ("23");  
// Correctly converts the text "23" to the integer 23  
  
double z = Double.parseDouble (" 3.14159 ");  
// Correctly converts the text "3.14159" to 3.14159  
  
int a = Integer.parseInt (" Twenty three ");  
// Causes the program to crash  
  
double b = Double.parseDouble ("pi");  
// Causes the program to crash
```

You might wonder why the computer isn't smart enough to know that "23" means 23. Remember, the computer has no intelligence. If something is marked as text, it doesn't know that it can interpret it as a number. What kind of data something is depends on its type, which doesn't change. We'll discuss types more deeply in [Chapter 3](#).

The next example uses these two type conversion methods with methods from JOptionPane in a GUI-based solution to subproblem 1 of the bouncing ball problem.

Example 2.5: GUI input

We can change the solution given in [Program 2.1](#) to use the GUI-based input tools in JOptionPane

Program 2.1 this equivalent GUI-based Java program.

Program 2.3: Program to get the height, coefficient of restitution, and number of bounces using a GUI. (GetInputGUI.java)

```
1 import javax.swing.*;
2
3 public class GetInputGUI {
4     public static void main ( String [] args ) {
5         String title = " Bouncing Ball : Subproblem 1";
6
7         // Declare variables to hold input data
8         double height , coefficient ;
9         int bounces ;
10
11        // Declare user prompt strings
12        String enterHeight = " Enter the height :";
13        String enterCoefficient =
14            " Enter restitution coefficient :";
15        String enterBounces = " Enter the number of bounces :";
16
17        // Prompt the user , get data , and convert it
18        String response = JOptionPane.showInputDialog (null ,
19            enterHeight , title , JOptionPane.QUESTION_MESSAGE );
20        height = Double.parseDouble ( response );
21        response = JOptionPane.showInputDialog (null ,
22            enterCoefficient , title , JOptionPane.QUESTION_MESSAGE );
23        coefficient = Double.parseDouble ( response );
24        response = JOptionPane.showInputDialog (null ,
25            enterBounces , title , JOptionPane.QUESTION_MESSAGE );
26        bounces = Integer.parseInt ( response );
27    }
28 }
```

The code in this program is very similar to [Program 2.1](#) until [line 18](#). At that point, we use the `showInputDialog()` method to read a String version of the height from the user. On the next line, we have to convert this String version into the `double` version that we store in the `height` variable. The next four lines read in the coefficient of restitution and the number of bounces and convert them to their appropriate numerical types. ■

2.3.4 A few operations

Basic math

To make our code useful, we can perform operations on values and variables. For example, we used

the expression $35 + 7$ as an argument to the `System.out.print()` method to print 42 to the screen. We can use the add (+), subtract (-), multiply (*), and divide(/) operators on numbers to solve arithmetic problems. These operators work the way you expect them to (except that division has a few surprises). We'll go into these operators and others more deeply in [Chapter 3](#). Here are examples of these four operators used with integer and floating-point numbers.

```
int a = 2 + 3;          // a will hold 5
int b = 2 - 3;          // b will hold -1
int c = 2 * 3;          // c will hold 6
int d = 2 / 3;          // d will hold 0 ( explained later )

double x = 1.6 + 3.2;   // x will hold 4.8
double y = 1.6 - 3.2;   // y will hold -1.6
double z = 1.6 * 3.2;   // z will hold 5.12
double w = 1.6 / 3.2;   // w will hold 0.5
```

Other operations

These basic operations can mix values and variables together. As we will discuss later, they can be arbitrarily complicated with order of operations determining the final answer. Nevertheless, we also need ways to accomplish other mathematical operations such as raising a number to a power or finding its square root. The `Math` class has methods that perform these and other functions. To raise a number to a power, we call `Math.pow()` with two arguments: first the base and then the exponent. To find the square root, we pass the number to the `Math.sqrt()` method.

```
double p = Math.pow (3.0 , 2.5) ;
// Raises 3 to the power 2.5 , approximately 15.588457
double q = Math.sqrt (2.0) ;
// Finds the square root of 2.0 , approximately 1.4142136
```

Example 2.6: Compute height

We compute the final height of the ball in subproblem 2 of the bouncing ball problem. To do so, we have to multiply the height by the coefficient of restitution raised to the power of the number of bounces. The following program does so, using the `Math.pow()` method.

Program 2.4: Program to compute height of a ball after bounces. (`ComputeHeight.java`)

```
1 public class ComputeHeight {
2     public static void main ( String [] args ) {
3         // Use dummy values to test subproblem 2
4         double height = 15, coefficient = 0.3;
5         int bounces = 10;
6         // Compute height after bounces
7         double bounceHeight = height * Math.pow( coefficient , bounces );
8         System.out.println ( bounceHeight ); // For testing
```

```
9     }
10 }
```

Program 2.4 is only focusing on subproblem 2, but, if we want to test it, we need to supply some dummy values for height, coefficient, and bounces, since these are read in by the solution to subproblem 1. Likewise, the output statement on the last line of the main() method is just for testing purposes. The complete solution has more complex output. ■

String concatenation

Just as we can add numbers together, we can also “add” pieces of text together. In Java, text has the type String. If you use the + operator between two values or variables of type String, the result is a new String that is the *concatenation* of the two previous String values, meaning that the result is the two pieces of text pasted together, one after the other. Concatenation doesn’t change the String values you are concatenating.

At this point, it becomes confusing if you mix types (String, int, double) together when doing mathematical operations. However, feel free to concatenate String values with any other type using the + operator. When you do so, the other type is automatically converted into a String. This behavior is useful since any String is easy to output. Here are a few examples of String concatenation.

```
String word1 = " tomato ";
String word2 = " sauce ";
String text1 = word1 + word2 ;
// text1 will contain " tomatoesauce "
String text2 = word1 + " " + word2 ;
// text2 will contain " tomato sauce "
String text3 = " potato " + word1 ;
// text3 will contain " potato tomato "
String text4 = 5 + " " + word1 + "es";
// text4 will contain "5 tomatoes "
```

Example 2.7: Display height

With String concatenation, subproblem 3 becomes a bit easier. We concatenate the results together with an appropriate message and then use the System.out.println() method for output. The following program does so.

Program 2.5: Program to display height of a ball using the command line. (DisplayHeightCLI.java)

```
1 public class DisplayHeightCLI {
2     public static void main ( String [] args ) {
3         // Use dummy values to test subproblem 3
4         int bounces = 10;
5         double bounceHeight = 2.0;
6         String message = " After " + bounces +
```

```

7     " bounces the height of the ball is: " +
8     bounceHeight + " feet ";
9     System.out.println( message );
10    }
11 }

```

Program 2.5 is only focusing on subproblem 3, but, if we want to test it, we need to supply dummy values for bounces and bounceHeight, since these are generated by the solution to earlier subproblems.

The same concatenation can be used for GUI output as well. The only difference is the use of JOptionPane.showMessageDialog() instead of System.out.println().

Program 2.6: Program to display height of a ball using a GUI. (DisplayHeightGUI.java)

```

1 import javax.swing.*;
2
3 public class DisplayHeightGUI {
4     public static void main ( String [] args ) {
5         String title = " Bouncing Ball : Subproblem 3";
6         // Use dummy values to test subproblem 3
7         int bounces = 10;
8         double bounceHeight = 2.0;
9         String message = " After " + bounces +
10            " bounces the height of the ball is: " +
11            bounceHeight + " feet ";
12         JOptionPane.showMessageDialog ( null , message , title ,
13             JOptionPane.INFORMATION_MESSAGE );
14     }
15 }

```

2.3.5 Java formatting

Writing good Java code has some similarities to writing effectively in English. There are rules you have to follow in order to make sense, but there are also guidelines that you should follow in order to make your code easier to read for yourself and everyone else.

Variable and class naming

Java programs are filled with variables, and each variable should be named to reflect its contents. Variable names are essentially unlimited in length (although the JVM you use may limit this length to thousands of characters). A tremendously long variable name can be hard to read, but abbreviations can be worse. You want the meaning of your code to be obvious to others and to yourself when you come back days or weeks later.

Imagine you are writing a program that sells fruit. Consider the following names for a variable that keeps track of the number of apples.

Name	Attributes
a	Too short, gives no useful information
apps	Too short, vague, could mean applications or appetizers
cntr	Too short, vague, could mean center
counter	Not bad, but counting what?
theVariableUsedToCountApples	Too long for no good reason
appleCounter	Very clear
apples	Concise and clear, unless there are multiple apple quantities such as applesSold and applesBought

Mathematics is filled with one letter variables, partly because there is a history of writing mathematics on chalkboards and paper. Clarity is more important than brevity with variables in computer programs. Some variables need more than one word to be descriptive. In that case, programmers of Java are encouraged to follow *camel case*. In camel case, multi-word variables and methods start with a lowercase letter and then use an uppercase letter to mark the beginning of each new word. It is called camel case because the uppercase letters are reminiscent of the humps of a camel. Examples include lastValue, symbolTable, and makeHamSandwich().

By convention, class names should always begin with a capital letter, but they also use camel case with a capital letter for the beginning of each new word. Examples include LinkedList, JazzPiano, and GlobalNuclearWarfare.

Another convention is that constants, variables whose value never changes, have names in all uppercase, separated by underscores. Examples include PI,

TRIANGLE_SIDES, and UNIVERSAL_GRAVITATIONAL_CONSTANT.

Spaces are not allowed in variable, method, or class names. Recall that a name in Java is called an identifier. The rules for identifiers specify that they must start with an uppercase or lowercase letter (or an underscore) and that the remaining characters must be letters, underscores, or numerical digits. Thus, mötleyCrüe, Tupac, and even the absurd _____5 are legal identifiers, but Motley Crue and 2Pac are not. Java has support for many of the world's languages, allowing identifiers to contain characters from Chinese, Thai, Devanagari, Cyrillic, and other scripts.

Remember that keywords cannot be used as identifiers. For example, **public**, **static**, and **class** are all keywords in Java and can never be the names of classes, variables, or methods.

White space

Although you are not allowed to have spaces in a Java identifier, you can usually use white space (spaces, tabs, and new lines) wherever you want. Java ignores extra space. Thus, this line of code:

```
int x = y + 5;
```

is equivalent to this one:

```
int x=y +5;
```

We chose to type our earlier example of a program performing output as follows:

```
public class Example {  
    public static void main ( String [] args ) {  
        System.out.print (42) ;  
    }  
}
```

However, we could have been more chaotic with our use of whitespace:

```
public  
class      Example {  
public  
    static void  
main ( String      [  
        ] args  
    ) {  
        System.  
out  
        .print (42  
) ; } }
```

Or used almost none at all:

```
public class Example { public static void main ( String [] args ){ System.  
    out.print (42) ;}}
```

These three programs are identical in the eyes of the Java compiler, but the first one is easier for a human to read. You should use whitespace to increase readability. Don't add too much whitespace with lots of blank lines between sections of code. On the other hand, don't use too little and cramp the code together. Whenever code is nested inside of a set of braces, indent the contents so that it is easy to see the hierarchical relationship.

The style we present in this book puts the left brace ({}) on the line starting a block of code. Another popular style puts the left brace on the next line. Here is the same example program formatted in this style:

```
public class Example  
{  
    public static void main ( String [] args )  
    {  
        System.out.print (42) ;  
    }  
}
```

There are people (including some authors of this book) who prefer this style because it is easier to see where blocks of code begin and end. However, the other style uses less space, and so we use it throughout the book. You can make your own choices about style, but be consistent. If you work for a

software development company, they may have strict standards for code formatting.

Comments

As we mentioned before, you can leave comments in your code whenever you want someone reading the code to have extra information. Java has three different kinds of comments. We described single-line comments, which start with a `//` and continue until the end of the line.

If you have a large block of text you want as a comment, you can create a block comment, which starts with a `/*` and continues until it reaches a `*/`.

Beyond leaving messages for other programmers, you can also “comment out” existing code. By making Java code a comment, it no longer affects the program execution. This practice is very common when programmers want to remove or change some code but are reluctant to delete it until the new version of the code has been tested.

The third kind of comment is called a documentation comment and superficially looks a lot like a block comment. A documentation comment starts with a `/**` and ends with a `*/`. These comments are supposed to come at the beginning of classes and methods and explain what they are used for and how to use them. A tool called javadoc is used to run through documentation comments and generate an HTML file that users can read to understand how to use the code. This tool is one of the features that has contributed greatly to the popularity of Java, since its libraries are well-documented and easy to use. However, we do not discuss documentation comments deeply in this book.

Here is our example output program heavily commented.

```
/**  
 * Class Example prints the number 42 to the screen.  
 * It contains an executable main () method.  
 */  
  
public class Example {  
    /*  
     * The main () method was last updated by Barry Wittman.  
     */  
  
    public static void main ( String [] args ) {  
        System.out.print (42) ; // answer to everything  
    }  
}
```

Comments are a wonderful tool, but clean code with meaningful variable names and careful use of whitespace doesn’t require too much commenting. Never hesitate to comment, but always ask yourself if there is a way to write the code so clearly that a comment is unnecessary.

2.4 Solution: How to solve problems

The problem solving steps given in [Section 2.2](#) are sound, but they depend on being able to implement your planned solution in Java. We have introduced far too little Java so far to expect to solve **all** the

problems that can be solved with a computer in this chapter. However, we can show the solution to the bouncing ball problem and explain how our solution works through the software development lifecycle.

2.4.1 Bouncing ball solution (command line version)

In [Example 2.2](#) we made sure that we understood the problem and then formed a three-part plan to read in the input, compute the height of the bounce, and then output it.

In [Program 2.1](#), we implemented subproblem 1, reading the input from the command line. In [Program 2.4](#), we implemented subproblem 2, computing the height of the final bounce. In [Program 2.5](#), we implemented subproblem 3, displaying the height that was computed. In the final, integrated program, the portion of the code that corresponds to solving subproblem 1 is below.

```
1 import java.util.*;
2
3 public class BouncingBallCLI {
4     public static void main ( String [] args ) {
5         // Solution to subproblem 1
6         // Create an object named in for input
7         Scanner in = new Scanner ( System.in );
8
9         // Declare variables to hold input data
10        double height , coefficient ;
11        int bounces ;
12
13        // Declare user prompt strings
14        String enterHeight = " Enter the height : ";
15        String enterCoefficient =
16            " Enter restitution coefficient : ";
17        String enterBounces = " Enter the number of bounces : ";
18
19        System.out.println ( " Bouncing Ball " );
20
21        // Prompt the user and read data from the keyboard
22        System.out.println ( " Bouncing Ball : Subproblem 1 " );
23        System.out.print ( enterHeight );
24        height = in.nextDouble ();
25        System.out.print ( enterCoefficient );
26        coefficient = in.nextDouble ();
27        System.out.print ( enterBounces );
28        bounces = in.nextInt ();
```

With the imports, class declaration, and `main()` method set up by the solution to subproblem 1, the solution to subproblem 2 is very short.

```
30 // Solution to subproblem 2
31 double bounceHeight = height * Math.pow( coefficient , bounces );
```

The solution to subproblem 3 and the braces that mark the end of the main() method and then the end of the class only take up a few additional lines.

```
33 // Solution to subproblem 3
34 String message = " After " + bounces +
35     " bounces the height of the ball is: " +
36     bounceHeight + " feet ";
37 System.out.println ( message );
38 }
39 }
```

2.4.2 Bouncing ball solution (GUI version)

If you prefer a GUI for your input and output, we can integrate the GUI-based versions of the solutions to subproblems 1, 2, and 3 from Programs 2.1, 2.4, and 2.6. The final program is below. It only differs from the command line version in a few details.

Program 2.7: Full program to compute the height of the final bounce of a ball and display the result with a GUI. (BouncingBallGUI.java)

```
1 import javax.swing.*;
2
3 public class BouncingBallGUI {
4     public static void main ( String [] args ) {
5         // Solution to sub - problem 1
6         String title = " Bouncing Ball ";
7         double height , coefficient ;
8         int bounces ;
9
10        // Declare user prompt strings
11        String enterHeight = " Enter the height : ";
12        String enterCoefficient =
13            " Enter restitution coefficient : ";
14        String enterBounces = " Enter the number of bounces : ";
15
16        // Prompt the user , get data , and convert it
17        String response = JOptionPane.showInputDialog ( null ,
18            enterHeight , title , JOptionPane.QUESTION_MESSAGE );
19        height = Double.parseDouble ( response );
20        response = JOptionPane.showInputDialog ( null ,
21            enterCoefficient , title , JOptionPane.QUESTION_MESSAGE );
22        coefficient = Double.parseDouble ( response );
```

```

23     response = JOptionPane.showInputDialog (null ,
24         enterBounces , title , JOptionPane.QUESTION_MESSAGE );
25     bounces = Integer.parseInt ( response );
26
27     // Solution to sub - problem 2
28     double bounceHeight = height * Math.pow( coefficient , bounces );
29
30     // Solution to sub - problem 3
31     String message = " After " + bounces +
32         " bounces the height of the ball is: " +
33         bounceHeight + " feet ";
34     JOptionPane.showMessageDialog (null , message , title ,
35         JOptionPane.INFORMATION_MESSAGE );
36 }
37 }
```

2.4.3 Testing and maintenance

Testing and maintenance are key elements of the software engineering lifecycle and often take more time and resources than the rest. However, we only discuss them briefly here.

The ball bouncing problem is not complex. There are a few obvious things to test. We should pick a “normal” test case such as a height of 15 units, a coefficient of restitution of 0.3, and 10 bounces. The height should be $15 \times 0.3^{10} = 0.0000885735$. The result computed by our program should be the same, ignoring floating point error. We can also check some boundary test cases. If the coefficient of restitution is 1, the ball should bounce back perfectly, reaching whatever height we input. If the coefficient of restitution is 0, the ball doesn’t bounce at all, and the final height should be 0.

Our code does not account for users entering badly formatted data like two instead of 2. Likewise, our code does not detect invalid values such as a coefficient of restitution greater than 1 or a negative number of bounces. An industrial-grade program should. We’ll discuss testing further in [Chapter 16](#).

As with most of the problems we discuss in this book, issues of maintenance will not apply. We don’t have a customer base to keep happy. However, it is a good thought exercise to imagine a large-scale version of this program that can solve many different kinds of physics problems. Who are likely to be your clients? What are the kinds of bugs that are likely to creep into such a program? How would you provide bug-fixes and develop new features?

2.5 Concurrency: Solving problems in parallel

2.5.1 Parallelism and concurrency

The terms *parallelism* and *concurrency* are often confused and sometimes used interchangeably. Parallelism or parallel computing occurs when multiple computations are being performed at the same time. Concurrency occurs when multiple computations may interact with each other. The distinction is subtle since many parallel computations are concurrent and vice versa.

An example of parallelism without concurrency is two separate programs running on a multicore system. They are both performing computations at the same time, but, for the most part, they are not interacting with each other. (Concurrency issues can arise if these programs try to access a shared resource, such as a file, at the same time.)

An example of concurrency without parallelism is a program with multiple threads of execution running on a single-core system. These threads will not execute at the same time as each other. However, the OS or runtime system will interleave the execution of these threads, switching back and forth between them whenever it wants to. Since these threads can share memory, they can still interact with each other in complicated and often unpredictable ways.

With multicore computers, we want good and effective parallelism, computing many things at the same time. Unfortunately, striving to reach parallelism often means struggling with concurrency and carefully managing the interactions between threads.

2.5.2 Sequential versus concurrent programming

Imagine that the evil Lellarap aliens are attacking Earth. They have sent an extensive list of demands to the world's leaders, but only a few people, including you, have mastered their language, Lellarapian. To save the people of Earth, it is imperative that you translate their demands as quickly as possible so that world leaders can decide what course of action to take. If you do it alone, as illustrated in [Figure 2.5\(a\)](#), the Lellaraps might attack before you finish.

In order to finish the work faster, you hire a second translator whose skills in Lellarap are as good as yours. As shown in [Figure 2.5\(b\)](#), you divide the document into two nearly equal parts, Document A and Document B. You translate Document A and your colleague translates Document B. When both translations are complete, you merge the two, check the translation, and send the result to the world's leaders.

Translating the demands alone is a *sequential* approach. In this context, sequential mean non-parallel. Translating the demands with two people is a parallel approach. It is concurrent as well because you have to decide on how to split up the document and how to merge it back together.

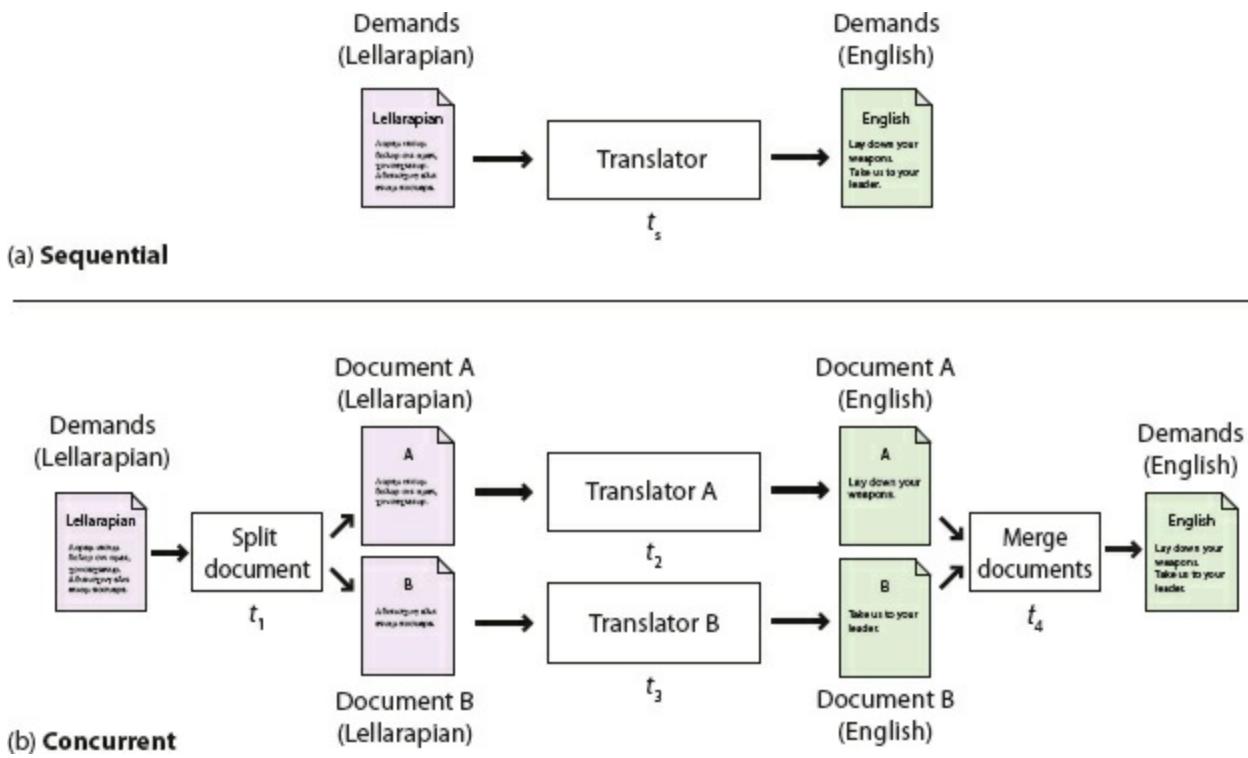


Figure 2.5: (a) Translation by one translator. Time t_s gives the sequential time taken. (b) Translation by translators A and B. Times t_1 , t_2 , t_3 , and t_4 give the times needed to do each component of the concurrent approach.

If you were to write a computer program to translate the demands using the sequential approach, you would produce a *sequential program*. If you write a computer program that uses the approach shown in Figure 2.5(b), it would be a *concurrent program*. A concurrent program is also referred to as a *multi-threaded* program. *Threads* are sequences of code that can execute independently and access each other's memory. Imagine you are one thread of execution and your colleague is another. Thus, the concurrent approach will have at least two threads. It may have more if separate threads are used to divide up the document and merge it back together.

Because we're interested in the time the process takes, we have labeled different tasks in Figure 2.5 with their running times. We let t_s be the time for one person to complete the translation. The times t_1 through t_4 mark the times needed to complete tasks 1 through 4, indicated in Figure 2.5(b). Exercise 2.10 asks you to calculate these times for the sequential and concurrent approaches.

Exercise 2.10

2.5.3 Kinds of concurrency

A sequential program, like the single translator, uses a single processor on a multi-processor system or a single core on a multicore chip. To speed up the solution of a program on a multicore chip, it may be necessary to divide a problem so that different parts of it can be executed concurrently.

This process of dividing up a problem falls into the category of *domain decomposition*, *task decomposition*, or some combination of the two. In domain decomposition, we take a large amount of data or elements to be processed and divide up the data among workers that all do the same thing to different parts of the data. In task decomposition, each worker is assigned a different task that needs to be done. The following two examples explore each of these approaches.

Example 2.8: Domain decomposition

Suppose we have an autonomous robot called the Room Rating Robot or R³. The R³ can measure the area of any home. Suppose that we want to use an R³ to measure the area of the home with two floors sketched in [Figure 2.6](#).

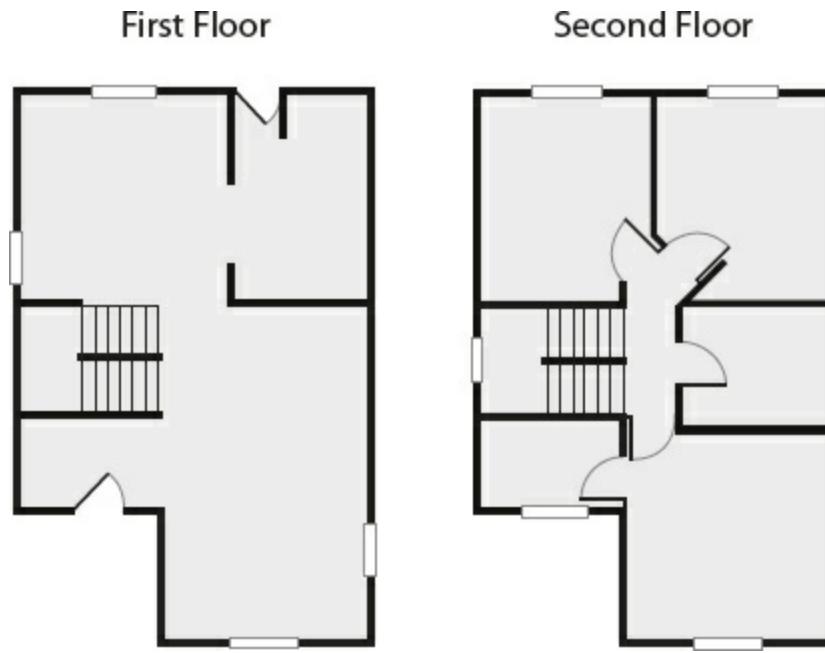


Figure 2.6: A home with two floors.

One way to measure the area is to put an R³ at the entrance of the home on the first floor and give it the following instructions:

1. Initialize total area to 0
2. Find area of next room and add to total area
3. Repeat Step 2 until all rooms have been measured
4. Move to second floor
5. Repeat Step 2 until all rooms have been measured
6. Display total area

By following these steps, the R³ will systematically go through each room, measure its area, and add the value to the total area found so far. It will then climb up the stairs and repeat the process for the second floor. It would then add the areas from the two floors and give us the total living area of the house. This is a sequential approach for measuring the area.

Now suppose we have two R³ robots, named R³A and R³B. We can put R³A on the first floor and R³B on the second. Both robots are then instructed to find the area of the floor they're on using steps very similar to the ones listed above for a sequential solution. When done, we add together the answers from R³A and R³B to get the total. This is a concurrent (and also parallel) approach for

measuring the living area of a home with two floors. Using two robots in this way can speed up the time it takes to measure the area. ■

Exercise 2.11

In the example above, the tasks are the same, i.e., measuring the area, but are performed on two different input domains, i.e., the floors. Thus, both robots are performing essentially the same operations but on different floors. This type of task division is also known as domain decomposition. Here, to achieve concurrency, we take the domain of the problem (the house) and divide it into smaller subdomains (its floors). Then, each processor (or robot in this example) performs the same task on each subdomain. When done, the final answer is found by combining the answers. Running the robots on each floor is purely parallel, but combining the answers is concurrent since some interaction between the robots is necessary.

Exercise 2.12

Another way of solving a problem concurrently is to divide it into fundamentally different tasks. The tasks could be executed on different processors and perhaps on different input domains. Eventually, some coordination of the tasks must be done to generate the final result. The next example illustrates such a division.

Example 2.9: Task decomposition

Let's expand the problem given in [Example 2.8](#). R³ robots can do more than just measure area. In addition to calculating the living area of a home, we want an R³ robot to check if the electrical outlets are in working condition. The robot should give us the area of the house as well as a list of electrical outlets that are not working.

This problem can be solved in a sequential manner with just one robot. One way to do so would be for a robot to make a first pass through all floors and rooms and compute the living area. It can then make a second pass and make a list of electrical outlets that are not working.

One way to solve this problem concurrently is to assign R^{3A} to measure the area and R^{3B} to identify broken electrical outlets. Once the respective tasks are assigned, we place the robots at the entrance to the house and activate them. It is possible that the two robots will bump into each other while working, and that's one of the difficulties of concurrency. The burden is on the programmer to give instructions so that the robots can avoid or recover from collisions. After the robots are done, we ask R^{3A} for the living area of the house and R^{3B} for a list of broken outlets. ■

2.6 Summary

In this chapter, we introduced an approach for developing software to solve a problem with a computer. A number of examples illustrated how to move from a problem statement to a complete Java program. Although we have given rough explanations of all the Java programs in this chapter, we encourage you to play with each program to expand its functionality. Several exercises prompt you to do just that. It is impossible to learn to program without actively practicing programming. Never be afraid of “breaking” the program. Only by breaking it, changing it, and fixing it will your understanding grow.

In addition to the software development lifecycle, we introduced several building blocks of Java syntax including classes, main() methods, import statements, and variable declarations. We also gave a preview of different variable types (`int`, `double`, and `String`) and operations that can be used with them. Material about types and operations on them is covered in depth in the next chapter. Furthermore, we discussed input and output using Scanner and `System.out.print()` for the command line interface and JOptionPane methods for a GUI.

Finally, we introduced the notions of sequential and concurrent solutions to problems and clarified the subtle difference between parallelism and concurrency.

Exercises

Conceptual Problems

- 2.1 When solving a problem using a computer, what problem is solved by the programmer and what problem is solved by the program written by the programmer? Are they the same?
- 2.2 In [Example 2.4](#), we declared all variables to be of type double. How would the program behave differently if we had declared all the variables with type int?
- 2.3 What is the purpose of the statement at [line 6](#) in [Program 2.1](#)?
- 2.4 Explain the difference between a declaration and an assignment statement.
- 2.5 Is the following statement from [line 17](#) of [Program 2.7](#) a declaration, an assignment, or a combination of the two?

```
String response = JOptionPane.showInputDialog (
    null, enterHeight, title,
    JOptionPane.QUESTION_MESSAGE);
```

- 2.6 When would you prefer using the JOptionPane class for output over `System.out.print()`? When might you prefer `System.out.print()` to using JOptionPane?
- 2.7 Review [Program 2.7](#) and identify all the Java keywords used in it.
- 2.8 Try to recompile [Program 2.7](#) after removing the `import` statement at the top. Read and explain the error message generated by the compiler.
- 2.9 Explain the difference between parallel and concurrent tasks. Give examples of tasks that are parallel but not concurrent, tasks that are concurrent but not parallel, and tasks that are both.

Concurrency

- 2.10 Refer to [Figure 2.5](#). Suppose that you and your colleague translate from English to Lellarapiar at the rate of 200 words per hour. Suppose that the list of demands contains 10,000 words.

(a) Compute t_s , the time for you to translate the entire document alone, assuming that, after

Concurrency

translation, you perform a final check at the rate of 500 words per hour.

- (b) Now assume that the task of splitting up the document and handing over the correct part to your colleague takes 15 minutes. Also, the task of receiving the translated document from your colleague and merging with the one you translated takes another 15 minutes. After merging the two documents, you do a final check for correctness at a rate of 500 words per hour. Calculate the total time to complete the translation using this concurrent approach. Let us refer to this time as t_c .
- (c) One way to calculate the speedup of a concurrent solution is to divide the sequential time by the concurrent time. In our case, the speedup is t_s/t_c . Using the values you have computed in (a) and (b), calculate the speedup.
- (d) Suppose that you have a total of two colleagues willing to help you with the translation. Assuming that the three of you will perform the translation and that the times needed to split, merge, and check are unchanged, calculate the total time needed. Then, compute the speedup.
- (e) Now suppose that there are an unlimited number of people willing and able to help you with the translation. Will the speedup keep on increasing as you add more translators? Explain your answer.

2.11 In [Example 2.8](#), what aspect of a multicore system do the robots represent?

Concurrency

2.12 In [Example 2.8](#), suppose that you have two R^3 robots available. You would like to use these to measure the living area of a single floor home. Suggest how two robots could be programmed to work concurrently to measure the living area faster than one.

Concurrency

Programming Practice

2.13 Write a program that prompts the user for three integers from the command line. Read each integer in using the `nextInt()` method of a `Scanner` object. Compute the sum of these values and print it to the screen using `System.out.print()` or `System.out.println()`.

2.14 Expand the program from Exercise 2.13 so that it finds the average of the three numbers instead of the sum. (Hint: Try dividing by 3.0 instead of 3 to get an average with a fractional part. Then, store the result in a variable of type `double`.)

GUI

2.15 Rewrite your solution to Exercise 2.14 so that it uses a `JOptionPane`-based GUI instead of `Scanner` and `System.out.print()`.

2.16 Copy and paste [Program 2.1](#) into the Java IDE you prefer. Compile and run it and make sure that the program executes as intended. Then, add statements to prompt the user for the color of the ball and read it in. Store the color in a `String` value. Add an output statement that mentions the color of the ball.

- 2.17 Rewrite your solution to Exercise 2.16 so that it uses a JOptionPane-based GUI instead of Scanner and System.out.print().
- 2.18 In [Example 2.4](#), we assumed that the speed is given in miles per hour and the time in hours. Change [Program 2.2](#) to compute the distance traveled by a moving object given its speed in miles per hour but time in seconds. You will need to perform a conversion from seconds to hours before you can find the distance.
- 2.19 A program can use both a command line interface and a GUI to interact with a user. Write a program that uses the Scanner class to read a String value containing the user's favorite food. Then display the name of the food using JOptionPane.
- 2.20 Use the complete software development cycle to write a program that reads in the lengths of two legs of a right triangle and computes the length of its hypotenuse.
1. Make sure you understand the problem. How can you apply the Pythagorean formula ($a^2 + b^2 = c^2$) to solve it?
 2. Design a solution by listing the steps you will need to take to read in the appropriate values, find the answer, and then output it.
 3. Implement the steps as a Java program.
 4. Test the solution with several known values. For example, a right triangle with legs of lengths 3 and 4 has a hypotenuse of length 5. Which values cause errors? How should your program react to those errors?
 5. Consider what other features your program should have. If your intended audience is children who are learning geometry, should your error handling be different from an audience of architects?

Chapter 3

Primitive Types and Strings

Originality exists in every individual because each of us differs from the others. We are all primary numbers divisible only by ourselves.

—Jean Guitton

3.1 Problem: College cost calculator

Perhaps you're a student. Perhaps you aren't. In either case, you must be aware of the rapidly rising cost of a college education. The motivating problem for this chapter is to create a Java program that can estimate the cost of a college education, including room and board. It starts by reading a first name, a last name, the per-semester cost of tuition, the monthly cost of rent, and the monthly cost of food as input from a user. Many students take out loans for college. In fact, student debt has surpassed credit card debt in the United States. We can implement a feature to read in an interest rate and the number of years expected to pay off the loan.

After taking all this data as input, we want to calculate the yearly cost of such an education, the four year cost, the monthly loan payment, and the total cost of the loan over time. Furthermore, we want to output this information on the command line in an attractive way, customized with the user's name. Below is a sample execution of this program.

```
Welcome to the College Cost Calculator!
```

```
Enter your first name:      Barry
Enter your last name:       Wittman
Enter tuition per semester: $174.15
Enter rent per month:       $350
Enter food cost per month:  $400
Annual interest rate:      .0937
Years to pay back your loan: 10
```

```
College costs for Barry Wittman
```

```
*****
```

```
Yearly cost:          $43830.00
Four year cost:        $175320.00
Monthly loan payment: $2256.14
Total loan cost:       $270736.57
```

Samples from a command line interface can be confusing because it is difficult to see what is output and what is input. In this case, we have marked the input in green so that it looks similar the way that Eclipse marks user input. In this program, the names, the tuition, the rent, the food, the interest rate, and the years to pay back the loan are taken as input. Note that the dollar signs are not part of the input and are outputted as a cue to the user and to give visual polish.

If you are following the software development lifecycle, we hope you have a good understanding of this problem, but there are a few mathematical details worth addressing. First, the yearly cost is twice the semester tuition plus 12 times the rent and food costs. The four year cost is simply four times the yearly cost. The monthly loan payment amount, however, requires a formula from financial mathematics. Let P be the amount of the loan (the principal). Let J be the monthly interest rate (the annual interest rate divided by 12). Let N be the number of months to pay back the loan (years of the loan times 12). Let M be the monthly payment we want to calculate, then:

$$M = P \times \frac{J}{1 - (1 + J)^{-N}}$$

If you use the concepts and syntax from the previous chapter carefully, you might be able to solve this problem without reading further. However, there is a depth to the ideas of types and operations that we have not explored fully. Getting a program to work is not enough. Programmers must understand every detail and quirk of their tools to avoid potential bugs.

3.2 Concepts: Types

Every operation inside of a Java program manipulates data. Often, this data is stored in *variables*, which look similar to variables from mathematics. Consider the following mathematical equation.

$$x + 3 = 7$$

In this equation, x has the value 4, and it always will. You can set up another equation in which x has a different value, but it won't change in this one. Java variables are different. They are locations where you can store something. If you decide later that you want to change what you stored there, you can put something else into the same location, overwriting the old value.

In contrast to a variable is a *literal*. A literal is a concrete value that does not change, though it can be stored in a variable. Numbers like 4 or 2.71828183 are literals. We need to represent text and single characters in Java as well, and there are literals like “grapefruit segment” and ‘X’ that fill these roles.

In Java, both variables and literals have *types*. If you think of a variable as a box where you can hold something, its type is the shape of the box. The kinds of literals that can be stored in that box must have a matching shape. In the last chapter, we introduced the type `int` to store integer values and the type `double` to store floating point values. Java is a *strongly typed* language, meaning that, if we declare a variable of type `int`, we can only put `int` values into it (with a few exceptions).

This idea of a type takes some getting used to. From a mathematical perspective 3 and 3.0 are identical. However, if you have an `int` variable in Java, you can store 3 into it, but you can't store 3.0. Types never change, but you can convert a value from one type to an equivalent value in another type.

3.2.1 Types as sets and operations

Before we go any further, let's look deeper at what a type really is. We can think of a type as a set of elements. For example, `int` represents a set of integer values (specifically, the integers between –

2,147,483,648 and 2,147,483,647). Consider the following declarations:

```
int x;  
int y;
```

These declarations indicate that the variables named x and y can only contain integer values in this range. Furthermore, a type only allows specific operations. In Java, the `int` type allows addition, subtraction, multiplication, division, and several other operations we will talk about in [Section 3.3](#), but you cannot directly raise an `int` value to a power in Java. Let's assume that x is of type `int`. As we discussed in the previous chapter, the expression `x + 2` performs addition between the variable represented by x and the literal 2. Some languages use the operator `^` to mean raising a number to a power. Following this notation, some beginning Java programmers are tempted to write `x ^ 2` to compute x squared. The `^` operator does have a meaning in Java, but it does not raise values to a power. Other combinations of operators are simply illegal, such as `x # 2`.

The idea of using types this way gives structure to a program. All operations are well-defined. For example, you know that adding two `int` values together will give you another `int` value. Java is a *statically typed* language. This means that it can analyze all the types you're using in your program at compile-time and warn you if you're doing something illegal. Consequently, you'll get a lot of compiler warnings and errors, but you can be more confident that your program is correct if all the types make sense.

3.2.2 Primitive and reference types in Java

As shown in [Figure 3.1\(a\)](#), there are two kinds of types in Java: *primitive types* and *reference types*. Primitive types are like boxes that hold single, concrete values. The primitive types in Java are `byte`, `short`, `int`, `long`, `float`, `double`, `boolean`, and `char`. These are types provided by the designers of Java, and it is not possible to create new ones. Each primitive type has a set of *operators* that are legal for it. For example, all the numerical types can be used with `+` and `-`. We'll talk about these types and their operators in great detail in [Section 3.3](#).

Reference types work differently from primitive types. For one thing, a reference variable **points at** an object. This means that when you assign one reference variable to another, you are not getting a whole new copy of the object. Instead, you're getting another arrow that points at the same object. The result is that performing an operation on one reference can effectively change another reference, if they are both pointing at the same object.

Another difference is that reference variables (or simply *references*) do not have a large set of operators that work on them. Every variable can be used with the assignment (`=`) and the comparison (`==`) operators. Every variable can also be concatenated with a String by using the `+` operator, but even if two objects represent numerical values, they cannot be added together with the `+` operator.

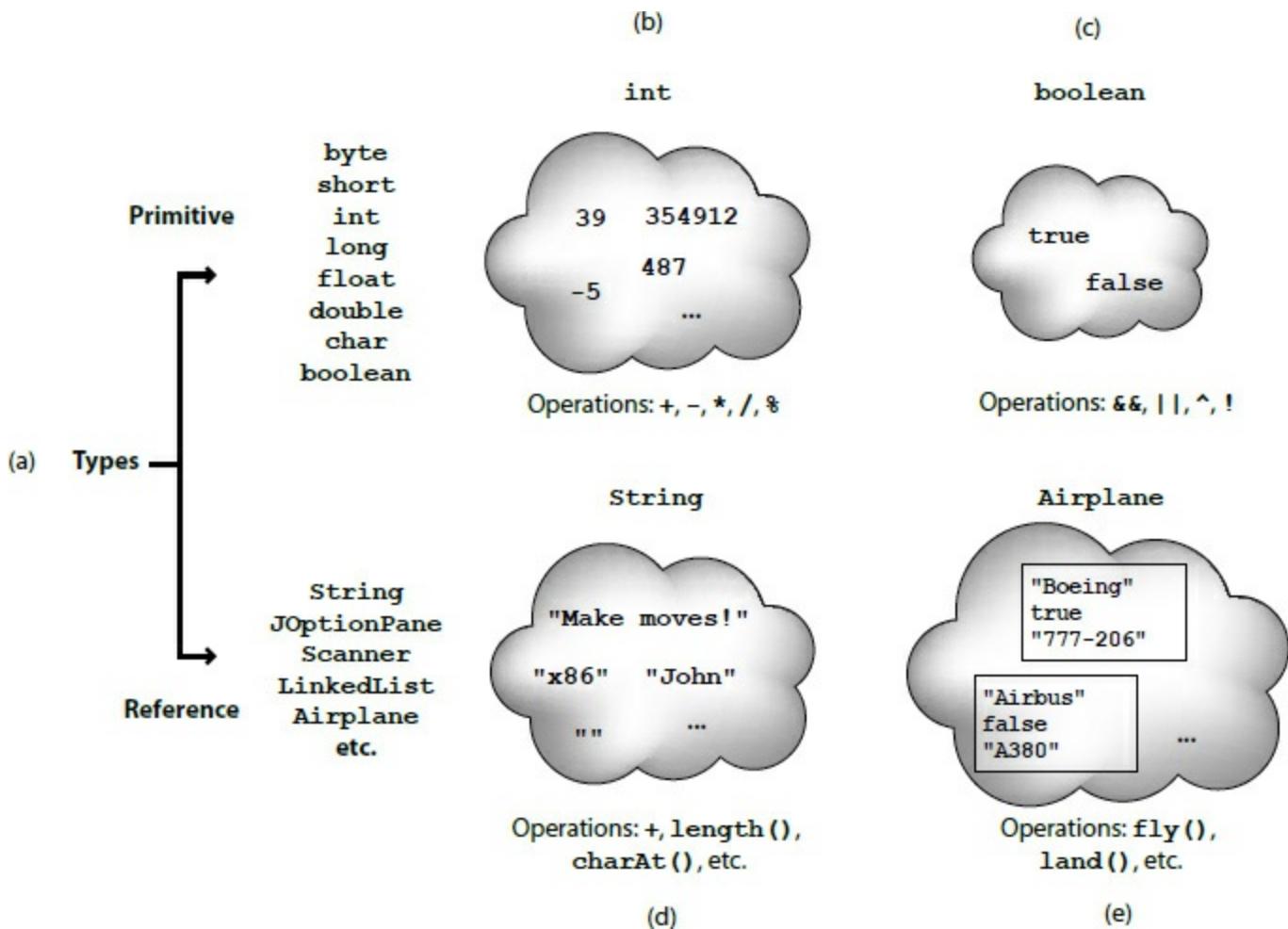


Figure 3.1: (a) The two categories of types in Java. The `int` primitive type (b), the `boolean` primitive type (c), the `String` reference type (d), and a possible Aircraft reference type (e) are represented as sets of items with operations.

References should still be thought of as types defining a set of objects and operations that can be done with them. Instead of using operators, however, references use *methods*. You have seen methods such as `System.out.print()` and `JOptionPane.showInputDialog()` in the previous chapter. A method generally has a dot (.) before it and always has parentheses afterwards. These parentheses contain the input *parameters* or *arguments* that you give to a method. Using operators on primitive types is convenient, but on the other hand, there is no limit to the number, kind, or complexity of methods that can be used on references.

Another important feature of reference types is that anyone can define them. So far, you have seen the reference types `String` and `JOptionPane`. As we will discuss later, `String` is an unusual reference type in that it is built deeply into the language. There are a few other types like this (such as `Object`), but most reference types could have been written by anyone, even you.

To create a new type, you write a class and define data and methods inside of it. If you wanted to define a type to hold airplanes, you might create the `Airplane` class and give it methods such as `takeOff()`, `fly()`, and `land()` because those are operations that any airplane should be able to do.

Once a class has been defined, it is possible to *instantiate* an *object*. An object is a specific instance of the type. For example, the type might be `Airplane`, but the object might be referenced by a variable called `sr71Blackbird`. Presumably, this object has a weight, a maximum speed, and other

characteristics that mark it as a Lockheed SR-71 “Blackbird,” the famous spy plane. To summarize: The object is a concrete instance of the data. The reference is the variable that gives a name to (points to) the object. The type is the class that both the variable and the object have, which defines what kinds of data the object has and what operations it can perform.

Exercise 3.12

The following table lists some of the differences between primitive types and reference types.

Primitive Types	Reference Types
Created by the designers of Java	Created by any Java programmer
Use operators to perform operations	Use methods to perform operations
There are only eight different primitive types	The number of reference types is unlimited and grows every time someone creates a new class
Hold a specific numbers of bytes of data depending on the type	Can hold arbitrary amounts of data
Assignment copies a value from one place to another	Assignment copies an arrow that points at an object
Declaration creates a box to hold values	Declaration creates an arrow that can point at an object, but only instantiation creates a new object

3.2.3 Type safety

Why do we have types? There are weakly typed languages where you can store any value into almost any variable. Why bother with all these complicated rules? Most assembly languages have no notion of types and allow the programmer to manipulate memory directly.

Because Java is strongly typed, the type of every variable, whether primitive or reference, must be declared prior to its use. This constraint allows the Java compiler to perform many safety and sanity checks during compilation, and the JVM performs a few more during execution. These checks avoid errors during program execution that might otherwise be hard to find. These errors could lead to catastrophic failures of the program.

The Ariane 5 rocket is an example of a catastrophic failure due to a type error. On its first flight, the rocket left its flight path and eventually exploded. The failure was caused because of errors that resulted after converting a 64-bit floating point to 16-bit signed integer value. The converted value was larger than the integer could hold, resulting in a meaningless value.

Converting from one type to another is called *casting*. The Ariane 5 failure was due to a problem with casting that was not caught. Even in Java, it is possible for a human being to circumvent type safety with irresponsible casting.

3.3 Syntax: Types in Java

In this section we will dig deeper into the type system in Java, starting with variables and moving on

to the properties of the eight primitive types and the properties of String and other reference types.

3.3.1 Variables and literals

To use a variable in Java, you must first *declare* it, which sets aside memory to hold the variable and attaches a name to that space. Declarations always follows the same pattern. The type is written first followed by the identifier, or name, for the variable. Below we declare a variable named value of type int.

```
int value ;
```

Note how we use the same pattern to declare a reference variable named creature of type Wombat.

```
Wombat creature ;
```

You are always free to declare a variable and then end the line with a semicolon (;), but it is common to *initialize* a variable at the same time. The following line simultaneously declares value and initializes it to 5.

```
int value = 5;
```

Pitfall: Multiple declarations

Don't forget that you are declaring and initializing in a line like the above. Beginning Java programmers sometimes try to declare a variable more than once, as in the following:

```
int value = 5;  
int value = 10;
```

Java will not allow two variables with the same name to exist in the same block of code. The programmer probably intended the following, which reuses variable value and replaces its contents with 10.

```
int value = 5;  
value = 10;
```

This error is more common when several other lines of code are between the two assignments.

In some of the examples above, we have stored the value 5 into our variable value. The symbol 5 is an example of a *literal*. A literal is a value represented directly in code. It cannot be changed, but it can be stored into variables that have the same type. The values stored into variables come from literals, input, or more complicated expressions. Just like variables, literals have types. The type of 5 is int while the type of 5.0 is double. Other types have literals written in ways we'll discuss below.

3.3.2 Primitive types

The building blocks of all Java programs are primitive types. Even objects must fundamentally contain primitive types deep down inside. There are eight primitive types. Half of them are used to

represent integer values, and we'll start by looking at those.

Integers: byte, short, int, and long

A variable intended to hold integer values can be declared with any of the four types `byte`, `short`, `int`, or `long`. All of them are signed (holding positive and negative numbers) and represent numbers in two's complement. They only differ by the range of values that each type can hold. These ranges and the number of bytes used to represent variables from each type are given in [Table 3.1](#).

Type	Bytes	Range		
<code>byte</code>	1	-128	to	127
<code>short</code>	2	-32,768	to	32,767
<code>int</code>	4	-2,147,483,648	to	2,147,483,647
<code>long</code>	8	-9,223,372,036,854,775,808	to	9,223,372,036,854,775,807

Table 3.1: Ranges for primitive integer types in Java.

Note that the range of `byte` is included in that of `short`, of `short` in that of `int`, and so on. We say that `short` is *broader* than `byte`, `int` is broader than `short`, and `long` is broader than `int`.

Exercise 3.1

A variable declared with type `byte` can only represent 256 different values, the integers in the range -128 to 127. Why use `byte` at all, then? Since a `byte` value only takes up a single byte, it can save memory, especially if you have a list of variables called an *array*, which we will discuss in [Chapter 6](#). However, too narrow of a range will result in underflow and overflow. Java programmers are advised to stick with `int` for general use. If you need to represent values larger than 2 billion or smaller than -2 billion, use `long`. Once you are an experienced programmer, you may occasionally use `byte` and `short` to save space, but they should be used sparingly and for a clear purpose.

Example 3.1: Integer variables

Consider the following declarations.

```
byte age ;  
int numberOfRooms ;  
long foreverCounter = 0;
```

The first of these statements declares `age` to be a variable of type `byte`. This declaration means that `age` can assume any value from the range for `byte`. For a human being, this limitation is reasonable (but dangerously close to the limit) since there is no documented case of a person living more than 122 years. Similarly, the next declaration declares `numberOfRooms` to be of type `int`. The last declaration declares `foreverCounter` to be of type `long` and initializes it to 0.

Since `age` is a variable, its value can change during program execution. Note that the above

declaration of age does not assign a value to it. When they are declared, all integer variables are set to 0 by Java. However, to make sure that the programmer is explicit about what he or she wants, the compiler will give an error in most cases if a variable is used without first having its value set.

Like any other integer variable, we can assign age a value as follows.

```
age = 32;
```

Doing so assigns the value 32 to variable age. Note that the Java compiler would not complain if you were to assign -10 to the variable age, even though it is impossible for a human to have a negative age (at least, without a time machine). Java attaches no meaning to the name you give to a variable. ■

Earlier, we said that variables had to match the type of literals you want to store into them. In the example above, we declared age with type **byte** and then stored 32 into it. What is the type of 32? Is it **byte**, **short**, **int**, or **long**? By default, all integer literals have type **int**, but they can be used with **byte** or **short** variables provided that they fit within the range. Thus, the following line causes an error.

```
byte fingers = 128;
```

If you want to specify a literal to have type **long**, you can append l or L to it. Thus, 42 is an **int** literal, but 42L is a **long** literal. You should always use L since l can be difficult to distinguish from 1.

At the time of this writing, Java 7 is the newest version of Java, but it has not yet become popular or widespread. In Java 7, you are allowed to put any number of underscores (_) inside of numerical literals to break them up for the sake of readability. Thus, 123_45_6789 might represent a social security number, or you could use underscores instead of commas to write three million as 3_000_000. To use this syntax, you must have a Java 7 compiler and be sure that your code will never need to be compiled in an earlier version of Java. Note that you should **never** use a comma in a numerical Java literal, no matter which version of Java.

Floating point numbers: **float** and **double**

To represent numbers with fractional parts, Java provides two floating point types, **double** and **float**. Because of limits on floating point precision discussed in [Chapter 1](#), Java cannot represent all rational or real numbers, but these types provide good approximations. If you have a variable that takes on floating point values such as 3.14, 1.707×10^{25} , 9.8, and so on, it ought to be declared as a **double** or a **float**.

Example 3.2: Floating point declarations

Consider the following declarations.

```
float roomArea ;  
double avogadro = 6.02214179 E23
```

The first of the above two statements declares roomArea to be of type **float**. Note that the declaration does not initialize roomArea to any value. Similar to integer primitive types, an

uninitialized floating point variable contains 0.0, but Java usually forces the programmer to assign a value to a variable before using it. The second of the above two statements declares `avogadro` to be a variable of type `double` and initializes it to the well-known Avogadro constant $6.02214179 \times 10^{23}$. Note the use of E to mean “ten to the power of.” In Java, you could write 0.33×10^{-12} as `0.33E-12`, or the number -4.325×10^{18} as `-4.325E18` (or even `-4.325E+18` if you would like to write the sign of the exponent explicitly). ■

Accuracy in number representation

As discussed in [Chapter 1](#), integer types within their specified ranges have their exact representations. For example, if you assign 19 to a variable of type `int` and then print this value, you always get exactly 19. Floating point numbers do not have this guarantee of exact representation.

Example 3.3: Floating point accuracy

Try executing the following statements from within a Java program.

```
double test = 0.0;  
test += 0.1;  
System.out.println (test);  
test += 0.1;  
System.out.println (test);  
test += 0.1;
```

Since we are adding 0.1 each time, one would expect to see outputs of 0.1, 0.2, and 0.3. The first two numbers print as expected, but the third number prints out as 0.3000000000000004. It may seem counterintuitive, but 0.1 is a repeating decimal in binary, meaning that it cannot be represented exactly using the 64-bit IEEE floating point standard. The `System.out.println()` method hides this ugliness by rounding the output past a certain level of precision, but by the third addition, the number has drifted far enough away from 0.3 that an unexpected number peeks out. ■

Variables of type `float` give you an accuracy of about 6 decimal digits while those of type `double` give about 15 decimal digits. Does the accuracy of floating point number representation matter? The answer to this question depends on your application. In some applications, 6-digit accuracy may be adequate. However, when doing large-scale simulations, such as computing the trajectory of a spacecraft on a mission to Mars, 15-digit accuracy might be a matter of life or death. In fact, even `double` precision may not be enough. There is a special `BigDecimal` class which can perform arbitrarily high precision calculations, but due to its low speed and high complexity, it should only be used in those rare situations when a programmer requires a much higher level of precision than what `double` provides.

Java programmers are recommended to use `double` for general purpose computing. The `float` type should only be used in special cases where storage or speed are critical and accuracy is not. Because of its greater accuracy, `double` is considered a broader type than `float`. You can store `float` values in a `double` without losing precision, but the reverse is not true.

All floating point literals in Java have type `double` unless they have an f or F appended on the end. Thus, 3.14 is a `double` literal, but 3.14f is a `float` literal.

Floating point output

Formatting output for floating point numbers has an extra complication compared to integers: How many digits after the decimal point should be displayed? If you are representing money, it is common to show exactly two digits after the decimal point. By default, all of the non-zero digits are shown.

Instead of using `System.out.print()`, you can use `System.out.format()` to control formatting. When using `System.out.format()`, the first argument to the method is a *format string*, a piece of text that gives all the text you want to output as well as special format specifiers that indicate where other data is to appear and how it should be formatted. This method takes an additional argument for each format specifier you use. The specifier `%d` is for integer values, the specifier `%f` is for floating point values (including both `float` and `double` types), and the specifier `%s` is for text. Consider the following example:

```
System.out.format ("%s! I broke %d records in %f seconds.\n",
  "Bob", 3, 2.4985);
```

The output of this code is

Bob! I broke 3 records in 2.4985 seconds.

This kind of output is based on the `printf()` function used for output in the C programming language. It allows the programmer to have a holistic picture of what the final output might look like, but it also gives control of formatting through the format specifiers. For example, you can choose the number of digits for a floating point value to display after the decimal point by putting a `.` and the number between the `%` and the `f`.

```
System.out.format ("$ %.2 f\n", 123.456789);
```

The output of this code is

\$123.46

rounding the last digit appropriately. To learn about other ways to use format strings to manipulate output, read the documentation at <http://download.oracle.com/javase/7/docs/api/java/util/Formatter.html#syntax>.

Basic arithmetic

The following table lists the arithmetic operators available in Java. All of these operators can be used on both the integer primitive types and the floating point primitive types.

Operator	Meaning
+	Add
-	Subtract
*	Multiply
/	Divide
%	Modulus (remainder)

The first four of these should be familiar to you. Addition, subtraction, and multiplication work as you would expect, provided that the result is within the range defined for the types you're using, but division is a little confusing. If you divide two integer values in Java, you'll get an integer as a result. If there would have been a fractional part, it will be truncated, not rounded. Consider the following.

```
int x = 1999/1000;
```

In normal mathematics, $1,999 \div 1,000 = 1.999$. In Java, $1999/1000$ yields 1, and that's what is stored in x. For floating point numbers, Java works much more like normal mathematics.

```
double y = 1999.0/1000.0;
```

In this case, y contains 1.999. The literals 1999.0 and 1000.0 have type **double**. The type of y does not affect the division, but it had to be **double** to be a legal place to store the result.

Pitfall: Unexpected integer division

It's easy to focus on the variable and forget about the types involved in the operation. Consider the following.

```
double z = 1999/1000;
```

Because z has type **double**, it seems that the result of the division should be 1.999. However, the dividend and the divisor have type **int**, and the result is 1. This value is converted into **double** and stored in z as 1.0. This mistake is more commonly seen in the following scenario.

```
double half = 1/2;
```

The code looks fine at first, but 1/2 yields 0. If the result is to be stored in a **double** variable, it is better to multiply by 0.5 instead of dividing by 2.

You may not have thought about this idea since elementary school, but the division operator (/) finds the quotient of two numbers. The modulus operator (%) finds the remainder. For example, $15/6$ is 2, but $15 \% 6$ is 3 because 6 goes into 15 twice with 3 left over. The modulus operator is usually used with integer values, but it is also defined to work with floating point values in Java. It's easy to dismiss the modulus operator because we don't often use it in daily life, but it is incredibly useful in programming. On its face, it allows us to see the remainder after division. This idea can be applied to see if a number is even or odd. It can also be used to compress a large range of random integers to a

smaller range. Keep an eye out for it. We'll use it many times in this book.

Precedence

Although all the previous examples use only one mathematical operator, you can combine several operators and operands into a larger expression like the following.

```
((a + b) * (c + d)) \% e
```

Such expressions are evaluated from left to right, using the standard order of operations: The `*` and `/` (and also `%`) operators are given precedence over the `+` and `-` operators. Like in mathematics, parentheses have the highest precedence and can be used to add clarity. Thus, the order of evaluation of `a + b / c` is the same as `a + (b / c)` but different from `(a + b) / c`.

Example 3.4: Order of operations

Consider the following lines of code.

```
int a = 31;  
int b = 16;  
int c = 1;  
int d = 2;  
a = b + c * d - a / b / d;
```

What is the result? The first operation to be evaluated is `c * d`, yielding 2. The next is `a / b`, yielding 1, which is then divided by `d`, yielding 0. Next `b + 2` gives 18, and `18 - 0` is still 18. Thus, the value stored in `a` is 18.

Your inner mathematician might be nervous that `a` is used in the expression on the right side of the assignment and is also the variable where the result is stored. This situation is very common in programming. The value of `a` doesn't change until after all the math has been done. The assignment always happens last. ■

All of the operators we have discussed so far are *binary* operators. This use of the word "binary" has nothing to do with base 2. A binary operator takes two things and does something, like adding them together. A *unary* operator takes a single operand and does something. The `-` operator can be used as a unary operator to negate a literal, variable, or expression. A unary negation has a higher precedence than the other operators, just like in mathematics. In other words, the variable or expression will be negated before it is multiplied or divided. The `+` operator can be used anywhere you would use a unary negation, although it doesn't actually do anything. Consider the following statements.

```
int a = - 4;  
int b = -c + d * -(e * f);  
int s = +-t + (-r);
```

Shortcuts

Some operations happen frequently in Java. For example, increasing a variable by some amount is a

common task. If you want to increase the value of variable value by 10, you can write the following.

```
value = value + 10;
```

Although the statement above is not excessively long, increasing a variable is common enough that there's shorthand for it. To achieve the same effect, you can use the `+=` operator.

```
value += 10;
```

The `+=` operator gets the value of the variable, in this case `value`, adds whatever is on its right side, in this case `10`, and stores the result back into the variable. Essentially, it saves you from writing the name of the variable twice. And `+=` is not the only shortcut. It is only one member of a family of shortcut operators that perform a binary operation between the variable on the left side and the expression on the right side and then store the value back into the variable. There is a `-=` operator that decreases a variable, a `*=` operator that scales a variable, and several others, including shortcuts for bitwise operations we cover in the next subsection.

Operator	Example	Meaning
<code>+=</code>	<code>a += b;</code>	<code>a = a + b;</code>
<code>-=</code>	<code>a -= b;</code>	<code>a = a - b;</code>
<code>*=</code>	<code>a *= b;</code>	<code>a = a * b;</code>
<code>/=</code>	<code>a /= b;</code>	<code>a = a / b;</code>
<code>%=</code>	<code>a %= b;</code>	<code>a = a % b;</code>
<code>&=</code>	<code>a &= b;</code>	<code>a = a & b;</code>
<code>^=</code>	<code>a ^= b;</code>	<code>a = a ^ b;</code>
<code> =</code>	<code>a = b;</code>	<code>a = a b;</code>
<code><<=</code>	<code>a <<= b;</code>	<code>a = a << b;</code>
<code>>>=</code>	<code>a >>= b;</code>	<code>a = a >> b;</code>
<code>>>>=</code>	<code>a >>>= b;</code>	<code>a = a >>> b;</code>

These assignment shortcuts are useful and can make a line shorter and easier to read.

Pitfall: Weak type checking with assignment shortcuts

Because you can lose precision, it is not allowed to store a `double` value into an `int` variable.

Thus, the following lines of code are illegal and will not compile.

```
int x = 0;  
x = x + 0.1;
```

In this case, the check makes a lot of sense. If you were able to add 0.1 to 0 and then store that value into an `int` variable, the fractional part would be truncated, keeping 0 in the variable. However, this safeguard against lost precision is not done with assignment shortcuts. Even though we expect the following lines to be functionally identical to the previous ones, they will compile (but still do nothing).

```
int x = 0;  
x += 0.1;
```

This kind of error can cause problems when the program expects the value of `x` to grow and eventually reach some level.

There are also two unary shortcuts. Incrementing a value by one and decrementing a value by one are such common operations that they get their own special operators, `++` and `--`.

Operator	Example	Meaning
<code>++</code>	<code>a++;</code>	<code>a = a + 1;</code>
<code>--</code>	<code>a--;</code>	<code>a = a - 1;</code>

Using either an increment or decrement changes the value of a variable. In all other cases, the use of an assignment operator is required to change a variable. Even in the binary shortcuts given before, the programmer is reminded that an assignment is occurring because the `=` symbol is present.

Both the increment and decrement operators come in prefix and postfix flavors. You can write the `++` (or the `--`) in front of the variable you are changing or behind it.

```
int value = 5;  
value++; //now value is 6  
++value; //now value is 7  
value--; //value is 6 again
```

When used in a line by itself, each flavor works exactly the same. However, the incremented (or decremented) variable can also be used as part of a larger expression. In a larger expression, the prefix form increments (or decrements) the variable **before** the value is used in the expression. Conversely, the postfix form gives back a copy of the original value, effectively incrementing (or decrementing) the variable **after** the value is used in the expression. Consider the following example.

```
int prefix = 7;  
int prefixResult = 5 + ++prefix;  
  
int postfix = 7;  
int postfixResult = 5 + postfix++;
```

After the code is executed, the values of `prefix` and `postfix` are both 8. However, `prefixResult` is 13 while `postfixResult` is only 12. The original value of `postfix`, which is 7, is added to 5, and then the increment operation happens afterwards.

Pitfall: Increment confusion

Incrementing a variable in Java is a very common operation. Expressions like `i++` and `++i` pop up so often that it is easy to forget exactly what they mean. Programmers occasionally forget that they are shorthand for `i = i + 1` and begin to think of them as a fancy way to write `i + 1`.

When confused, a programmer might write something like the following.

```
int i = 14;  
i = i++;
```

At first glance, it may appear that the second line of code really means `i = i = i + 1`. Assigning `i` an extra time is pointless, but it does no harm. However, remember that the postfix version gives back a copy of the original value, before it has been incremented. In this case, `i` will be incremented, but then its original value will be stored back into itself. In the code given above, the final value of `i` is still 14.

In general it is unwise to perform increment or decrement operations in the middle of larger expressions, and we advise against doing so. In some cases, code can be shortened by cleverly hiding an increment in the middle of some other expression. However, when reading back over the code, it always takes a moment to be sure that increment or decrement is doing exactly what it should. The additional confusion caused by this cleverness is not worth the line of code saved. Furthermore, the compiler will translate the operations into exactly the same bytecode, meaning that the shorter version is no more efficient than the longer version.

Nevertheless, many programmers enjoy squeezing their code down to the smallest number of lines of code possible. You may have to read code that uses increments and decrements in clever (if obscure) ways, but you should always strive to make your own code as readable as possible.

Bitwise operators

In addition to normal mathematical operators, Java provides a set of *bitwise* operators corresponding to the operations we discussed in [Chapter 1](#). These operators perform bitwise operations on integer values. The bitwise operators are `&`, `|`, `^`, and `~` (which is unary). In addition, there are bitwise *shift* operators: `<<` for signed left shift, `>>` for signed right shift, and `>>>` for unsigned right shift. There is no unsigned left shift operator in Java.

Operator	Name	Description
&	Bitwise AND	Combines two binary representations into a new representation that has 1s in every position that both the original representations have a 1
	Bitwise OR	Combines two binary representations into a new representation that has 1s in every position that either of the original representations have a 1
^	Bitwise XOR	Combines two binary representations into a new representation that has 1s in every position that the original representations have different values
~	Bitwise NOT	Takes a representation and creates a new representation in which every bit is flipped from 0 to 1 and 1 to 0
<<	Signed left shift	Moves all the bits the specified number of positions to the left, shifting 0s into the rightmost bits
>>	Signed right shift	Moves all the bits the specified number of positions to the right, padding the left with copies of the sign bit
>>>	Unsigned right shift	Moves all the bits the specified number of positions to the right, padding with 0s

When used with **byte** and **short**, all bitwise operators will automatically convert their operands to 32-bit **int** values. It is crucial to remember this conversion since the number of bits used for representation is a fundamental part of bitwise operators.

The following example shows these operators in use. In order to understand the output, you need to understand how integers are represented in the binary number system, which is discussed in [Section 1.3](#).

Exercise 3.5

Example 3.5: Binary operators in Java

The following code shows a sequence of bitwise operations performed with the values 3 and -7. To understand the results, remember that, in 32-bit two's complement representation, $3 = 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0011$ and $-7 = 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1001$.

```
int x = 3;
int y = -7;
int z = x & y;
System.out.println ("x & y\t= " + z);
z = x | y;
System.out.println ("x | y\t= " + z);
```

```
z = x ^ y;  
System.out.println ("x ^ y\t= " + z);  
z = x << 2;  
System.out.println ("x << 2\t= " + z);  
z = y >> 2;  
System.out.println ("y >> 2\t= " + z);  
z = y >>> 2;  
System.out.println ("y >>> 2\t= " + z);
```

The output of this fragment of code is:

```
x & y    = 1  
x | y    = -5  
x ^ y    = -6  
x << 2   = 12  
y >> 2   = -2  
y >>> 2 = 1073741822
```

Note how the escape sequence \t is used to put a tab character in the output, making the results line up. ■

Why use the bitwise operators at all? Sometimes you may read data as individual **byte** values, and you might need to combine four of these values into a single **int** value. Although the signed left shift (<<) and signed right shift (>>) are, respectively, equivalent to repeated multiplications by 2 or repeated divisions by 2, they are faster than doing these operations over and over. Finally, some of these operations are used for cryptographic or random number generation purposes.

Casting

Sometimes you need to use different types (like integers and floating point values) together. Other times, you have a value in one type, but you need to store it in another (like when you are rounding a **double** to the nearest **int**). Some combinations of operators and types are allowed, but others cause compiler errors.

The guiding rule is that Java allows an assignment from one type to another, provided that no precision is lost. That is, we can copy a value of one type into a variable of another type, provided that the destination variable has a broader type than the source value. The next few examples illustrate how to convert between different numerical types.

Example 3.6: Upcast with integers

Consider the following statements.

```
short x = 341;  
int y = x;
```

Because the type of y is **int**, which is broader than **short**, the type of x, it is legal to assign the value

in x to variable y. In the assignment, a value with the narrower type `short` is converted to an equivalent value with the broader type `int`. Converting from a narrower type to a broader type is called an *upcast* or a *promotion*, and Java allows it with no complaint. Most languages allow upcasts without any special syntax because it is always safer to move from a narrower, more restrictive type to a broader, less restrictive one. ■

Example 3.7: Downcast error

Consider these statements that declare variables a, b, and c and compute a value for c.

```
int a = 10;  
int b = 2;  
byte c;  
c = a + b;
```

If you try compiling these statements as part of a Java program, you get an error message like the following.

```
Error: possible loss of precision  
found: int  
required: byte
```

The compiler generates the error above because the sum of two `int` values is another `int` value, which could be greater than the maximum value you can store in c, of type `byte`. In this example, you know that the value of 12 does not exceed the maximum of 127, but the Java compiler is inherently cautious. It complains whenever the type of the expression to be evaluated is broader than the type of the destination variable. ■

Exercise 3.2

Example 3.8: Upcast from integers to floating point

Integers are automatically converted to floating point when needed. Consider the following statement.

```
double tolerance = 3;
```

The literal 3 has type `int`, but it is automatically converted to the floating point value 3.0 with type `double`. Again, `double` (and also `float`) are considered broader types than any integer types. Consequently, this type conversion is an upcast and is completely legal.

Upcasts also occur with arithmetic operations. Whenever you try to do arithmetic with two different numerical types, the narrower type is automatically upcast to the broader one.

```
double value = 3 + 7.2;
```

In this statement, 3 is automatically upcast to its `double` version 3.0 because 7.2 has the broader `double` type. ■

In order to perform a downcast, the programmer has to mark that he or she intends for the

conversion to happen. A downcast is marked by putting the result type in parentheses before the expression you want converted. The next example illustrates how to cast a **double** value to type **int**.

Example 3.9: Downcast from double to int

The following statements cause a compiler error because an expression with type **double** cannot be stored into a variable with type **int**.

```
double roomArea = 3.5;  
int houseArea = roomArea * 4.0;
```

A downcast can lose precision, and that's why Java doesn't allow it. Sometimes a downcast is necessary, and you can override Java's type system with an explicit cast. To do so, we put the expected (or desired) result type in parentheses before the expression. In this case (and many others), it is also necessary to surround the expression with parentheses so that the entire expression (and not just `roomArea`) is converted to type **int**.

```
double roomArea = 3.5;  
int houseArea = ( int ) ( roomArea * 4.0 ) ;
```

In this case, the expression has value 14.0. Consequently, the **int** version is 14. In general, the value could have a fractional part. When casting from a floating point type to an integer type, the fractional part is truncated **not** rounded. Consider the following statement:

```
int count = ( int ) 15.99999;
```

Mathematically, it seems obvious that 15.99999 should be rounded to the nearest **int** value of 16, but Java does not do this. Instead, the code above stores 15 into `count`. If you want to round the value, Java provides a method for rounding in the `Math` class. The rounding (instead of truncating) version is given below.

```
int count = ( int ) Math.round ( 15.99999 ) ;
```

The value given back by `Math.round()` has type **long**. The designers of the `Math` class did this so that the same method could be used to round large **double** values into a **long** value, since the result might not fit in an **int** value. Since **long** is a broader type than **int**, we have to downcast the result to an **int** so that we can store it in `count`. ■

Exercise 3.10

Example 3.10: Conversion from double to float

Consider the following declaration and assignment of variable `roomArea`.

```
float roomArea ;  
roomArea = 2.0;
```

This assignment is illegal in Java, and the compiler gives an error message like the following:

```
Error: possible loss of precision
found: double
required: float
```

As we mentioned earlier, the literal 2.0 has type **double**. When you try to assign a **double** value to a **float** variable, there is always a risk that precision will be lost. The best way to avoid the error above is to declare roomArea with type **double**. Alternatively, we could store the **float** literal 2.0f into roomArea. We could also assign 2 instead of 2.0 to roomArea, since the upcast from **int** is done automatically.

Exercise 3.3

Remember, you should almost always use the **double** type to represent floating point numbers. Only in rare cases when you need to save memory should you use **float** values. By making it illegal to store 2.0 into a **float** variable, Java is encouraging you to use high precision storage. ■

Numerical types and the conversions between them are critical elements of programming in Java, which has a strong mathematical foundation. In addition to these numerical types, Java also provides two other types that represent individual characters and Boolean values. We examine these next.

Characters: **char**

Sentences are made up of words. Words are made up of letters. Although we have discussed many powerful tools for representing numbers in Java, we need a way to represent letters and other characters that we might find in printed text. Values with the **char** type are used to represent individual letters.

In the older languages of C and C++, the **char** type used 8 bits for storage. From [Chapter 1](#), you know that you can represent up to $2^8 = 256$ values with 8 bits. The Latin alphabet, which is used to write English, uses 26 letters. If we need to represent upper and lower case letters, the 10 decimal digits, punctuation marks, and quite a few other special symbols, 256 values is plenty. However, people all over the world use computers and want to store text from their language written in their script digitally. Taking the Chinese character system alone, some Chinese dictionaries list over 100,000 characters!

Java uses a standard called the UTF-16 encoding to represent characters. UTF-16 is part of a larger international standard called Unicode, which is an attempt to represent most of the world's writing systems as numbers that can be stored digitally. Most of the inner workings of Unicode aren't important for day-to-day Java programming, but you can visit <http://www.unicode.org/> if you want more information.

In Java, each variable of type **char** uses 16 bits of storage. Therefore, each character variable could assume any value from among a total of $2^{16} = 64,768$ possibilities (although a few of these are not legal characters). Here are a few declarations and assignments of variables of type **char**.

```
char letter = 'A';
char punctuation = '?';
char digit = '7';
```

We are storing **char** literals into each of the variables above. Most of the **char** literals you will use commonly are made by typing the single character you want in **single** quotes ('), such a '**z**'. These characters can be upper- or lowercase letters, single numerical digits, or other symbols.

The space character literal is ' ', but some characters are harder to represent. For example, a new line (the equivalent of pressing <enter>) is represented as a single character, but we can't type a single quote, hit <enter>, and then type the second quote. Instead, the character to represent a new line is '\n', which we will refer to simply as a *newline*. Every **char** variable can only hold a single character. It appears that '\n' has multiple characters in it, but it does not. The use of the backslash (\) marks an *escape sequence*, which is a combination of characters used to represent a specific difficult to type or represent character. Here is a table of common escape sequences.

Escape Sequence	Character
\n	Newline
\t	Tab
\'	Single quote
\\	Backslash

Remember, everything inside of a computer is represented with numbers, and each **char** value has some numerical equivalent. These numbers are arbitrary but systematic. For example, the character '**a**' has a numerical value of 97, and '**b**' has a numerical value of 98. The codes for all of the lowercase Latin letters are sequential in alphabetical order. (The codes for uppercase letters are sequential too, but there is a gap between them and the lowercase codes.)

Some Unicode characters are difficult to type because your keyboard or operating system has no easy way to produce the character. Another kind of escape sequence allows you to specify any character by its Unicode value. There are large tables listing all possible Unicode characters by numerical values. If you want to represent a specific literal, you type '\uxxxx' where xxxx is a hexadecimal number representing the value. For example, '\u0064' converted into decimal is $16 \times 6 + 4 = 100$, which is the letter '**d**'.

Example 3.11: Printing single characters

If you print a **char** variable or literal directly, it prints the character representation on the screen. For example, the following statement prints A not 65, the Unicode value of '**A**'.

```
System.out.println('A');
```

However, the Unicode values **are** numbers. If you try to perform arithmetic on them, Java will treat them like numbers. For example, the following statement adds the integer equivalents of the characters ($65+66 = 131$), concatenates the sum with the String "**C**", and concatenates the result with a String representation of the **int** literal 999. The final output is 131C999.



```
System.out.println('A' + 'B' + "C" + 999);
```

Booleans: boolean

If you are new to programming, it may seem useless to have a type designed to hold only true and false values. These values are called *Boolean values*, and the logic used to manipulate them turns out to be crucial to almost every program. We use them to represent conditions in [Chapters 4, 5](#), and beyond.

To store these truth values, Java uses the type **boolean**. There are exactly two literals for type **boolean**: **true** and **false**. Here are two declarations and assignments of **boolean** variables.

```
boolean awesome = true;  
boolean testFailed = false;
```

If we could only store these two literals, **boolean** variables would have limited usefulness. However, Java provides a full range of *relational* operators that allow us to compare values. Each of these operators generates a **boolean** result. For example, we can test to see if two numbers are equal, and the answer is either **true** or **false**. All Java relational operators are listed in the table below. Assume that all variables used in the **Example** column have a numeric type.

Symbol	Read as	Example
<code>==</code>	equal to	<code>x + 3 == y * 2</code>
<code>!=</code>	not equal to	<code>x != y / 4</code>
<code><< (less than)@< (less than)</code>	less than	<code>x < 3.5</code>
<code><=</code>	less than or equal to	<code>x <= y</code>
<code>></code>	greater than	<code>x > y+1</code>
<code>>=</code>	greater than or equal to	<code>x + y >= z</code>

Example 3.12: Boolean variables

The following declarations and assignments illustrate some uses of **boolean** variables. Note the use of the relational operators `==` and `>`.

```
int x = 3;  
int y = 4;  
boolean same = (x == 3);  
same = (x == y);  
boolean xIsGreater = (x > y);
```

In the first use of `==` above, the value of `same` is **true** because the value of `x` is 3. In the second comparison, the value of `same` is **false** because the values of `x` and `y` are different. The value of `xIsGreater` is also **false** since the value of `x` is not greater than the value of `y`. ■

In addition to the relational operators, Java also provides *logical* operators that can be used to combine or negate **boolean** values. These are the logical AND (`&&`), logical OR (`||`), logical XOR (`^`), and logical NOT (`!`) operators.

Name	Operator	Description
AND	&&	Returns true if both values are true
OR		Returns true if either value is true
XOR	^	Returns true if values are different
NOT	!	Returns the opposite of the value

All of these operators, except for NOT, are binary operators. Logical AND is used when you want your result to be **true** only if both the operands being combined evaluate to **true**. Logical OR is used when you want your result to be **true** if either operand is **true**. Logical XOR is used when you want your result to be **true** if one but not both of your operands is **true**. The unary logical NOT operator (!) results in the opposite value of its operand, switching **true** to **false** or **false** to **true**. Both the relational operators and the logical operators are described in greater detail in [Chapter 4](#).

3.3.3 Reference types

Now we will move on to reference types, which vastly outnumber the primitive types, with new types created all the time. Nevertheless, the primitive types in Java are important, partly because they are the building blocks for reference types.

Recall that a variable with a reference type does not contain a concrete value like a primitive variable. Instead, the value it holds is a reference or arrow pointing to the “real” object. It’s like a name for an object. When you declare a reference variable in Java, it starts off pointing at nothing, represented by the special literal **null**. For example, the following code creates a Wombat variable called w, which doesn’t point at anything. Wombat w;

```
Wombat w;
```

To create an object in Java, you use the **new** keyword followed by the name of the type and parentheses, which can either be empty or contain data you want to use to initialize the object. This process is called invoking the *constructor*, which creates space for the object and then initializes it with the values you specify or with default values if you leave the parentheses empty. Below we invoke the default Wombat constructor and point the variable w at the resulting object.

```
w = new Wombat();
```

Alternatively, the Wombat type might allow you to specify its mass in kilograms when creating one, as follows.

```
w = new Wombat(26.3);
```

Assignment of reference types points the two references to the same object. Thus, we can have two different Wombat references pointing at the same object.

```
Wombat w1 = new Wombat(26.3);
```

```
Wombat w2 = w1;
```

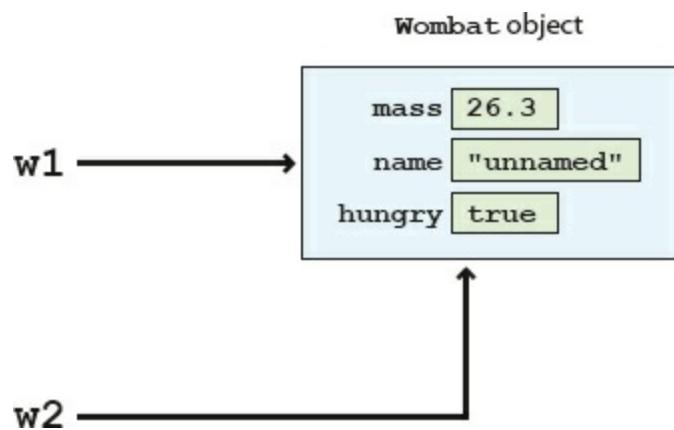


Figure 3.2: Two Wombat references pointing at the same object.

Then, anything we do to w1 will affect w2 and vice versa. For example, we can tell w1 to eat leaves using the `eatLeaves()` method.

```
w1. eatLeaves();
```

Perhaps this will increase the mass of the object that w1 points at to 26.9 kilograms. But the mass of the object that w2 points at will be increased as well, because they are the **same object**. Since primitive variables hold values and not references to objects, this kind of code works very differently with them. Consider the following.

```
int a = 10;
int b = a;
a = a + 5;
```

In this code, a is initialized to have a value of 10 and b is initialized to have whatever value a has, namely 10. The third line increases the value of a to 15, but b is still 10.

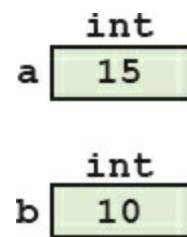


Figure 3.3: Because they are primitive, `int` variables store values, not references.

Now that we've highlighted some of the differences between primitive and reference types, we explain the String type more deeply. You use it frequently, but it has a few unusual features that are not shared by any other reference types.

String basics

The String type is used to represent text in Java. A String object contains a sequence of zero or more `char` values. Unlike every other reference type, there is a literal form for String objects. These literals are written with the text you want to represent inside of double quotes ("), such as "Fight the power!".

You can declare a String reference and initialize it by setting it equal to another String reference or a String literal. Like any other reference, you could leave it uninitialized.

Exercise 3.18

There is a difference between an uninitialized String (a reference that points to `null`) and a String of length 0. A String of length 0 is also known as an *empty string* and is written `""`. The space character (`' '`) and escape sequences such as `'\n'` can be a part of a String and add to its length. For example, `"ABC"` contains three characters, but the String `"A B C"` has five, because the spaces on each side of `'B'` count. The next example illustrates some ways of defining and using the String type.

Exercise 3.17

Example 3.13: String assignment

The following declarations define two String references named `greeting` and `title` and initialize each with a literal.

```
String greeting = "Bonjour !"  
String title = "French Greeting";
```

As you have seen in [Chapter 2](#), String values can be output using `System.out.print()` and `JOptionPane` methods.

```
System.out.println( greeting );  
JOptionPane.showMessageDialog( null , greeting , title , JOptionPane.  
    INFORMATION_MESSAGE );
```

The first statement above displays `Bonjour!` on the terminal. The second statement creates a dialog box with the title `French Greeting` and the message `Bonjour!` ■

String operations

In [Chapter 2](#), you saw that we can *concatenate* two String objects into a third String object using the `+` operator. This operator is unusual for a reference type. Almost all other reference types are only able to use the assignment operator (`=`) and the comparison operator (`==`). Like other reference types, the String class provides methods for interaction. We introduce a few String methods in this section and subsequent sections, but the String class defines many more.

Example 3.14: String concatenation

Here is another example of combining String objects using the `+` operator.

```
String argument = "the cannon";  
String phrase = "No argument but " + argument + "!";
```

In these statements, we initialize `argument` to `"the cannon"`. We then compute the value of `phrase` by adding, or concatenating, three String values: `"No argument but "`, the value of `argument`, and `"!"`. The result is `"No argument but the cannon!"`. If `argument` had been initialized to `"a pie in the face"`, then

phrase would point to “[No argument but a pie in the face!](#)”.

Another way of concatenating two String objects is by using the String concat() method.

```
String argument = " the cannon ";
String exclamation = "!";
String phraseStart = "No argument but ";
String phrase = phraseStart.concat ( argument );
phrase = phrase.concat ( exclamation );
```

This sequence of statements gives the same result as the one above it using the + operator. In practice, the concat() method is rarely used because the + operator is so convenient. Note that String objects in Java are *immutable*, meaning that calling a method on a String object will never change it. In the code above, calling concat() creates new String objects. The phrase reference points first at one String then it points at a new String on the next line. In this case the **reference** can be changed, but a **String object** never changes once it has been created. This distinction is a subtle but important one. ■

A host of other methods can be used on a String just like concat(). For example, the length of a String can be found using the length() method. The following statements prints 30 to the terminal.

```
String motto = " Fight for your right to party !";
System.out.println (motto.length());
```

String literals are String objects as well, and you can call methods on them. The following code stores 11 into letters.

```
int letters = " cellar door ". length();
```

Remember that a String is a sequence of **char** values. If you want to find out what **char** is at a particular location within a String, you can use the charAt() method.

This method is called with an **int** value giving the index you want to know about. Indexes inside of a String start at 0, not at 1. Zero-based numbering is used extensively in programming, and we discuss it further in [Chapter 6](#). (It may help if you think of the index as the number of characters that appear before the character at the specified index.) The next example shows how charAt() can be used.

Example 3.15: Examining the **char** value at an index

To see what **char** is at a given location, we call charAt() with the index in question, as shown below.

```
String word = "antidisestablishmentarianism";
char letter = word.charAt (11);
```

In this case, letter is assigned the value ‘**b**’. Remember, indexes for **char** values inside of a String start with 0. Thus, the **char** at index 0 is ‘**a**’, the **char** at index 1 is ‘**n**’, the **char** at index 2 is ‘**t**’, and so on. If you count up to the twelfth **char** (which has index 11), it should be ‘**b**’.

Every **char** inside of a String counts, whether it is a letter, a digit, a space, punctuation, or some other symbol.

```
String text = "^\_^ l337 # haxor # skillz !";
System.out.println (text.charAt (10));
```

This code prints out h since 'h' is the eleventh **char** (with index 10) in text. ■

A contiguous sequence of characters inside of a String is called a *substring*. For example, a few substrings of “Throw your hands in the air!” are “T”, “Throw”, “hands”, and “ur ha”. Note that “Ty” is not a substring because these characters do not appear next to each other.

The String class provides the `indexOf()` method to find the position of a substring, as shown in the next example.

Example 3.16: String search

Suppose we wish to find a String inside of another String. To do so, we call the `indexOf()` method on the String we’re searching inside of, with the String we’re searching for as the argument.

```
String countries = " USA Mexico China Canada ";
String search = " China ";
System.out.println (countries.indexOf (search));
```

The `indexOf()` method returns an **int** value that gives the position of the String we’re searching for. In the code above, the output is 11 because “China” appears starting at index 11 inside the `countries` String. (Alternatively, there are 11 characters before “China” in the String.) If the given substring cannot be found, the `indexOf()` method returns -1. For example, -1 will be printed to the terminal if we replace the print statement above with the following.

```
System.out.println (countries.indexOf (" Honduras "));
```

■ There are several other methods provided by String that we introduce as the need arises. If you are curious, you should look into the Java documentation for String at <http://download.oracle.com/javase/7/docs/api/java/lang/String.html> for a complete list of available methods.

3.3.4 Assignment and comparison

Both assigning one variable to another and testing two variables to see if they are equal to each other are important operations in Java. These operations are used on both primitive and reference types, but there are subtle differences between the two that we discuss below.

Assignment statements

Assignment is the act of setting one variable to the value of another. With a primitive type, the value held inside one variable is copied to the other. With a reference type, the arrow that points at the object is copied. All types in Java perform assignment with the assignment operator (=).

As we have discussed, values can be computed and then assigned to variables as in the following statement.

```
int value = Integer.parseInt ( response );
```

In Java, a statement that computes a value and assigns it is called an *assignment statement*. The generic form of the assignment statement is as follows.

```
identifier = expression ;
```

Here, identifier gives the name of some variable. For example, in the statement above, value is the name of the variable.

The right-hand side of an assignment statement is an expression that returns a value that is assigned to the variable on the left-hand side. Even an assignment statement can be considered an expression, allowing us to stack multiple assignments into one line, as in the following code.

```
int a, b, c;  
a = b = c = 15;
```

The Java compiler checks for type compatibility between the left and the right sides of an assignment statement. If the right-hand side is a broader type than the left-hand side (or is completely mismatched), the compiler gives an error, as in the following cases.

```
int number = 4.9;  
String text = 9;
```

Comparison

Comparing two values to see if they are the same uses the comparison operator (==) in Java. With primitive types, this kind of check is intuitive: The comparison is **true**, if the two values are the same. With reference types, the value held by the variable is the arrow pointing to the object. Two reference variables could point to different objects with identical contents and return **false** when compared to each other. The following gives examples of these comparisons.

Example 3.17: Comparison

Consider the following lines of code.

```
int x = 5;  
int y = 2 + 3;  
boolean z = (x == y);
```

The value of variable z is **true** because x and y contain the same values. If x were assigned 6 instead, z would be **false**.

Now, consider the following code:

```
String thing1 = new String (" Magical mystery ");  
String thing2 = new String (" Magical mystery ");  
String thing3 = new String (" Tragical tapestry ");
```

This code declares and initializes three String values. Although it is possible to store String literals

directly without invoking a String constructor, we are using this style of String creation to make our point because Java can do some confusing optimizations otherwise. Variables thing1 and thing2 point to String values that contain identical sequences of characters. Variable thing3 points to a different String. Consider the following statement.

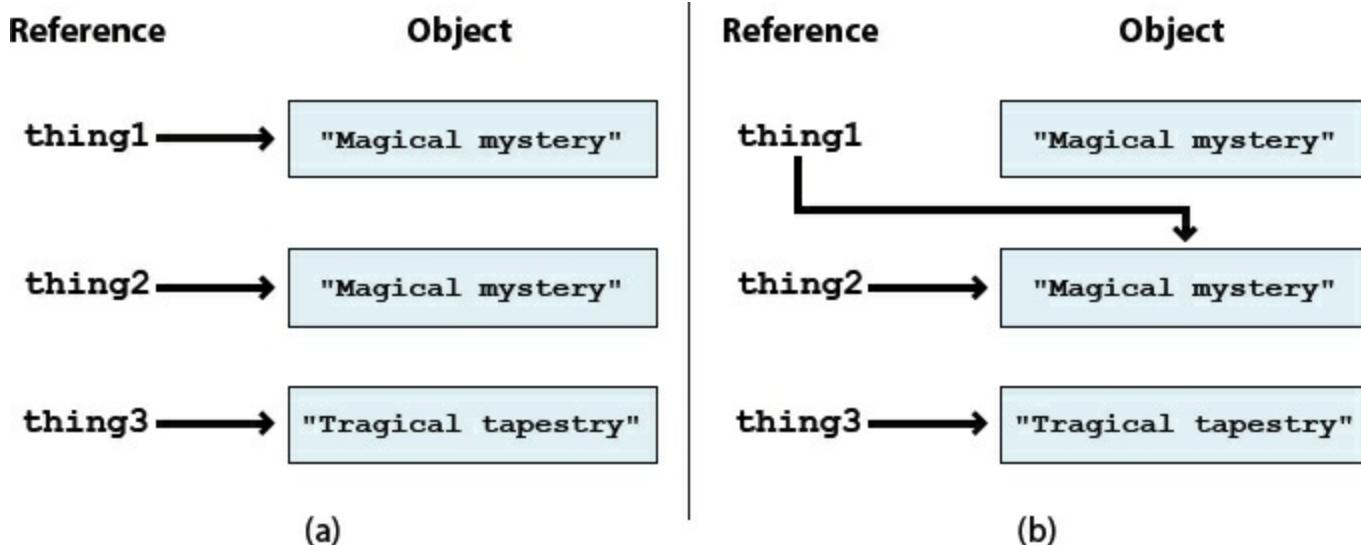


Figure 3.4: Objects thing1, thing2, and thing3 (a) in their initial states and (b) after the assignment `thing1 = thing2;`.

```
boolean same = (thing1 == thing3);
```

In this case the value of same is clearly **false** because the two String values are not the same. What about the following case?

```
boolean same = (thing1 == thing2);
```

Again, same contains **false**. Although, thing1 and thing2 point at identical objects, they point at **different** identical objects. Since the value held by a reference is the arrow that points to the object, the comparison operator only shows that two references are the same if they point at the same object.

To better understand comparison between reference types, consider [Figure 3.4\(a\)](#), which shows three different objects. Note that each reference points at a distinct object, even though two objects have the same contents.

Now consider the following assignment.

```
thing1 = thing2 ;
```

As shown in [Figure 3.4\(b\)](#), this assignment points reference thing1 to the same location as reference thing2. Then, `(thing1 == thing2)` would be **true**.

The `==` operator is generally not very useful with references, and the `equals()` method should be used instead. This method compares the contents of objects in whatever way the designer of the type specifies. For example,

```
thing1.equals (thing2 )
```

is **true** when `thing1` and `thing2` are pointing at distinct but identical String objects. ■

Exercise 3.13

3.3.5 Constants

In addition to normal variables, we can define *named constants*. A named constant is similar to a variable of the same type except that its value cannot be changed once set. A constant in Java is declared like any other variable with the addition of the keyword **final** before the declaration.

The convention in Java (and many other languages) is to name constants with all capital letters. Because camel case can no longer be used to tell where one word starts and another ends, an *underscore* (_) is used to separate words. Here are a few examples of named constant declarations.

```
final int POPULATION = 25000;  
final double PLANCK_CONSTANT = 6.626E -34;  
final boolean FLAG = false ;  
final char FIRST_INITIAL = 'A';  
final String MESSAGE = "All your base are belong to us.";
```

In this code, the value of `POPULATION` is 25000 and cannot be changed. For example, if you now write `population = 30000;` on a later line, your compiler will give an error. `PLANCK_CONSTANT`, `FLAG`, `FIRST_INITIAL`, and `MESSAGE` are also defined as named constants. Because of the syntax Java uses, these constants are sometimes referred to as *final variables*.

In the case of `MESSAGE` and all other reference variables, being **final** means that the reference can never point at a different object. Even with a **final** reference, the objects themselves can change if their methods allow it. (Since they are immutable, String objects can never change.)

Named constants are useful in two ways. First, a well-named constant can make your code more readable than using a literal. Second, if you do need to change the value to a different constant, you only have to change it in one place. For example, if you have used 25000 in five different places in your program, changing it to 30000 requires five changes. If you have used `POPULATION` throughout your program instead of a literal, you only have to change it in one place.

3.4 Syntax: Useful libraries

Computer software is difficult to write, but many of the same problems come up over and over. If we had to solve these problems every time we wrote a program, we'd never get anywhere. Java allows us to use code other people have written called *libraries*. One selling point of Java is its large standard library that can be used by any Java programmer without special downloads. You have already used the `Scanner` class, the `Math` class, and perhaps the `JOptionPane` class, which are all part of libraries. Below, we'll go deeper into the `Math` class and a few other useful libraries.

3.4.1 The Math library

Basic arithmetic operators are useful, but Java also provides a rich set of mathematical methods through the `Math` class. [Table 3.2](#) lists a few of the methods available. For a complete list of methods

provided by the Math class at the time of writing, visit <http://download.oracle.com/javase/7/docs/api/java/lang/Math.html>.

Example 3.18: Math library usage

Here is a program that uses the Math.pow() method to compute compound interest. Unlike Scanner and JOptionPane, the Math class is imported by default in Java programs and requires no explicit import statement.

Method	Sample use	Purpose
Trigonometric functions		
cos()	<pre>double adjacent = hypotenuse * Math.cos(theta);</pre>	Find the cosine of the argument.
sin()	<pre>double opposite = hypotenuse * Math.sin(theta);</pre>	Find the sine of the argument.
tan()	<pre>double opposite = adjacent * Math.tan(theta);</pre>	Find the tangent of the argument.
Exponentiation and logarithms		
exp()	<pre>double population = 250 * Math.exp(0.03 * time);</pre>	Compute e^x , where x is the argument.
log()	<pre>double digits = Math.log(1000000);</pre>	Compute the natural logarithm of the argument.
pow()	<pre>double money = principal * Math.pow(1.0 + rate, time);</pre>	Compute a^b , where a and b are the first and second arguments.
Miscellaneous		
random()	<pre>double percent = Math.random();</pre>	Generate a random number x where $0.0 \leq x < 1.0$.
round()	<pre>long items = Math.round(material);</pre>	Round to the nearest long (or nearest int when rounding a float).
sqrt()	<pre>double hypotenuse = Math.sqrt(a*a+b*b);</pre>	Compute the square root of the argument.

Table 3.2: A sample of methods available in the Java Math class. Arguments to trigonometric methods are given in radians.

Program 3.1: Program to compute interest earned and new balance.
(CompoundInterestCalculator.java)

```

1 import java.util.*;
2
3 class CompoundInterestCalculator {
4     public static void main (String [] args) {
5         Scanner in = new Scanner (System.in);
6         System.out.println ("Compound Interest Calculator ");
7         System.out.println ();
8         System.out.print ("Enter starting balance : ");
9         double startingBalance = in. nextDouble ();
10        System.out.print ("Enter interest rate : ");
11        double rate = in. nextDouble ();
12        System.out.print ("Enter time in years : ");
13        double years = in. nextDouble ();
14        System.out.print ("Enter compounding frequency : ");
15        double frequency = in. nextDouble ();
16        double newBalance = startingBalance *
17            Math.pow (1.0 + rate / frequency , frequency * years );
18        double interest = newBalance - startingBalance ;
19        System.out.println ("Interest earned : $" + interest );
20        System.out.println ("New balance : $" + newBalance );
21    }
22 }
```

■ In addition to methods, the Math library contains named constants including Euler's number e and π . These are written in code as Math.E and Math.PI, respectively. For example, the following assignment statement computes the circumference of a circle with radius given by the variable radius, using the formula $2\pi r$.

```
double circumference = 2* Math.PI* radius ;
```

3.4.2 Random numbers

Random numbers are often needed in applications such as games and scientific simulations. For example, card games require a random distribution of cards. To simulate a deck of 52 cards, we could associate an integer from 1 to 52 with each card. If we had a list of these values, we could swap each value in the list with a value at a random location later in the list. Doing so is equivalent to shuffling the deck.

Java provides the Random class in package java.util to generate random values. Before you can generate a random number with this class, you need to create a Random object as follows.

```
Random random = new Random ();
```

Here we have created an object named random of type Random. Depending on the kind of random value you need, you can use the nextInt(), nextBoolean(), or nextDouble() to generate a random value.

of the corresponding type.

```
// Random integer with all values possible  
int balance = random.nextInt();  
  
// Random integer between 0 (inclusive) and 130 (exclusive )  
int humanAge = random.nextInt(130);  
  
// Random boolean value  
boolean gender = random.nextBoolean();  
  
// Random floating - point value between 0.0 ( inclusive )  
// and 1.0 ( exclusive )  
double percent = random.nextDouble();
```

In these examples, *inclusive* means that the number could be generated, while *exclusive* means that the number cannot be. Thus, the call `random.nextInt(130)` generates the integers 0 through 129, but never 130. Exclusive upper bounds on ranges of random values are very common in programming.

To generate a random `int` between values a and b , not including b , use the following code, assuming you have a `Random` object named `random`.

```
int count = random.nextInt(b - a) + a;
```

The `nextInt()` method call generates a value between 0 and $b - a$, and adding a shifts it into the range from a up to (but not including) b .

Generating a random `double` between values a and b is similar except that `nextDouble()` always generates a value between 0.0 and 1.0, not including 1.0. Thus, you must scale the output by $b - a$ as shown below.

```
double value = random.nextDouble() * (b - a) + a;
```

The following example illustrates a potential use of random numbers in a video game.

Example 3.19: Dragon attribute generation

Suppose you are designing a video in which the hero must fight a dragon with random attributes. [Program 3.2](#) generates random values for the age, height, gender, and hit points of the dragon.

Program 3.2: Program to set attributes of a randomly generated dragon for a video game.
(DragonAttributes.java)

```
1 import java.util.*;  
2  
3 public class DragonAttributes {  
4     public static void main ( String [] args ) {  
5         Random random = new Random();
```

```

6     int age = random.nextInt (100) + 1;
7     double height = random.nextDouble () *30;
8     boolean gender = random.nextBoolean ();
9     int hitPoints = random.nextInt (51) + 25;
10    System.out.println (" Dragon Statistics ");
11    System.out.println ("Age :\t" + age );
12    System.out.format (" Height :\t%.1 f feet \n", height );
13    System.out.println (" Female :\t" + gender );
14    System.out.println ("Hit points :\t" + hitPoints );
15 }
16 }
```

Note that we begin by importing `java.util.*` to include all the classes in the `java.util` package, including `Random`. At [line 5](#), we create an object `random` of type `Random`. At [line 6](#), we use it to generate a random `int` between 0 and 99, to which we add 1, making an age between 1 and 100. To generate the height, we multiply a random `double` by 75, yielding a value between 0.0 and 75.0 (exclusive). Since there are only two choices for a dragon's gender, we generate a random `boolean` value, interpreting `true` as female and `false` as male. Finally, we determine the number of hit points the dragon has by generating a random `int` between 0 and 50, then add 25 to it, yielding a value between 25 and 75.

Because we are using random values, the output of [Program 3.2](#) changes every time we run the program. Sample output is given below.

```

Dragon Statistics
Age:      90
Height:    13.7 feet
Female:    true
Hit points: 67
```



If you only need a random `double` value, you can generate a number between 0.0 and 1.0 (exclusive) using the `Math.random()` method from the `Math` class. This method is a quick and dirty way to generate random numbers without importing `java.util.Random` or creating a `Random` object.

The random numbers generated by the `Random` class and by `Math.random()` are *pseudorandom* numbers, meaning that they are generated by a mathematical formula instead of truly random events. Each number is computed using the previous one, and the starting number is determined using time information from the OS. For most purposes, these pseudorandom numbers are good enough. Since each number can be predicted from the previous one, pseudorandom numbers are insufficient for some security applications. For those cases, Java provides the `SecureRandom` class, which is slower than `Random` but produces random numbers that are much harder to predict.

3.4.3 Wrapper classes

Reference types have methods that allow a user to interact with them in many useful ways. The primitive types (`byte`, `short`, `int`, `long`, `float`, `double`, `char`, and `boolean`) do not have methods, but we

sometimes need to manipulate them with methods or store them in a place that can only take a reference type.

To deal with such situations, Java uses *wrapper classes*, reference types that correspond to each primitive type. Following Java conventions for class names, the wrapper types all start with an uppercase letter but are otherwise similar to the name of the primitive type they support: Byte, Short, Integer, Long, Float, Double, Character, and Boolean.

String to numerical conversions

A common task for a wrapper class is to convert a String representation of a number such as “[37](#)” or “[2.097](#)” to its corresponding numeric value. We had such a situation in [Program 2.3](#), where we did the conversion as follows.

```
String response = JOptionPane.showInputDialog (null , enterHeight , title ,  
    JOptionPane.QUESTION_MESSAGE );  
height = Double.parseDouble ( response );
```

This code uses the `JOptionPane.showInputDialog()` method to get from the user the height from which a ball is dropped. This method always returns data as a String. In order for us to do computation with the value, we need to convert it to a numeric type, such as an int or a `double`. To do so, we use the appropriate `Byte.parseByte()`, `Short.parseShort()`, `Integer.parseInt()`, `Long.parseLong()`, `Float.parseFloat()`, or `Double.parseDouble()` method.

The following example shows conversions from a String to a number using three of these methods.

Example 3.20: String to numeric conversion

Consider the following statements that show how a string can be converted to a numerical value.

```
String text = "15";  
int count = Integer.parseInt ( text );  
float value = Float.parseFloat ( text );  
double tolerance = Double.parseDouble ( text );
```

In this example, we declare a String object named `text` and initialize it to “[15](#)”. Since `text` is a String and not a number, arithmetic expressions such as (`text*29`) are illegal.

To use the String “[15](#)” in a numerical computation, we need to convert it to a number. We used the `Integer.parseInt()`, `Float.parseFloat()`, and `Double.parseDouble()` methods to convert the String to `int`, `float`, and `double` values, respectively. Each method gives us 15 stored as the appropriate type. ■

Exercise 3.19

What happens if the String “[15.5](#)” (or even “[cinnamon](#)”) is given as input to the `Integer.parseInt()` method? If the String is not formatted as the appropriate kind of number, Java throws a `NumberFormatException`, probably crashing the program. An *exception* is an error that happens in the middle of running a program. We discuss how to work with exceptions in [Chapter 12](#).

When working with **char** values, it can be useful to know whether a particular value is a digit, a letter, or has a particular case. It may also be useful to convert a **char** to upper or lower case. Here is a partial list of the methods provided by the Character wrapper class to do these tasks.

Method	Purpose
isDigit(char value)	Returns true if value is a numerical digit and false otherwise.
isLetter(char value)	Returns true if value is a letter and false otherwise.
isLetterOrDigit(char value)	Returns true if value is a digit or a letter and false otherwise.
isLowerCase(char value)	Returns true if value is a lower case letter and false otherwise.
isUpperCase(char value)	Returns true if value is an upper case letter and false otherwise.
isWhitespace(char value)	Returns true if value is a whitespace character such as space, tab, or newline and false otherwise.
toLowerCase(char value)	Returns a lower case version of value, with no change if it is not a letter.
toUpperCase(char value)	Returns an upper case version of value, with no change if it is not a letter.

For example, the variable `test` contains **true** after the following code is executed.

```
boolean test = Character.isLetter ('x');
```

And the variable `letter` contains '**M**' after the following code is executed.

```
char letter = Character.toUpperCase ('m');
```

These methods can be especially useful when processing input.

Maximum and minimum values

As you recall from [Chapter 1](#), integer arithmetic in Java has limitations. If you increase a large positive number past its maximum value, it becomes a large magnitude negative number, a phenomenon called overflow. Conversely, if you decrease a large magnitude negative number past its minimum value, it becomes a large positive number, a phenomenon called underflow.

With floating point numbers, increasing their magnitudes past their maximum values results in special values that Java reserves to represent either positive or negative infinity, as the case may be. If a floating point value gets too close to zero, it eventually rounds to zero.

In addition to useful conversion methods, the numerical wrapper classes also have constants for the maximum and minimum values for each type. Instead of trying to remember that the largest positive int value is 2,147,483, 647, you can use the equivalent `Integer.MAX_VALUE`.

The `MAX_VALUE` constants are always the largest positive number that can be represented with the corresponding type. The `MIN_VALUE` is more confusing. For integer types, it is the largest magnitude negative number. For floating point types, it is the smallest positive non-zero value that can

be represented. Here is a table listing all these constants.

Constant	Meaning
Byte.MAX_VALUE	Most positive value a <code>byte</code> value can have
Byte.MIN_VALUE	Most negative value a <code>byte</code> value can have
Short.MAX_VALUE	Most positive value a <code>short</code> value can have
Short.MIN_VALUE	Most negative value a <code>short</code> value
Integer.MAX_VALUE	Most positive value an <code>int</code> value can have
Integer.MIN_VALUE	Most negative value an <code>int</code> value can have
Long.MAX_VALUE	Most positive value a <code>long</code> value can have
Long.MIN_VALUE	Most negative value a <code>long</code> value can have
Float.MAX_VALUE	Largest absolute value a <code>float</code> value can have
Float.MIN_VALUE	Smallest absolute value a <code>float</code> value can have
Double.MAX_VALUE	Largest absolute value a <code>double</code> value can have
Double.MIN_VALUE	Smallest absolute value a <code>double</code> value can have

The wrap-around nature of integer arithmetic means that adding 1 to Integer.MAX_VALUE results in Integer.MIN_VALUE. Note that all integer arithmetic in Java is done assuming type `int`, unless explicitly specified otherwise. Thus, Short.MAX_VALUE + 1 does not overflow to a negative value unless you store the result into a `short`. The same rules apply to underflow.

Exercise 3.6

Overflow and underflow do not work in the same way with the floating point numbers represented by `float` and `double`. The expression Double.MAX_VALUE + 1 results in Double.MAX_VALUE because 1 is so small in comparison that it is lost in rounding error. However, 1.5*Double.MAX_VALUE results in Double.POSITIVE_INFINITY, a constant used to represent a value larger than Double.MAX_VALUE. Since Double.MIN_VALUE is the smallest non-zero number, Double.MIN_VALUE - 1 evaluates to -1.0.

Exercise 3.7

Exercise 3.8

3.5 Solution: College cost calculator

In this chapter, we have introduced and more fully explained many aspects of manipulating data in Java, including declaring variables, assigning values, performing simple arithmetic and more advanced math, inputting and outputting data, and using the type system, which has small differences for primitive and reference types. Our solution to the college cost calculator problem posed at the beginning of the chapter uses all of these features at some level.

We present this solution below. The first step in our solution is to import `java.util.*` so that we can use the `Scanner` class. Then, we start the enclosing `CollegeCosts` class, begin the `main()` method, print a welcome message for the user, and create a `Scanner` object.

```

1 import java.util.*;
2
3 public class CollegeCosts {
4     public static void main ( String [] args ) {
5         System.out.println (
6             " Welcome to the College Cost Calculator !");
7         Scanner in = new Scanner ( System.in );

```

Next is a sequence of prompts to the user interspersed with input done with the Scanner object. The program reads the user's first name as a `String`, the user's last name as a `String`, the per-semester tuition cost as a `double`, the monthly cost of rent as a `double`, the monthly cost of food as a `double`, the interest rate for the loan as a `double`, and the number of years needed to pay back the loan as an `int`.

```

9     System.out.print (" Enter your first name \n\t");
10    String firstName = in.next ();
11    System.out.print (" Enter your last name \n\t");
12    String lastName = in.next ();
13    System.out.print (" Enter tuition per semester \t$");
14    double semesterTuition = in.nextDouble ();
15    System.out.print (" Enter rent per month \t$");
16    double monthlyRent = in.nextDouble ();
17    System.out.print (" Enter food cost per month \t$");
18    double monthlyFood = in.nextDouble ();
19    System.out.print (" Annual interest rate \n\t");
20    double annualInterest = in.nextDouble ();
21    System.out.print (" Years to pay back your loan \n");
22    int years = in.nextInt ();

```

The next segment of code completes the computations needed. First, it finds the total yearly cost by doubling the semester cost, multiplying the monthly rent and food costs by 12, and summing the answers together. The four year cost is simply four times the yearly cost. To find the monthly payment, we find the monthly interest by dividing the annual interest rate by 12 and plugging this value into the formula from the beginning of the chapter. Finally, the total cost of the loan is the monthly payment times 12 times the number of years.

```

24    double yearlyCost = semesterTuition * 2.0 +
25        ( monthlyRent + monthlyFood ) * 12.0;
26    double fourYearCost = yearlyCost * 4.0;
27    double monthlyInterest = annualInterest / 12.0;
28    double monthlyPayment = fourYearCost * monthlyInterest /
29        (1.0 - Math.pow (1.0 + monthlyInterest ,
30            -years * 12.0) );
31    double totalLoanCost = monthlyPayment * 12.0 * years ;

```

All that remains is to print out the output. First, we output a header describing the following output

as college costs for the user. Using `System.out.format()` as described in Subsection 3.3.2, we print out the yearly cost, four year cost, monthly loan payment, and total cost, all formatted with dollar signs, 2 places after the decimal point, and tabs so that the output lines up.

```
33     System.out.println ("\\nCollege costs for " +
34         firstName + " " + lastName );
35     System.out.println (
36         " *****");
37     System.out.print (" Yearly cost :\t\t$");
38     System.out.format (" %.2f\n", yearlyCost );
39     System.out.print (" Four year cost :\t\t$");
40     System.out.format (" %.2f\n", fourYearCost );
41     System.out.print (" Monthly loan payment :\t\t$");
42     System.out.format (" %.2f\n", monthlyPayment );
43     System.out.print (" Total loan cost :\t\t$");
44     System.out.format (" %.2f\n", totalLoanCost );
45 }
46 }
```

3.6 Concurrency: Expressions

In Section 2.5, we introduced the ideas of task and domain decomposition that could be used to solve a problem in parallel. By splitting up the jobs to be done (as in task decomposition) or dividing a large amount of data into pieces (as in domain decomposition), we can attack a problem with several workers and finish the work more quickly.

3.6.1 Splitting expressions

Performing arithmetic is some of the only Java syntax we have introduced that can be used to solve problems directly, but evaluating a single mathematical expression usually does not warrant concurrency. If the terms in the expression are themselves complex functions (such as numerical integrations or simulations that produce answers), it might be reasonable to evaluate these functions concurrently.

In this section, we will give an example of splitting an expression into smaller sub-expressions that could be evaluated concurrently. The basic steps underlying the concurrent evaluation of expressions are the following.

- Identify sub-expressions that are independent of each other.
- Create a separate thread that evaluates each sub-expression.
- Combine the results from each thread to obtain a final answer.

While this sequence of steps looks simple, each step can be complex. Worse, being careless at any

step could result in a concurrent solution that runs slower than the sequential solution or even gives the wrong answer. The following example illustrates these steps.

Example 3.21: Split expression

Consider the following statement:

```
double value = f(a,b)*g(c);
```

This statement evaluates methods $f()$ and $g()$, multiplies the computed values, and assigns the result to variable $value$. In [Figure 3.5](#), we show two ways of evaluating the expression $f(a,b)*g(c)$. [Figure 3.5\(a\)](#) shows sequential evaluation of the expression, where $f()$ is computed, $g()$ follows, and then the two results are multiplied to get the final value. [Figure 3.5\(b\)](#) shows evaluation of the expression in which $f()$ and $g()$ are evaluated concurrently instead.

On a multicore processor, the computation of $f()$ and $g()$ could be carried out on separate cores. We can create one thread for each method and wait for the threads to complete. Upon completion, we can retrieve the results of each computation and multiply them together as in [Figure 3.5\(b\)](#). [Program 3.3](#) illustrates this concurrent approach.

In [Program 3.3](#), we create two objects named $fThread$ and $gThread$ at lines 3 and 4, respectively. Both of these objects have types that extend the `Thread` class, which means that they can be made to run independently. Object $fThread$ needs two arguments (3.14 and 2.99 in this example), and $gThread$ needs one (5.55).

Program 3.3: Program for concurrent evaluation of an expression. (`SplitExpression.java`)

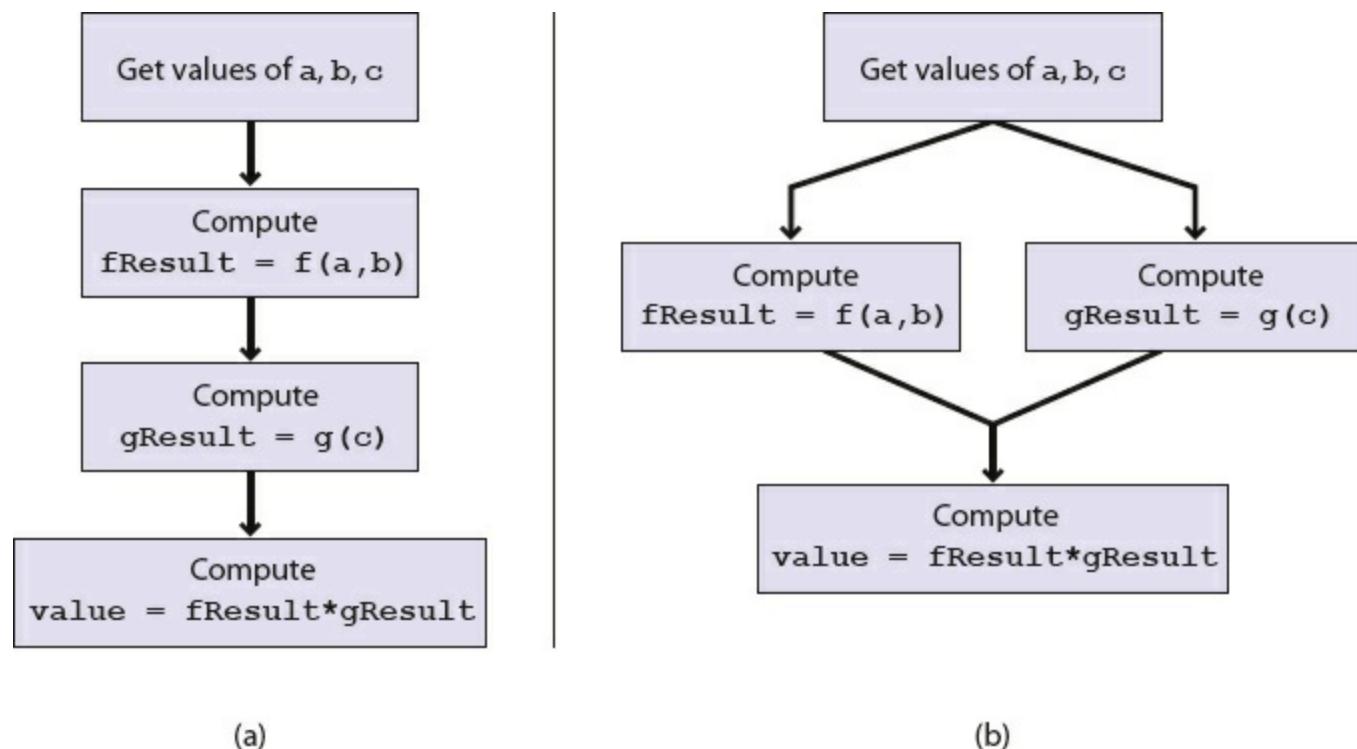


Figure 3.5: Computation of $value = f(a,b)*g(c)$ with (a) sequential and (b) concurrent approaches.

```
1 public class SplitExpression {  
2     public static void main ( String [] args ) {
```

```

3     ComputeF fThread = new ComputeF (3.14 , 2.99) ;
4     ComputeG gThread = new ComputeG (5.55) ;
5     fThread.start ();
6     gThread.start ();
7     try {
8         fThread.join ();
9         gThread.join ();
10        double fResult = fThread.getResult ();
11        double gResult = gThread.getResult ();
12        double answer = fResult * gResult ;
13        System.out.println (" Result of f: " + fResult );
14        System.out.println (" Result of g: " + gResult );
15        System.out.println (" Final answer : " + answer );
16    }
17    catch ( InterruptedException e){
18        System.out.println (" Computation interrupted !");
19    }
20}
21}

```

Once the thread objects have been created, we start the threads at lines 5 and 6. Every object whose type is Thread (or a child of Thread, which we will discuss in [Chapter 11](#)) has a start() method which begins its execution as a separate thread.

How do we know when a thread is done executing? Every Thread object has a join() method. If some code calls a thread's join() method, the method will not return until the thread is finished. When code is waiting for a thread to finish, it is possible for it to be interrupted if some other thread has gotten tired of the code waiting around doing nothing. If that happens, an InterruptedException is thrown. Exceptions are the way that Java deals with errors and other unusual situations. We will discuss them further in [Chapter 12](#), but, for now, you only need to know that code (like the join() method) that can cause certain kinds of exceptions (like the InterruptedException) needs to be enclosed in a **try** block. After the **try** block comes a **catch** block that says what to do in the even of that exception. In our case, we print out “Computation interrupted!”

Once the threads have completed their respective tasks, the execution of [Program 3.3](#) resumes at [line 10](#), where we obtain the result of the computation done by fThread by calling its getResult() method. On the next line, we call the getResult() method on gThread to obtain its result. Note that we could have called these getResult() methods before the join() calls, but the computations might not have completed, yielding invalid or incorrect results (or crashing the program). Finally, at [line 12](#), these two computed values are multiplied to get the final result, which is assigned to value and printed. ■

We would like to show how classes ComputeF and ComputeG are written, but we will hold off since they use concepts relating to methods, class design, and inheritance that we will not cover until [Chapters 8, 9, and 11](#).

If you don't understand all the elements of [Program 3.3](#), don't despair. We're trying to give you an example of what concurrency looks like in Java, but you cannot be expected to master all the details at this stage. However, concurrency in Java will often follow the steps shown:

1. Creation of Thread (or children of Thread) objects
2. Calling the start() method on these objects to start them executing
3. Calling the join() method on them to wait for them to finish
4. Retrieving the results (if any) of the computations done by the objects

3.6.2 Care in splitting expressions

The above example illustrates how you could split an expression and evaluate it concurrently. Note the following points when deciding whether or not to use concurrency. First, your program will run faster concurrently only if the work done is complex enough that its computation takes significantly longer than the time to create the necessary threads. In the example above, the methods f() and g() must be complex enough that it takes a significant amount of time to evaluate them. Otherwise, concurrency will not reduce the running time. This aspect of speedup is explained in detail in [Chapter 13](#).

Second, splitting an expression (or any complex sequence of computations) is easy when its individual components are independent. If they are interdependent, splitting requires care or subtle programming errors can occur. Consider the expression $f(a)+g(b)$ and suppose that f() modifies the value of b during execution. Such a modification is called a *side effect*. This side effect creates a dependency between f() and g(). Concurrent execution of these two methods must be done carefully, if it can be done at all. [Chapter 14](#) discusses concurrency in the presence of dependencies.

3.7 Summary

In a strongly typed language such as Java, types are an important concept. Every literal and variable in Java has a type, which specifies the possible values items with that type could have and the operations that can be done with them. Types are used to catch programming errors at compile time.

Java has a small set of primitive types such as `int` and `double`, which hold single values and use operators to manipulate them. Java also has reference types, which use primitive types as building blocks, can be created by any Java programmer, can contain arbitrarily complex data, and are manipulated with methods. One of the most commonly used reference types is `String`, which is used to store text of any length.

A number of library classes have been provided by the developers of Java. Programs performing mathematical operations beyond simple arithmetic may need to use methods from the `Math` class. Programs that need to generate random numbers can use methods from the `Random` class. Conversions and other useful manipulations of primitive types are provided by wrapper classes.

We also gave a taste of the syntax for creating, running, and waiting for the completion of threads.

Such threads could be used to speed up the evaluation of mathematical computations on multicore processors, but only if the computations are long, complex, and not too interdependent.

Exercises

Conceptual Problems

- 3.1 What is the difference between the set of integers from mathematics and the sets defined by `int` and `long`?
- 3.2 In [Example 3.7](#), the sum of two `int` variables was another `int` value, which could not be stored into a `byte` variable. Would this code have worked if variables `a` and `b` had been declared with type `byte`? What if `a` was assigned 121 and `b` was assigned 98?
- 3.3 The following three statements are legal Java (if properly included inside of a method). However, if we changed 2 to 2.0 or 5 to 5.0, the statements would not be legal. Explain why.

```
float roomArea = 2;  
float homeArea = 5;  
float area = roomArea * homeArea ;
```

- 3.4 Consider the following variable declarations.

```
int x = 3, y = 4, z = -9;  
float p = 3.99f, q = -9.89f;  
int population1 = 15000, population2 = 8000;  
final double MAXIMUM_LEVEL = 350;  
double limitPerCapita = 0.03;  
int age = 14;  
final int MAXIMUM_AGE = 23;  
boolean allowed = false ;
```

Now evaluate each of the following expressions to `true` or `false`.

- (a) `MAXIMUM_LEVEL/population1 > limitPerCapita && MAXIMUM_LEVEL/population2 < limitPerCapita`
- (b) `MAXIMUM_LEVEL/population1 > limitPerCapita || MAXIMUM_LEVEL/population2 < limitPerCapita`
- (c) `age < MAXIMUM_AGE && allowed`
- (d) `(x < y && y > z) || (p > q && population1 < population2)`

- 3.5 Evaluate the following expressions by hand and then check the results with a Java compiler.
 - (a) `5 & 6`

- (b) $5 \mid 6$
- (c) $5 \wedge 6$
- (d) ~ 5
- (e) $5 >> 2$
- (f) $5 << 2$
- (g) $5 >>> 2$

3.6 Evaluate the following expressions by hand and then check the results with a Java compiler.

- (a) `Byte.MIN_VALUE - 1`
- (b) `Byte.MAX_VALUE + 1`
- (c) `Integer.MIN_VALUE - 1`

3.7 Evaluate the following expressions by hand and then check the results with a Java compiler.

- (a) `Float.MAX_VALUE + 1`
- (b) `Double.MAX_VALUE - 1`
- (c) `-Double.MAX_VALUE - 1`
- (d) `-Double.MIN_VALUE - 1`
- (e) `-Double.MIN_VALUE + 1`

3.8 When evaluated in Java, the expression `2*Double.MAX_VALUE` results in `Double.POSITIVE_INFINITY` to indicate that the maximum representable value has been exceeded. However, the expression `Double.MAX_VALUE + 1` results in `Double.MAX_VALUE`. Why doesn't the second case yield `Double.POSITIVE_INFINITY` well?

3.9 What is printed when the following statements are executed?

```
System.out.println (15 + 20) ;  
System.out.println ("15" + 20) ;  
System.out.println ("15" + 15 + 20));
```

3.10 For each of the following Java expressions, indicate the types of each value being used and the type of the result when the expression is evaluated.

- (a) $3 + 4$
- (b) $3 + 4.0$
- (c) $3.0 + 4.0$
- (d) $3.0f + 4.0$
- (e) (`double`)($3 + 4$)

- (f) `(double)(3.0 + 4.0)`
- (g) `Math.round(3 * 4.2)`
- (h) `Math.round(3.2 * 4.9)`
- (i) `Math.round(15.5 * 4.0)`
- (j) `(int)(15.5 * 4.0)`
- (k) `Math.round(3.154)`

Concurrency

3.11 For each of the following expressions, determine the maximum amount of concurrency that can be achieved. Using a diagram similar to [Figure 3.5\(b\)](#), show how the computation of each expression will proceed. Assume there are no side effects. Note that you can create separate threads for multiple instances of method `f()`.

- (a) `f(a) + f(b) + f(c)`
- (b) `f(a * g(b))`
- (c) `f(g(a)) + f(b) + f(c)`

3.12 Answer the following questions about types, values, and references.

- (a) What is the difference between a value and the type that the value has?
- (b) In Java, the primitive type `int` represents a limited set of integers, not the entire set of integers from mathematics. Why is this the case? Why didn't the designers of Java allow `int` to represent all integers?
- (c) How are operations defined for reference types?
- (d) Explain the subtle difference between a reference and an object in Java.

3.13 Consider the following declarations of three Car objects.

```
Car car1 = new Car("Mercedes", "C300 Sport", 75000);  
Car car2 = new Car("Pontiac", "Vibe", 17000);  
Car car3 = new Car("Mercedes", "C300 Sport", 75000);
```

Let `same` be a variable of type `boolean`. What is the value of variable `same` after each of the following statements? Assume that the `equals()` method will return `true` if all of the attributes specified by the constructors for the two objects are the same.

- (a) `same = (car1 == car2);`
- (b) `same = (car1 == car3);`
- (c) `same = car1.equals(car3);`
- (d) `car2 = car3;`

(e) same = (car2 == car1);

(f) same = (car2 == car3);

(g) same = car2.equals(car3);

3.14 Characters 'a' and 'A' have Unicode values \u0061 and \u0041, respectively. Give the representation of these two characters as 16-bit unsigned binary integers.

3.15 Assuming that each character occupies 16 bits (two bytes) in memory and is coded using Unicode, use hexadecimal numbers to show how the word "Java" will be represented in computer memory. Unicode values for the Latin alphabet are the same as the values for the older ASCII standard. You can find a listing of these values on many websites such as <http://www.asciitable.com/>.

3.16 What is the output from the following sequence of statements?

```
String p = " Break it";
String q = " down like this !";
System.out.println ((p + q).length());
```

3.17 What is the output from the following sequence of statements? Note that r contains a single space character.

```
String p = " This is not a string .";
String q = "";
String r = " ";
System.out.println ((p + q + r).length());
```

Programming Practice

3.18 Try compiling the following program and observe the error reported by the compiler.

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

Now initialize the greeting object and rerun the program. Why does the program compile now?

3.19 Write a Java program that prompts the user to enter the number of rooms in her home, uses a Scanner object to read the input into an int variable named rooms, and then outputs the value on the screen. If you compile and execute your program and type in the value 3.5, can you explain the output you see?

3.20 Convert the college cost calculator solution given in [Section 3.5](#) to use JOptionPane methods for both input and output. Use the header giving the user's first and last name for the title of the output dialog and omit the line of asterisks. If you put all the output in a single String with a newline (\n) separating each line, the output will display properly.

Chapter 4

Selection

Life is a sum of all your choices.

—Albert Camus

4.1 Problem: Monty Hall simulation

There is a famous mathematical puzzle called the Monty Hall problem, based on the television show *Let's Make a Deal* hosted by the eponymous Monty Hall. In this problem, you are presented with three doors. Two of the three doors have junk behind them. One randomly selected door conceals something like a pile of gold. If you can choose that door, you win the gold. After you make an initial choice, Monty, who knows which door the pile of gold is behind, will open one of the two other doors, always picking a door with junk behind it. If you chose the gold door, Monty will pick between the two junk doors randomly. After opening a door, Monty gives you a chance to switch to the other unopened door. You decide to switch or not, the appropriate door is opened, and you win either junk or a pile of gold, depending on your luck.

As it turns out, it is always a better strategy to switch doors. If you keep your initial choice, you will win the gold with probability $\frac{1}{3}$. However, if you switch doors, you will win with probability $\frac{2}{3}$. The problem is counterintuitive and leads many people, including mathematicians and people holding advanced degrees, to the incorrect answer.

Think about it this way: Suppose you could pick two doors to open, and if the gold was behind either one of them, you would win. Clearly, your probability of winning would be $\frac{2}{3}$. Monty allows you this option. Just pick the two doors you want and tell Monty the third. He reveals one of your two initial doors as junk, and you switch to the other one.

If you still aren't convinced, that's fine. Your goal is to write a program that simulates the Monty Hall dilemma, allowing a user to guess a door and then, potentially, switch. Once you have written the simulation, you can choose to play repeatedly and see how well you do if you switch.

A Monty Hall scenario has two significant features that distinguish it from problems in previous chapters. First, randomness play a role. Generating random numbers has become an important part of computer science, and most languages provide programmers with tools for generating random or nearly random numbers. Recall the Random class from [Chapter 3](#). With an object of type Random called random, you can generate a random int between 0 and n - 1 by calling random.nextInt(n).

The second and much more important feature of Java in the solution to this problem is the element of choice. A random door is chosen to hide gold, and the program must react appropriately. The user chooses a door, and the program must carefully choose another door to open in response. Finally, the user must decide whether or not he or she wants to switch his or her choice. Inherent in this problem is the idea of *conditional execution*. Every program from the previous chapter runs sequentially, line by line. With conditional execution, only some of the code may be executed, depending on input from

the user or the values that random numbers take. Previously, every program was deterministic, a series of inevitable consequences. Now, that linear, one-thing-follows-another paradigm has split into complex trees and webs of possible program executions.

4.2 Concepts: Choosing between options

Before we get to random numbers and the complex choices involved in the Monty Hall problem, let's talk about the simplified approach that most programming languages take to conditional execution. When we come to a point in a program where there is a choice to be made, we can think of it as the question, "Do I want to perform this series of tasks?" Like the classic game of 20 Questions, these questions in Java generally only have two answers: "yes" or "no." If the answer is "yes," the program completes Task A, otherwise it completes Task B. It is easier to design programming languages that can handle yes-or-no questions than any general question. If you have studied logic in the past, you have probably run across *Boolean logic*. Boolean logic gives a set of rules, similar to the rules of traditional algebra, that can be applied to a system with only two values: true and false.

4.2.1 Simple choices

Because we want to build a system using only yes-or-no questions, Boolean logic is a perfect fit for computer science. To conform with other computer scientists, we try to think of conditions in terms of true and false, instead of yes and no. Thus, we can begin to formulate the kinds of choices we want to make:

If it is raining outside, then I will take my umbrella.

This statement is a very simple program, even though it is not one executed by a computer. The person following this program asks herself, "Is it raining today?" If the answer is "yes," then she will take her umbrella. We can abstract this idea a bit further by saying that *raining outside* is a condition p and that *taking my umbrella* is an action a . In other words, if p is true, then do a . We have not specified what is to be done if p is not true, although it is assumed that the actor in this drama will not take an umbrella.

If we want to view p as a decision to make, we can specify what happens if it is not true. For example, we could formulate another choice:

If I have at least \$50 in my pocket, then I will eat a lobster dinner, otherwise I will eat fast food.

In this case, we let *having at least \$50* be condition q , *eating a lobster dinner* be action b , and *eating fast food* be action c . Now we have created a decision. If q is true, the person will do action b , but, if it is false, she will do action c .

4.2.2 Boolean operations

Even by itself, the ability to pick between two options is powerful, but we can augment this ability in a couple of ways. First, we don't have to rely on simple conditions. Using Boolean logic, we can make arbitrarily complex conditions.

If I am bored, or it is late and I can't sleep, then I will watch television.

Someone following this program will watch television if he is bored or if it is late and he also cannot sleep. We can break the condition into three sub-conditions: *I am bored* is condition x , *it is late* is condition y , and *I can't sleep* is condition z . We have connected these three conditions together using the words “and” and “or.” These two simple words represent powerful concepts in Boolean logic, AND and OR. When two conditions are combined with AND, the result is true only if both conditions are true. When two conditions are combined with OR, the result is true if either of the conditions is true.

We can create a table called a *truth table* to show all the possible values certain conditions can take. We are going to use the symbol \wedge to represent the concept of AND and the symbol \vee to represent the concept of OR. We will also abbreviate true to T and false to F.

x	y	$x \wedge y$
T	T	T
T	F	F
F	T	F
F	F	F

Table 4.1: Given a condition x , a condition y , and the condition made by $x \wedge y$, this truth table shows all possible values. As stipulated, $x \wedge y$ is true only when both x and y are true.

x	y	$x \vee y$
T	T	T
T	F	T
F	T	T
F	F	F

Table 4.2: This truth table gives all the values for $x \vee y$. As you can see, $x \vee y$ is true if x or y are true.

Note that there is confusion with the use of the word “or” in English. Sometimes “or” is used in an exclusive sense to mean one or the other but not both, as in, “Would you like lemonade or iced tea with your meal?” In logic, this exclusive or exists as well and is called XOR. This difference gives another reason for a formally structured language like mathematics or Java to express ourselves precisely. When two conditions are connected with XOR, the result is true if one or the other but not both conditions are true. We use the symbol \oplus to represent the XOR operation in the truth table below.

x	y	$x \oplus y$
T	T	F
T	F	T
F	T	T
F	F	F

Table 4.3: This truth table gives all the values for $x \oplus y$.

The operations AND, OR, and XOR are all binary operations like addition and multiplication. They connect two conditions together to get a result. There is a single unary operation in Boolean logic, the NOT operator. A NOT simply reverses a condition. If a condition is true, then NOT applied to that condition will yield false, and vice versa. Here is a truth table for NOT, using the symbol \neg to represent the NOT operation.

x	$\neg x$
T	F
F	T

Now that we have nailed down some notation for Boolean logic, we can express the complicated expression that sent us down this path in the first place. Recall that x is *I am bored*, y is *it is late*, and z is *I can't sleep*. Let d be the action *I will watch television*. We can express the choice in this way: If $x \vee (y \wedge z)$, then do d . Using this notation, we have expressed precisely the conditions for watching television, using parentheses to clear up the ambiguity present in the original statement. If we can map individual conditions to Boolean variables, we can build conditions of arbitrary complexity.

4.2.3 Nested choices

Making one choice is all well and good, but in life and computer programs, we may have to make many interrelated choices. For example, if you choose to eat at a seafood restaurant, then you might choose between eating shrimp and lobster, but, if you choose instead to eat at a steakhouse, the options of shrimp and lobster might not be available.

A *nested* choice is one that sits inside of another choice you have already made. We could describe choices of restaurant and meal as follows.

If I want seafood, then I will eat at Sharky's, otherwise I will eat at the Golden Calf. When dining at Sharky's, if I have at least \$50, I will order the lobster, otherwise I will order the shrimp. When dining at the Golden Calf, if I have at least \$30, I will order the filet mignon, otherwise I will order the pork chops.

The previous description is long, but it precisely expresses the decisions our imaginary diner might make. This description in English has drawbacks: It is long and repetitive, and the grouping of specific meal choices with specific restaurants is not clear.

In the next section, we discuss the Java syntax that allows us to express the same sorts of decision patterns. Unlike English, Java has been designed to make these sequences of decisions clear and easy to read.

4.3 Syntax: Selection in Java

With some theoretical background on the kinds of choices we are interested in making, we are going to discuss the Java syntax used to describe these choices. It was no accident that we kept repeating the word “if,” because the main Java language feature for making choices is called an **if** statement.

4.3.1 if statements

The designers of Java studied **Boolean** logic and created a type called **boolean**. Every condition used by an **if** statement must evaluate to a **boolean** value, which can only be one of two things: **true** or **false**.

For example, we could have a **boolean** variable called **raining**. Stored in this variable is the value **true** if it is raining and **false** if it isn’t. Using Java syntax, we could encode our first example in which our actor takes her umbrella if it is raining.

```
if( raining ) {  
    umbrella.take();  
}
```

The action taken if it is raining is done by calling a *method* on an *object*. We’ll discuss objects and methods further in [Chapters 8](#) and [9](#). What we’re focusing on now is that the line `umbrella.take();` is executed only if `raining` has the value **true**. Nothing is done if it is **false**. [Figure 4.1](#) shows this pattern of conditional execution followed by all **if** statements.

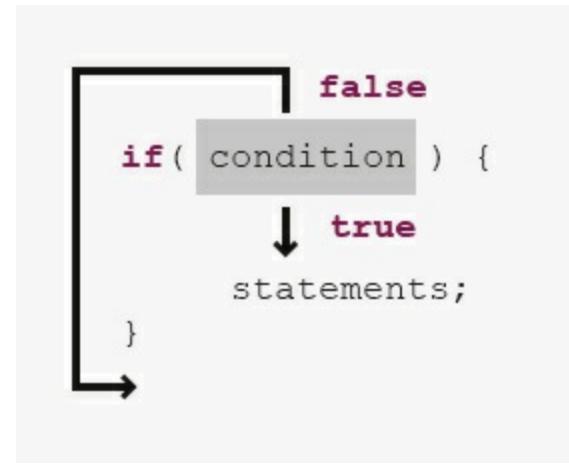


Figure 4.1: Execution goes inside the **if** statement when its condition is **true** and skips past it otherwise.

Our descriptions of logical scenarios from the previous section used the word “then” to mark the actions that would be done if a condition was true. Some languages use `then` as a keyword, but Java does not. Instead, note the left brace (`{`) and the right brace (`}`) that enclose the executable line `umbrella.take();`. These braces serve the same role as the word “then,” clearly marking the action to

be performed if a condition is true. Braces are unambiguous because they mark a start and an end. If there are many actions to be done, they can all be put inside the braces, and there will be no question as to which actions are associated with a given **if** statement.

For example, we may also need to close the window and put on a raincoat if it is raining. We might accomplish these tasks in Java as follows.

```
if( raining ) {  
    umbrella.take();  
    window.close();  
    raincoat.putOn();  
}
```

Within a matching pair of braces ({}), called a *block* of code, execution proceeds normally, line by line. First, the JVM will cause the umbrella to be taken, then the window to be closed, and finally the raincoat to be put on.

If only a single line of code is contained within a block of code, the braces can be left out. For example, many experienced Java programmers would have written our first example as follows.

```
if( raining )  
    umbrella.take();
```

For beginning Java programmers, however, it is a good idea to use braces even when you don't need to. Without braces, code can appear to be doing one thing when it really is doing another.

Since programmers must often choose between two alternatives, Java provides an **else** statement to specify code that should be run if the condition of the **if** statement is false.

Let **fiftyDollars** be a **boolean** variable that is **true** if we have at least \$50 and is **false** otherwise. Now, we can choose between two dining options based on how much money we have.

```
if( fiftyDollars ) {  
    lobsterDinner.eat();  
}  
else {  
    fastFood.eat();  
}
```

This Java code matches the logical statements we wrote before. If we have enough money, we'll eat a lobster dinner, otherwise, we'll eat fast food. As with an **if** statement, we use braces to mark a block of code for an **else** statement, too. Since a single line of code will be executed in each case, the braces are optional here. We could have written code with the same functionality as follows.

```
if( fiftyDollars )  
    lobsterDinner.eat();  
else  
    fastFood.eat();
```

Figure 4.2 shows the pattern of conditional execution followed by all **if** statements that have a matching **else** statement.

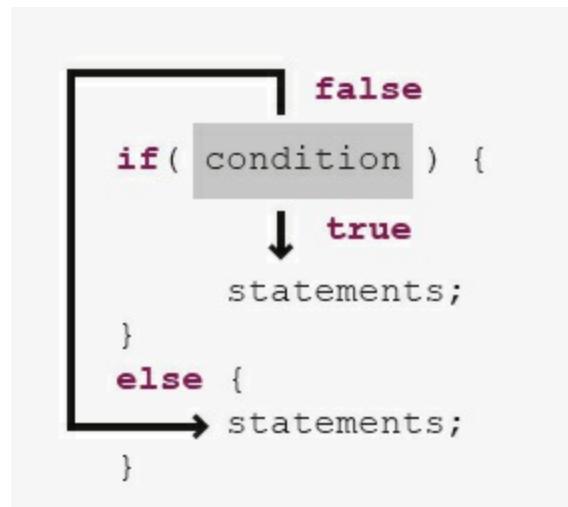


Figure 4.2: Execution goes inside the **if** statement when its condition is **true** and jumps into the **else** statement otherwise.

Pitfall: Misleading indentation

Indentation is used to make the code more readable, but Java ignores whitespace, meaning that the indentation has no effect on the execution of the code. To demonstrate, let's assume that our imaginary diner knows he will get a stomachache after eating fast food. Thus, he will take some Pepto-Bismol after eating it. If you modified the code above, which does not contain braces, you might get the following.

```
if( fiftyDollars )
    lobsterDinner.eat ();
else
    fastFood.eat ();
    peptoBismol.take ();
```

Although it looks like both `fastFood.eat();` and `peptoBismol.take();` are within the block of the **else** statement, only `fastFood.eat();` is. The line `peptoBismol.take();` is not part of the **if-else** structure at all and will be executed no matter what. The correct way to program this decision is below.

```
if( fiftyDollars )
    lobsterDinner.eat ();
else {
    fastFood.eat ();
    peptoBismol.take ();
}
```

4.3.2 The boolean type and its operations

Recall that Java uses the type **boolean** for values that can only be true or false. Just like the numerical types **double** and **int**, the **boolean** type has specific operations that can be used to combine them together. By design, these operations correspond exactly to the logical operations we described before. Here is a table giving the Java operators that are equivalent to the logical Boolean operations.

Math		Java	
Name	Symbol	Operator	Description
AND	\wedge	<code>&&</code>	Returns true if both values are true
OR	\vee	<code> </code>	Returns true if either value is true
XOR	\oplus	<code>^</code>	Returns true if values are different
NOT	\neg	<code>!</code>	Returns the opposite of the value

Using these operators, we can create **boolean** values and combine them together.

```
boolean x = true ;  
boolean y = false ;  
boolean z = !(( x || y ) ^ (x && y));
```

When this code is executed, the value of `z` will be **false**. Although it is perfectly legal to perform **boolean** operations this way, it is much more common to combine them “on the fly” inside of the condition of an **if** statement. Recall the statement from the previous section:

If I am bored, or it is late and I can't sleep, then I will watch television.

If we let `bored`, `late`, and `canSleep` be **boolean** variables whose values indicate if we are bored, if it is late, and if we can sleep, respectively, we can encode this statement in Java like so.

```
if( bored || ( late && ! canSleep ) )  
    television.watch();
```

Combining the `||` operator with other `||` operators is both commutative and associative: order and grouping doesn’t matter. Likewise, combining the `&&` operator with other `&&` operators is also commutative and associative. However, once you start mixing `||` with `&&`, it is a good idea to use parentheses for grouping. If, in the above example, `bored` is **true**, `late` is **false**, and `canSleep` is **true**, then the expression `bored || (late && !canSleep)` will be **true**. However, with the same values, the expression `bored || late && !canSleep` will be **false**.

Now that we are discussing ordering, it is important to note that `||` and `&&` are *short circuit* operators. Short circuit means that, if the value of the expression can be determined without evaluating the rest of it, the JVM will not bother to compute any more of the expression. With `||` this situation arises because **true OR anything else is still true**. With `&&` this situations arises because **false AND anything else is still false**.

```
if( true || ( late && ! canSleep && isTired && isHungry ) ||
```

```
( wantsToFindOutWhatHappensNextInHisFavoriteShow ||
    likesTV )) )
```

The condition of this **if** statement will always evaluate to **true** and its body will always be executed. Because Java knows this, it will not even bother to execute any of the code after the first **|** operator. This short circuit evaluation is done at run time and will work if the value of a variable at the beginning of an OR clause is **true**. It need not be the literal **true**.

```
if( false && (( late || ! canSleep || isTired || isHungry ) &&
    ( wantsToFindOutWhatHappensNextInHisFavoriteShow ||
        likesTV )) )
```

The condition of this **if** statement will always evaluate to **false** and its body will not be executed. As before, nothing after the first **&&** will even be executed. If you are combining literals and **boolean** values with the **||** and **&&** operators, it makes no difference that short circuit evaluation occurs. However, if a method call is part of the clauses, your code might miss valuable side-effects. For example, let the **boolean** variable **working** be **false** in the following.

```
if( working && doSomethingImportant() )
```

In this case, the **doSomethingImportant()** method must return a **boolean** value to be a valid statement. Still, if **working** is **false**, the **doSomethingImportant()** method will not be called. As soon as the JVM realizes that it is applying the **&&** operation to a **false** value, it will give up. In many cases, doing so is fine. In fact, programmers will sometimes exploit this feature to allow code in a method like **doSomethingImportant()** to run only if it is safe to do so. In this case, if we assume that we always want to run the **doSomethingImportant()** method (because it does something important) every time the condition of the **if** statement is evaluated, we need to restructure the code. For example, we can reverse the order of the two terms in the AND clause to achieve this effect. Alternatively, Java provides non-short circuit versions of the **||** and **&&** operators, namely **|** and **&**, if you need to force full evaluation.

You may have been wondering where the majority of **boolean** values come from. Most computer programs do not ask the user a long series of true or false questions before spitting out an answer. Most **boolean** values in Java programs are the result of comparisons, often of numerical data types.

It is useful to compare two numbers to see if one is larger, smaller, or equal to the other. For example, you might have a **double** variable called **pressure** that gives the water pressure in a hydraulic system. Perhaps you also have a constant called **CRITICAL_PRESSURE** that gives the maximum safe pressure for your system. You can compare these values using the **>** operator.

```
if( pressure > CRITICAL_PRESSURE )
    emergencyShutdown();
```

This code allows you to call the appropriate emergency method when pressure is too high. Of course, the **>** operator is not the only way to compare two values in Java. We list all the relational operators in [Chapter 3](#), but [Table 4.4](#) below shows them again in a mathematical context.

Name	Math	Java	
Name	Symbol	Operator	Description
Equals	=	==	true if the two values are equal
Not Equals	≠	!=	true if the two values are not equal
Less Than	<	<	true if the first value is strictly less than the second
Less Than or Equals	≤	<=	true if the first value is less than or equal to the second
Greater Than	>	>	true if the first value is strictly greater than the second
Greater Than or Equals	≥	>=	true if the first value is greater than or equal to the second

Table 4.4: Relational operators in Java.

The concepts and mathematical symbols for these operators should be familiar from mathematics. There are a few differences from the mathematical versions of these ideas that are worth pointing out. First, only easy-to-type symbols are used for Java operators. Thus, we need two characters to represent most operators in the language. These operators can be used to compare any numerical type with any other numerical type, including `char`. In the case of mismatched types, such as an `int` and a `double`, the lower precision type is automatically cast to the higher precision type. Care should be taken when using the `==` operator with floating point types because of rounding errors. For example, the expression `(1.0/3.0 == 0.3333333333)` always evaluates to `false`.

The `==` operator is not the same as the `=` operator from previous chapters. In Java, the double equal sign `==` is used to compare two things while the single equal sign `=` is used to assign one thing to another.

Confusion can also arise because, in the mathematical world, relational symbols are used to make a statement: $x < y$ is an announcement or a discovery that the value contained in x is, in fact, smaller than the value contained in y . In the Java world, the statement `x < y` is a **test** whose answer is **true** if the value contained in `x` is smaller than the value contained in `y` and **false** otherwise. Using these operators means performing a test at a specific point in the code, asking a question about the values that certain variables or literals (or the results of method calls) have at that moment in time. In another sense, using these comparisons is a way to take numerical data and convert it into the language of `boolean` values. Note that the following statement does not compile in Java.

```
if( 4 )
    x = y + z;
```

To be used in an `if` statement, the value 4 must be first compared with some other numerical type to

yield a **true** or **false**.

Pitfall: Assignment instead of equality

Along these lines, a common pitfall is to forget one of the equal signs in the comparison operator.

```
if( x = 4 )  
    x = y + z;
```

Again, this code will not compile. If it did, the variable `x` would be assigned the value 4, which would in turn be given to the `if` statement, but an `if` statement does not know what to do with anything other than a `boolean` value. Extreme care should be taken when comparing two `boolean` values. For example, we might have two `boolean` values `genderA` and `genderB`, corresponding to the genders of two different people. Let's say that the value of each one is `true` if the person is female and `false` otherwise. We could create an `if` statement that would work only if their genders are the same.

```
if( genderA == genderB )  
    makeRoommates();
```

This code correctly calls the `makeRoommates()` method only if the two individuals have the same gender. However, a tiny mistake in the code could yield the following.

```
if( genderA = genderB )  
    makeRoommates();
```

In this case, `genderA` would be assigned to whatever `genderB` is. Then, that value would be given to the `if` statement. In this situation, the `makeRoommates()` method will be called only if `genderB` is `true`, meaning female. Thus, the two people will become roommates if the second one is female, and the gender of the first person won't be considered. Unlike the `x = 4` example, this code will compile with no warning.

The next few examples illustrate the use of the `if` statement. They also use some methods from class Math.

Example 4.1: Leap year

In the standard Gregorian calendar, leap years occur roughly once every four years. During leap years, the month of February has 29 days instead of 28. This extra day makes up for the fact that it takes almost 365.24 days for the earth to orbit the sun. Unfortunately, the orbit of the earth around the sun does not match up in any exact way with the rotation of the earth. So, there are exceptions to the rule of every four years.

In fact, the official definition for a leap year is a year that is evenly divisible by 4, except for those years that are evenly divisible by 100, with the exception to the exception of years that are evenly divisible by 400. For example, 1988 was a leap year because it was divisible by 4. The year 1900 was not a leap year because it was divisible by 100 but not by 400, and the year 2000 was a leap year because it was divisible by 400.

Recall that the mod operator (%) allows us to find the remainder after integer division. Thus, if n

% 100 gives zero, n has no remainder after being divided by 100 and must be evenly divisible by 100.

Program 4.1: A program that prompts the user for a year and then determines whether or not it is a leap year. (LeapYear.java)

```
1 import java.util.*;
2
3 public class LeapYear {
4     public static void main ( String [] args ) {
5         Scanner in = new Scanner ( System.in );
6         System.out.print (" Please enter a year : ");
7         int year = in. nextInt ();
8         if( year % 400 == 0 )
9             System.out.println ( year + " is a leap year." );
10        else if( year % 100 == 0 )
11            System.out.println ( year + " is not a leap year." );
12        else if( year % 4 == 0 )
13            System.out.println ( year + " is a leap year." );
14        else
15            System.out.println ( year + " is not a leap year." );
16    }
17 }
```

As with all of the programs in this section, we begin by importing `java.util.*`, which is needed for the `Scanner` class for input. The program prompts the user for a year and reads it in. If the year is evenly divisible by 400, the program outputs that it is a leap year. Otherwise, if the year is evenly divisible by 100, the program outputs that it is not a leap year. Otherwise, if the year is evenly divisible by 4, the program outputs that it is a leap year. Finally, if all the other conditions have failed, the program outputs that the year is not a leap year. ■

Example 4.2: Quadratic formula

The quadratic formula is a useful tool from mathematics. Using this formula, you can solve equations of the form $ax^2+bx+c = 0$. As you might recall, the quadratic equation that gives the solutions is:

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

The b^2-4ac part of the formula is called the *discriminant*. If the discriminant is positive, there will be two real answers to the equation. If the discriminant is negative, there will be two complex answers to the equation. Finally, if the discriminant is zero, there will be a single real answer to the problem. If you want to write a program to solve quadratic equations for you, it should take these three possibilities into account.

```

1 import java.util.*;
2
3 public class Quadratic {
4     public static void main ( String [] args ) {
5         Scanner in = new Scanner ( System.in );
6         System.out.println (" This program solves quadratic " +
7             " equations of the form ax ^2 + bx + c = 0.");
8         System.out.print (" Please enter a value for a: ");
9         double a = in.nextDouble ();
10        System.out.print (" Please enter a value for b: ");
11        double b = in.nextDouble ();
12        System.out.print (" Please enter a value for c: ");
13        double c = in.nextDouble ();
14        double discriminant = b*b - 4*a*c;
15        if( discriminant == 0.0 )
16            System.out.println (" The answer is x = " + (-b /(2* a)));
17        else if( discriminant < 0.0 )
18            System.out.println (" The answers are x = " + (-b /(2* a))
19                +
20                " + " + Math.sqrt (- discriminant ) + "i and x = " +
21                (-b / (2* a)) + " - " + Math.sqrt (- discriminant ) + "
22                i);
23        else
24            System.out.println (" The answers are x = " + (-b /(2* a) +
25                Math.sqrt ( discriminant )) + " and x = " +
26                (-b /(2* a) - Math.sqrt ( discriminant ))));
}
}

```

This program begins by prompting the user and reading in values for a, b, and c. Then, it computes the discriminant. In the first case, we want to test to see if the discriminant is zero. If the discriminant was not zero but is negative, we account for this situation in the next case. We compute the real and complex parts separately and output the two answers. Finally, if the discriminant is positive, we find the two answers and output them. Note that braces were not needed for the `if`, `else-if`, and `else` blocks because each is composed of only a single line of code. Although these `System.out.println()` method calls may take up more than one line visually, Java interprets them as single lines because they each only have a single semicolon (`;`).

The line `if(discriminant == 0.0)` is dangerous since we are using `double` values. Because of rounding errors, the discriminant might not be exactly zero even if it should be, mathematically. Industrial strength code would probably check to see if the absolute value of the discriminant is less than a very small number (such as 0.00000001). Values that small would then be treated as if they were zero. ■

Example 4.3: 20 Questions

In the time-honored game of 20 Questions, one person mentally chooses something, and the other participants must guess what the thing is by asking questions whose answer is either “yes” or “no.” In one popular version, the person who chooses the thing starts by declaring whether it is animal, vegetable, or mineral.

Using counting principles from math, 20 yes-or-no questions makes it possible to differentiate $2^{20} = 1,048,576$ items. If you are told ahead of time whether the thing is animal, vegetable, or mineral, it should be possible to guess over 3 million items! We are not yet ready to deal with such a large range of possibilities. To keep the size of the code reasonable, let’s narrow the field to 10 different items: a lizard, an eagle, a dolphin, a human, some lead, a diamond, a tomato, a peach, a maple tree, and a potato.

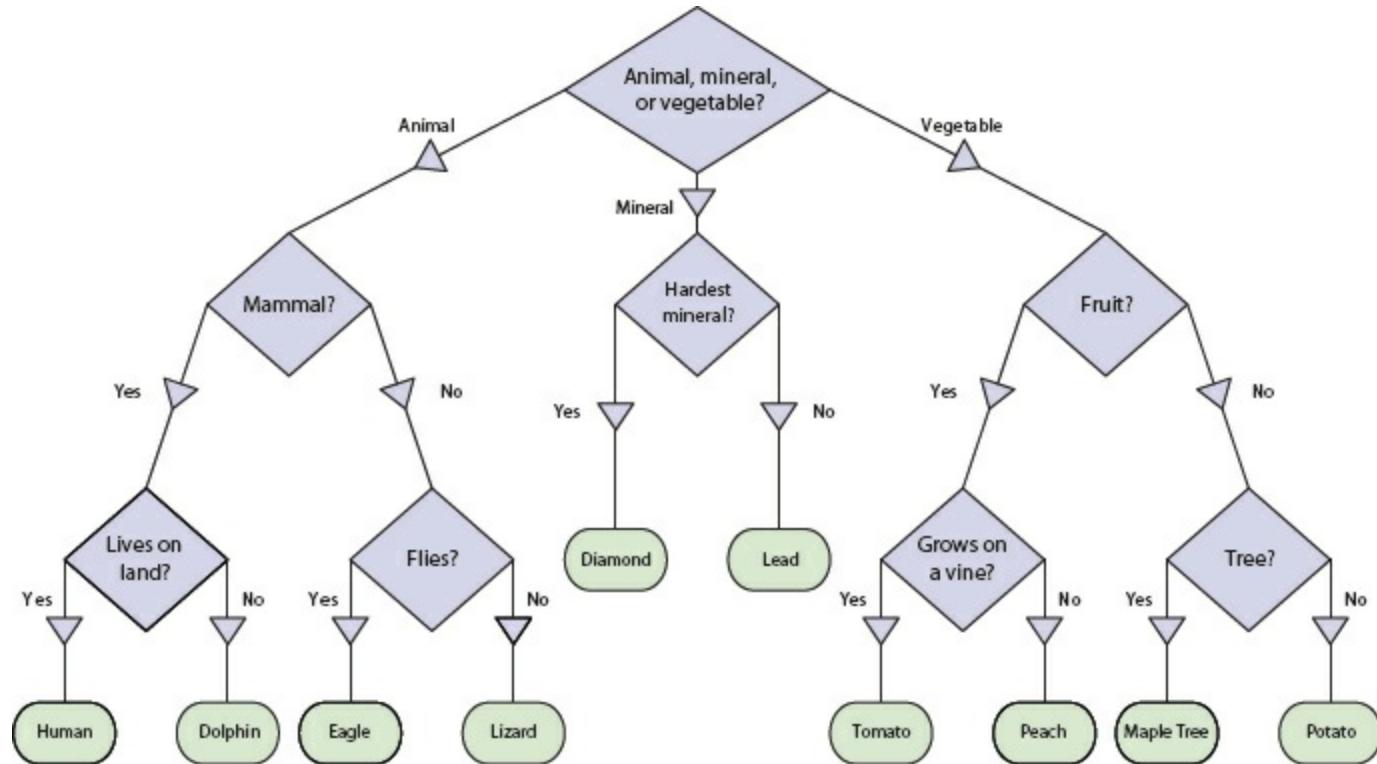


Figure 4.3: Decision tree to distinguish 10 items.

Using these items, we can construct a tree of decisions to make, starting with the decision between animal, vegetable, and mineral. If the thing is an animal, we could then ask if it is a mammal. If it is a mammal, we could ask if it lives on land, deciding between human and dolphin. If it is not a mammal, we could ask if it flies, deciding between an eagle and a lizard. We can construct similar questions for the things in the vegetable and mineral categories, matching [Figure 4.3](#).

Program 4.3: Program to navigate the possible choices in the decision tree. ([TwentyQuestions.java](#))

```
1 import java.util.*;
2
3 public class TwentyQuestions {
4     public static void main ( String [] args ) {
5         Scanner in = new Scanner ( System.in );
```

```
6 System.out.print ("Is it an animal , vegetable , or mineral ?" +  
7     "('a ','v ', or 'm '): ");  
8 String response = in.next ().toLowerCase ();  
9 if( response.equals ("a") ) {  
10    System.out.print ("Is it a mammal ? ('y' or 'n '): ");  
11    response = in.next ().toLowerCase ();  
12    if( response.equals ("y") ) {  
13       System.out.print (  
14           " Does it live on land ? ('y' or 'n '): ");  
15       response = in.next ().toLowerCase ();  
16       if( response.equals ("y") )  
17          System.out.println ("It 's a human.");  
18       else // assume "n"  
19          System.out.println ("It 's a dolphin.");  
20    }  
21    else { // assume "n"  
22       System.out.print (" Does it fly ? ('y' or 'n '): ");  
23       response = in.next ().toLowerCase ();  
24       if( response.equals ("y") )  
25          System.out.println ("It 's an eagle.");  
26       else // assume "n"  
27          System.out.println ("It 's a lizard.");  
28    }  
29 }  
30 else if( response.equals ("v") ) {  
31    System.out.print ("Is it a fruit ? ('y' or 'n '): ");  
32    response = in.next ().toLowerCase ();  
33    if( response.equals ("y") ) {  
34       System.out.print (  
35           " Does it grown on a vine ? ('y' or 'n '): ");  
36       response = in.next ().toLowerCase ();  
37       if( response.equals ("y") )  
38          System.out.println ("It 's a tomato.");  
39       else // assume "n"  
40          System.out.println ("It 's a peach.");  
41    }  
42    else { // assume "n"  
43       System.out.print ("Is it a tree ? ('y' or 'n '): ");  
44       response = in.next ().toLowerCase ();  
45       if( response.equals ("y") )  
46          System.out.println ("It 's a maple tree.");  
47       else // assume "n"  
48          System.out.println ("It 's a potato.");
```

```

49     }
50 }
51 else { // assume "m"
52     System.out.print (
53         "Is it the hardest mineral? ('y' or 'n'): ");
54     response = in.next ().toLowerCase ();
55     if( response.equals ("y") )
56         System.out.println ("It 's a diamond.");
57     else // assume "n"
58         System.out.println ("It 's lead.");
59 }
60 }
61 }

```

The code in this example is straightforward, although even 10 items makes for a lot of `if` and `else` blocks. Other than the `if-else` statements, only simple input and output are needed to make the program function. For proper String comparison, it is necessary to use the `equals()` method to test if two String values are the same.

Note that we have added comments specifying what we assume is the case for each `else` block. If we were being more careful, we should test for the “`y`” and “`n`” cases and then give an error message when the user inputs something unexpected, like “`x`” or “`149`” or even “`no`”. Again, note that no braces are needed for the final `if-else` blocks in which the guess is made, since each of these guesses requires only a single line of code.

You might be curious how to make a real 20 Questions game that could learn over time. To do so, many more programming tools are necessary: repetition, data structures (so that you can organize the questions), and file input and output (so that you can store new information permanently). These concepts are covered in later chapters. ■

4.3.3 switch statements

The `if` statement is the workhorse of Java conditional execution. With enough care, you can craft code that can make any fixed sequence of decisions with arbitrary complexity. Even so, the `if` statement can be a little clumsy because it only allows you to choose between two alternatives. After all, a conditional can only be `true` or `false`. Certainly, decisions can be nested, allowing for more than two possibilities, but long lists of possibilities can be cumbersome.

For example, imagine that we want to create a program that determines the appropriate gift for a wedding anniversary. Below is a table of traditional categories of gifts based on the anniversary year.

Year	Gift	Year	Gift
1	Paper	13	Lace
2	Cotton	14	Ivory
3	Leather	15	Crystal
4	Fruit	20	China
5	Wood	25	Silver
6	Candy / Iron	30	Pearl
7	Wool / Copper	35	Coral
8	Bronze / Pottery	40	Ruby
9	Pottery / Willow	45	Sapphire
10	Tin / Aluminum	50	Gold
11	Steel	55	Emerald
12	Silk / Linen	60	Diamond

Let year be a variable of type `int` containing the year in question. A structure of `if-else` statements that can determine the appropriate gift based on the year is below.

```
String gift ;
if( year == 1 )
    gift = " Paper ";
else if( year == 2 )
    gift = " Cotton ";
else if( year == 3 )
    gift = " Leather ";
else if( year == 4 )
    gift = " Fruit ";
else if( year == 5 )
    gift = " Wood ";
else if( year == 6 )
    gift = " Candy / Iron ";
else if( year == 7 )
    gift = " Wool / Copper ";
else if( year == 8 )
    gift = " Bronze / Pottery ";
else if( year == 9 )
    gift = " Pottery / Willow ";
else if( year == 10 )
    gift = " Tin / Aluminum ";
else if( year == 11 )
    gift = " Steel ";
```

```

else if( year == 12 )
    gift = " Silk / Linen ";
else if( year == 13 )
    gift = " Lace ";
else if( year == 14 )
    gift = " Ivory ";
else if( year == 15 )
    gift = " Crystal ";
else if( year == 20 )
    gift = " China ";
else if( year == 25 )
    gift = " Silver ";
else if( year == 30 )
    gift = " Pearl ";
else if( year == 35 )
    gift = " Coral ";
else if( year == 40 )
    gift = " Ruby ";
else if( year == 45 )
    gift = " Sapphire ";
else if( year == 50 )
    gift = " Gold ";
else if( year == 55 )
    gift = " Emerald ";
else if( year == 60 )
    gift = " Diamond ";
else
    gift = "No traditional gift ";

```

This code stores the correct value in gift. Note that we are using the feature of if statements that treats an entire if statement as one statement. If we used braces to group things properly, the code would become unreadable and unmanageably large.

```

String gift ;
if( year == 1 ) {
    gift = " Paper ";
}
else {
    if( year == 2 ) {
        gift = " Cotton ";
    }
    else {
        if( year == 3 ) {
            gift = " Leather ";
        }
        else if( year == 4 ) {
            gift = " Wool ";
        }
        else if( year == 5 ) {
            gift = " Silk / Linen ";
        }
        else if( year == 6 ) {
            gift = " Lace ";
        }
        else if( year == 7 ) {
            gift = " Ivory ";
        }
        else if( year == 8 ) {
            gift = " Crystal ";
        }
        else if( year == 9 ) {
            gift = " China ";
        }
        else if( year == 10 ) {
            gift = " Silver ";
        }
        else if( year == 11 ) {
            gift = " Pearl ";
        }
        else if( year == 12 ) {
            gift = " Coral ";
        }
        else if( year == 13 ) {
            gift = " Ruby ";
        }
        else if( year == 14 ) {
            gift = " Sapphire ";
        }
        else if( year == 15 ) {
            gift = " Gold ";
        }
        else if( year == 16 ) {
            gift = " Emerald ";
        }
        else if( year == 17 ) {
            gift = " Diamond ";
        }
        else
            gift = "No traditional gift ";
    }
}

```

```
    }
} else {
    if( year == 4 ) {
        gift = " Fruit ";
    }
}
.
```

It appears that there is some kind of **else if** construct in Java, but there is not. Still, careful use of the rules for braces allows us to write code that nicely expresses a list of alternatives, even if the true compiler interpretation looks a little different.

Another way of expressing a long sequence of choices is by using a **switch** statement. A **switch** statement takes a single integer type value (**int**, **long**, **short**, **byte**, **char**) or a String and jumps to a case corresponding to the input. We can recode the anniversary gift example using a **switch** statement as follows.

```
String gift ;
switch( year ) {
    case 1: gift = " Paper "; break ;
    case 2: gift = " Cotton "; break ;
    case 3: gift = " Leather "; break ;
    case 4: gift = " Fruit "; break ;
    case 5: gift = " Wood "; break ;
    case 6: gift = " Candy / Iron "; break ;
    case 7: gift = " Wool / Copper "; break ;
    case 8: gift = " Bronze / Pottery "; break ;
    case 9: gift = " Pottery / Willow "; break ;
    case 10: gift = " Tin / Aluminum "; break ;
    case 11: gift = " Steel "; break ;
    case 12: gift = " Silk / Linen "; break ;
    case 13: gift = " Lace "; break ;
    case 14: gift = " Ivory "; break ;
    case 15: gift = " Crystal "; break ;
    case 20: gift = " China "; break ;
    case 25: gift = " Silver "; break ;
    case 30: gift = " Pearl "; break ;
    case 35: gift = " Coral "; break ;
    case 40: gift = " Ruby "; break ;
    case 45: gift = " Sapphire "; break ;
    case 50: gift = " Gold "; break ;
    case 55: gift = " Emerald "; break ;
    case 60: gift = " Diamond "; break ;
    default : gift = "No traditional gift "; break ;
}
```

Just like an **if** statement, a **switch** statement always has parentheses enclosing some argument. Unlike an **if**, the argument of a **switch** must be some kind of data that can be expressed as an integer or a String, not a **boolean**. For each of the possible values you want the **switch** to handle, you write a **case** statement. A **case** statement consists of the keyword **case** followed by a constant value, either a literal or a named constant, then a colon. When executed, the JVM jumps to the matching **case** label and starts executing code there. If there is no matching **case** label, the JVM goes to the **default** label. If there is no **default** label, the entire **switch** statement is skipped.

One unusual feature of **switch** statements is that execution *falls through* **case** statements. This means that you can use many different **case** statements for a single segment of executable code. The execution of code in a **switch** statement jumps out when it hits a **break** statement. However, **break** statements are not required, as shown in this **switch** statement that gives location information for all of the telephone area codes in New York state.

```
String location = "";
switch ( code ) {
    case 917: location = " Cellular ";
    case 212:
    case 347:
    case 646:
    case 718: location += " New York City "; break;

    case 315: location = " Syracuse "; break;

    case 516: location = " Nassau County "; break;

    case 518: location = " Albany "; break;

    case 585: location = " Rochester "; break;

    case 607: location = " South Central New York "; break;

    case 631: location = " Suffolk County "; break;

    case 716: location = " Buffalo "; break;

    case 845: location = " Lower Hudson Valley "; break;

    case 914: location = " Westchester County "; break;

    default : location = " Unknown Area Code "; break;
}
```

As you can see, five different area codes are used by New York City. By leaving out the `break` statements, values of 212, 347, 646, and 718 all have “[New York City](#)” stored into location. Area code 917 was originally designated for cellular phones and pagers although now it has some landlines. By cleverly putting the statement for 917 ahead of the other New York City entries, a value of 917 first stores “[Cellular:](#)” into location and then falls through and appends “[New York City](#)”. For each of these five area codes, execution in the `switch` statement ends only when the `break` statement is reached.

The remaining nine area codes are by themselves. Each of them does a single assignment and then breaks out of the `switch` block. Finally, the `default` label is used if the area code is not one of the ones specified. Note that we have ordered the (non-NYC) area codes in ascending order for the sake of readability. As you can see with the 917 example, there is no rule about the ordering of the labels. Even the `default` label can occur anywhere in the `switch` block you want, although it is common to put it at the end. Also, the `break` after the `default` label is unnecessary because execution exits the `switch` block anyway. Nevertheless, it is always wise to end on a `break`, in the event that you add more cases in later.

Carelessness is always something to watch out for in `switch` statements. Leaving out a `break` statement can cause disastrous and difficult to discover bugs. The compiler does not warn you about missing `break` statements, either. It is entirely your job to use them appropriately. Because of the dangers involved, it is often better to use `if-else` statements. Any `switch` statement can be rewritten as some combination of `if-else` statements, but the reverse is not true. The main benefit of `switch` statements is the ability to list many alternatives clearly. Their drawbacks include the ease of making a mistake, an inability to express ranges of data or most types (`double`, `float`, or any reference type other than `String`), and limited expressive power. They should be used only when their benefit of clearly displaying a list of data outweighs the drawbacks. Note that Java 7 added `String` values as legal input to a `switch`. If you are coding in Java 7 or later, you can use `String` literals for your cases, but then your code is not compatible with earlier versions.

Next we give a number of examples to help you get more familiar with `switch` statements.

Example 4.4: Days in the month

The use of `switch` statements is usually a little more special purpose than `if` statements. Nevertheless, there are many problems where their fall-through behavior can be useful. Imagine that you need to write a program that gives the length of each month (assume that February always has 28 days). Given the month as a number, we can easily write a program that maps the number of the month to the number of days it contains.

Program 4.4: This program computes the number of days in a given month. (DaysInMonth.java)

```
1 import java.util.*;
2
3 public class DaysInMonth {
4     public static void main ( String [] args ) {
5         Scanner in = new Scanner ( System.in );
6         System.out.print ( " Please a month number (1 -12) : " );
```

```

7     int month = in.nextInt ();
8     int days = 0;
9     switch ( month ) {
10        case 2: days = 28; break ;
11
12        case 4:
13        case 6:
14        case 9:
15        case 11: days = 30; break ;
16
17        case 1:
18        case 3:
19        case 5:
20        case 7:
21        case 8:
22        case 10:
23        case 12: days = 31; break ;
24    }
25    System.out.println ("The month you entered has " +
26        days + " days.");
27 }
28 }
```

This program has a single label for February setting days to 28. Then, there are labels for April, June, September, and November, months that each have 30 days. Finally, the large block of January, March, May, July, August, October, and December all set days to 31. It would be easy to extend this code to prompt the user for a year so that you could integrate the leap year code from above for the February case. Note also that we do not have a **default** label. You might want to set days to some special value (like -1) for invalid months. ■

Example 4.5: Ordinal numbers

The term *ordinal numbers* refers to numbers that are used for ordering within a set of items: first, second, third, and so on. When writing these numbers with numerals in English, it is common to append two letters to the end of the numeral to give the reader a clue that these numerals should be read with their ordinal names: 1st, 2nd, 3rd, and so on.

Unlike most things in English, the rules for deciding which two letters are relatively simple. If the number ends in a 1, the letters “st” should generally be used. If the number ends in a 2, the letters “nd” should generally be used. If the number ends in a 3, the letters “rd” should generally be used. For most other numbers, the letters “th” should be used. We can use a **switch** statement to write a program to give the correct ordinal endings for most numbers as follows.

Program 4.5: This program appends the appropriate suffix to a numeral to make it an ordinal.
(*Ordinals.java*)

```

1 import java.util.*;
2
3 public class Ordinals {
4     public static void main ( String [] args ) {
5         Scanner in = new Scanner ( System.in );
6         System.out.print ("Please enter a positive number : ");
7         int number = in.nextInt ();
8         String ending ;
9         switch ( number % 10 ) {
10            case 1: ending = "st"; break ;
11            case 2: ending = "nd"; break ;
12            case 3: ending = "rd"; break ;
13            default : ending = "th"; break ;
14        }
15        System.out.println ("Its ordinal version is "
16                           + number + ending + ".");
17    }
18 }
```

This program prompts and then reads in an `int` from the user. We then find the remainder of number when it is divided by 10, yielding its last digit. Based on this digit, we can pick from the four possibilities and output the correct ordinal number in most cases. Unfortunately, the names for English numbers do not follow the normal pattern of tens place name followed by ones place name between 11 and 19, inclusive, and the ordinals for any number ending in 11, 12, or 13 will be given the wrong suffix by our code. We leave a more complete solution as an exercise. ■

Exercise 4.12

Example 4.6: Astrology

Many cultures practice astrology, a tradition that the time of a person's birth impacts his or her personality or future. One important element of Chinese astrology is their zodiac, consisting of 12 animals. Each consecutive year in a 12-year cycle corresponds to an animal. Because this system repeats, the year one is born in modulo 12 identifies the animal. Below is a table giving these values. For example, if you were born in 1979, $1979 \bmod 12 \equiv 11$, and thus you would be a Ram. Note that this arrangement is based on years in the Gregorian calendar. Chinese astrologers do not list the Monkey as the first animal in the cycle.

Unfortunately, this table is not very accurate because it is based on numbering from the Gregorian calendar. The years in question actually start and end based on Chinese New Year, which occurs between January 21 and February 20. As a consequence, you may miscalculate your animal if your birthday is early in the year. Let's ignore this problem for the moment and write a program using a `switch` statement designed to correctly output the animal corresponding to an input birth year.

Year	Animal modulo 12
------	------------------

Monkey 0

Rooster 1

Dog 2

Boar 3

Rat 4

Ox 5

Tiger 6

Rabbit 7

Dragon 8

Snake 9

Horse 10

Ram 11

Program 4.6: This program determines a Chinese zodiac animal based on birth year.
(ChineseZodiac.java)

```
1 import java.util.*;
2
3 public class ChineseZodiac {
4     public static void main ( String [] args ) {
5         Scanner in = new Scanner ( System.in );
6         System.out.print (" Please enter a year : ");
7         int year = in.nextInt ();
8         String animal = "";
9         switch ( year % 12 ) {
10             case 0: animal = " Monkey "; break ;
11             case 1: animal = " Rooster "; break ;
12             case 2: animal = " Dog "; break ;
13             case 3: animal = " Boar "; break ;
14             case 4: animal = " Rat "; break ;
15             case 5: animal = " Ox "; break ;
16             case 6: animal = " Tiger "; break ;
17             case 7: animal = " Rabbit "; break ;
18             case 8: animal = " Dragon "; break ;
19             case 9: animal = " Snake "; break ;
20             case 10: animal = " Horse "; break ;
21             case 11: animal = " Ram "; break ;
22         }
23         System.out.println ("The Chinese zodiac animal for "
24             " this year is: " + animal );
25     }
26 }
```

Sign	Symbol	Date Range
Aries	The Ram	March 21 to April 19
Taurus	The Bull	April 20 to May 20
Gemini	The Twins	May 21 to June 20
Cancer	The Crab	June 21 to July 22
Leo	The Lion	July 23 to August 22
Virgo	The Virgin	August 23 to September 22
Libra	The Scales	September 23 to October 22
Scorpio	The Scorpion	October 23 to November 21
Sagittarius	The Archer	November 22 to December 21
Capricorn	The Sea-Goat	December 22 to January 19
Aquarius	The Water Bearer	January 20 to February 19
Pisces	The Fishes	February 20 to March 20

In Western astrology, an important element associated with a person's birth is also called a zodiac sign. The dates for determining this kind of zodiac sign are given by the preceding table.

If you want to implement the rules for this zodiac in code, a **switch** statement is a good place to start, but you also have to put **if** statements for each month to test the exact range of dates.

Program 4.7: This program determines Western zodiac signs based on birth month and day.
(WesternZodiac.java)

```

1 import java.util.*;
2
3 public class WesternZodiac {
4     public static void main ( String [] args ) {
5         Scanner in = new Scanner ( System.in );
6         System.out.print (" Please enter a month number (1 -12): ");
7         int month = in.nextInt ();
8         System.out.print (" Please enter a day number in that month (1 -31) : ");
9         int day = in.nextInt ();
10        String sign = "";
11        switch ( month ) {
12            case 1: if( day < 20 )
13                sign = " Capricorn ";
14            else
15                sign = " Aquarius ";
16            break ;
17            case 2: if( day < 20 )
18                sign = " Aquarius ";
19            else
20                sign = " Pisces ";

```

```
21      break ;
22 case 3: if( day < 20 )
23     sign = " Pices ";
24 else
25     sign = " Aries ";
26 break ;
27 case 4: if( day < 20 )
28     sign = " Aries ";
29 else
30     sign = " Taurus ";
31 break ;
32 case 5: if( day < 21 )
33     sign = " Taurus ";
34 else
35     sign = " Gemini ";
36 break ;
37 case 6: if( day < 21 )
38     sign = " Gemini ";
39 else
40     sign = " Cancer ";
41 break ;
42 case 7: if( day < 23 )
43     sign = " Cancer ";
44 else
45     sign = " Leo ";
46 break ;
47 case 8: if( day < 23 )
48     sign = " Leo ";
49 else
50     sign = " Virgo ";
51 break ;
52 case 9: if( day < 23 )
53     sign = " Virgo ";
54 else
55     sign = " Libra ";
56 break ;
57 case 10: if( day < 23 )
58     sign = " Libra ";
59 else
60     sign = " Scorpio ";
61 break ;
62 case 11: if( day < 22 )
63     sign = " Scorpio ";
```

```

64     else
65         sign = " Sagittarius ";
66     break ;
67 case 12: if( day < 20 )
68     sign = " Sagittarius ";
69 else
70     sign = " Capricorn ";
71 break ;
72 }
73 System.out.println ("The zodiac sign is: " + sign );
74 }
75 }
```

This program is just slightly more complex than the program for the Chinese zodiac. You still need to jump to 12 different cases (numbered 1-12 instead of 0-11), but additional day information is needed to pin down the sign. ■

4.4 Solution: Monty Hall

We now return to the Monty Hall simulation described at the beginning of the chapter. Recall that the Random class allows us to generate all kinds of random values. To implement this simulation successfully, our program must make all the decisions needed to set up the game for the user as well as respond to the user's input. We begin with the `import` statement that is necessary to use both the Scanner and Random class and then define the MontyHall class.

```

1 import java.util.*;
2
3 public class MontyHall {
4     public static void main ( String [] args ) {
5         Random random = new Random ();
6         int winner = random.nextInt (3) ;
7         Scanner in = new Scanner ( System.in );
8         System.out.print (" Choose a door ( enter 0, 1, or 2): ");
9         int choice = in.nextInt ();
10        int alternative ;
11        int open ;
```

In the `main()` method we first decide which of the three doors is the winner. To do so, we instantiate a `Random` object and use it to generate a random number that is either 0, 1, or 2 by calling the `nextInt()` method with an argument of 3. We could have added 1 to this value to get a random choice of 1, 2, or 3, but many counting systems in computer science start with 0 instead of 1. You might as well get used to it. Next, we prompt the user to pick from the three doors and read the choice. Finally, we declare two more `int` values to keep track of which door to open and which door is the alternative that the user can choose to change over to.

```

13     if( choice == winner ) {
14         int low ;
15         int high ;
16         if( choice == 0 ) {
17             low = 1;
18             high = 2;
19         }
20         else if( choice == 1 ) {
21             low = 0;
22             high = 2;
23         }
24         else { // choice == 2
25             low = 0;
26             high = 1;
27         }
28         // randomly choose between other two doors
29         double threshold = random.nextDouble ();
30         if( threshold < 0.5 ) {
31             alternative = low ;
32             open = high ;
33         }
34         else {
35             alternative = high ;
36             open = low;
37         }
38     }

```

Now we have to navigate a complicated series of decisions. In this segment of code, we are tackling the possibility that the user happened to choose the winning door. To obey the rules of the game, we must randomly pick which of the two other doors to open. First, we determine which are the other two doors and save them in low and high, respectively. Then, we generated a random number. If the random number is less than 0.5, we keep the lower numbered door as an alternative choice for the user and open the higher numbered door. If the random number is greater than or equal to 0.5, we do the opposite.

```

39     else {
40         alternative = winner ;
41         if( choice == 0 ) {
42             if( winner == 1 )
43                 open = 2;
44             else
45                 open = 1;
46         }
47         else if( choice == 1 ) {

```

```

48     if( winner == 0 )
49         open = 2;
50     else
51         open = 0;
52 }
53 else { // choice == 2
54     if( winner == 0 )
55         open = 1;
56     else
57         open = 0;
58 }
59 }
```

This **else** block covers the case that the player did not pick the winning door the first time. Unlike the previous code segment, we no longer have a choice of which door to open. This time, we must always make the winner the alternative for the user to pick. Then, we simply determine which door is leftover so that we can open it. Note that the braces surrounding the blocks for each of the braces surrounding the blocks for each of the three possible values of choice are not necessary but are included for readability.

```

60     System.out.println ("We have opened Door " + open +
61         ", and there is junk behind it!");
62     System.out.print ("Do you want to change to Door " +
63         alternative + " from Door " + choice +
64         "? ( Enter 'y' or 'n'): ");
65     String change = in. next ();
66     if( change.equals ("y") )
67         choice = alternative ;
68     System.out.println ("You chose Door " + choice );
69     if( choice == winner )
70         System.out.println (" You win a pile of gold !");
71     else
72         System.out.println (" You win a pile of junk.");
73 }
74 }
```

This final segment of code informs the user which door has been opened and prompts the user to change his or her decision. Depending on the final choice, the program says whether or not the user wins gold or junk.

4.5 Concurrency: Selection

The selection primitives (**if** and **switch** statements) seem to have little to do with concurrency or parallelism. Selection allows you to choose between alternatives while concurrency is about the

interaction between different threads of execution. As it turns out, there are two reasons why selection and concurrency are deeply related to each other.

The first reason is that selection is one of the most basic tools in Java. It is impossible to go more than a few lines of a code without encountering a selection primitive, usually an **if** statement. Concurrent programs are not exempt from this dependence on **if** statements. Making decisions is at the heart of all programming languages running on all computers.

The second reason is more troubling and is related to a problem with some concurrent programs called a *race condition*, which is discussed in great detail in [Chapter 14](#). Remember, one of the biggest challenges of programming a computer is thinking in a completely sequential and logical way. Each line of code is executed one after the other. Adding in **if** statements means that some code is executed only if a condition is true and skipped otherwise. Consider the following fragment of code:

```
if( ! matches.areLit() && ! flyingSparks ) {  
    storageRoom.enter();  
    dynamite.unpack();  
}
```

In this **if** statement, the imaginary agent only enters the storage room and unpacks the dynamite if the matches are not lit and there are no flying sparks. When execution reaches the first line inside the **if** block, we are certain that `matches.areLit()` returned **false** and `flyingSparks` is **false**. This is a one-time check. If the first thing that happens inside the **if** block is code that lights the matches, Java will **not** jump out of the **if** statement.

As always, the programmer is responsible for making an **if** statement that makes sense. It is possible that entering the storage room or unpacking the dynamite causes sparks to fly or matches to burst into flames spontaneously, but it seems unlikely. If the `storageRoom` and `dynamite` objects were written by other people, we would expect their documentation to explain unusual side-effects of this kind. In a sequential program, the programmer can be reasonably sure that it is safe to unpack the dynamite.

Consider another fragment of code:

```
matches.light();  
flyingSparks = sparklers.light( matches );
```

This code appears to light the matches and uses the lit matches to set some sparklers on fire. Presumably, if the process was successful, `flyingSparks` will have the value **true**. This code is reasonable and potentially helpful. If you were celebrating the 4th of July or needed to signal a passing helicopter to rescue you from a desert island, lighting sparklers could be a great idea. This sparkler-lighting code could occur before the dynamite-unpacking code or after it, but the protection of the **if** statement keeps our hero from being blown up if he tries to unpack the dynamite with lit sparklers, in a sequential program.

In a concurrent program, all bets are off. Another thread of execution can be operating at the very same time. It's as if our hero is trying to unpack the dynamite while the villain is lighting sparklers and tossing them into the storage room. If the thread of execution gets to the **if** statement and makes

sure that the matches aren't lit and that there are no flying sparks, it continues onward. If sparks start flying after that check, it still continues onward, oblivious of the fact. Even though this risk of explosion exists, it depends on the timing of the two (or more) concurrent threads of execution. It might be possible to run a program 1,000 times with no problem. But if the timing is wrong on the 1,001st time, **BOOM!**

At this point, you do not need to worry about values inside your **if** statements being changed by other segments of code, but that problem is at the heart of why concurrent programming can be so difficult. Whether or not you are programming concurrently, it is always important to keep in mind the assumptions your code makes and the way different parts of your program interact with each other.

Exercises

Conceptual Problems

- 4.1 Given that x , y , and z are propositions in Boolean logic, make a truth table for the expression $(\neg(x \wedge \neg y) \oplus \neg z)$.
- 4.2 What is the value of the Boolean expression $\neg((T \oplus F) \wedge \neg(F \vee T))$?
- 4.3 The calculation to determine the leap year given in [Example 4.1](#) uses three **if** statements and three **else** statements. Write the leap year calculation using a single **if** and a single **else**. Feel free to use **boolean** connectors such as `||` and `&&`.
- 4.4 The XOR operator (`^`) is useful for combining **boolean** values, but it can be replaced with a more commonly used **relational** operator in Java. Which one?
- 4.5 De Morgan's laws are the following, which show that the process of negating a clause changes an AND to an OR and vice versa.

$$\neg(x \wedge y) = \neg x \vee \neg y$$

$$\neg(x \vee y) = \neg x \wedge \neg y$$

Create truth tables to verify both of these statements.

- 4.6 Use De Morgan's laws given above to rewrite the following statement in Java to an equivalent statement that contains no negations.

```
boolean value = !(( x != 4) && (y < 2));
```

- 4.7 Consider the following fragment of code.

```
int x = 5;
int y = 3;
if( y > 10 && (x = 10) > 5 )
    y++;
System.out.println ("x: " + x);
System.out.println ("y: " + y);
```

What is the output? Is the output changed if the condition of the **if** statement is changed to $y > 10 \& (x = 10) > 5$? Why?

4.8 Consider the following fragment of code.

```
int a = 7;  
if( a++ == 7 )  
    System.out.println (" Seven ");  
else  
    System.out.println (" Not seven ");
```

What is the output? Is the output changed if the condition of the **if** statement is changed to $++a == 7$? Why? Note: It is generally wise to avoid increment, decrement, and assignment statements in the condition of an **if** statement because of the confusion that can arise.

Programming Practice

- 4.9 (a) Write a program that reads in two **double** values and prints the larger of the two of them.
(b) Expand the ideas from the previous program into a program that reads in **three double** values and prints the largest of the three out. Note: You should use nested **if** statements.

4.10 Write programs that:

- (a) Read an **int** value from the user specifying a certain number of cents. Use **if** statements to print out the name of the corresponding coin in U.S. currency according to the table below. If the value doesn't match any coin, print no coin.

Cents	Coin
1	penny
5	nickel
10	dime
25	quarter
50	half-dollar
100	dollar

- (b) Read a **String** value from the user that gives one of the 6 coin names given in the table above. Use **if** statements to print out the corresponding number of cents for the input. If the name doesn't match any coin, print unknown coin.

4.11 Re-implement both parts from Exercise 4.10 using **switch** statements instead of **if** statements. Note: You cannot use **switch** statements for (b) unless you are using Java 7 or later.

4.12 Expand the program given in [Example 4.5](#) to give the correct suffixes (always “th”) for numbers that end in 11, 12, and 13. Use the modulus operator to find the last two digits of the number. Using either an **if** statement, a **switch** statement, or a combination, check for those three cases before going into the normal cases.

- 4.13 At the bottom of Section 4.3.3, we use a **switch** statement to determine the location of various area codes in New York state. Write an equivalent fragment of code using **if-else** statements instead.
- 4.14 Every member of your secret club has an ID number. These ID numbers are between 1 and 1,000,000 and have two special characteristics: They are multiples of 7 and all end with a 3 in the one's place. For example, 63 is the smallest such value, and 999,943 is the largest such value. Write a program that prompts the user for an **int** value, read it in, and then say whether or not it could be used as an ID number. Note: You need to use the **%** operator in two different ways to test the value correctly.
- 4.15 According to the North American Numbering Plan (NANP) used by the United States, Canada, and a number of smaller countries, a legal telephone number takes the form XYY-XYY-YYYY, where X is any digit 2-9 and Y is any digit 0-9. Write a program that reads in a String from the user and verifies that it is a legal NANP phone number. The length of the entire String must be 12. The fourth and eighth characters in the String (with indexes 3 and 7) must be hyphens (-), and all the remaining digits must be in the correct range. Use the **charAt()** method of the **String** class to get the **char** value at each index. Note: There are several ways to structure the **if** statements you need to use, but the number of conditions may become large. (23 or more!)
- 4.16 Re-implement the solution to the Monty Hall program given in [Section 4.4](#) using **JOptionPane** to GUI generate GUIs for input and output.

Chapter 5

Repetition

Q-Tip: *You on point, Phife?*

Phife Dawg: *All the time, Tip.*

Q-Tip: *You on point, Phife?*

Phife Dawg: *All the time, Tip.*

Q-Tip: *You on point, Phife?*

Phife Dawg: *All the time, Tip.*

Q-Tip: *Well, then grab the microphone and let your words rip.*

—A Tribe Called Quest

5.1 Problem: DNA searching

The world of bioinformatics is the intersection between biology and computer science. Mapping the human genome would have been impossible without computers. Sequencing genomes, determining the function of specific genes, the analysis and prediction of protein structures, and biomedical imaging are just a few of the areas under the umbrella of bioinformatics. Much fascinating research is being done in this area as biologists become better programmers and computer scientists learn more about biology.

Because of its fundamental importance and the incredible amount of information involved, with tens or hundreds of millions of base pairs of DNA in each human chromosome, DNA is a central focus of bioinformatics. As you may know, a DNA strand is made up of a sequence of four nucleotide bases: adenine, cytosine, guanine, and thymine. These bases are usually abbreviated as A, C, G, and T, respectively.

Searching for a specific DNA subsequence within a larger sequence is a common task for biologists to perform. Your goal is to write a program that will search for a subsequence and report how many times it was found within the sequence. For example, if you are given the sequence ATTAGACCATA and asked to search for CAT, your program should output 1, since there is exactly 1 occurrence of CAT within ATTAGACCATA. One feature of this problem that makes it more interesting is that occurrences can overlap. For example, given the sequence TATTATTAGATTA and asked to search for TATTA, the correct answer is 2. The sequence begins with a TATTA, but the third T in the sequence is also the first T in a second instance of TATTA.

In [Chapter 4](#), you learned tools that allow you to do comparisons and make choices based on the results. These tools will still be useful. For example, when you come across a **char** in the sequence, you know how to compare it to a **char** in the subsequence you are searching for. The tools you do not yet have are those that allow repetition. Because this problem requires the program to process a DNA sequence of arbitrary length, we will need some way to perform a repetitive action.

5.2 Concepts: Repetition

You now know how to write choices into a Java program, but, so far, each choice can only be made once. So, if you want the computer to do a lot of things, you have to type a lot of things. One of the big disadvantages of computers is that they have no intelligence: They can follow instructions blindly, but they cannot do anything else. One of the big advantages of computers is that they are fast. Modern computers can perform mathematical operations billions of times faster than human beings. So, we need to give it some instructions that allow it to do tasks over and over. Such an instruction must have two components to be useful: It must have a way to change the task slightly each time so that each task is useful. It must also have a way to decide when to stop, otherwise it will continue forever.

The first component is the more subtle one. Crafting a set of instructions so that each repetition of the task does the appropriate thing will be different for every problem. The second component is easier to describe: We are going to rely on Boolean logic, just as we did for conditional statements. The main tool for repetition in Java and many other languages is called a *loop*. The body of a loop contains the task to be performed by the loop. The rest of the loop, at the very least, contains a condition. Every time we finish the task given in the body of the loop, we will check the condition. If the condition is true, we will do the task again. If the condition is false, we are done with the loop and can move on to the code that comes afterward.

One of the difficulties of programming a computer is that we must be very explicit. Even the most obvious tasks must be spelled out meticulously. Let's consider a simple task, one that we perform every day. If we are in a room and we want to leave, we simply walk out of the nearest door. Assuming there is only one door in the room, how can we describe this process by breaking it down into the steps we (literally) take? Perhaps we could say the following.

Walk toward the door until you reach it.

This statement is a little more specific than *Leave the room*, but it does not conform nicely to the paradigm of a loop, that is, a clearly separated task and a condition. The following is better.

While you are not at the door,

Take a step toward the door.

Now we have good separation between the work done and the condition for repeating. What is the task performed in the body of this loop and what is the condition? The task is taking a step towards the door. The condition is **not** being at the door. It seems a little awkward to include that “not,” but, in our definition of loops, the body is executed as long as the condition is true.

In a loop, we call each execution of the body of the loop an *iteration*. When we say that a program *iterates* over the statements in a loop, we are referring to a single pass through the body of a loop. In this case, the loop will iterate however many times there are steps to the door from the starting position. It is even possible that the loop will iterate zero times: The person following this set of instructions might already be at the door.

It's hard to get away from numbers in a computer program, especially since everything is

fundamentally stored as numbers inside of a computer. So, the most common kind of loop is one that iterates a fixed number of times. For example, your morning exercise routine might include jumping rope 100 times. We could formulate a loop to do that like so.

Set your counter to 0.

While your counter is less than 100,

Jump rope.

Increase your counter by 1.

This loop required some set up to make sure that the counter value started at the right place. Then, the work done by the loop is the actual rope jumping and the counter increment. The condition is the counter being less than 100. Note that this is strictly **less** than 100. After the first jump, the counter will be incremented to 1. After the 100th jump, the counter will be incremented to 100. Since 100 is **not** less than 100, the loop will exit. If the condition was the counter being less than or equal to 100, the person following the instructions would jump 101 times.

Input can also be a factor in loop repetitions. For example, you might be a soldier training in the U.S. Marine Corps. Perhaps your drill sergeant has commanded you to do push-ups until he says you can stop. We might formulate a loop to do this as follows.

Do:

Push-up.

Ask the drill sergeant if you can stop.

While the answer is “no.”

As is the case with user input, you must often go into the loop at least once to get the input. This loop requires the soldier to do at least one push-up before asking to stop. Some systems might use input but have other constraints. A more realistic version of this loop might be the following.

Do:

Push-up.

Ask the drill sergeant if you can stop.

While the answer is “no” and you haven’t collapsed.

Remember, the condition for a loop should be a Boolean, and the loop runs as long as the condition is true. However, there is no reason why the Boolean cannot be a complicated expression using all the Boolean logic we have come to know and love.

Of course it is possible to *nest* loops. Nesting loops means putting one loop inside of another, similar to the way that conditional statements could be nested inside of other conditional statements. Just like any other statement, an inner loop will be run as many times as the outer loop runs. Of course, the statements inside of the inner loop will be run according to the conditions of that loop. So, if an outer loop runs 10 times and an inner loop runs 50 times, a statement in the body of an inner loop

would run 500 times!

As an example, if you are working out, you might do several sets of bench presses with a fixed number of reps in each set. If you did 3 sets of 15 bench presses each, your workout program might look like this:

Set your set counter to 0.

While your set counter is less than 3,

Set your rep counter to 0.

While your rep counter is less than 15,

Do a bench press.

Increase your rep counter by 1.

Rest for 2 minutes.

Increase your set counter.

This way of describing the work out program seems tedious. Most of the description is structural: conditions for the loops and increments for the counters. The only “real” activities are the bench press and the resting. As you can see, the bench press is inside the inner rep loop and will be executed 15 times each time for each complete execution of the inner rep loop. Since the inner rep loop sits inside the outer set loop, it will be executed 3 times, giving a grand total of 45 bench presses. Resting, however, is after the inner rep loop but still contained in the outer set loop and will be executed 3 times, totaling 6 minutes of rest.

As with conditionals, writing out loops in English is tedious and imprecise. In the next section, we will discuss the tools for writing loops in Java. Because Java was designed with loops as a central tool, we can write loops much more succinctly than in English, squeezing a lot of information into a small space. Because we pack so much information into them, loops can look daunting at first. Remember that the syntax we will introduce is only the formal Java way of expressing a condition and a list of instructions to execute repeatedly.

5.3 Syntax: Loops in Java

The Java programming language contains three differently named kinds of loops: **while** loops, **for** loops, and **do-while** loops. All of them allow you to write code that will be executed repeatedly. In fact, any program that uses one style of loops to solve a problem could be converted to use either of the other two kinds. The three kinds are provided in Java partly so that it is easy to code certain typical kinds of repetition and partly because the C language, an ancestor of Java, contained these three. We will begin by describing **while** loops because they have the simplest form and then move on to the other two kinds. We will then explain the syntax for nesting together multiple loops and finally discuss several of the common pitfalls encountered by programmers who are coding loops.

5.3.1 while loops

Superficially, the syntax of a **while** loop resembles an **if** statement. It starts with the keyword **while**

followed by a **boolean** condition in parentheses with a block of code surrounded by braces (`{ }`) afterward. This similarity is not accidental. The only difference between the two is that the body of the **if** statement will run only single time, while the body of the **while** loop will run as long as the condition remains **true**. [Figure 5.1](#) shows the pattern of execution for **while** loops.

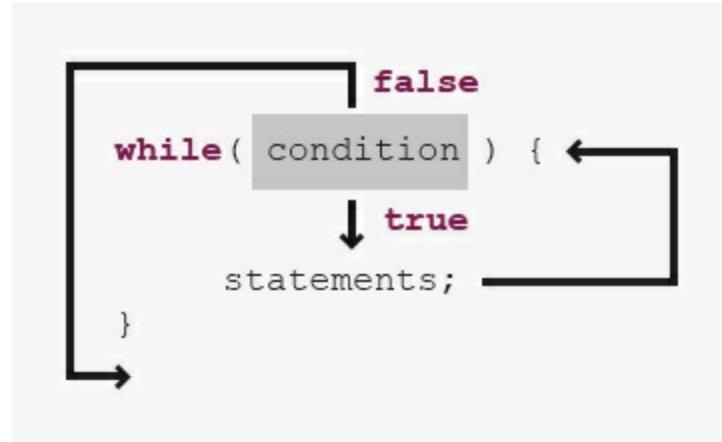


Figure 5.1: If the condition is **true**, all of the statements in the body of the loop are executed, and then the condition is checked again. When the check is **false**, execution skips past the body of the loop.

If we assume that the **boolean** value `atDoor` says whether or not we have reached the door and the method `walkTowardsDoor()` allows us to take one step closer to the door, we could formulate our example from the beginning of the previous section as follows.

```
while ( ! atDoor ) {  
    atDoor = walkTowardsDoor ();  
}
```

Here we assume that the `walkTowardsDoor()` method gives back a **boolean** value that is **true** if we have reached the door and **false** otherwise. Unless the `walkTowardsDoor()` method is able to change the value of `atDoor`, the loop will repeat forever, a phenomenon known as an *infinite loop*.

```
while ( true ) {  
    System.out.println (" Help me!");  
}
```

This loop is an example of an infinite loop. If you run this code inside of a program, it will print out an endless succession of `Help me!` messages. Be prepared to stop the program by typing `Ctrl-C` (hold down the Control key and press C) because it will not end otherwise. Not all infinite loops are this obvious. A programmer will not usually use **true** as the condition of a loop, but doing so is not always wrong. Some loops are expected to continue for quite some time with no definite end. To leave a loop abruptly, you can use the **break** command.

```
while ( true ) {  
    System.out.println (" Help me!");  
    break ;  
}
```

This loop will only print out a single Help me! before exiting. A **break** command can be used with an **if** statement to make a loop that repeats more than once.

```
int counter = 0;
while ( true ) {
    System.out.print (" the loop ");
    counter++;
    if( counter >= 3 )
        break ;
}
System.out.println ("is on fire !");
```

This loop will print out the loop the loop the loop is on fire! Of course, the **break** statement unnecessarily complicates the code. We could have written equivalent code as follows.

```
int counter = 0;
while ( counter < 3 ) {
    System.out.print (" the loop ");
    counter++;
}
System.out.println ("is on fire !");
```

Now we move on to a more complicated example that can print out the binary equivalent of a number.

Example 5.1: Binary Conversion

As we discussed in [Chapter 1](#), binary numbers are the building blocks of every piece of data inside of a modern computer's memory. Integers are stored in binary. The representation of floating point numbers is more complicated, but it also uses 1s and 0s. Even the **char** data type and the String values built from them are fundamentally stored as binary numbers. For this reason, computer scientists tend to be familiar with the base 2 number system and how to convert between it and base 10, our usual number system.

In base 10, the number 379 is equal to $3 \cdot 100 + 7 \cdot 10 + 9 \cdot 1 = 3 \cdot 10^2 + 7 \cdot 10^1 + 9 \cdot 10^0$. Moving from right to left, the value of each place increases by a factor of 10. A binary number is the same, except that the increase is by a factor of 2 and no single digit is greater than 1. Thus, the number $101011_2 = 1 \cdot 2^5 + 0 \cdot 2^4 + 1 \cdot 2^3 + 0 \cdot 2^2 + 1 \cdot 2 + 1 \cdot 2^0 = 1 \cdot 32 + 0 \cdot 16 + 1 \cdot 8 + 0 \cdot 4 + 1 \cdot 2 + 1 \cdot 0 = 43$. In binary, the number $379 = 101111011_2$.

To convert a number n to binary, we first find the largest power of 2 that is not larger than n . Then, we begin a repetitive process that stops when the power of 2 under consideration is 0. If 2 raised to the current power is bigger than n , we print out a 0 because that power is too big for n . Otherwise, we print out a 1, subtract 2 raised to that power from n , and move on to the next smaller power of 2. This process will print a 0 for every power of 2 that is not in n and a 1 for every one that is, giving exactly the definition of a number written in base 2.

Program 5.1: This program outputs a binary representation of a decimal number.
(DecimalToBinary.java)

```
1 import java.util.*;
2
3 public class DecimalToBinary {
4     public static void main ( String [] args ) {
5         Scanner in = new Scanner ( System.in );
6         System.out.print (" Please enter a base 10 number : ");
7         int number = in.nextInt ();
8         int power = 1;
9         while ( power <= number /2 )
10             power *= 2;
11         while ( power > 0 ) {
12             if( power > number )
13                 System.out.print (0);
14             else {
15                 System.out.print (1);
16                 number -= power ;
17             }
18             power /= 2;
19         }
20     }
21 }
```

The first **while** loop in this program doubles the value of power until doubling it again would make it larger than number. We go up to and including number/2, otherwise we would stop when power was larger than number. After that loop, we begin repeatedly checking to see if a given power of 2 is bigger than the value left in number. If it is, we know that we do not use that power. If it is not, we do and must remove that power from the value of number.

You may have been tempted to solve this problem by determining if a given number is even or odd. If it is even, then you record a 0, and if it is odd, then you record a 1. You could then divide the number by two and repeat the process of determining whether it is even or odd. You could continue this process until the number became 0. This procedure requires only a single **while** loop and would give the digits of the number in base 2. Unfortunately, you would get the digits in reverse order. Because we write our numbers with the most significant digit on the left, we had to use the code given above to first find the largest value and work backwards, in order to determine the binary digits in the correct sequence. ■

5.3.2 for loops

Let's return to our code that prints out the loop the loop is on fire!

```
int counter = 0;
while ( counter < 3 ) {
```

```

System.out.print (" the loop ");
counter++;
}
System.out.println ("is on fire !");

```

This code involves some initialization, a condition, and an update, as many loops do. The initialization sets counter to 0. The condition checks to make sure that counter is less than 3. The update increments counter by 1 every iteration of the loop. These three elements are so common that a special kind of loop called the **for** loop was designed with them explicitly in mind. Most **for** loops are dependent on a single counting variable. To make the loop easy to read, the initialization, condition, and update, all of which relate to this variable, are pulled into the header of the loop. We could code the previous **while** loop example more cleanly, using a **for** loop, as follows.

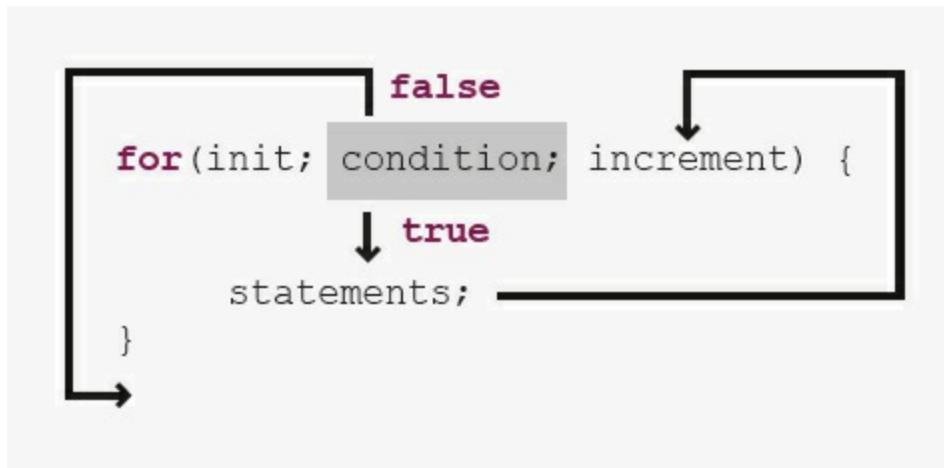
```

for ( int i = 0; i < 3; i++ ) {
    System.out.print (" the loop ");
}
System.out.println ("is on fire !");

```

The header of a **for** loop consists of those three parts: the initialization, the condition, and the update, all separated by semicolons. [Figure 5.2](#) shows the pattern of execution for **for** loops.

You may have noticed that we have changed the variable name used within the loop from counter to i. Doing so does not change the function of the code. We did so because using the variables i, j, and sometimes k is a very common practice with **for** loops. By using variables named like this, we are indicating that the variable is just a dummy counter that we are using to make the loop work, not some variable with a grander purpose. Also, with three uses of a single variable in the header of a **for** loop, a long variable name will take up a lot of space.



[Figure 5.2](#): The loop is initialized. If the condition is **true**, all of the statements in the body of the loop are executed, followed by the increment step. Then the condition is checked again. When the check is **false**, execution skips past the body of the loop.

for loops are used in Java programs more than the other two loops. They work well when you know how many times you want to iterate through the loop, which you often do. You can think of the first part of the **for** loop header as the starting point, the second part as the ending point, and the third

part as how you get from the start to the end. Many beginning programmers get stuck on the idea that every **for** loop starts with **int i = 0** and ends with **i++**. While this pattern is often true, there are many other ways to use a for loop. For example, we could print the powers of 2 that are less than 1000.

```
for ( int i = 1; i < 1000; i *= 2 ) {  
    System.out.println (i);  
}
```

This segment of code prints out 1, 2, 4, 8, 16, 32, 64, 128, 256, and 512 on separate lines, which are the powers of 2 from 2^0 up to 2^9 . As you know from discussion about binary numbers, powers of two have a special interest to computer scientists. Both of the examples of **for** loops we have given have only had a single executable line in the body of the loop. Like **if** statements, loops only require braces if their bodies have more than one executable line. Many of the **while** loops from the previous subsection could have been written without braces.

Just because a **for** loop already has a counting mechanism doesn't mean that we will not need other variables to perform useful tasks. For example, given a String, we could try to find the letter of the alphabet in the String which is closest to the end of the alphabet. For the String "Pluto is no longer a planet", the latest letter in the alphabet is 'u'. To write code that will do this job, we must use the counting variable from the **for** loop as an *index* into the String. Then, we must also have a temporary variable where we keep the latest letter found so far. To get the *i*th **char** from a String, we can use the **charAt()** method. The index of the first **char** in a String is 0, and the index of the last **char** is one less than the length of the String.

```
String s = "The quick brown fox jumps over the lazy dog.";  
String lower = s.toLowerCase ();  
char latest = ' ';  
char c;  
for ( int i = 0; i < lower.length (); i++ ) {  
    c = lower.charAt (i);  
    if( c >= 'a' && c <= 'z' && c > latest )  
        latest = c;  
}  
System.out.println ("The latest character in the alphabet " + " from your message is: " + latest + ".");
```

The first thing we do in this example is convert **s** to lower case, so that we are comparing all **char** values in the same case. Next, we run through **lower**, starting at index 0 and going until we reach the end of the String. For each **char**, we check to see if it is an alphabetic character and then if it is later in the alphabet than our current **latest**. If it is, we store it into **latest**. After the loop, we print out the value in **latest**. We have chosen the **char** ' ' because it is numerically earlier than all the letters in the alphabet. If the output is a space, we would know that none of the characters in **s** were alphabetic.

For the example given, the latest character in the alphabet is 'z' because of the word "lazy". One weakness in this code is that it will always search through the entire String, even if the letter 'z' has already been found. For the String "The quick brown fox jumps over the lazy dog.", we are not wasting too much time. However, if the String were "Zanzibar!" followed by the full text of *War and*

Peace, we would be wasting thousands and thousands of operations reading characters when we knew that '**z**' was going to be the latest letter, no matter what. So, we can rewrite our for loop so that it quits early if it reaches a '**z**'.

```
for ( int i = 0; i < lower.length (); i++ ) {  
    c = lower.charAt (i);  
    if( c >= 'a' && c <= 'z' && c > latest )  
        latest = c;  
    if( latest == 'z' )  
        break ;  
}
```

This version of the **for** loop will break out immediately if the latest is already a '**z**'. This code will work efficiently, but many professional programmers discourage the use of **break** except when absolutely necessary (like in a **switch** statement). If a **break** is used to exit the loop, this logic can be encoded into the condition of the loop. Thus, the same loop written with better style would be the following.

```
for ( int i = 0; i < lower.length () && latest != 'z'; i++ ) {  
    c = lower.charAt (i);  
    if( c >= 'a' && c <= 'z' && c > latest )  
        latest = c;  
}
```

For this final version of the loop, we have made the conditional portion of the header more complex. The comparison using **<** gives a **boolean** that we combine using **&&** with the **boolean** from the comparison using **!=**. As always, remember that the loop will continue iterating as long as the condition is **true**. Since we need both parts of the condition to be **true** to continue executing, we use the **&&** operator to connect them.

We apologize to international readers for focusing on the Latin alphabet used by English and many other Western European languages. It should be possible to make a localized version of this example with any alphabet by checking the return value of **Character.isLetter(c)**, which is valid for all single-character Unicode values, although the idea of alphabetical order does not really apply to some character systems like the hanzi and kanji of Chinese and Japanese. Regardless, using the **Character.isLetter()** method is recommended for almost all applications, since it is more general and more readable.

Example 5.2: Primality testing

Prime numbers are numbers whose only factors are 1 and themselves. If you have encountered prime numbers before, they probably seemed like a mathematical curiosity and nothing more. In fact, prime numbers are the basis of a very practical application of mathematics: cryptography. With the use of some math and very large prime numbers, computer scientists have devised techniques that make shopping online safer.

These techniques are beyond the scope of this book, but we can at least write some code to

determine if a number n is prime. To do so, we can simply divide n by all the numbers between 2 and $n - 1$. If none of the numbers divide it evenly, it must be prime. Here is this basic solution.

Program 5.2: This program gives a naive approach for testing if a number is prime.
(PrimalityTester0.java)

```
1 import java.util.*;
2
3 public class PrimalityTester0 {
4     public static void main ( String [] args ) {
5         Scanner in = new Scanner ( System.in );
6         System.out.print (" Please enter a number : ");
7         long number = in.nextLong ();
8         boolean prime = true ;
9         for ( long i = 2; i < number && prime ; i++ )
10             if( number % i == 0 )
11                 prime = false ;
12         if( prime )
13             System.out.println ("" + number + " is prime.");
14         else
15             System.out.println ("" + number + " is not prime.");
16     }
17 }
```

This program has a **for** loop that runs from 2 up to number - 1, provided that we don't find a number that evenly divides number. This optimization means that the program will output the moment that it knows that the number is not prime, but we will still have to wait for it to check all the other possibilities before it is sure that the number is prime.

One insight that we can use to make the program more efficient is that, after checking 2, we don't have to divide it by any even numbers. So, we can do half the checking with a few simple modifications.

Program 5.3: This program gives a slightly cleverer approach for testing if a number is prime.
(PrimalityTester1.java)

```
1 import java.util.*;
2
3 public class PrimalityTester1 {
4     public static void main ( String [] args ) {
5         Scanner in = new Scanner ( System.in );
6         System.out.print (" Please enter a number : ");
7         long number = in.nextLong ();
8         boolean prime = ( number % 2 != 0 );
9         for ( long i = 3; i < number && prime ; i += 2 )
```

```

10     if( number % i == 0 )
11         prime = false ;
12     if( prime )
13         System.out.println ("" + number + " is prime.");
14     else
15         System.out.println ("" + number + " is not prime.");
16     }
17 }
```

This version of the program sets the **boolean** variable prime to **false** if number is divisible by 2 and **true** otherwise. Then, it starts the search at 3 and continues in jumps of 2. Although we are saving half the time, we can still do better. Note that if a number n is divisible by 2, then it is also divisible by $\frac{n}{2}$. So, if a number is **not** divisible by 2, it is not divisible by any number larger than $\frac{n}{2}$. If it is not divisible by 2 or 3, then it is not divisible by any number larger than $\frac{n}{3}$. If it is not divisible by 2 or 3 or 4, it is not divisible by any number larger than $\frac{n}{4}$, and so on. Thus, we do not have to check all the way up to $n - 1$. If we are checking to see if n is divisible by x and learning that n is not divisible by anything larger than $\frac{n}{x}$, the point where $x = \frac{n}{x}$ is when $x = \sqrt{n}$. Thus, we only need to search up to \sqrt{n} , which will save even more time.

Program 5.4: This program gives a much faster approach for testing if a number is prime.
(PrimalityTester2.java)

```

1 import java.util.*;
2
3 public class PrimalityTester2 {
4     public static void main ( String [] args ) {
5         Scanner in = new Scanner ( System.in );
6         System.out.print (" Please enter a number : ");
7         long number = in.nextLong ();
8         boolean prime = ( number % 2 != 0 );
9         long root = ( long ) Math.sqrt ( number );
10        for ( long i = 3; i <= root && prime ; i += 2 )
11            if( number % i == 0 )
12                prime = false ;
13        if( prime )
14            System.out.println ("" + number + " is prime.");
15        else
16            System.out.println ("" + number + " is not prime.");
17    }
18 }
```

Note in this version of the program we do go up to and including root, because there is the possibility that number is a perfect square. ■

Example 5.3: DNA reverse complement

DNA is usually double stranded, with each base paired to another specific base, called its complementary base. The following table shows the association between each base and its complementary base.

Base	Abbreviation	Complementary Base
Adenine	A	T
Cytosine	C	G
Guanine	G	C
Thymine	T	A

A simple but common task is finding the reverse complement of a DNA sequence. The reverse complement of a DNA sequence is its sequence of complementary bases given in reverse order. For example, the reverse complement of ACATGAG is CTCATGT. This sequence is found by first finding the complement of ACATGAG, which is TGTACTC, and then reversing its order.

We will write a program that finds the reverse complement of a DNA sequence entered by a user. This sequence will be entered as a sequence of characters made up of the four abbreviations for the bases: A, C, G, and T. We will store this sequence as a String and perform some manipulations on it to get the reverse complement.

Program 5.5: This program finds the reverse complement of a DNA sequence.
(ReverseComplement.java)

```
1 import java.util.*;
2
3 public class ReverseComplement {
4     public static void main ( String [] args ) {
5         Scanner in = new Scanner ( System.in );
6         System.out.print ( " Please enter a DNA sequence : " );
7         String sequence = in.next ().toUpperCase ();
8         String complement = "";
9         for ( int i = 0; i < sequence.length (); i++ )
10             switch ( sequence.charAt ( i ) ) { // get complements
11                 case 'A': complement += "T"; break ;
12                 case 'C': complement += "G"; break ;
13                 case 'G': complement += "C"; break ;
14                 case 'T': complement += "A"; break ;
15             }
16         String reverseComplement = "";
17         // reverse the complement
18         for ( int i = complement.length () - 1; i >= 0; i-- )
19             reverseComplement += complement.charAt ( i );
20         System.out.println ( " Reverse complement : " +
```

```
21     reverseComplement );
22 }
23 }
```

This example first creates a String filled with the complement of the base pairs from the input String. Then, in a second step, it creates a new String that is the reverse of the complement sequence. Note how complement is created by appending the **char** corresponding to the complementary base at the end of complement. If we inserted each **char** at the beginning of complement, we would not need to reverse in a separate step.

Program 5.6: This program more cleverly finds the reverse complement of a DNA sequence.
(CleverReverseComplement.java)

```
1 import java.util.*;
2
3 public class CleverReverseComplement {
4     public static void main ( String [] args ) {
5         Scanner in = new Scanner ( System.in );
6         System.out.print (" Please enter a DNA sequence : ");
7         String sequence = in.next ().toUpperCase ();
8         String reverseComplement = "";
9         for ( int i = 0; i < sequence.length (); i++ )
10             switch ( sequence.charAt ( i ) ) { // get complements
11                 case 'A': reverseComplement = "T" +
12                     reverseComplement ; break ;
13                 case 'C': reverseComplement = "G" +
14                     reverseComplement ; break ;
15                 case 'G': reverseComplement = "C" +
16                     reverseComplement ; break ;
17                 case 'T': reverseComplement = "A" +
18                     reverseComplement ; break ;
19             }
20         System.out.println (" Reverse complement : " +
21             reverseComplement );
22     }
23 }
```



5.3.3 do-while loops

Use this rule of thumb for deciding which kind of loop to use: If you know how many times you want the loop to execute, use a **for** loop. If you don't know how many times you want it to execute, use a **while** loop. Clearly, this rule is not iron-clad. In the previous example, we used a **for** loop even though it would stop executing as soon as a '**Z**' was encountered. Nevertheless, it seems like we have

covered all of the possible situations with `while` and `for` loops. When should we use `do-while` loops? The simple answer is: never.

You never **have** to use a `do-while` loop. With a little bit of effort, you use a single kind of loop for every job. The key difference between a `do-while` loop and a regular `while` loop is that a `do-while` loop will always run at least once. Neither of the other two loops give you that guarantee. The syntax for a `do-while` loop is a `do` at the top of a loop body enclosed in braces, with a normal `while` header at the end, including a condition in parentheses, followed by a semicolon. [Figure 5.3](#) shows the pattern of execution for `do-while` loops.

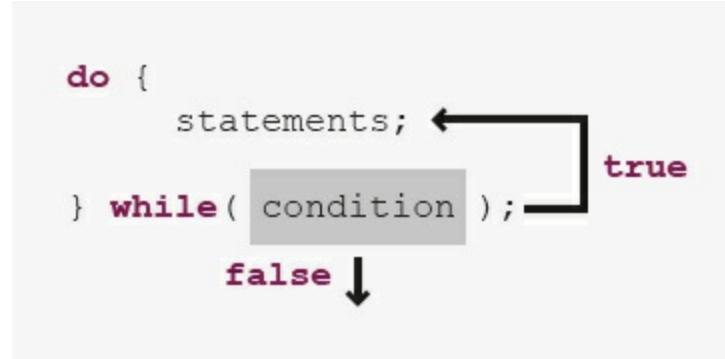


Figure 5.3: The statements in the body of the loop are executed, and then the condition is checked. When the check is `false`, execution skips past the body of the loop. A `do-while` loop is guaranteed to run at least once.

We can use a `do-while` loop to print out the first 10 perfect squares as follows.

```
int x = 1;  
do {  
    System.out.println (x*x);  
    x++;  
} while ( x <= 10 );
```

This loop behaves exactly the same as the following loop.

```
int x = 1;  
while ( x <= 10 ) {  
    System.out.println (x*x);  
    x++;  
}
```

The time when a `do-while` loop is really going to shine is when your program will work incorrectly if the loop doesn't run at least once. This situation often occurs with input, when the loop must run at least once before checking the condition. For example, imagine that you want to write a program that picks a random number between 1 and 100 and lets the user guess what it is until the user gets it right. You need a loop because it is a repetitive activity, but you need to let the user guess at least once so that you can check to see if he or she was right. The following program fragment does exactly that.

```

Scanner in = new Scanner ( System.in );
Random random = new Random ();
int guess = 0;
int number = Math.nextInt ( 100 ) + 1;
do {
    System.out.print ( " What is your guess ? " );
    guess = in.nextInt ();
} while ( guess != number );
System.out.println ( " You got it! The number was " + number + " ." );

```

You could perform the same function with a **while** loop, but you will need to get some input from the user before the loop starts. Using the **do-while** loop is a little more elegant.

5.3.4 Nested loops

As with **if** statements, it is possible to nest loops inside of other loops. In the simplest case, you may have some repetitive activity that itself needs to be performed several times. For example, when you were younger, you probably had to learn your multiplication tables. For each number, a multiplication table gave the value of the product of that number by every integer between 1 and 12. We can write code to print out the multiplication table for every number from 1 to 10 by simply repeating the process.

```

for ( int number = 1; number <= 10; number ++ ) {
    for ( int factor = 1; factor <= 12; factor ++ ) {
        System.out.println ( number + " x " + factor + " = " + ( number * factor ) );
    }
    System.out.println ();
}

```

The outer loop incrementing `number` will run 10 times. The inner loop incrementing `factor` runs 12 times for each iteration of the outer loop. So, the code in the inner loop will run a total of 120 times. Every 12 iterations, the inner loop will stop, and an extra blank line will be added by the `System.out.println()` method in the outer loop.

Example 5.4: Triangular numbers

The sequence consisting of 1, 3, 6, 10, 15, and so on is known as the triangular numbers. The i^{th} triangular number is the sum of the first i integers. They are called triangular numbers because they can be drawn as equilateral triangles in a very natural way, if you use a number of dots equal to the number.

We can use nested loops to print out the first n triangular numbers, where n is specified by the user.

Program 5.7: This program prints out triangular numbers. (`TriangularNumbers.java`)

```

1 import java.util.*;
2

```

```

3 public class TriangularNumbers {
4     public static void main ( String [] args ) {
5         Scanner in = new Scanner ( System.in );
6         System.out.print (" How many triangular numbers ? ");
7         int n = in. nextInt ();
8         int sum ;
9         for ( int i = 1; i <= n; i++ ) {
10             sum = 0;
11             for ( int j = 1; j <= i; j++ )
12                 sum += j;
13             System.out.println ( sum );
14         }
15     }
16 }
```

As you can see, the outer loop iterates through each of the n different triangular numbers. Then, the inner loop does the summation needed to compute the given triangular number. Producing a sequence of triangular numbers this way is, unfortunately, not the most efficient way to do it. Nested loops are an effective way to solve many problems, particularly certain types of problems using arrays, but we can generate triangular numbers using only a single **for** loop. The key insight is that we can keep track of the previous triangular number and add i to it, as i increases.

Program 5.8: This program prints out triangular numbers more cleverly.
(CleverTriangularNumbers.java)

```

1 import java.util.*;
2
3 public class CleverTriangularNumbers {
4     public static void main ( String [] args ) {
5         Scanner in = new Scanner ( System.in );
6         System.out.print (" How many triangular numbers ? ");
7         int n = in. nextInt ();
8         int triangular = 0;
9         for ( int i = 1; i <= n; i++ ) {
10             triangular += i;
11             System.out.println ( triangular );
12         }
13     }
14 }
```

By removing the inner **for** loop, the total amount of work needed is greatly reduced. ■

5.3.5 Common pitfalls

With great power comes great responsibility. The power to repeat things a large number of times

means that we can also repeat our mistakes a large number of times. Many classic bugs occur as a result of logical or typographical errors in loops. Below we will list a few of the most common.

Pitfall: Infinite loops

It is possible to create a loop that never terminates. Your program may be taking a long time to finish, but if it takes much longer than you expect, an infinite loop might be the culprit. Infinite loops might occur because you forgot to include an appropriate statement to advance a counter.

```
int number = 1;  
while ( number <= 100 )  
    System.out.println ( number );  
}
```

This code is presumably intended to print out the first 100 integers, but there is no code that increases the value of `number`. As a consequence, the number 1 will be printed out over and over until the user stops the program from executing. Usually, the cause is more subtle, as in the following code.

```
for ( int i = 0; i < 10; i += 0.5 )  
    System.out.println (" Half a step forward , half a step back ... " );
```

One might expect this code to print out 20 lines of output. However, remember that `i` is an `int`. Adding 0.5 to 0 and then casting it to an `int` gives 0 again. What is particularly insidious about this loop is that it compiles without even a warning in Java. Usually conversion from a `double` to an `int` requires an explicit cast, but the `+=` operator (and other similar operators) behave a little differently for technical reasons.

Pitfall: Almost infinite loops

Many loops are truly infinite; others take a really long time. For example, if you intended to run a loop down from 10 to 0, but increment your counter instead of decrementing it, overflow means that you will eventually get to a number less than 0, but it will take more than 2 billion increments instead of the expected 10 decrements.

```
for ( int i = 10; i > 0; i++ )  
    System.out.println (i);  
System.out.println (" Blast off!");
```

This loop will significantly slow your code. Everyone will be so tired of waiting that they might leave the space shuttle launch. Of course, another problem with almost infinite loops is that you are dealing with the wrong values. No one expects to hear the number 2147483647 in a countdown.

Pitfall: Fencepost errors

Perhaps the most common loop errors are fencepost errors, often known as off-by-one errors. The name “fencepost” comes from a related mistake that someone might make when putting up a fence. Imagine that you want to erect a 10 meter long chain link fence and you need to have a support post every meter, how many posts do you need? In fact, we have not given you enough information to answer the question correctly. If your fence is built in a straight line, then you will need 11 posts so that you have a post at each end. However, if your fence is a rectangular enclosure, say 3 meters by 2 meters, you will only need 10 posts.

In loops, fencepost errors are often due to zero-based counting. A for loop that iterates 10 times is below.

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

Of course, sometimes we need one-based counting instead. After being used to zero-based counting, a programmer might make the following loop that incorrectly iterates 9 times.

```
for ( int i = 1; i < 10; i++ )  
    System.out.println (i);
```

The correct version that iterates 10 times is below.

```
for ( int i = 1; i <= 10; i++ )  
    System.out.println (i);
```

If you want to iterate n times, start at 0 and go up to but not including n or start at 1 and go up to and including n . To keep loop headers consistent, some programmers always start at 0 and then adjust the values inside the loop, printing out $i + 1$ in this case.

Pitfall: Skipped loops

A loop runs as long as its condition is `true`. For `for` loops and `while` loops, this could mean that the loop is never even entered. Sometimes, that behavior is intended by the programmer. Sometimes, the programmer made a mistake.

For example, we can write a program that will add any number of positive values. When the user is finished using the adder, he or she enters a negative number. This negative number, called a *sentinel value*, tells the program to stop executing the loop. Below is an incorrect implementation of such a program.

```
Scanner in = new Scanner ( System.in );  
int number = 0;  
int sum = 0;  
while ( number > 0 ) {  
    sum += number ;  
    System.out.print (" Enter the next number to add : ");  
    number = in.nextInt ();  
}  
System.out.println (" The total sum is " + sum );
```

This loop will never be executed because 0 is not greater than 0. The program could be changed by making the condition of the `while` loop `number >= 0`. Doing so will allow the user to enter 0 as input, which is fine since it does not change the value of the sum. If you want to force the user to enter only numbers greater than zero, you could change the loop into a `do-while` loop.

Pitfall: Misplaced semicolons

The idea of a statement in Java is often amorphous in the minds of beginning programmers. An entire loop (with any number of loops nested inside of it) is just one statement. An executable statement ending with a semicolon is one statement as well, even when that executable statement is empty. Thus, the following is a legal (but infinite) loop.

```
int i = 100;  
while ( i > 0 ); {  
    System.out.println (i);  
    i --;  
}  
System.out.println (" Ready or not , here I come !");
```

This code was supposed to count down from 100, just like in the game of Hide and Seek; however, there is a semicolon after the condition of the `while` loop. This semicolon is treated like an executable statement that does nothing. As a consequence, the `while` loop does the single statement, checks if the condition is `true` (which it is), and continues to do the empty statement and check the condition, forever. The extra braces enclose two statements unnecessarily, but Java allows extra braces, as long as they are evenly matched.

This error is common especially for those new to loops and conditional statements and are in the habit of putting semicolons after everything. A misplaced semicolon does not always result in an infinite loop. Here is the `for` loop version of the same code, also with a semicolon inserted after the loop header.

```
for ( int i = 100; i > 0; i-- ); {  
    System.out.println (i);  
}  
System.out.println (" Ready or not , here I come !");
```

This version of the code will execute similarly, except the decrement is built into the header of the loop. So, the loop will execute the empty statement, but it will also decrement `i`. This code will decrement `i` 100 times, then print out 0 exactly once, then print Ready or not, here I come !.

There are some cases when an empty statement for a loop body is actually useful, although it is never necessary. In future chapters, we will point out situations in which you may wish to use an empty statement this way.

5.4 Solution: DNA searching

Below we give a solution to the DNA searching problem posed at the beginning of the second half of this chapter. Our solution prints out the index within the main String when it finds a match with the pattern it is looking for. Afterwards, it prints out the total number of matches. Our code also does error checking to make sure that the user only enters valid DNA sequences containing the letters A, C, G, and T. We begin our code with the standard `import` statement and class definition.

```

1 import java.util.*;
2
3 public class DNASearch {
4     public static void main ( String [] args ) {
5         Scanner in = new Scanner ( System.in );
6         String sequence , subsequence ;
7         boolean valid ;
8         char c;

```

The `main()` method instantiates a `Scanner` object and declares both of the `String` variables we'll need to store the DNA sequences. The method also declares a `boolean` and a `char` we will use for input checking.

```

10    do {
11        System.out.print (
12            " Enter the DNA sequence you wish to search in: ");
13        sequence = in. next (). toUpperCase ();
14        valid = true ;
15        for ( int i = 0; i < sequence.length () && valid ; i ++ ) {
16            c = sequence.charAt (i);
17            if( c != 'A' && c != 'C' && c != 'G' && c != 'T') {
18                System.out.println (" Invalid DNA sequence !");
19                valid = false ;
20            }
21        }
22    } while ( ! valid );

```

Next, the user is prompted for a DNA sequence to search in. This `String` stored in `sequence` is converted to upper case just in case the user is not being consistent. The inner `for` loop in this code is checking each `char` inside of `sequence`. If any `char` is not an '`A`', '`C`', '`G`', or '`T`', then `valid` is set to `false`. As a result, the `for` loop terminates. Also, the `do-while` loop repeats the prompt and gets a new `String` for `sequence` from the user. This outer `do-while` loop continues as long as the user keeps entering invalid DNA sequences.

```

24    do {
25        System.out.print (
26            " Enter the subsequence you wish to search for : ");
27        subsequence = in. next (). toUpperCase ();
28        valid = true ;
29        for ( int i = 0; i < subsequence.length () && valid ; i ++ )
30        {
31            c = subsequence.charAt (i);
32            if( c != 'A' && c != 'C' && c != 'G' && c != 'T') {
33                System.out.println (" Invalid DNA sequence !");
34                valid = false ;

```

```

34         }
35     }
36 } while ( ! valid );

```

The code used to input subsequence while doing error checking is virtually identical to the code to input sequence.

```

38     int found = 0;
39     for ( int i = 0; i < sequence.length () -
40             subsequence.length () + 1; i++ ) {
41         for ( int j = 0; j < subsequence.length (); j++ ) {
42             if( subsequence.charAt (j) != sequence.charAt (i+j))
43                 break ;
44             if( j == subsequence.length () - 1 ) { // matches
45                 System.out.println (" Match found at index " + i);
46                 found++;
47             }
48         }
49     }

```

The actual workhorse of the search is found in these nested **for** loops. The outer loop iterates through every index in sequence, until it comes to an index that is too late to be the start of a new subsequence (since the subsequence would be too long to fit anymore). This happens to be when the value of *i* is greater than or equal to `sequence.length() - subsequence.length() + 1`. It may take some thought to verify that this condition is the correct one. One way to think about this problem is by noting that, when sequence and subsequence have the same length, you need to check starting at index 0 of sequence but not any later indexes. Also, if subsequence is one **char** longer than sequence, there can never be a match. In that case, the value of `sequence.length() - subsequence.length() + 1` would be 0. Since 0 is not less than 0, the outer **for** loop would never execute.

The inner **for** loop iterates through the length of subsequence, making sure that every **char** in sequence, starting at the appropriate offset, exactly matches a **char** in subsequence. If, at any point, the two **char** values do not match, the inner **for** loop will immediately exit, using the **break** command. However, on the last iteration of the inner **for** loop, when *j* is one less than the length of subsequence, we know that all of subsequence matched a part of sequence. As a result, we print out the index of sequence where subsequence started and increment the found counter.

If you know the `String` class well, you can use the `indexOf()` method to replace the inner **for** loop. We leave that approach as an exercise.

Exercise 5.8

```

51     if( found == 1 )
52         System.out.println (" One match found.");
53     else
54         System.out.println ( found + " matches found.");
55 }

```

Finally, we print out the total number of matches found. In order to avoid awkward output like 1 matches found., we used an **if-else** to customize the output based on the value of found.

The ideas needed to correctly implement the solution are not difficult, but catching all the off-by-one errors and getting every detail right takes care. There is also more than one way to code this solution. For example, we could have written the nested loops that do the searching as follows.

```
int found = 0;
for ( int i = 0; i < sequence.length () -
    subsequence.length () + 1; i++ ) {
    for ( int j = 0; j < subsequence.length () &&
        subsequence.charAt (j) == sequence.charAt (i + j); j++ )
        if( j == subsequence.length () - 1 ) { // matches
            System.out.println (" Match found at index " + i);
            found++;
        }
}
}
```

This design is preferred by many since it removes the **break**. By using an empty statement, it is possible to move the check to see if the matching process is done outside of the inner **for** loop.

```
int found = 0;
int j;
for ( int i = 0; i < sequence.length () -
    subsequence.length () + 1; i++ ) {
    for ( j = 0; j < subsequence.length () &&
        subsequence.charAt (j) == sequence.charAt (i + j); j++ );
        if( j == subsequence.length () ) { // matches
            System.out.println (" Match found at index " + i);
            found++;
        }
}
}
```

In this case, note that we must declare **j** outside of the inner **for** loop, since it will be used outside. This approach is more efficient because we only need to perform the check once. Notice also that the condition of the **if** statement has changed. Now, we know that all of subsequence matches because the loop ran to completion. If the loop did not run to completion, then **j** would be smaller than **subsequence.length()** and the loop must have terminated because the two **char** values did not match. Although more efficient, some programmers would avoid this approach because it uses the confusing syntax in which the body of the for loop is a single empty statement followed by a semicolon. Likewise, the logic about exiting the loop and the condition of the **if** statement is murkier.

5.5 Concurrency: Loops

Many programmers use concurrency for speedup. They want their programs to run faster. Most programs that run for a long time use loops to do repetitive tasks. If these loops are doing the same operation to many different pieces of data, we may be able to speed up the process by splitting up the data and letting different threads operate on their own segment of the data. Splitting up data this way is called *domain decomposition* which allows us to achieve *data parallelism*. These topics are discussed further in [Section 13.3](#).

Performing repetitive tasks is one of the great strengths of computers. For most programs that run a long time, incredible amounts of computation are being done inside of (usually nested) loops. Domain decomposition will not work for all of these programs. Some cannot be parallelized at all, but this book is about finding problems that can have parallel and concurrent solutions.

In [Chapter 13](#), we will introduce tools for writing a concurrent program with different threads of execution running at the exactly the same time and potentially interacting. Using only the power of loops, you can see parallelism in action now.

Example 5.5: Parallelism without threads

Consider the problem of computing the sum of the sines of a range of integers. At its heart is a loop from the start of the range to the end.

```
for ( int i = start ; i <= end ; i++ )  
    sum += Math.sin( start );
```

If we want to allow the user to specify the start and the end and print out the sum, we need to make a program with a little bit of input and output around this loop.

Program 5.9: Program to add the sines of all integers in a range specified by the user.
(SumSines.java)

```
1 import java.util.Scanner ;  
2  
3 public class SumSines {  
4     public static void main ( String [] args ) {  
5         Scanner in = new Scanner ( System.in );  
6         System.out.print ( " Enter starting value : " );  
7         int start = in.nextInt ();  
8         System.out.print ( " Enter ending value : " );  
9         int end = in.nextInt ();  
10  
11         double sum = 0;  
12         for ( int i = start ; i <= end ; i++ )  
13             sum += Math.sin ( start );  
14  
15         System.out.println ( " Sum of sines : " + sum );
```

```
16      }  
17 }
```

If you compile and run this program with 1 as the start value and 100000000 as the end, the answer should be 1.7136493465700542. One hundred million values is a lot to find the sine for. Depending on your machine, this task should take between 10 seconds and over a minute. Try to time how long this takes as accurately as possible.

Now, open a total of four console windows and navigate them all to the directory with SumSines.class in it. Run SumSines in each one. For the first console, enter 1 as the start and 25000000 as the end. For the second, enter 25000001 and 50000000. For the third, enter 50000001 and 75000000. For the last, enter 75000000 and 100000000. Once they have run, you should get, respectively, 1.4912473269134603, -0.6795491754132104, -0.2893142602684644, and 1.1912654553381272. If you add these together using a calculator, you should get 1.7136493465699127, which is almost exactly the same answer we got before. (Floating point rounding errors cause the slight difference.)

If you try to start them computing at about the same time, you can try to see how long it takes for all of them to complete. Did it take less time than before? If you have a single core processor, it might have taken just as long or longer. If you have a dual-core processor, it should have taken less time, and if you have a quad core processor, even less. Since we are dividing the problem into four pieces, we will not expect to see any improvement with more than four cores.

Most operating systems provide a graphical way of viewing the load on each processor. If you examine your CPU usage while running those programs, you should see it spike up when the programs start and then come down when they finish. For multiple cores, how did we say which core we wanted each program to run on? We didn't. In general, it is difficult to specify which core we want to run a program, process, or thread on. The OS does the job of scheduling and picks a free processor when it needs to run a program. It is even possible for programs and threads to change from one core to another while running if the OS needs to balance out the workload.

This sines example is similar to [Example 13.10](#) which we will cover in [Chapter 13](#). As you may have noticed, running four programs is not convenient. You have to open several windows, you have to type starting and ending points very carefully, and you have to combine the answers at the end since your programs cannot interact directly with each other. Features of Java will make this job easier, allowing us to run more than one thread of execution at a time without the need to run multiple programs by hand. ■

Exercise 5.15

Exercises

Conceptual Problems

- 5.1 If you have a String containing a long text and you want to count the number of words in the text that begin with the letter '[m](#)', which of the three kinds of loops would you use, and why?
- 5.2 In [Example 5.2](#), our last version of the primality tester PrimalityTester2 computes the square

root of the number being tested. Instead of computing this value before the loop, how would performance be affected by changing the head of the **for** loop to the following?

```
for ( long i = 3; i <= Math.sqrt ( number ) && prime ; i += 2)
```

5.3 How many different DNA sequences of length n are there?

5.4 There are three different errors in the following loop intended to print out the numbers 1 through 10. What are they?

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

5.5 Consider the following code containing nested **for** loops.

```
Scanner in = new Scanner ( System.in );  
int n = in.nextInt ();  
int count = 0;  
for ( int i = 1; i <= n; i++ )  
    for ( int j = 1; j <= i; j++ )  
        count ++;
```

In terms of the value of n , how many times is count incremented? If it is not immediately obvious, trace through the execution of the program by hand or run the code for several different values of n and try to detect a pattern.

Programming Practice

5.6 Write a program that converts base 10 numbers into base 3 numbers. If you find that task too easy, write a program that will convert base 10 numbers to any base in the range 2 to 16. Hint: Use letters A through F, in order, to represent digits larger than 9.

5.7 The greatest common divisor (GCD) of two integers is the largest integer that divides both of them evenly. The GCD for any two positive integers is at least 1 and at most the smaller of the two numbers. Write a program that prompts a user for two **int** values and finds their GCD. Although there are more efficient methods, you can count down from either number. If the counter ever divides **both** numbers evenly, it is the GCD. The counter is guaranteed to divide them both if it reaches 1.

5.8 In the solution to the DNA searching problem given in [Section 5.4](#), we used two **for** loops to find occurrences of a DNA subsequence inside of a larger sequence. Professional Java developers would have used a single **for** loop and the `indexOf()` method in the `String` class. One version of this method returns the index of a substring within a `String` object, starting from a particular offset, as shown below.

```
String text = " fun dysfunction ";
```

```
String search = "fun";
System.out.println (" Location : " + text.indexOf ( search , 4));
```

This code will output Location: 7 since the first occurrence of “fun” from index 4 or later starts at index 7. If there are no more occurrences of the substring beyond the starting index, the method will return -1. Rewrite the solution to the DNA searching problem, replacing the inner searching **for** loop with the `indexOff()` method.

5.9 Write a program that reads a number n from a user and then prints all possible DNA sequences of length n . Be careful not to supply too large of a value when you run this program. Hint: Represent the sequence as a String. On each iteration, focus on the last **char** in the String. If it is an 'A', change it to a 'C'. If it is a 'C', change it to a 'G'. If it is a 'G', change it to a 'T'. If it is a 'T', change it back to an 'A', but “carry” the increment over to the next **char**, like a rolling odometer. You will have to design loops that can deal with carries that cascade across multiple indexes.

5.10 Re-implement the solution to the DNA searching program given in [Section 5.4](#) using `JOptionPane` to generate GUIs for input and output.

GUI

Experiments

5.11 Using a **for** loop, record the Monty Hall simulation so that you can run it 100 times, always choosing to switch doors. Keep a record of how many times you win. Change your code again to run the Monty Hall simulation 100 more times, always choosing to keep your initial choice. Again, keep a record of how many times you win. Compare the two records. Choosing to switch should perform roughly twice as well as keeping the first door. Increase the number of iterations to 1,000 and then 10,000 times. Does the performance of switching get closer to twice the performance of not switching?

5.12 Write three nested **for** loops, each of which run 1,000 times. Increment a counter in the innermost **for** loop. If that counter starts at 0, its final value should be 1,000,000,000. Time how long your program takes to run to completion using either a stopwatch or, if you are on a Unix or Linux system, the `time` command. Feel free to increase and decrease the amount that each loop runs to see the effect on the time. However, if you increase the values of all three loops too much, you may have to wait longer than you want.

5.13 In [Section 5.3.5](#), one of the common loop mistakes we discuss is an almost infinite loop. Create your own almost infinite loop that runs from 10 to 0, incrementing instead of decrementing. Time the execution of your program. Unlike our example, do not use an output statement or your code will take too long to run. How much longer would your code take to run if you used a `long` instead of an `int`?

5.14 In [Example 5.2](#), we gave three programs to test a number for primality. Run each of these prime testers on a large prime such as 982,451,653 and time them. Is there a significant difference in the running time of `PrimalityTester0` and `PrimalityTester1`? What about `PrimalityTester1` and `PrimalityTester2`?

5.15 In [Example 5.5](#), we ran four programs at the same time to solve a problem in parallel. Use the same framework (combined with your knowledge of primes from [Example 5.2](#)) to write a program that can see how many prime numbers are in a user specified range of integers. Then, use it to find the total number of primes between 2 and 500,000,000. Now, run two copies of the program with one starting at 2 and going up to 250,000,000 and the other starting at 250,000,001 and going up to 500,000,000. If you add the numbers together, do you get the same answer? (If not, there is a bug in your program.) Now, divide the work into four pieces. How much quicker, if at all, is running all four programs instead of one? Does one of the four pieces run significantly faster or slower than the others?

Concurrency

Chapter 6

Arrays

Too much of a good thing can be wonderful.

—Mae West

6.1 Introduction

With one exception, all of the types we have talked about in this book have held a single value. For example, an `int` variable can only contain a single `int` value. If you try to put a second `int` value into a variable, it will overwrite the first.

The `String` type, of course, is the exception. `String` objects can contain `char` sequences of any length from 0 up to a practically limitless size (the theoretical maximum length of a Java `String` is `Integer.MAX_INT`, more than 650 times the length of *War and Peace*). As remarkable as `String` objects are, this chapter is about a more general kind of list called an *array*. We can create arrays to hold a list of any type of variable in Java.

The ability to work with lists expands the scope of problems that we can solve. Beyond simple lists, we can use the same tools to create tables, grids, and other structures to solve fascinating problems like the one that comes next.

6.2 Problem: Conway's Game of Life

Some physicists insist that the rules governing the universe are horribly complicated. Some insist that the fundamental laws are simple and only their overall interaction is complex. With the power to do simulations quickly, computer scientists have shown that some systems can exhibit very complex interactions using simple rules. Perhaps the best known examples of these systems are *cellular automata*, of which Conway's Game of Life is the most famous.

The framework for Conway's Game of Life is an infinite grid made up of squares called cells. Some cells in the grid are black (or alive, to give it a biological flavor), and the rest are white (or dead). Any given cell has 8 neighbors as shown in the figure below.

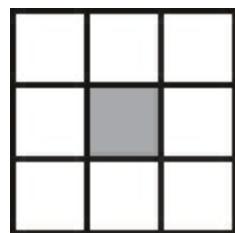


Figure 6.1: Cell (shown in gray) surrounded by 8 neighboring cells.

The pattern of living and dead cells at any given time on the grid is called a *generation*. To

determine the next generation, we use the following rules.

1. If a living cell has fewer than two living neighbors, it dies from loneliness.
2. If a living cell has more than three neighbors, it dies because of overcrowding.
3. If a living cell has exactly two or three living neighbors, it lives to the next generation.
4. If a dead cell has exactly three live neighbors, it becomes a living cell.

These four simple rules allow for surprisingly complex interactions. The patterns that emerge from applying these rules to a starting configuration of living and dead cells strike a balance between complete chaos and rigid order. As the name of the game implies, the similarity to biological patterns of development can be surprising.

Your problem is to create a Life simulator of size $n \times m$, specific values for which will be discussed below. The program should simulate the process at a speed that is engaging to watch, with a new generation every tenth of a second. The program should begin by randomly making 10% of all the cells living and the rest dead.

6.2.1 Terminal limitations

One problem you might be worrying about is how to display the simulation. In [Chapter 15](#) you will learn how to make a graphical user interface (GUI) that can display a grid of cells in black and white and much more interesting things as well. For now, the main tool that we can use for output is still the terminal. The output method we recommend is printing an asterisk ('*') for a living cell and a space (' ') for a dead one. In this way you can easily see the patterns form on the screen and change over time.

The classic terminal is not very big. For this reason, we suggest that you set the height of your simulation to 24 rows and the width of your simulation to 80 columns. These dimensions conform to the most ancient terminal sizes. If your terminal screen is much larger, you can change the width and height later to have a larger simulation. The ideal form of the game is infinite, as well. Because our size is so limited, we must deal with the problem of a cell on the boundary. Anything beyond the boundaries should be counted as dead. Thus, a cell right on the edge of the simulation grid can have a maximum of 5 neighbors. A cell in one of the four corners can only have 3 neighbors.

In order to give the appearance of smooth transitions, you need to print out each new generation of cells quickly, and in the same locations as the previous generation. Simply printing one set after another will not achieve this effect unless your terminal screen is exactly 24 rows tall. So, you will need to clear the screen each time. In an effort to be platform independent, Java does not provide an easy way to clear the terminal screen. A quick and simple hack is to simply print out 100 blank lines before printing the next generation. This hack will work as long as your terminal is not significantly more than 100 rows in height. If it is, you will need to print a larger number of blank rows.

Finally, you need to wait a short period of time between generations so that the user can see each configuration before it is cleared away and replaced by the next one. The simplest way to do this is by having your program go to sleep for a short period of time, a tenth of a second as we suggested

before. The code to make your program sleep for that amount of time is:

```
try { Thread.sleep(100); }  
catch (InterruptedException e) {}
```

We will explain this code in much greater detail in [Chapter 13](#). The key item of importance is the number passed into the `sleep()` method. This value is the number of milliseconds you want your program to sleep. 100 milliseconds is, of course, one tenth of a second.

In order to simulate the Game of Life, we need to store information, namely the liveness or deadness of cells, in a grid. First, we need to discuss the simpler task of storing data in a list.

6.3 Concepts: Lists of data

Lists of data are of paramount importance in many different parts of life: shopping lists, packing lists, lists of employees working at a company, address books, top ten lists, and more. Lists are even more important in programming. As you know, one of the great strengths of computers is their speed. If we have a long list of data, we can use that speed to perform some operation on all the data quickly. E-mail contact lists, entries in a database, and cells in a spreadsheet are just a few of the most obvious ways that lists come up in computer applications.

Even in Java, there are many different ways to record a list of information, but a list is only one form of *data structure*. As the name implies, a data structure is a way to organize data, whether in a list, in a tree, in sorted order, in some kind of hierarchy, or in any other way that might be useful. We will only talk about the array data structure in this chapter, but other data structures will be discussed in [Chapter 18](#). Below we give a short explanation of some of the attributes any given data structure might have.

6.3.1 Data structure attributes

Contents: Keeping only a single value in a data structure defeats the purpose of a data structure. But, if we can store more than a single value, must all of those values come from the same type? If a data structure can hold several different types of data, we call it *heterogeneous*, but if it can only hold one type of data, we call it *homogeneous*.

Size: The size of a data structure may be something that is fixed when it is created or it could change on the fly. If a data structure's size or length can change, we call it a *dynamic* data structure. If the data structure has a size or length that can never change after it has been created, we call it a *static* data structure.

Element Access: One of the reasons that there are so many different data types is that different ways of structuring data are better or worse for different tasks. For example, if your task is to add a list of numbers, then you are expecting to access every single element in the list. However, if you are searching for a word in a dictionary, you do not want to check every dictionary entry to find it.

Some data structures are optimized so that you can efficiently read, insert, or delete only a single item, often the first (or last) item in the data structure. Some data structures only allow you to move through the structure sequentially, one item after another. Such a data structure has what is called *sequential access*. Still others allow you to jump randomly to any point you want inside the data structure efficiently. These data structures have what is called *random access*. Advanced programmers take into account many different factors before deciding which data structure is best suited to their problem.

6.3.2 Characteristics of an array

Now that we've defined these attributes, we can say that an array is a homogeneous, static data structure with random access. An array is homogeneous because it can only hold a list of a single type of data, such as `int`, `double`, or `String`. An array is static because it has a fixed length that is set only when the array is instantiated. An array also has random access because jumping to any element in the array is very fast and takes about the same amount of time as jumping to any other.

An array is a list of a specific type of elements that has length n , a length specified when the array is created. Each of the n elements is indexed using a number between 0 and $n - 1$. Once again, zero-based counting rears its ugly head. Consider the following list of items:

$$\{9, 4, 2, 1, 6, 8, 3\}$$

If this list is stored in an array, the first element, 9, would have index 0, 4 would have index 1, and so on, finishing at 3 with an index of 6, although the total number of items is 7. Not all languages use zero-based counting for array indexes, but many do, including C, C++, and Java. The reason that languages like C originally used zero-based counting for indexes is that the variable corresponding to the array is an address inside the computer's memory giving the first element in the array. Thus, an index of 0 is 0 times the size of an element added to the starting address, and an index of 5 is 5 times the size of an element added to the starting address. So, zero based indexes gave a quick way for the program to compute where in memory a given element of an array is.

6.4 Syntax: Arrays in Java

The idea of a list is not mysterious. Indexing each element of the list using numbers is natural, even if the numbers start at 0 instead of 1. Even so, arrays are the source of many errors that cause Java programs to crash. Below we explain the basics of creating arrays, indexing into arrays, and using arrays with loops. Then there is an extra subsection explaining how to send data from a file to a program as if the file were being typed in by a user. Using this technique can save you a lot of time when you are experimenting with arrays.

6.4.1 Array declaration and instantiation

To create an array, you usually need to create an array variable first. Remember that an array is a homogeneous data structure, meaning that it can only store elements of a single type. When you create an array variable, you have to specify what that type is. To declare an array variable, you use the type

it is going to hold, followed by square brackets ([]), followed by the name of the variable. For example, if you want to create an array called numbers that can hold integers, you would type the following.

```
int [] numbers ;
```

If you have some C or C++ programming experience, you might be used to the brackets being on the other side of the variable, like so.

```
int numbers [];
```

In Java, both declarations are perfectly legal and equivalent. However, the first declaration is preferred from a stylistic perspective. It follows the pattern of using the type (an array of `int` values in this case) followed by the variable name as the syntax for a declaration.

As we said, arrays are also static data structures, meaning that their length is fixed at the time of their creation. Yet we did not specify a length above. This declaration has not yet created an array, just a variable that can point at an array. In the second half of this chapter, we will further discuss this difference between the way an array is created and the way an `int` or any other variable of primitive type is created. To actually create the array, we need to use another step, involving the keyword `new`. Here is how we instantiate an array of `int` type with 10 elements.

```
numbers = new int [10];
```

We use the keyword `new`, followed by the type of element, followed by the number of elements the array can hold in square brackets. This new array is stored into `numbers`. In other words, the variable `numbers` is now a name for the array. Commonly, the two steps of declaring and instantiating an array will be combined into one line of code.

```
int [] numbers = new int [10];
```

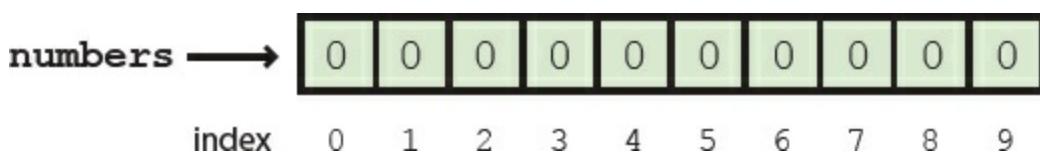
It is always possible to separate the two steps. In some cases, a single variable might be used to point at an array of one particular length, then changed to point at an array of another length, and so on, as below.

```
int [] numbers ;
numbers = new int [10];
numbers = new int [100];
numbers = new int [1000];
```

Here, the variable `numbers` starts off pointing at no array. Next, it is made to point at a new array with 10 elements. Then, it is made to point at a new array with 100 elements, ignoring the 10 element array. Finally, it is made to point at an array with 1,000 elements, ignoring the 100 element array. Remember, the arrays themselves are static; their lengths cannot change. The array type variables, however, can point at different arrays with different lengths, provided that they are still the right type (in this case `int`).

What values are inside the array when it is first created? Let's return to the case where `numbers`

points at a new array with 10 elements. Each of those elements contains the `int` value 0, as shown below.



Whenever an array is instantiated, each of its n elements is set to some default value. For `int`, `long`, `short`, and `byte` this value is 0. For `double` and `float`, this value is 0.0. For `char`, this value is '\0', a special unprintable character. For `boolean`, this value is `false`. For `String` or any other reference type, this value is `null`, a special value that means there is no object at that address.

It is also possible to use a list to initialize an array. For example, we can create an array of type `double` that contains the values 0.5, 1.0, 1.5, 2.0, and 2.5 using the following code.

```
double [] increments = {0.5, 1.0, 1.5, 2.0, 2.5};
```

This line of code is equivalent to using the `new` keyword to create a `double` array with 5 elements and then setting each to the values shown.

6.4.2 Indexing into arrays

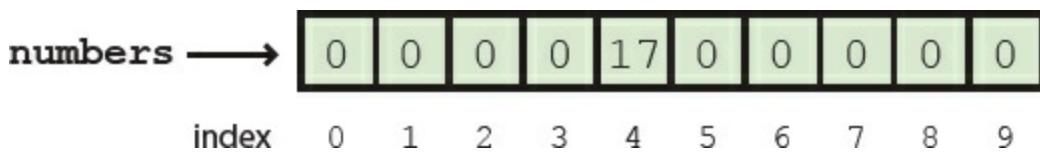
To use a value in an array, you must *index* into the array, using the square brackets once again. Returning to the example of the `int` array `numbers` with length 10, we can read the value at index 4 from the array and print it out.

```
System.out.println( numbers [4] );
```

Of course, the value of `numbers[4]` is 0 and so 0 is all that will be printed out. We can set the value at `numbers[4]` to 17 as follows.

```
numbers [4] = 17;
```

Then, if we try to print out `numbers[4]`, 17 will be printed. The contents of the `numbers` array will look like this.



The key thing to understand about indexing into an array is that it gives you an element of the specified type. In other words, `numbers[4]` is an `int` variable in every possible sense. You can read its value. You can change its value. You can pass it into a method. It can be used anywhere a normal `int` can be used, as in the following example.

```
int x = numbers [4];
double y = Math.sqrt ( numbers [2]) + numbers [4];
numbers [9] = (int )(y*x);
```

Executing this code will store 17 into x and 17.0 into y. Then, the product of those two, 289, will be stored into numbers[9]. Remember, in Java, the type on the left and the type on the right of the assignment operator (=) must match, except in cases of automatic casting, like storing an int value into a double variable. Since they have the same type, it makes sense to store an element of an int array like numbers[4] into an int variable like x. However, an array of int values cannot be stored into an int type.

```
int z = numbers ;
```

This code will cause a compiler error. What would it mean? You can't put a list of variables into a single variable. And the converse is true as well.

```
numbers = 31;
```

This code will also cause a compiler error. A single value cannot be stored into a whole list. You have to specify an index where it can be stored. Furthermore, you must be careful to specify a legal index. No negative index will ever be legal, and neither will an index greater than or equal to the number of elements in the array.

```
numbers [10] = 99;
```

This code will compile correctly. If you remember, we instantiated the array that numbers points at to have 10 elements, numbered 0 through 9. Thus, we are trying to store 99 into the element that is one index after the last legal element. As a result, Java will cause an error called an ArrayIndexOutOfBoundsException to happen, which will crash your program.

6.4.3 Using loops with arrays

One reason to use arrays is to avoid declaring 10 separate variables just to have 10 int values to work with. But once you have the array, you will often need an automated way to process it. Any of the three kinds of loops provides a powerful tool for performing operations on any array, but the for loop is an especially good match. Here is an example of a for loop that sets the values in an array to their indexes.

```
int [] values = new int [100];
for ( int i = 0; i < 100; i++)
    values [i] = i;
```

This sample of code shows how easy it is to iterate over every element in an array with a for loop, but it has a flaw in its style. Note that the number 100 is used twice: once in the instantiation of the array and a second time in the termination condition of the for loop. This fragment of code works fine, but if the programmer changes the length of values to be 50 or 500, the bounds of the for loop will also need to change. Furthermore, the length of the array might be determined by user input.

To make the code both more robust and readable, we can use the length field of the values array for the bound of the for loop.

```
int [] values = new int [100];
```

```
for ( int i = 0; i < values.length ; i++ )  
    values [i] = i;
```

The length field gives the length of the array that values points to. If the programmer wants to instantiate the array with a different length, that's fine. The length field will always reflect the correct value. Whenever possible, use the length field of arrays in your code. Note that the length field is read-only. If you try to set values.length to a specific value, your code will not compile.

Setting the values in an array is only one possible task you can perform with a loop. Let's assume that an array of type **double** named data has been declared, instantiated, and filled with user input. We could sum all its elements using the following code. A more elegant way to do the same summation is discussed in [Section 6.9.1](#).

```
double sum = 0.0;  
for ( int i = 0; i < data.length ; i++ )  
    sum += data [i];  
System.out.println ("The sum of your data is: " + sum );
```

So far, we have only discussed operations on the values in an array. It is important to realize that the order of those values can be equally important. We are going to create an array of **char** type named letters, initialized with some values, and then reverse the order of the array.

```
char [] letters = {"b", "y", "z", "a", "n", "t", "i", "n", "e"};  
int start = 0;  
int end = letters.length - 1;  
char temp ;  
while ( start < end ) {  
    temp = letters [ start ];  
    letters [ start ] = letters [ end ];  
    letters [ end ] = temp ;  
    start ++;  
    end --;  
}  
for ( int i = 0; i < letters.length ; i++ )  
    System.out.print ( letters [i] );
```

This code will print out enitnazyb. After initializing the letters array, we declare start and end, giving them the values 0, the first index of letters, and letters.length - 1, the last valid index of letters, respectively. Then, the **while** loop continues as long as the start is less than the end. The first three lines of each iteration of the **while** loop will swap the **char** at index start with the **char** at index end. The two lines after that will increment and decrement start and end, respectively. When the two meet in the middle, the entire array has been reversed. The simple **for** loop at the end prints out each **char** in letters. Of course, we could have printed out the array elements in reverse order without changing their order, but we wanted to reverse them, perhaps because we will need them reversed in the future.

6.4.4 Redirecting input

With arrays and loops, we can process a lot of data, but testing programs that process a lot of data can be tedious. Instead of typing data into the terminal, we can read data from a file. In Java, file I/C is a messy process that involves several objects and method calls. We're going to talk about it in depth in [Chapter 20](#), but for now we can use a quick and easy workaround.

If you create a text file using a simple text editor, you can *redirect* the file as input to a program. Everything you have written in the text file is treated as if it were being typed into the command line by a person. To do so, you type the command using java to run your class file normally, type the < sign, and then type the name of the file you want to use as input. For example, if you have a text file called numbers.txt that you want to use as input to a program stored in Summer.class, you could do so as follows.

```
java Summer < numbers.txt
```

Redirecting input this way is not a part of Java. Instead, it is a feature of the terminal running under your OS. Not all operating systems support input redirection, but virtually every flavor of Linux and Unix do, as well as the Windows command line and the Mac OS X terminal. We could write the program mentioned above and give it the simple task of summing all the numbers it gets as input.

Program 6.1: This program sums a list of numbers given as input. (Summer.java)

```
1 import java.util.*;
2
3 public class Summer {
4     public static void main ( String [] args ) {
5         Scanner in = new Scanner ( System.in );
6         System.out.print (" How many numbers do you want to add ? ");
7         int n = in.nextInt ();
8         int sum = 0;
9         for ( int i = 0; i < n; i++ ) {
10             System.out.print (" Enter next number : ");
11             sum += in.nextInt ();
12         }
13         System.out.println ("The sum of the numbers is " + sum );
14     }
15 }
```

Now, we can type out a file with a list of numbers in it and save it as numbers.txt. To conform with the program we wrote, we should also put the total count of numbers as the first value in the file. You can put each number on a separate line or just leave a space between each one. As long as they are separated by white space, the Scanner object will take care of the rest. You will have to type the numbers into the file once, but then you can test your program over and over with that file.

If you do run the program with the file you've created, you'll notice that the program still prompts you once for the total count of numbers and then prompts you many times to enter the next number. With redirected input, all that text runs together in a bizarre way. All the input is coming from

numbers.txt. If you expect a program to read strictly from redirected input, you can design your code a little differently. For one thing, you don't need to have explicit prompts for the user. For another, you can use a number of special methods from the Scanner class. The Scanner class has a several methods like `hasNextInt()` and `hasNextDouble()`. These methods will examine the input and see if there is another legal `int` or `double` and return `true` or `false` accordingly. If you expect a file to have only a long sequence of `int` values, you can use `hasNextInt()` to determine if you have reached the end of the file or not. Using `hasNextInt()`, we can simplify the program and remove the expectation that the first number gives the total count of numbers.

Program 6.2: This program sums a list of numbers given as input without prompting the user.
(QuietSummer.java)

```
1 import java.util.*;
2
3 public class QuietSummer {
4     public static void main ( String [] args ) {
5         Scanner in = new Scanner ( System.in );
6         int sum = 0;
7         while ( in.hasNextInt () )
8             sum += in.nextInt ();
9         System.out.println ("The sum of the numbers is " + sum );
10    }
11 }
```

On the other hand, you might be interested in the output of a program. The output could be very long or it might take a lot of time to produce or you might want to store it permanently. For these situations, it is possible to *redirect output* as well. Instead of printing to the screen, you can send the output to a file of your choosing. The syntax for this operation is just like the syntax for input redirection except that you use the `>` sign instead of `<`. To run QuietSummer with input from numbers.txt and output to sum.txt, we could do the following.

```
java QuietSummer < numbers.txt > sum.txt
```

You would be free to examine sum.txt at any time with your text editor of choice. When using output redirection, it makes more sense run to QuietSummer than Summer. If we had run Summer, all of that unnecessary output prompting the user to enter numbers would be saved in sum.txt.

6.5 Examples: Array usage

Here are a few examples of practical array usage. We are going to discuss some techniques useful mostly for searching and sorting. Searching for values in a list seems mundane, but it is one of the most practical tasks that a computer scientist routinely carries out. By making a computer do the work, it saves human beings countless hours of tedious searching and checking. Another important task is sorting. Sorting a list can make future searches faster and is the simplest way to find the

median of a list of values. Sorting is a fundamental part of countless real world problems.

In the examples below, we will first discuss finding the largest (or smallest) value in a list, move on to sorting lists, and then talk about a task that searches for words, like a dictionary look up.

Example 6.1: King of the hill

Finding the largest value input by a user is not difficult. Applying that knowledge to an array is pretty straightforward as well. This simple task is also a building block of the sorting algorithm we will discuss below. The key to finding the largest value in any list is to keep a temporary variable that records the largest value found so far. As we go along, we update the variable if we find a larger value. The only trick is initializing the variable to some appropriate starting value. We could initialize it to zero, but what if entire list of numbers is negative? Then, our answer would be larger than any of the numbers in the list. If our list of numbers is of type `int`, we could initialize our variable to `Integer.MIN_VALUE`, the smallest possible `int`. This approach works, but you have to remember the name of the constant and it does not improve the readability of the code.

When working with an array, the best way to find the largest value in the list is by setting your temporary variable to the first element (index 0) in the array. Below is a short snippet of code that finds the largest value in an `int` array named `values` in exactly this way.

```
int largest = values [0];
for ( int i = 1; i < values.length ; i++ )
    if( values [i] > largest )
        largest = values [i];
System.out.println ("The largest value is " + largest );
```

Note that the `for` loop starts at 1 not 0. Because `largest` is initialized to be `values[0]`, there is no reason to repeat that value. Doing so would still give the correct answer, but it wastes a tiny amount of time.

What is the feature of this code that makes it find the largest value? The key is the `>` operator. With the change of a single character, we could find the smallest value instead.

```
int smallest = values [0];
for ( int i = 1; i < values.length ; i++ )
    if( values [i] < smallest )
        smallest = values [i];
System.out.println ("The smallest value is " + smallest );
```

In addition to the necessary change from `>` to `<`, we also changed the output and the name of the variable to avoid confusion. Now we will show how repeatedly finding the smallest value in an array can be used to sort it. The largest value could be used equally well, but we will use the smallest. ■

Example 6.2: Selection sort

Sorting is the bread and butter of computer scientists. Much research has been devoted to finding the fastest ways to sort a list of data. The rest of the world assumes that sorting a list of data is trivial

because computer scientists have done such a good job solving this problem. The name of the sorting algorithm we are going to describe below is *selection sort*. It is **not** one of the fastest ways to sort data, but it is simple and easy to understand.

The idea behind selection sort is to find the smallest element in an array and put it at index 0 of the array. Then, from the remaining elements, find the smallest element and put it at index 1 of the array. The process continues, filling the array up from the beginning with the smallest values until the entire array is sorted. If the length of the array is n , we will need to look for the smallest element in the array $n - 1$ times. By putting the code that searches for the smallest value inside of an outer loop, we can write a program that does selection sort of **int** values input by the user as follows.

```
1 import java.util.*;
2
3 public class SelectionSort {
4     public static void main ( String [] args ) {
5         Scanner in = new Scanner ( System.in );
6         int n = in.nextInt ();
7         int [] values = new int [n];
8         int smallest ;
9         int temp ;
10        for ( int i = 0; i < values.length ; i++ )
11            values [i] = in.nextInt ();
```

This program is not very long, but there's a lot going on. After instantiating a Scanner, we read in the total number of values the list will hold. We cannot rely on the `hasNextInt()` method to tell us when to stop reading values. We need to know up front how many values we are going to store so that we can instantiate our array with the appropriate length. Then, we read each value into the array using the first **for** loop.

```
12        for ( int i = 0; i < n - 1; i++ ) {
13            smallest = i;
14            for ( int j = i + 1; j < n; j++ )
15                if( values [j] < values [ smallest ] )
16                    smallest = j;
17            temp = values [ smallest ];
18            values [ smallest ] = values [i];
19            values [i] = temp ;
20        }
```

The next **for** loop is where the actual sort happens. We start at index 0 and then try to find the smallest value to be put in that spot. Then, we move on to index 1, and so on, just as we described before. Note that we only go up to $n - 2$. We don't need to find the value to put in index $n - 1$, because the rest of the list has the $n - 1$ smallest numbers in it and so the last number must already be the largest. If you look carefully, you will notice that the inner **for** loop has the same overall shape as the loop used to find the smallest value in the previous example; however, there is one key difference.

Instead of storing the **value** of the smallest number in `smallest`, we now store the **index** of the smallest number. We need to store the index of the smallest number so that, in the next step, we can swap the corresponding element with the element at `i`, the spot in the array we are trying to fill. The three lines after the inner `for` loop are a simple swap to do exactly that.

```
21     System.out.print (" The sorted list is: ");
22     for ( int i = 0; i < values.length ; i++ )
23         System.out.print ( values [i] + " " );
24     }
25 }
```

After all the sorting is done, the final `for` loop prints out the newly sorted list. This program gives no prompts for user input, so it is well designed for input redirection. If you are going to make a file containing numbers you want to sort with this program, make sure that the first number is the total count of numbers in the file.

Again, this program sorts the list in ascending order (from smallest to largest). If you wanted to sort the list in descending order, you would only need to change the `<` to a `>` in the comparison of the inner `for` loop, although other changes are recommended for the sake of readability. ■

Example 6.3: Word search

In this example, we will read in a list of words and a long passage of text and keep track of the number of times each word in the list occurs in the passage. This kind of text searching has many applications. Similar ideas are used in a spell checker that needs to look up words in a dictionary. The incredibly valuable find and replace tools in modern word processors use some of the same techniques.

To make this program work, however, we need to read in a (potentially long) list of words and then a lot of text. We are forced to use input redirection (or some other file input) because typing this text in multiple times would be tedious. When we get to [Chapter 20](#), we will talk about ways to read from multiple files at the same time. Right now, we can only redirect input from a single file and so we are forced to put the list of words at the top of the file, followed by the text we want to search through.

```
1 import java.util.*;
2
3 public class WordCount {
4     public static void main ( String [] args ) {
5         Scanner in = new Scanner ( System.in );
6         int n = in.nextInt ();
7         String [] words = new String [n];
8         int [] counts = new int [n];
9         String temp ;
10        for ( int i = 0; i < words.length ; i++ )
11            words [i] = in.next ().toLowerCase ();
```

As in the last example, this program begins by reading in the length of the list of words. Then, it instantiates the String array words to hold these words. It also instantiates an array counts of type int to keep track of the number of times each word is found. By default, each element in counts is initialized to 0. The first **for** loop in the program reads in each word and stores it into the array words.

```
12     while ( in.hasNext () ) {  
13         temp = in.next ().toLowerCase ();  
14         for ( int i = 0; i < n; i++ )  
15             if( temp.equals ( words [i] ) ) {  
16                 counts [i]++;  
17                 break ;  
18             }  
19     }  
20     System.out.println ("The word counts are : ");
```

The **while** loop reads in each word from the text following the list and stores it in a variable called temp. Then, it loops through words and tests to see if temp matches any of the elements in the list. If it does, it increases the value of the element of counts that has the same index and breaks out of the inner **for** loop.

```
21     for ( int i = 0; i < words.length ; i++ )  
22         System.out.println ( words [i] + " " + counts [i]);  
23     }  
24 }
```

After all the words in the text have been processed, the final **for** loop prints out each word from the list, along with its counts.

This program uses two different arrays for bookkeeping: words contains the words we are searching for and counts contains the number of times each word has been found. These two arrays are separate data structures. The only link between them is the code we wrote to maintain the correspondence between their elements.

To give a clear picture of how this program should behave, here is a sample input file with two paragraphs from the beginning of *The Counte of Monte Cristo* by Alexandre Dumas.

```
7  
and  
at  
bridge  
for  
pilot  
vessel  
walnut
```

On the 24th of February, 1815, the look-out at Notre-Dame de la Garde signalled the three-master, the Pharaon from Smyrna, Trieste, and Naples.

As usual, a pilot put off immediately, and rounding the Chateau d'If, got on board the vessel between Cape Morgion and Rion island.

Immediately, and according to custom, the ramparts of Fort Saint-Jean were covered with spectators; it is always an event at Marseilles for a ship to come into port, especially when this ship, like the Pharaon, has been built, rigged, and laden at the old Phocean docks, and belongs to an owner of the city.

And here is the output one should get from running WordCount with input redirected from the file given above.

The word counts are:

and 6

at 3

bridge 0

for 1

pilot 1

vessel 1

walnut 0

For this example, the program works fine. However, our program would have given incorrect output if ship, spectators, or several other words in the text had been on the word list. You see, the next() method in the Scanner class reads in String values separated by white space. The word ship appears twice in the text, but the second instance is followed by a comma. Since the words are separated by white space only, the String “*ship*, ” does not match the String “*ship*”. Dealing with punctuation is not difficult, but it would increase the length of the code, and we leave it as an exercise. ■

Exercise 6.8

Example 6.4: Statistics

Imagine that you are a teacher who has just given an exam. You want to produce statistics for the class so that the students have some idea how well they have done. You want to write a Java program to help you produce the statistics, to save time now and in the future.

The statistics you want to collect are listed in the following table.

Statistic	Description
Maximum	Maximum score
Minimum	Minimum score

Mean	Average of all the scores
Standard Deviation	Sample standard deviation of the scores
Median	Middle value of the scores when ordered

[Example 6.1](#) covered how to find the maximum and minimum scores in a list. The mean is simply the sum of all the scores divided by the total number of scores. Standard deviation is a little bit trickier. It gives a measurement of how spread out the data is. Let n be the number of data points, label each data point x_i , where $1 \leq i \leq n$, and let \bar{x} be the mean of all the data points. Then, the formula for the sample standard deviation is as follows.

$$\sigma = \sqrt{\frac{1}{n-1} \sum_{i=1}^n (x_i - \bar{x})^2}$$

Finally, if you sort a list of numbers in order, the median is the middle value in the list, or the average of the two middle values, if the list has an even length.

These kinds of statistical operations are very useful and are packaged into many important business applications such as Microsoft Excel. This version will have a simple interface whose input comes from the command line. First, the total number of scores will be entered. Then, each score should be entered one by one. After all the data has been entered, the program should compute and output the five values.

Below we give the solution to this statistics problem. Several different tasks are combined here, but each of them should be reasonably easy to solve after the previous examples.

```

1 import java.util.*;
2
3 public class Statistics {
4     public static void main ( String [] args ) {
5         Scanner in = new Scanner ( System.in );
6         int n = in.nextInt ();
7         int [] scores = new int [n];
8         for ( int i = 0; i < n; i++ )
9             scores [i] = in.nextInt ();

```

In our solution, the `main()` method begins by reading in the total number of scores and declaring an `int` array of that length named `scores`. Then, we read in each of the scores and store them into `scores`.

```

10     int max = scores [0];
11     int min = scores [0];
12     int sum = scores [0];
13     for ( int i = 1; i < n; i++ ) {
14         if( scores [i] > max )
15             max = scores [i];
16         if( scores [i] < min )
17             min = scores [i];

```

```
18     sum += scores [i];  
19 }
```

Here we declare variables max, min, and sum to hold, respectively, the maximum, minimum, and sum of the elements in the array. Then, we set all three variables to the value of the first element of the array. These initializations make the following code work. In a single **for** loop, we find the maximum, minimum, and sum of all the values in the array. We could have done so with three separate loops, but this approach is more efficient. Setting max and min to scores[0] follows the pattern we have used before, but setting sum to the same value is also necessary in this case. Because the loop iterates from 1 up to scores.length - 1, we must include the value at index 0 in sum. Alternatively, we could have set sum to 0 and started the **for** loop at i = 0.

```
20 double mean = (( double ) sum )/n;  
21 System.out.println (" Maximum \t\t" + max );  
22 System.out.println (" Minimum \t\t" + min );  
23 System.out.println (" Mean \t\t\t" + mean );
```

In this short snippet of code, we compute the mean, being careful to store it into a variable of type **double**, and then print out the three statistics we have already computed.

```
24 double variance = 0;  
25 for ( int i = 0; i < n; i++ )  
26     variance += ( scores [i] - mean )*( scores [i] - mean );  
27 variance /= (n - 1);  
28 System.out.println (" Standard Deviation \t" + Math.sqrt (variance ));
```

At this point, we use the mean we have already computed to find the sample standard deviation. Following the formula for sample standard deviation, we subtract the mean from each score, square the result, and add it to a running total. Although the formula for sample standard deviation uses the bounds 1 to n , we translate them to 0 to $n - 1$ because of zero-based array numbering. Dividing the total by $n - 1$ gives the sample variance. Then, the square root of the variance is the standard deviation.

```
29 int temp ;  
30 for ( int i = 0; i < n - 1; i++ ) {  
31     min = i;  
32     for ( int j = i + 1; j < n; j++ )  
33         if( scores [j] < scores [min] )  
34             min = j;  
35     temp = scores [ min ];  
36     scores [ min ] = scores [i];  
37     scores [i] = temp ;  
38 }  
39 double median ;  
40 if( n % 2 == 0 )
```

```

41     median = ( scores [n /2] + scores [n/2 + 1]) /2.0;
42
43     else
44         median = scores [n /2];
45
46     System.out.println (" Median \t\t" + median );
47 }

```

To find the median, we use our selection sort code. Note that we have reused the variable min to hold the smallest value found so far, instead of declaring a new variable such as smallest. Some programmers might object to doing so, since we run the risk of interpreting the variable as the minimum value in the entire array, as it was before. Either approach is fine. If you worry about confusing people reading your code, add a comment.

After the array has been sorted, we need to do a quick check to see if its length is odd or even. If its length is even, we need to find the average of the two middle elements. If its length is odd, we can report the value of the single middle element.

Note that some of the statistics we found, such as the maximum, minimum, or mean, could be computed using loops without an array for storage. However, the last two tasks need to store all of the values at once in order to work. Finding the sample standard deviation of a list of values requires its mean. At least two passes over the data are needed to compute the sample standard deviation: one to find the mean and another to apply the equation for sample standard deviation. ■

6.6 Concepts: Multidimensional lists

In the previous half of the chapter, we focused on lists of data and how to store them in Java in arrays. The arrays we have discussed already are *one-dimensional* arrays. That is, each element in the array has a single index that refers to it. Given a specific index, an element will have that index, come before it, or come after it. These kinds of arrays can be used to solve a huge number of problems involving lists or collections of data.

Sometimes, the data needs to be represented with more structure. One way to provide this structure is with a *two-dimensional* array. You can think of a two-dimensional array as a table of data. Instead of using a single index, a two-dimensional array has two indexes. Usually, we think about these dimensions as rows and columns. Below is a table of information that gives the distances in miles between the five largest cities in the United States.

City	New York	Los Angeles	Chicago	Houston	Phoenix
New York	0	2,791	791	1,632	2,457
Los Angeles	2,791	0	2,015	1,546	373
Chicago	791	2,015	0	1,801	1,181
Houston	1,632	1,546	1,801	0	1,176
Phoenix	2,457	373	1,181	1,176	0

The position of each number in the table is a fundamental part of its usefulness. We know that the

distance from Chicago to Houston is 1,801 miles because that number is in the Chicago row and the Houston column. A two-dimensional array shares almost all of its properties with a one-dimensional array. It is still a homogeneous, static data structure with random access. If the example above were made into a Java array, the numbers themselves would be the elements of the array. The names of the cities would need to be stored separately, perhaps in an array of type String.

There is no reason to confine the idea of a two-dimensional list to a table of values. Many games are played on a two-dimensional grid. One of the most famous such games is chess. As with so many other things in computer science, we must come up with an abstraction that mirrors reality and allows us to store the information inside of a computer. For chess, we will need an 8×8 two-dimensional array. We can represent each piece in the board with a **char**, using the encoding given in [Table 6.1](#).

Piece	Encoding
Pawn	'P'
Knight	'N'
Bishop	'B'
Rook	'R'
Queen	'Q'
King	'K'

Table 6.1: Encoding chess pieces with the **char** type

Index	0	1	2	3	4	5	6	7
0	'R'	'N'	'B'	'Q'	'K'	'B'	'N'	'R'
1	'P'							
2								
3								
4								
5					'p'			
6	'p'	'p'	'p'	'p'		'p'	'p'	'p'
7	'r'	'n'	'b'	'q'	'k'	'b'	'n'	'r'

Table 6.2: Using upper case characters for black pieces and lower case characters for white pieces, we could represent a game of chess after a classic king's pawn open by white as shown.

Observe that, just as with one-dimensional arrays, the indexes for rows and columns in two-dimensional arrays also use zero-based counting.

After the step from one-dimensional arrays to two-dimensional arrays, it is natural to wonder if there can be arrays of even higher dimension. We can visualize a two-dimensional array as a table, but a three-dimensional array is harder to visualize. Nevertheless, there are uses for three-dimensional arrays.

Consider a professor who is taking a survey of students in her course. She wants to know how many students there are in each of three categories: gender, class level, and major. If she treats each of these as a dimension and assigns an index to each possible value, she could store the results in a three-dimensional array. For gender she could pick male = 0 and female = 1. For class level she could pick freshman = 0, sophomore = 1, junior = 2, senior = 3, and other = 4. Assuming it is a computer science class, for major she could pick computer science = 0, math = 1, other science = 2, engineering = 3, humanities = 4. Using this system she could compactly store the number of students in any combination of categories she was interested in. For example, the total number of female sophomore engineering students would be stored in the cell with gender index 1, class level index 1, and major index 3.

Three dimensions is usually the practical limit when programming in Java. If you find an especially good reason to use four or higher dimensions, feel free to do so, but it should happen infrequently. The Java language has no set limit on array dimensions, but most virtual machines have the absurdly high limitation of 255 different dimensions.

6.7 Syntax: Advanced arrays in Java

Now that we have discussed the value of storing data in multidimensional lists, we will describe the Java language features that allow you to do so. The changes needed to go from one-dimensional arrays to two-dimensional and higher arrays are quite simple. First, we will describe how to declare, instantiate, and index into two-dimensional arrays. Then, we will discuss some of the ways in which arrays (both one-dimensional and higher) are different from primitive data types. Next, we will explain how it is possible make two-dimensional arrays in Java where the rows are not all the same length. Finally, we will cover some of the most common mistakes programmers make with arrays.

6.7.1 Multidimensional arrays

When declaring a two-dimensional array, the main difference from a one-dimensional array is an extra pair of brackets. If we wish to declare a two-dimensional array of type `int` in which we could store values like the table of distances above, we would do so as follows.

```
int [][] distances ;
```

As with one-dimensional arrays, it is legal to put the brackets on the other side of the variable identifier or, even more bizarrely, have a pair on each side.

Once the array is declared, it must still be instantiated using the `new` keyword before it can be used. This time we will use two pairs of brackets, with the number in the first pair specifying the number of rows and the number in the second pair specifying the number of columns.

```
distances = new int [5][5];
```

After the instantiation, we will have 5 rows and 5 columns, giving a total of 25 locations where `int` values can be stored. Indexing these locations is done by specifying row and column values in the

brackets. So, to fill up the table with the distances between cities given above we can use the following tedious code.

```
// New York
distances [0][1] = 2791;
distances [0][2] = 791;
distances [0][3] = 1632;
distances [0][4] = 2457;
// Los Angeles
distances [1][0] = 2791;
distances [1][2] = 2015;
distances [1][3] = 1546;
distances [1][4] = 373;
// Chicago
distances [2][0] = 791;
distances [2][1] = 2015;
distances [2][3] = 1801;
distances [1][4] = 1181;
// Houston
distances [3][0] = 1632;
distances [3][1] = 1546;
distances [3][2] = 1801;
distances [3][4] = 1176;
// Phoenix
distances [4][0] = 2457;
distances [4][1] = 373;
distances [4][2] = 1181;
distances [4][3] = 1176;
```

You will notice that we did not specify values for distances[0][0], distances[1][1], distances[2][2], distances[3][3], or distances[4][4], since each of these already has the default value of 0.

Much more often, multidimensional array manipulation will use nested **for** loops. For example, we could create an array with 3 rows and 4 columns, and then assign values to those locations such that they were numbered increasing across each row.

```
int [][] values = new int [3][4];
int number = 1;
for ( int i = 0; i < values.length ; i++ )
    for ( int j = 0; j < values [0].length ; j++ ) {
        values [i][j] = number ;
        number++;
    }
```

This code would result in an array filled up like the following table.

1	2	3	4
5	6	7	8
9	10	11	12

The bounds for the outer `for` loop in this example uses `values.length`, giving the total number of rows. Then, the inner `for` loops uses `values[0].length`, which is the length (number of columns) of the first row. In this case, all the rows of the array have the same number of columns, but this is not always true, as we will discuss later.

6.7.2 Reference types

All array variables are *reference* type variables, not simple values like most of the types we have discussed so far. A reference variable is a name for an object. You might recall that we described the difference between reference types and primitive types in [Section 3.2](#), but the only reference type we have considered in detail is `String`.

More than one reference variable can point at the same object. When one object has more than one name, this is called *aliasing*. The `String` type is immutable, meaning that an object of type `String` cannot change its contents. Arrays, however, are mutable, which means that aliasing can cause unexpected results. Here is a simple example with one-dimensional array aliasing.

```
int [] array1 = new int [10];
for ( int i = 0; i < array1.length ; i++ )
    array1 [i] = i;
int [] array2 = array1 ;
array2 [3] = 17;
System.out.println ( array1 [3]) ;
```

Surprisingly, the value printed out will be 17. The variables `array1` and `array2` are references to the same fundamental array. Unlike primitive values, the complete contents of `array1` are not copied to `array2`. Only one array exists because only one array has been created by the `new` keyword. So, when index 3 of `array2` is updated, index 3 of `array1` changes as well, because the two variables are simply two names for one array.

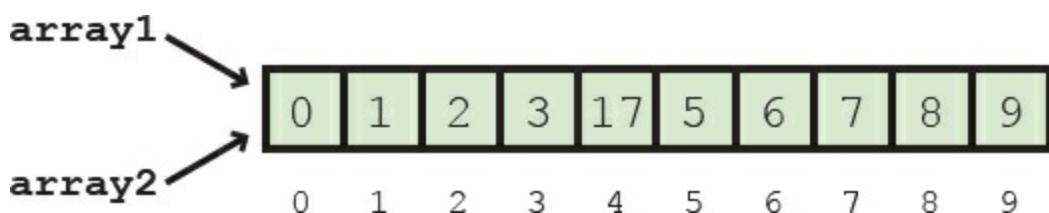


Figure 6.2: Two array references pointing to a single array object.

Sometimes this reference feature of Java allows us to write code that is confusing or has unexpected consequences. However, the benefit is that we can assign one array to another without incurring the expense of copying the entire array. If you had created an array with 1,000,000 elements, copying that array several times could get very expensive in terms of program running time.

The best rule of thumb for understanding reference types is that there is only one actual object for every call to `new`. The primary exception to this rule is that uses of `new` can be hidden from the user when they are in method calls.

```
String greeting = new String ("Hello");  
String pronoun = greeting.substring (0,2);
```

At the end of this code, the reference pronoun will point to an object containing the String “He”. The `substring()` method invokes `new` internally, generating a new `String` object completely separate from the `String` referenced by `greeting`. This code may look unusual because we are explicitly using `new` to make a `String` object containing “Hello”. The `String` class is different from every other class because it can be instantiated without using the `new` keyword. The line

`String greeting = "Hello";` implicitly calls `new` to create an object containing the String “Hello” and functions nearly the same as the similar line above.

6.7.3 Ragged arrays

We are ashamed to say that we have lied to you. In Java, there is no such thing as a multidimensional array. Instead, the examples of two-dimensional and three-dimensional arrays we have given above are actually arrays of arrays (of arrays). Thinking about multidimensional arrays in this way can give the programmer more flexibility.

If we return to the definition of the two-dimensional array with 3 rows and 4 columns, we can instantiate each row separately instead of as a block.

```
int [][] values = new int [3][];  
int number = 1;  
for ( int i = 0; i < values.length ; i++ ) {  
    values [i] = new int [4];  
    for ( int j = 0; j < values [i].length ; j++ ) {  
        values [i][j] = number ;  
        number ++;  
    }  
}
```

This code is functionally equivalent to the earlier code that instantiated all 12 locations at once. The same could be done with a three-dimensional array or higher. We can specify the length of each row independently, and, more bizarrely, we can give each row a different length. A multidimensional array whose rows have different lengths is called a *ragged array*.

A ragged array is usually unnecessary. The main reason to use a ragged array is to save space, when you have tabular data in which the lengths of each row varies a great deal. If the lengths of the rows vary only a little, it is probably not worth the extra hassle. However, if some rows have 10 elements and others have 1,000,000, the space saved can be significant.

We can apply the idea of ragged arrays to the table of distances between cities. If you examine this table, you will notice that about half the data in it is repeated, because the distance from Chicago to

Los Angeles is the same as the distance from Los Angeles to Chicago, and so on. We can store the data in a triangular shape to keep only the unique distance information.

City	New York	Los Angeles	Chicago	Houston	Phoenix
New York	0				
Los Angeles	2,791	0			
Chicago	791	2,015	0		
Houston	1,632	1,546	1,801	0	
Phoenix	2,457	373	1,181	1,176	0

We could create this table in code by doing the following.

```
distances = new int [5][];
// New York
distances [0] = new int [1];
// Los Angeles
distances [1] = new int [2];
distances [1][0] = 2791;
// Chicago
distances [2] = new int [3];
distances [2][0] = 791;
distances [2][1] = 2015;
// Houston
distances [3] = new int [4];
distances [3][0] = 1632;
distances [3][1] = 1546;
distances [3][2] = 1801;
// Phoenix
distances [4] = new int [4];
distances [4][0] = 2457;
distances [4][1] = 373;
distances [4][2] = 1181;
distances [4][3] = 1176;
```

With this table a user cannot simply type in `distances[0][4]` and hope to get the distance from New York to Phoenix. Instead, we have to be careful to make sure that the index of the first city is never larger than the index of the second city. If we are reading in the indexes of the cities from a user, we can write some code to do this check. Let `city1` and `city2`, respectively, contain the indexes of the cities the user wants to use to find the distances between.

```
if( city1 > city2 ) {
    int temp = city1 ;
    city1 = city2 ;
    city2 = temp ;
```

```
}
```

```
System.out.println ("The distance is: " + distances [ city1 ][ city2 ] + " miles ");
```

If we wanted to be even cleverer, we could eliminate the zero entries from the table, but then the ragged array would have one fewer row than the original two-dimensional array.

6.7.4 Common pitfalls

Even one-dimensional arrays make many new errors possible. Below we list two of the most common mistakes made with both one-dimensional and multidimensional arrays.

Pitfall: Array out of bounds

The length of an array is determined at runtime. Sometimes the number is specified in the source code, but it is always possible for an array to be instantiated based on user input. The Java compiler does not do any checking to see if you are in the right range. If your program tries to access an illegal element, it will crash with an `ArrayIndexOutOfBoundsException`.

```
int [] array = new int [100];  
for ( int i = 0; i <= array.length ; i++ )  
    array [i] = i;
```

Here is a classic example. By iterating through the loop one too many times, the program will try to store 100 into `array[100]`, when the last index of the array is 99. In C and C++, pointer arithmetic allowed a negative index to be valid for an array in some cases. In Java, a negative index will always throw an `ArrayIndexOutOfBoundsException`.

There are other less common causes for going outside of array bounds. Imagine that you are scanning through a file that has been redirected to `input`, keeping a count of the occurrences of each letter of the alphabet in the file.

```
Scanner in = new Scanner ( System.in );  
int [] counts = new int [26];  
String word ;  
while ( in.hasNext () ) {  
    word = in.next ().toLowerCase ();  
    for ( int i = 0; i < word.length (); i++ )  
        counts [ word.charAt (i) - 'a' ]++;  
}
```

This segment of code does a decent job of counting the occurrences of each letter. The `while` loop continues to execute as long as there is another String worth of data to read in the file. The inner `for` loop iterates through each `char` in the String and increments the appropriate element of the `counts` array. By subtracting the value `'a'`, we normalize the `char` values `'a'` through `'z'` to 0 through 25. However, if there is any punctuation in the file, simply subtracting `'a'` will not work. The Unicode value of `'.'`, for example, is 46. The Unicode value of `'a'` is 97. Subtracting 97 from 46 will make this code try to increment index -51 of the array. An additional check should be put into this code to make sure that the `char` value being examined is a letter.

Pitfall: Uninitialized reference arrays

Another problem only comes up with arrays of reference types. Whenever the elements of an array are primitive data types, memory for that type is allocated. Whenever the elements of the array are reference types, only references to objects, initialized to `null` are allocated. Because it's an array of primitive values, the following code works fine.

```
int [] primitives = new int [100];
primitives [67]++;

```

The following code, however, will cause a `NullPointerException`.

```
String [] references = new String [100];
int x = references [67].length ();

```

Arrays of reference types must initialize each element before using it. The `NullPointerException` could be avoided as follows.

```
String [] references = new String [100];
for ( int i = 0; i < references.length ; i++ )
    references [i] = new String ();
int x = references [67].length ();

```

In this case, there would be no error, although `references[67].length()` would still be 0, and that is probably not what the programmer intended.

A similar error can happen with multidimensional arrays.

```
int [][] table = new int [10][];
for ( int i = 0; i < table.length ; i++ )
    table [i][i] = i;

```

Because an array is itself a reference type, the `table` array contains 10 references to `null` for each of its 10 rows. Unless those rows are instantiated, the JVM will again throw a `NullPointerException` when attempting to access an `int` value in the table. This error confuses many beginner programmers because no reference types appear to be involved.

6.8 Examples: Two-dimensional arrays

Below we give some examples where two-dimensional arrays can be helpful. We start with a very simple calendar example, move on to matrix and vector multiplication useful in math, and finish with a game.

Example 6.5: Calendar

We are going to create a calendar that can be printed to the console to show which day of the week each day lands on. Our program will prompt the user for the day of the week the month starts on and for the total number of days in the month. Our program will print out labels for the seven days of the week, followed by numbering starting at the appropriate place, and wrapping such that each

numbered day of the month falls under the appropriate day of the week.

Program 6.3: This program prints a calendar for a given month, formatted week by week.
(Calendar.java)

```
1 import java.util.*;
2
3 public class Calendar {
4     public static void main ( String [] args ) {
5         String [][] squares = new String [7][7];
6         squares [0][0] = " Sun ";
7         squares [0][1] = " Mon ";
8         squares [0][2] = " Tue ";
9         squares [0][3] = " Wed ";
10        squares [0][4] = " Thu ";
11        squares [0][5] = " Fri ";
12        squares [0][6] = " Sat ";
13        for ( int i = 1; i < squares.length ; i++ )
14            for ( int j = 0; j < squares [i]. length ; j++ )
15                squares [i][j] = " ";
16        Scanner in = new Scanner ( System.in );
17        System.out.print (" Which day does your month start on?"
18                         + " (0 - 6) ");
19        int start = in.nextInt () // read starting day
20        System.out.print (" How many days does your month have ?"
21                         + " (28 - 31) ");
22        int days = in.nextInt () // read days in month
23        int day = 1;
24        int row = 1;
25        int column = start ;
26        while ( day <= days ) { // fill calendar
27            squares [ row ][ column ] = "" + day;
28            day++;
29            column++;
30            if( column >= squares [ row ].length ) {
31                column = 0;
32                row++;
33            }
34        }
35        for ( int i = 0; i < squares.length ; i++ ) {
36            for ( int j = 0; j < squares [i].length ; j++ )
37                System.out.print ( squares [i][j] + "\t" );
38            System.out.println ();
39        }
```

```
40      }  
41 }
```

First, our code creates a 7×7 array of type String called squares. The array needs 7 rows so that it can start with a row to label the days and then output up to 6 rows to cover the weeks. (Months with 31 days span parts of 6 different weeks if they start on a Friday or a Saturday.) The number of columns corresponds to the seven days of the week. Next, we initialize the first row of the array to abbreviations for each day of the week. Then, we initialize the rest of the array to be a single space.

Our program then prompts the user for the day the month starts on, using a **for** loop to print out the choices that have already been saved in squares. The program also prompts the user for the total number of days in the month.

The main work of the program is done by the **while** loop, which fills each square with a steadily increasing day number for each column, moving on to the next row when a row is filled. Finally, the two nested **for** loops at the end print out the contents of squares, putting a tab ('**\t**') between each column and starting a new line for each row. ■

Example 6.6: Matrix-vector multiplication

Arrays give a natural way to represent vectors and matrices. In 3D graphics and video game design, we can represent a point in 3D space as a vector with three elements: x , y , and z . If we want to rotate the three-dimensional point represented by this vector, we can multiply it by a matrix. For example, to rotate a point around the x -axis by θ degrees, we could use the following matrix.

$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos \theta & -\sin \theta \\ 0 & \sin \theta & \cos \theta \end{bmatrix}$$

Given an $m \times n$ matrix A , let A_{ij} be the element in the i^{th} row, j^{th} column. Given a vector v of length n , let v_i be the i^{th} element in the vector. To multiply A by v , we use the following equation to find the i^{th} element of the resulting vector v' .

$$v'_i = \sum_{j=1}^n A_{ij} \cdot v_j$$

By transforming this equation to Java code, we can write a program that can read in a three-dimensional point and rotate it around the x-axis by the amount specified by the user.

Program 6.4: This program uses matrix multiplication to rotate a point in three-dimensional space. (MatrixRotate.java)

```
1 import java.util.*;  
2  
3 public class MatrixRotate {  
4     public static void main ( String [] args ) {  
5         double [] point = new double [3];
```

```

6   System.out.println (" What point do you want to rotate ?");
7   Scanner in = new Scanner ( System.in );
8   System.out.print ("x: ");
9   point [0] = in. nextDouble ();
10  System.out.print ("y: ");
11  point [1] = in. nextDouble ();
12  System.out.print ("z: ");
13  point [2] = in.nextDouble ();
14  System.out.print (" What angle around the x- axis ? ");
15  double theta = Math.toRadians (in.nextDouble ());
16  double [][] rotation = new double [3][3];
17  rotation [0][0] = 1;
18  rotation [1][1] = Math.cos( theta );
19  rotation [1][2] = -Math.sin ( theta );
20  rotation [2][1] = Math.sin( theta );
21  rotation [2][2] = Math.cos( theta );
22  double [] rotatedPoint = new double [3];
23  for ( int i = 0; i < rotatedPoint.length ; i++ )
24    for ( int j = 0; j < point.length ; j++ )
25      rotatedPoint [i] += rotation [i][j]* point [j];
26  System.out.println (" Rotated point : [" + rotatedPoint [0] +
27    "," + rotatedPoint [1] + "," + rotatedPoint [2] + "]");
28 }
29 }
```

This program begins by declaring a array of type **double** to hold the vector and then reading three values from the user into it. Then, the program reads in the angle of rotation in degrees and converts it to radians. Next, we use the Math class to calculate the values in the rotation matrix. Note that we do not change the values that need to be zero. Finally, we use a **for** loop to perform the matrix-vector multiplication and then print out the answer. Again, the summing done by our calculations uses the fact that all elements of rotatedPoint are initialized to 0.0. ■

Example 6.7: Tic Tac Toe

Almost every child knows the game of Tic Tac Toe. Its playing area is a 3×3 grid. Players take turns placing X's and O's, trying to get three in a row. Strategically, it is not the most interesting game since two players who make no mistakes will always tie. Still, we present a program that allows two human players to play the game because the manipulations of a two-dimensional array in the program are similar to those for more complicated games such as Connect Four, checkers, chess, or Go. Our program will catch any attempt to play on a location that has already been played and will determine the winner, if there is one.

```

1 import java.util.*;
2
3 public class TicTacToe {
```

```

4  public static void main ( String [] args ) {
5      Scanner in = new Scanner ( System.in );
6      char [][] board = new char [3][3];
7      for ( int i = 0; i < board.length ; i++ )
8          for ( int j = 0; j < board [0].length ; j++ )
9              board [i][j] = ' ';
10     boolean turn = true ;
11     boolean gameOver = false ;
12     int row , column , moves = 0;
13     char shape ;

```

Games often give rise to complex programs, since rules that are intuitively obvious to humans may be difficult to state explicitly in Java. Our program begins by setting up quite a few variables and objects. First, we create a Scanner to read in data. Then, we declare and instantiate our 3×3 playing board as a two-dimensional array of type `char`. We want any unplayed space on the grid to be the `char` for a space, so we fill the array with `' '`. Next, we declare a `boolean` value to keep track of whose turn it is and another to keep track of whether the game is over or not. Finally, we declare variables to hold the row, the column, the number of moves that have been made so far and the current shape (`'X'` or `'O'`).

```

15     while ( ! gameOver ) {
16         shape = turn ? 'X' : 'O';
17         System.out.print ( shape + "s turn.Enter row (0 -2): ");
18         row = in.nextInt ();
19         System.out.print (" Enter column (0 -2): ");
20         column = in.nextInt ();
21         if( board [ row ][ column ] != ' ' )
22             System.out.println (" Illegal move ");
23         else {
24             board [ row ][ column ] = shape ;
25             moves++;
26             turn = ! turn ;
27             // print board
28             System.out.println ( board [0][0] + "|"
29                     + board [0][1] + "|" + board [0][2] );
30             System.out.println (" -----");
31             System.out.println ( board [1][0] + "|"
32                     + board [1][1] + "|" + board [1][2] );
33             System.out.println (" -----");
34             System.out.println ( board [2][0] + "|"
35                     + board [2][1] + "|" + board [2][2] + "\n");

```

The core of the game is a `while` loop that runs until `gameOver` becomes `true`. The first line of the body of this loop is an obscure Java shortcut often referred to as the *ternary operator*. This line is

really shorthand for the following.

```
if( turn )
    shape = 'X';
else
    shape = 'O';
```

The ternary operator works with a condition followed by a question mark and then two values separated by a colon. If the condition is `true`, the first value is assigned, otherwise the second value is assigned. It is perfect for situations like this where one value is needed when `turn` is `true` and another is needed when `turn` is `false`. The ternary operator is a useful trick, but it should not be overused.

After assigning the appropriate value to `shape`, our code reads in the row and column values for the current player's next move. If the row and column selected correspond to a spot that has already been taken, the program gives an error message. Otherwise, the program sets `board[row][column]` to the appropriate symbol, increments `moves`, and changes the value of `turn`. Then, it prints out the board. We point out that our program does not do any bounds checking on row and column. If a user tries to place a move at row 5 column 3, our program will try to do so and crash. Four additional clauses in the if statement could be used to add bounds checking.

```
36     // check rows
37     for ( int i = 0; i < board.length ; i++ )
38         if( board [i ][0] == shape && board [i ][1] ==
39             shape
40             && board [i ][2] == shape )
41             gameOver = true ;
42     // check columns
43     for ( int i = 0; i < board [0].length ; i++ )
44         if( board [0][ i ] == shape && board [1][ i ] ==
45             shape
46             && board [2][ i ] == shape )
47             gameOver = true ;
48     // check diagonals
49     if( board [0][0] == shape && board [1][1] == shape
50             && board [2][2] == shape )
51             gameOver = true ;
52     if( board [0][2] == shape && board [1][1] == shape
53             && board [2][0] == shape )
54             gameOver = true ;
55     if( gameOver )
56         System.out.println ( shape + " wins !");
57     else if( moves == 9 ){
58         gameOver = true ;
59         System.out.println (" Tie game !");
60     }
```

```
59         }
60     }
61 }
62 }
```

Perhaps the trickiest part of our Tic Tac Toe program is checking for a win. First we check each row to see if it contains three in a row. Then, we check each column. Finally, we check the two diagonals. If any of those checks ended the game, we announce a winner. Otherwise, if the number of moves has reached 9 with no winner, it must be a tie game. In a larger game (such as Connect Four, we would want to find better ways to automate checking rows, columns, and diagonals. For one thing, a very large board we mean that we would not want to check the entire thing each move. Instead, we could focus only on rows, columns, and diagonals affected by the last move. ■

Exercise 6.11

6.9 Advanced Syntax: Special array tools in Java

Arrays are fundamental data structures in many programming languages. There are often special syntactical tools or libraries designed to make them easier to use. In this section, we explore two advanced tools, the for-each loop and the Arrays utility class.

6.9.1 The for-each loop

In [Chapter 5](#) we describe three loops: **while** loops, **for** loops, and **do-while** loops. Although these are the only three loops in Java, there is a special form of the **for** loop designed for use with arrays (and some other data structures). This construct is usually called a *for-each loop*.

A for-each loop does not have the three-part header of a regular **for** loop. Instead, it is designed to iterate over the contents of an array. Inside its parentheses is a declaration of a variable with the same type of the elements of the array, then a colon (:), then the name of the array. Consider the following example of a for-each loop used to sum the values of an array of **int** values called `array`. As with all loops in Java, braces are optional if there is only one executable statement in the loop.

```
int sum = 0;
for ( int value : array )
    sum += value ;
```

This code functions in exactly the same way as the traditional **for** loop we would use to solve the same problem.

```
int sum = 0;
for ( int i = 0; i < array.length ; i++ )
    sum += array [i];
```

The advantage of the for-each loop is that it is shorter and clearer. There is also no worry about being off by one with your indexes. The for-each loop iterates over every element in the array, no indexes needed!

For-each loops can be nested or used inside of other loops. Consider the following nested for-each loops that print out all of the kinds of chess pieces, in both black and white colors.

```
String [] colors = {"Black", "White"};
String [] pieces =
    {"King", "Queen", "Rook", "Bishop", "Knight", "Pawn"};
for (String color : colors)
    for (String piece : pieces)
        System.out.println (color + " " + piece);
```

For-each loops do have a few drawbacks. They are designed for iterating through an entire array. It is ugly to try to make them stop early, and it is impossible to make them go back to previous values. They are also only designed for **read** access, not write access. The variable in the header of the for-each loop takes on each value in the array in turn, but assigning values to that variable have no effect on the underlying array. Consider the following **for** loop that assigns 5 to every value in array.

```
for (int i = 0; i < array.length; i++)
    array[i] = 5;
```

This kind of assignment is impossible in a for-each loop. The “equivalent” for-each loop does nothing. It assigns 5 to the local variable value but never changes array.

```
for (int value : array)
    value = 5;
```

Although for-each loops are great for arrays, they can also be used for any other data structures that implements the Iterable interface. We discuss interfaces in [Chapter 10](#) and dynamic data structures in [Chapters 18](#) and [19](#).

6.9.2 The Arrays class

The designers of the Java API knew that arrays were important and added a special `Arrays` class to manipulate them.

This class has a number of static methods that can be used to search for values in arrays, make copies of arrays, copy selected ranges of arrays, test arrays for equality, fill arrays with specific values, sort arrays, convert an entire array into a String representation, and more. The signatures of the methods below are given for **double** arrays, but most methods are overloaded to work with all primitive types and reference types.

Method	Purpose
<code>binarySearch(double[] array, double value)</code>	Returns index of value inside array or a negative number if it cannot be found. Adding 1 to the negative number and then negating it will give the index where the value would have been.
<code>copyOf(double[] array, int length)</code>	Returns a copy of array with length length, either truncated or padded if it doesn't match the length of array.
<code>copyOfRange(double[] array, int from, int to)</code>	Returns a copy of array from the range starting at from and going up to but not

array, int from, int to)	including to.
equals(double[] array1, double[] array2)	Returns true if array1 and array2 have the same number of elements, each pair of which is equal.
fill(double[] array, double value)	Fills array with copies of value.
sort(double[] array)	Sorts array using natural ordering. This method can fail for Object arrays in which the objects are not comparable.
toString(double[] a)	Returns a String containing representations of each element separated with commas.

Consult the API for more information. Even though tasks like fill() are simple, it is worth using the method from Arrays instead of writing your own. The methods in the Java API have often been tuned for speed and use special commands that are not accessible to regular Java programmers.

6.10 Solution: Conway's Game of Life

Here we present our solution to the Conway's Game of Life simulation. Our program is designed to run the simulation with 24 rows and 80 columns, although it would be easy to change those dimensions.

```

1 public class Life {
2     public static void main ( String [] args ) {
3         final int ROWS = 24;
4         final int COLUMNS = 80;
5         final int GENERATIONS = 500;
6         boolean [][] board = new boolean [ ROWS ][ COLUMNS ];
7         boolean [][] temp = new boolean [ ROWS ][ COLUMNS ];
8         boolean [][] swap ;
9         for ( int row = 0; row < ROWS ; row ++ )
10            for ( int column = 0; column < COLUMNS ; column ++ )
11                board [ row ][ column ] = ( Math.random () > 0.9 );

```

The main() method of our program starts by defining ROWS, COLUMNS, and GENERATIONS named constants using the `final` keyword. Next, we create **two** arrays with ROWS rows and COLUMNS columns. The board array will hold the current generation. The temp array will be used to fill in the next generation. Then, temp will be copied into board, and the process will repeat. The swap variable is just a reference we will use to swap board and temp. We randomly fill the board, making 10% of the cells living. Again, you may wish to play with this number to see how the patterns in the simulation are affected.

```

12     int total ;
13     for ( int generation = 0; generation < GENERATIONS ;
14         generation ++ ) {

```

```

15    for ( int row = 0; row < ROWS ; row ++ )
16        for ( int column = 0; column < COLUMNS ; column ++ ) {
17            total = 0;
18            for ( int i = Math.max ( row - 1, 0 );
19                  i < Math.min ( row + 2, ROWS ); i++ )
20                for( int j = Math.max ( column -1, 0 );
21                      j < Math.min ( column +2, COLUMNS ); j++ )
22                    if( (i != row || j != column )
23                        && board [i][j] )
24                      total++;
25            if( board [row ][ column ] )
26                temp [row ][ column ] = ( total == 2 ||
27                total == 3);
28            else
29                temp [row ][ column ] = ( total == 3);
30        }
31        swap = board ;
32        board = temp ;
33        temp = swap ;

```

The **for** loop at the beginning of this segment of code runs once for each generation we simulate. The two nested **for** loops examine each cell in board. The two **for** loops nested inside of those loops do the calculations to determine if a cell will be living or dead in the next generation. These inner loops start one row before the current row and finish one row after the current row. They do the same for columns. The `Math.max()` and `Math.min()` methods are used to keep the loops from going out of bounds of the array. When backing up a row or a column, the `Math.max()` methods make sure that we do not generate an index smaller than 0. When going forward a row or a column, the `Math.min()` methods make sure that we do not generate an index greater than `ROWS - 1` or `COLUMNS - 1`.

After these two innermost **for** loops have counted the total of living cells around the cell in question, we decide the fate of the cell for the next generation. If the cell is living and has exactly 2 or 3 living neighbors, it will continue to be living. If a cell is dead, it will come to life only if it has exactly 3 living neighbors. After we have stored the state of each cell in the next generation into `temp`, we swap `board` and `temp`, using the `swap` variable. We could have thrown out the old array stored in `board` instead of swapping it with `temp`, but then we would have to create a new array for `temp` each time, which is less efficient.

```

34        for ( int i = 0; i < 100; i++ )
35            System.out.println ();
36        for ( int row = 0; row < ROWS ; row ++ ) {
37            for ( int column = 0; column < COLUMNS ; column ++ )
38                if( board [row ][ column ] )
39                    System.out.print ("*");
40                else
41                    System.out.print (" ");

```

```

42         System.out.println ();
43     }
44     try { Thread.sleep (100) ; }
45     catch ( InterruptedException e ) {}
46   }
47 }
48 }
```

The first `for` loop in this segment prints 100 blank lines to clear the screen, as we explained earlier. The two nested `for` loops print out the state of the current generation, with a * for each living cell and a blank space for each dead one. After the output, the code sleeps for 100 milliseconds to give the effect of an animation. We will discuss exceptions in general in [Chapter 12](#) and give more information about the `Thread.sleep()` method in [Chapter 13](#).

6.11 Concurrency: Arrays

Arrays are critical to concurrent programming in Java. In [Chapter 13](#), we will explain how to create independent threads of execution, each of which is tied to a `Thread` object. If you have a dual, quad, or higher core computer, you might want to use two or four threads to solve a problem, but some programs can use hundreds. How can you keep track of all those `Thread` objects? In many cases, you will hold references to them in an array.

Arrays also hold large lists of data. It is common for threaded programs to share a single array which each thread reads and writes to. In this way, memory costs are kept low because there is only one copy of all the data. In the simplest case, each thread works on some portion of the array without interacting with the rest. Even then, how do you assign parts of the array to the different threads?

We will assume that each element of the array needs to be processed in some way. For example, we might want to record whether or not each `long` in an array is prime or not. If you have k threads and an array of length n where n happens to be a multiple of k , then it's easy: Each thread gets exactly n/k items to work on. For example, the first thread will work on indexes 0 through $\frac{n}{k} - 1$, the second thread will work on indexes $\frac{n}{k}$ through $\frac{2n}{k} - 1$, and so on, with the last thread working on indexes $\frac{(k-1)n}{k}$ through $n - 1$. Not every element in the array will require the same amount of computation, but we often assume that they do because it can be difficult to guess which elements will take more time to process.

What if the number of elements in the array is not a multiple of the number of threads? We still want to assign the work the work as fairly as possible. New programmers are sometimes tempted to use the same arithmetic from the case in which the threads evenly divide the length of the array: Each thread gets $\frac{n}{k}$ (using integer division) elements, and we stick the last thread with the leftovers. How bad can that be?

Exercise 6.7

This assignment of work can be very poorly balanced. Consider a case with 10 threads and 28 pieces of data. $\frac{28}{10} = 2$, using integer division. Thus, the first nine threads have 2 units of work to do,

but the last thread is stuck with 10! Not only is this unfair, it is inefficient. The person writing the program probably wants to minimize the total amount of time needed to finish the job. In this case, the time from when the first thread starts to when the last thread finishes is called the task's *makespan*. With this division of work, the makespan is 10 units of work.

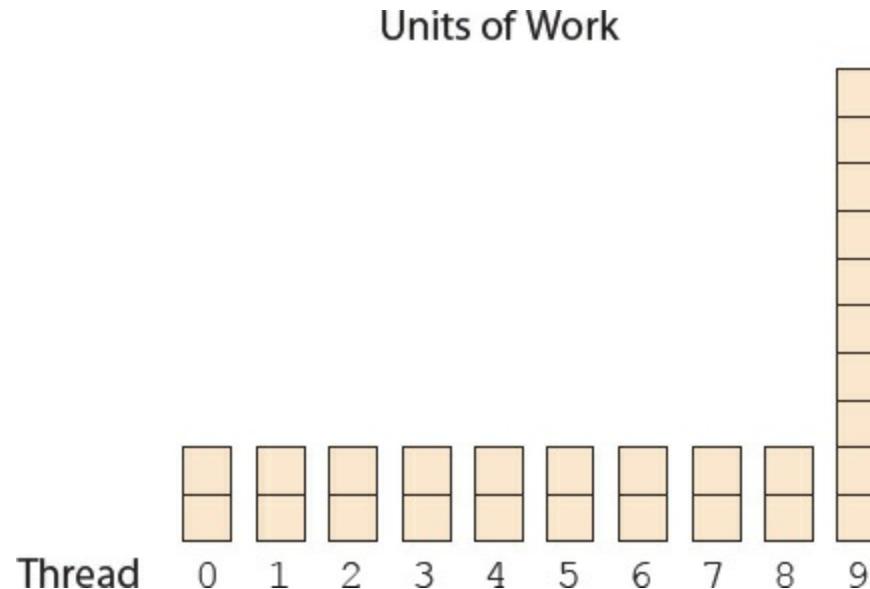


Figure 6.3: Units of work per thread using a naive strategy.

Example 6.8: Fairly assigning work

A simple way to fix this problem is to look at the value $n \bmod k$, the leftovers when you divide n by k . We want to spread those out over the first few threads. We know that any remainder will be smaller than k . If the index of the thread (starting at 0, of course) is less than the remainder, we add an extra element to its work. In this way, 28 units of work spread over 10 threads will give 3 elements to the first 8 threads and 2 elements to the rest. Using this strategy, the makespan becomes 3 units of work, a huge improvement over 10. Finding a way to spreading work across multiple threads to improve efficiency is a form of *load balancing*, a broad term for dividing work across computing resources.

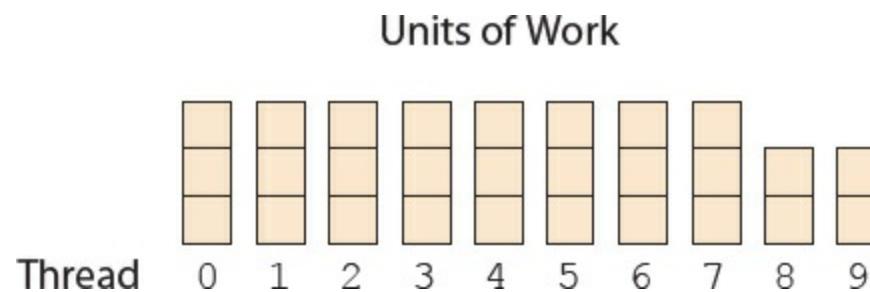


Figure 6.4: Units of work per thread using a fair strategy.

Program 6.5: Here is a short program that reads the length of an array and the number of threads from the user and then prints out the amount of work for each one. You should be able to adapt the ideas in it to your own multi-threaded programs in [Chapter 13](#). (`AssigningWork.java`)

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

```

2
3 public class AssigningWork {
4     public static void main ( String [] args ) {
5         Scanner in = new Scanner ( System.in );
6         System.out.print ( " How long is your array ? " );
7         int n = in.nextInt ();
8         System.out.print ( " How many threads do you have ? " );
9         int k = in.nextInt ();
10        int quotient = n / k;
11        int remainder = n % k;
12        int next = 0;
13        for ( int i = 0; i < k; i++ ) {
14            int work = quotient ;
15            if( i < remainder )
16                work++;
17            System.out.println ( " Thread " + i + " does " + work
18                            + " units of work , starting at index " + next
19                            + " and ending at index " + ( next + work - 1));
20            next += work ;
21        }
22    }
23 }
```



Exercises

Conceptual Problems

- 6.1 Why can't an array be used to hold an arbitrarily long list of numbers entered by a user? What are strategies that can be used to overcome this problem?
- 6.2 In future chapters, we will introduce a data structure called a *linked list*. A linked list is a homogeneous, dynamic data structure, with sequential access (unlike an array, which has random access). You can instantly jump to any place in an array, but you have to step through each element of a linked list to get to the one you want, even if you know its position in the list exactly. On the other hand, inserting values into the beginning of a linked list can be done in one step, while an array would need to be resized and have its contents copied over. List some tasks for which an array would be better than a linked list and vice versa.

- 6.3 Given the following code:

```

double [] array1 = new double [50];
double [] array2 = new double [50];
for ( int i = 0; i < array1.length ; i++ ) {
    array1 [i] = i + 1;
```

```
array2 [i] = array2.length - i;  
}  
array2 = array1 ;  
for ( int i = 1; i < array1.length ; i++ )  
    array1 [i] = array1 [i - 1] + array1 [i];
```

What is the value in `array2[array2.length - 1]` after this code is executed?

6.4 What error will be caused by the following code, and why?

```
String [] array = new String [100];  
System.out.println ( array [99].charAt (0) +  
    " is the first letter of the last String.");
```

6.5 An array of length n in Java typically takes n times the number of bytes for each element plus an additional 16 bytes of overhead. Since an `int` uses 4 bytes of storage, an array of 100 `int` elements would take 416 bytes. Consider the following three-dimensional array declaration and allocation.

```
int [[[ ] data = new int [10][5][20];
```

How many bytes are allocated for this array? Remember that the 16 byte overhead will occur repeatedly, since Java creates a three-dimensional array as an array of arrays of arrays.

6.6 Our original table of city distances allocates $5 \cdot 5 = 25$ `int` elements to store all the distances between the five cities, including repeats. How many `int` elements are allocated for the triangular, ragged array version of the city distance table? If we used the normal table style, n cities would require n^2 `int` elements. How many elements would the triangular, ragged array version allocate for n cities?

6.7 Consider the naive method of dividing an array of length n among k threads that was discussed in [Section 6.11](#): Each thread gets n/k (rounded down because of integer division) elements, and the last thread gets any extras. What mathematical expression describes how many extra elements are allocated to the last thread? Can you come up with an example in which the last element gets **all** the elements? What should have happened in this case using the other, more fair scheme for assigning the data to threads?

Concurrency

Programming Practice

6.8 In [Example 6.3](#), our code did not count ship, as an occurrence of ship because of the comma.

Rewrite the code from [Example 6.3](#) to remove punctuation from the beginning and end of a word. Use a loop that runs as long as the character at the beginning of a word is not a letter, replacing the word with a substring of itself that does not include the first character. Use a second loop to remove non-letters from the end of a word. Be careful to stop if the length of the String becomes 0, as with text that is entirely composed of non-letters.

6.9 In [Example 6.3](#), we wrote a program that counts the occurrences of each word from a list within a text. If the list of words to search within is long, it can take quite some time to search through the entire list. If the list of words were sorted, we could do a trick that would allow us to search much faster. We could play a “high-low” game, searching through the list by checking the middle word in the array. If that word is too late in the alphabet, repeat the search on the first half of the list. If it is too early in the alphabet, repeat the search on the second half of the list. By repeatedly dividing the list in half, until you either find the word you’re looking for or narrow your search down to a single incorrect word, you can search much faster. This kind of searching is called *binary search* and uses around $\log n$ comparisons to find an element in a list. In contrast, looking through the list one element at a time takes about n comparisons.

Rewrite the code from [Example 6.3](#) to use binary search, after applying selection sort from [Example 6.2](#). Although selection sort will take some extra time, you should more than make up the difference with such a fast search. To implement binary search, keep variables for the start, middle, and end of the list. Keep adjusting the three variables until the middle index has the word you are looking for or start and end reach each other. Remember to use the `compareTo()` method from the `String` class to compare words.

6.10 In [Example 6.4](#), we gave a program that finds the maximum, minimum, mean, standard deviation, and median of a list of values. Another statistic that is sometimes important is the *mode*, or most commonly occurring element. For example, in the list $\{1,2,3,3,3,5,6,6,10\}$, the mode is 3. Write a program that can determine the mode of a given list of `int` values. A list can have multiple modes if more than one element occurs with maximum frequency. For our purposes, we will consider any list with multiple modes to have no modes. You may wish to sort the list before starting the process of counting the frequency of each value.

6.11 We used the example of Tic Tac Toe in [Example 6.7](#) because a more complex game would have taken too much space to solve. The game of Connect Four (or the Captain’s Mistress, as it was originally called) pits two players against each other on a 6×7 vertical board. One player uses red checkers while the other uses black. The two players take turns dropping their checkers into columns of the board in which the checkers will drop to the lowest empty row, due to gravity. The goal of the game is to be the first to make four in a row of your color.

Implement a version of Connect Four for two human players, similar to the version of Tic Tac Toe we created. Many of the ideas are the same, but the details are more complicated. First, a player will only choose a particular column. Your program must then find which row a checker dropped into that column will fall to. Then, the process of counting four in a row is more difficult than the three in a row of Tic Tac Toe. You will need more loops to automate the process fully.

GUI 6.12 Once you have mastered the material in [Chapter 15](#), adapt the solution to Conway’s Game of Life from [Section 6.10](#) to display on a graphical user interface. You can use a `GridLayout` to arrange a large number of `JLabel` objects in a grid and update their background colors to `Color.BLACK` and `Color.WHITE` as needed, using the `setBackground()` method. (To make these colors visible, you will also need to call the `setOpaque()` method once on each `JLabel` with an argument of `true`.) The Game of Life is much more compelling with a real GUI instead.

of an improvised command line representation.

Experiments

6.13 Creating arrays with longer and longer lengths requires more processor time, since all of those elements must be initialized to some default value. Using an OS time command, determine the amount of time it takes to create an `int` array of length 10, 10,000, and 10,000,000. In all likelihood, the amount of time that instantiation of the array takes is a small part of the program, and you should see very little difference in those three times. However, time is not the only important resource. When you run a JVM, it has a default heap size that limits the amount of space you can use to create new objects, including arrays. When you exceed this size, your program will crash with an `OutOfMemoryError`. Experiment with different sizes of arrays until you can estimate the size of your heap within 5MB or so. This estimate will be very rough, since the JVM uses other memory in the background. For a more accurate picture, you can use the

`Runtime.getRuntime().maxMemory()` method to determine the maximum JVM memory size and the

`Runtime.getRuntime().totalMemory()` method to determine the total JVM memory being used.

6.14 Run the implementation of the word search program using the binary search improvement from Exercise 6.9. Use the OS time command to time the difference between the regular and binary search versions of the program with a long list of words. You may see very little difference on small input, but you can easily find a list of the 1,000 most commonly used words in English on the Internet along with long, copyright free texts from Project Gutenberg (<http://www.gutenberg.org/>). Combining these two into a single input should see a significant increase in speed for the binary search version relative to the regular version.

6.15 Generate input files consisting of 1,000, 10,000, and 100,000 random `int` values. Time our implementation of selection sort from [Example 6.2](#) running on each of these input files and redirecting output to output files. What is the behavior of the running time as the input length increases by a factor of 10? As a function of n , how many times does the body of the inner `for` loop run during selection sort? Does this function closely parallel the increase in running time?

Chapter 7

Simple Graphical User Interfaces

To a true artist only that face is beautiful which, quite apart from its exterior, shines with the truth within the soul.

—Mahatma Gandhi

7.1 Problem: Codon extractor

Recall from [Chapter 5](#) that we can record DNA as a sequence of nucleotide bases A, C, G, and T. Using this idea, we can represent any sequence of DNA using a String made up of those four letters such as “[ATGGAAGTATTAAATAG](#)”.

This particular sequence contains 18 bases and six *codons*. A codon is a three-base subsequence in DNA. Biologists are interested in dividing DNA into codons because a single codon usually maps to the production of a specific amino acid. Amino acids, in turn, are the building blocks of proteins. The DNA sequence above contains the six codons ATG, GAA, GTA, TTT, AAA, and TAG.

We want to write a program that extracts codons in order from DNA sequences entered by the user. The program must detect and inform the user of invalid DNA sequences (those containing letters other than the four bases). If the user enters a DNA sequence whose length is not a multiple of three, the final codon should be written with one or two asterisks (*), representing the missing bases.

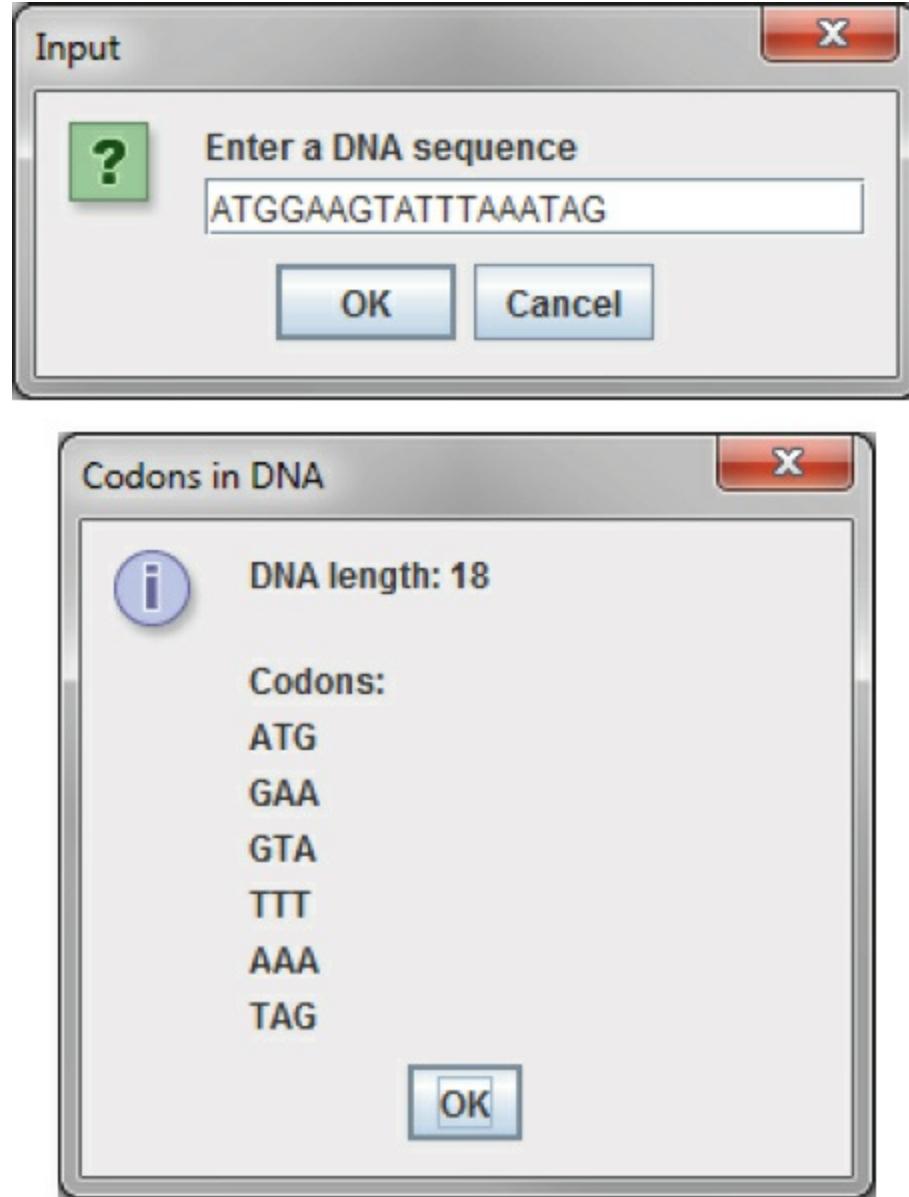
With your knowledge of String manipulation and loops, this problem should be easy. However, we want to solve it with a *graphical user interface*, not with the command line inter-action we have emphasized in previous chapters. That is, the input step should be done with a window that looks similar to the following.

And the corresponding output should look very much like this.

7.2 Concepts: User interaction

Many computer programs communicate with a human user. There are at least two ways in which this communication can happen. One way is to use command line input and output. In this case, a program prompts the user for an input and the user responds through the keyboard, usually completing the response by pressing the <return> or <enter> key. Another way to communicate is to use a graphical user interface or GUI. (Some people pronounce “GUI” to sound like “gooey,” but others say “G-U-I.”) In this case, the program displays a window consisting of one or more *widgets*, such as a button labeled “OK” or a text box in which the user can type some text. Widgets (also known as controls) can include buttons, labels, text areas, check boxes, menus, and many other pre-defined objects for user interaction. While the program waits for the user or does something in the background, the user has the option of using a combination of the keyboard and the mouse to respond to the program. While command line interfaces were dominant until the mid-70s, GUIs have become the prime mode of

communication between a program and a human user. This chapter focuses on the design of simple GUIs using a few built-in Java classes. [Chapter 15](#) introduces more advanced tools for constructing complex GUIs.



Example 7.1

[Figure 7.1\(a\)](#) shows a Java application interacting with a user through a command line interface. The application asks the user for a temperature value in degrees Fahrenheit, converts it to the equivalent Celsius, displays it, and prompts the user to enter another value. [Figure 7.1\(b\)](#) shows a similar application interacting with the user through a GUI. In this case, the application creates a window with five widgets (two text boxes and three buttons). The user enters a temperature value in the text box below either the Centigrade label or the Fahrenheit label and presses the appropriate Convert button. In turn the application displays the equivalent temperature in the other text box. ■

We describe the GUIs we will introduce in this chapter as simple because several aspects of GUI creation are hidden by the methods we will use. For example, these GUIs do not require the programmer to handle the details of events such as a user pressing an “OK” button or typing text into a text box and pressing the <enter> key. These events will be handled automatically by the tools we will introduce in this chapter. [Chapter 15](#) discusses the creation of more complex GUIs that require

the programmer to program event handling explicitly.

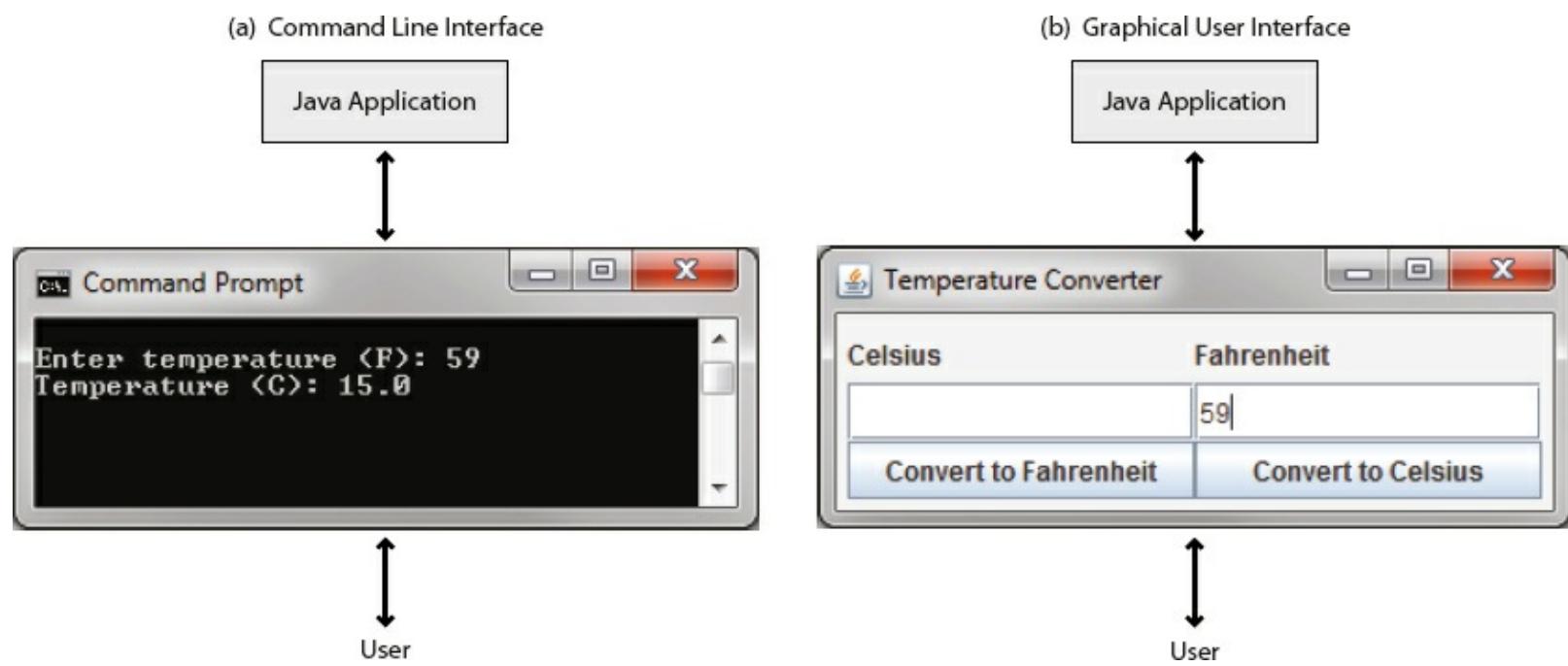


Figure 7.1: User interaction with a Java application (a) through a command line interface and (b) through a graphical user interface.

7.3 Syntax: Dialogs and the JOptionPane class

JOptionPane is a utility class for creating GUIs consisting most often of a single dialog. It offers a variety of ways to create useful dialogs quickly and easily and is part of the larger Java Swing GUI library. In this chapter, we will show you how to use the static methods and constants in JOptionPane to construct useful dialogs. Specifically, you will learn how to construct the following four types of dialogs.

Information: An information, or message, dialog displays a message to the user. Static method `showMessageDialog()` creates such a dialog. See [Figure 7.2](#) for an example of a message dialog.

Confirm: A confirm dialog asks a user to confirm a statement. Static method `showConfirmDialog()` creates such a dialog. This dialog may return user input as YES_OPTION, NO_OPTION, OK_OPTION, or CANCEL_OPTION. [See Figure 7.4](#) for an example of a Yes-No dialog.

Option: An option dialog asks the user to select one from an arbitrary set of options. Static method `showOptionDialog()` creates such a dialog. See [Figure 7.5](#) for an example.

Input: An input dialog is useful for obtaining data provided by the user. Static method `showInputDialog()` creates such a dialog. The user can input a String that might represent a number, a name, or any arbitrary piece of text. See [Figure 7.7](#) for an example.

The JOptionPane class can be used to create both *modal* and *non-modal* dialogs. A modal dialog is one that forces the user to interact with the dialog before the program can continue. Thus, the dialog is dismissed and the program execution resumes only after the user has responded. Modal dialogs are useful in situations where user input is required for the program to proceed further.

Exercise 7.8

A non-modal dialog is one that is displayed on the screen and does not require the user to interact with it for the underlying program to proceed. It is easy to create a modal dialog using the static methods in the JOptionPane class mentioned earlier. Creation of non-modal dialogs requires a bit more effort and is not covered in this chapter. In the remainder of this chapter we show how to use JOptionPane to create various types of modal dialogs.

Exercise 7.2

7.3.1 Generating an information dialog

Often programs need to generate a message for the user and request a response. The message might be a short piece of information, and the only response might be “OK.” The message might be more complex and require a more thoughtful response. In this section, we show how the Java utility class JOptionPane can generate a simple dialog whose sole purpose is to inform the user that a task has been completed. The next example shows how to generate such a dialog.

Example 7.2

[Program 7.1](#) creates a dialog to inform the user that the task it was assigned to perform is now complete. [Figure 7.2](#) shows the dialog generated by this program.

Program 7.1: Program to generate a simple dialog. (SimpleDialog.java)

```
1 import javax.swing.*;  
2  
3 public class SimpleDialog {  
4     public static void main ( String [] args ) {  
5         JOptionPane.showMessageDialog ( null,  
6             " Task completed. Click OK to exit ",  
7             " Simple Dialog ", JOptionPane.INFORMATION_MESSAGE );  
8         System.out.println ( " Done. " );  
9     }  
10 }
```

Let’s dissect [Program 7.1](#). Line 1 is needed to import classes used in this program. The swing package contains a number of classes needed to create a GUI, and JOptionPane is one such class. If you try to compile the program without line 1, it will fail.

In [Figure 7.2](#) the dialog titled “Simple Dialog” includes an icon, a message and a button labeled “OK.” This dialog is actually a *frame*, which is what windows are called in Java. We will discuss frames in greater detail in [Section 15.3](#).

Note that the appearance of the dialog may be different on your computer. Even though Java is platform independent, GUIs are customized based on the OS you are running. Each OS has a default *look and feel* (L & F) manager that specifies how widgets look and behave in your program. You can change the L&F manager, but not all managers are available on all operating systems.

[Line 5](#) and the next two lines use a static method to create a modal dialog. JOptionPane is a utility class and showMessageDialog() is a static method in this class. This method, along with the other three JOptionPane methods we discuss in this chapter is a *factory method*, meaning that it creates a new object (in this case some kind of dialog object) on the fly with specific attributes. In this example, the program is informing the user that a task has been completed. The method has the following four parameters.



Figure 7.2: A simple dialog generated using JOptionPane.

Component: The parent component in which the dialog is displayed. We use `null` in this example, which causes a default frame to be used, centering the dialog in the screen.

Message: The message to be displayed. In this example, we have “Task completed. Click OK to exit.”

Title: The title string used to decorate the dialog. In this example, it is “A Simple Dialog”.

Message Type: The type of the message to be displayed. In this example, we use the constant `INFORMATION_MESSAGE`.

Icon: The icon to be displayed in the dialog. If you have an object of type Icon, you can use it to customize your dialog. The `showMessageDialog()` is an overloaded method that can take several different sets of parameters. In this example, we used a version of the method that does not specify an icon.

Line 8 displays a message on the console which is not needed in this program but illustrates an interesting point. When you run SimpleDialog, you will notice that the “Done.” message displays on the console only after you have clicked the “OK” button in the dialog box. This is the modal behavior we mentioned earlier. The dialog blocks execution of the thread that generated it. ■

In the above example, we have displayed a message of type INFORMATION_MESSAGE. There are additional message types that could be used.

- ERROR_MESSAGE
- PLAIN_MESSAGE
- QUESTION_MESSAGE
- WARNING_MESSAGE

When used as parameters in showMessageDialog(), the constants above cause different default icons to be displayed in the dialog box. Figure 7.3 shows dialogs generated by showMessageDialog() when using JOptionPane.ERROR_MESSAGE, (left) and JOptionPane.WARNING_MESSAGE (right). Note the difference in the icons displayed towards the top left of the two dialogs.



Figure 7.3: Two dialogs generated using the showMessageDialog() method. The left dialog uses JOptionPane.ERROR_MESSAGE, and the right uses JOptionPane.WARNING_MESSAGE. The only difference is the icon displayed.

7.3.2 Generating a Yes-No confirm dialog

There are situations when a program needs to obtain a binary answer from the user, a “yes” or a “no.” The next example shows how to generate such a dialog and how to get the user’s response.

Example 7.3: Yes-No dialog

Consider a program that checks whether a student understands the difference between odd and even integers. The program generates a random integer x , presents it to the user, and asks the question, “Is x an odd integer?” The answer given by the user is checked for correctness, and the user is informed accordingly. Program 7.2 shows how to use the JOptionPane class to generate a dialog for such an interaction.

Program 7.2: Program that tests knowledge of odd and even integers with a Yes-No dialog. (OddEvenTest.java)

```
1 import javax.swing.*;
2 import java.util.*;
3
4 public class OddEvenTest {
5     public static void main ( String [] args ) {
6         Random random = new Random ();
7         String title = " Odd Even Test ";
8         int x = random.nextInt (10) ; // Random int from 0 to 9
9         String question = " Is " + x + " an odd integer ?";
10        int response = JOptionPane.showConfirmDialog (null,
11                                         question, title, JOptionPane.YES_NO_OPTION );
12        String message ;
13        // Response is YES_OPTION for Yes, NO_OPTION for No.
14        if (( response == JOptionPane.YES_OPTION && x%2 != 0 ) ||
15            ( response == JOptionPane.NO_OPTION && x%2 == 0 ))
16            message = " You 're right !";
17        else
18            message = "Sorry, that 's incorrect.";
19        JOptionPane.showMessageDialog (null, message, title,
20                                     JOptionPane.INFORMATION_MESSAGE );
21    }
22 }
```

Program 7.2 begins by declaring a random number generator named random. It then generates a random number and presents it to the user in a dialog created at line 10. Note the use of JOptionPane.YES_NO_OPTION as the last parameter in the showConfirmDialog() method at line 10. The generated dialog is shown in Figure 7.4(a). A second dialog is shown with a message dependent on whether the user gives the correct answer. The two different versions of this dialog are shown in Figure 7.4(b) and (c). Note that a call to showConfirmDialog() at line 10 returns the JOptionPane.YES_OPTION or the JOptionPane.NO_OPTION value depending on whether the user clicked the “Yes” or “No” button. ■

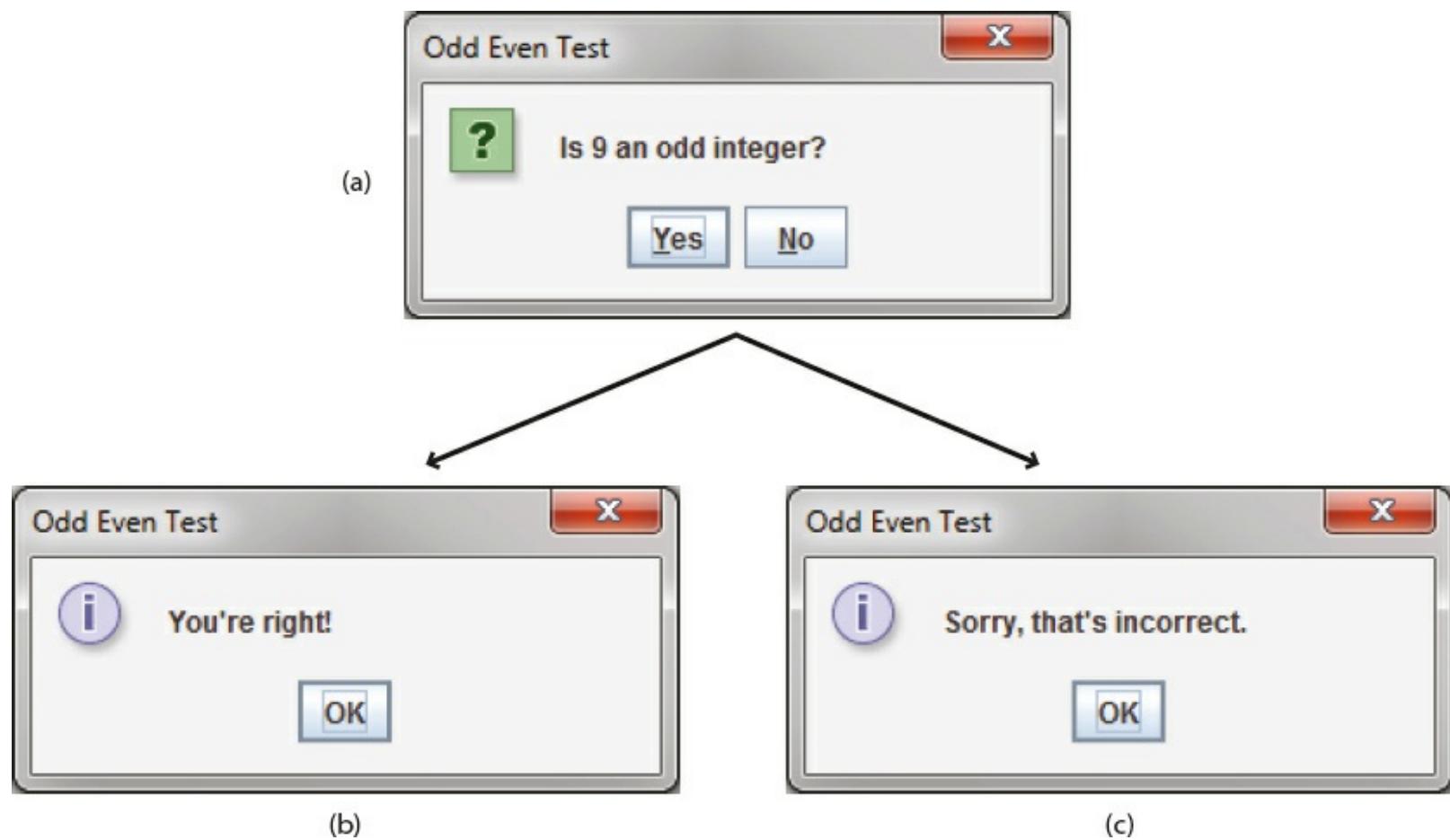


Figure 7.4: (a) A Yes-No dialog generated using JOptionPane. (b) Dialog in response to correct answer. (c) Dialog in response to incorrect answer.

Because we used YES_NO_OPTION, the dialog in Example 7.3 automatically generates two buttons labeled “Yes” and “No.” Dialogs can also use the YES_NO_CANCEL_OPTION to generate a dialog with “Yes,” “No,” and “Cancel” options. The return value from showConfirmDialog() is CANCEL_OPTION if the user presses the “Cancel” button.

Exercise 7.7

7.3.3 Generating a dialog with a list of options

The JOptionPane class can also be used to generate an arbitrary set of options as shown in the next example.

Example 7.4

Consider a program that asks the user to select the correct capital of a country from a given list of capitals. It shows three options and asks the user to select one from among the three. It then checks the user response for correctness and displays a suitable message. Program 7.3 performs these tasks. In this program, we call the showOptionDialog() method at line 10 to create a dialog with multiple options. In our case, the options are three names of capitals, and only one of them is correct. Figure 7.5 shows the dialog created.

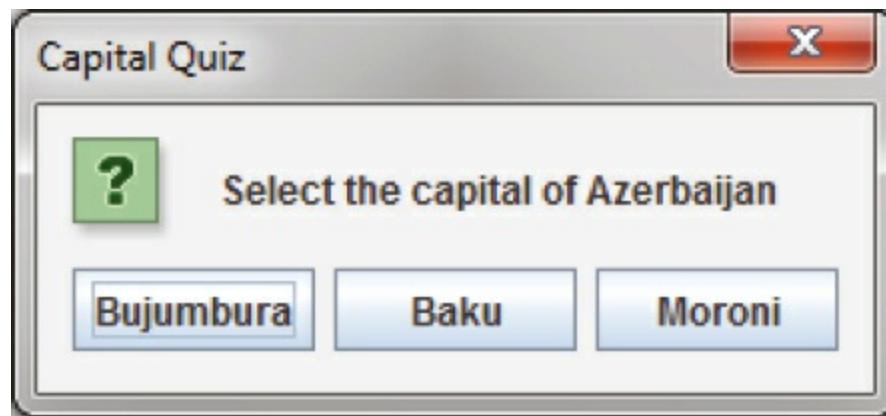
Exercise 7.3

Exercise 7.4

Program 7.3: Program to generate a dialog with programmer-defined options. (CapitalQuiz.java)

```

1 import javax.swing.*;
2
3 public class CapitalQuiz {
4     public static void main ( String [] args ) {
5         String title = " Capital Quiz ";
6         String country = " Azerbaijan ";
7         String [] capitals = {" Bujumbura "," Baku ", " Moroni "};
8         int correct = 1; // Baku is the correct answer
9         String question = " Select the capital of " + country ;
10        int response = JOptionPane.showOptionDialog (null,
11                question, title, JOptionPane.PLAIN_MESSAGE,
12                JOptionPane.QUESTION_MESSAGE, null, capitals, null );
13        // Response is 0, 1, or 2 for the three options
14        String message ;
15        if( response == correct )
16            message = " You 're right !";
17        else
18            message = "Sorry, the capital of " + country +
19            " is " + capitals [ correct ];
20        JOptionPane . showMessageDialog (null, message, title,
21                JOptionPane . INFORMATION_MESSAGE );
22    }
23 }
```

Figure 7.5: A dialog with programmer-defined options generated by [Program 7.3](#).

The `showOptionDialog()` method creates an options dialog, which is the most complicated (but also the most flexible) of all the dialogs. The array of `String` values provided as the second to last parameter to `showOptionDialog()` gives the labels for the buttons.

There are three `null` values passed into the method on line 10 in [Program 7.3](#). The first one

functions like the `null` used in [Program 7.2](#), specifying that the default frame should be used. The second specifies that the default icon should be used. In the next section, we will show how to specify a custom icon. The last parameter indicates the default button, which will have focus when the dialog is created. If the user hits <enter> instead of clicking, the button with focus is the button that will be pressed. ■

Exercise 7.9

7.3.4 Generating a dialog with a custom icon

A custom icon can be included in any dialog. Each of the methods in `JOptionPane` introduced earlier can take an icon as a parameter. The next example illustrates how to do so.

Example 7.5

[Program 7.4](#) shows how to use `showMessageDialog()` to generate a message dialog with a custom icon. Note the last parameter at [line 9](#). This parameter creates a new `ImageIcon` object from the file String ("bat.png" in this case). The resulting dialog appears in [Figure 7.6](#). Dialogs illustrated in earlier examples can also use an icon parameter to include a custom icon.

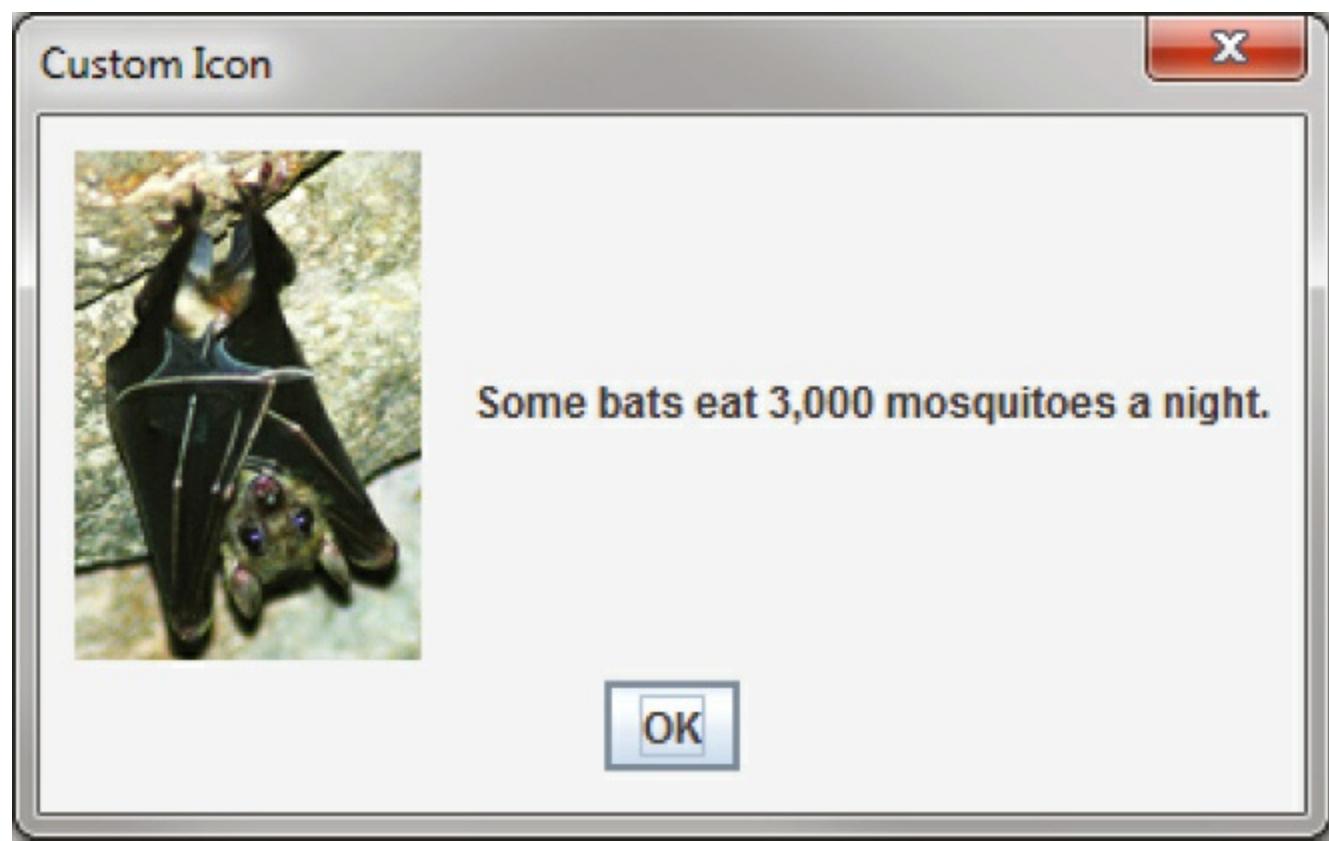


Figure 7.6: A dialog with a custom icon generated by [Program 7.4](#).

[Program 7.4](#): Program to generate a dialog with a custom icon. (`CustomIconDialog.java`)

```
1 import javax.swing.*;  
2  
3 public class CustomIconDialog {  
4     public static void main ( String [] args ){
```

```

5     String file = "bat.png";
6     String title = "Custom Icon";
7     String message = " Some bats eat 3 ,000 mosquitoes a night.";
8     JOptionPane.showMessageDialog (null, message, title,
9         JOptionPane.INFORMATION_MESSAGE, new ImageIcon ( file ));
10    }
11 }

```

Note that the icon shown above will not appear when you run this code unless you have a copy of bat.png in the same directory. ■

7.3.5 Generating an input dialog

An input dialog can read text data from the user. The showInputDialog() method in the JOptionPane class allows us to create such a dialog. We introduced the showInputDialog() method in [Section 2.3](#), but we give two more examples here to emphasize its similarity to the other JOptionPane factory methods and to show off some of its additional features.

Example 7.6: Input dialog

We want to write a program that asks a question about basic chemistry. [Program 7.5](#) shows how to display a question, obtain an answer from the user, check for the correctness of the answer, and report back to the user. At [line 7](#), the showInputDialog() method is used to generate the dialog shown in [Figure 7.7\(a\)](#). This method returns a String named response containing the text entered by the user in the dialog box. At [line 9](#), this String is converted to an int and saved into variable answer. This value is checked against the correct answer, and the showMessageDialog() method informs the user whether or not the answer is correct.

It is important to note that the user could type any sequence of characters in the dialog box. Try running [Program 7.5](#) and see what happens when you type “two,” instead of the number “2,” into the dialog box and press the “OK” button. The program will generate an exception indicating that the input String cannot be converted to an integer. Exercise 7.15 asks you to modify [Program 7.5](#) so it gracefully handles such exceptions.

Exercise 7.15

Exercise 7.15

Program 7.5: Program to generate a dialog to input data as text. (ChemistryQuizOne.java)

```

1 import javax.swing.*;
2
3 public class ChemistryQuizOne {
4     public static void main ( String [] args ) {
5         String title = " Atoms in Water ";
6         String query = " How many atoms are in a molecule of water ?"
7         ;
8         String response = JOptionPane.showInputDialog (null,

```

```
8     query, title, JOptionPane.QUESTION_MESSAGE );
9     int answer = Integer.parseInt ( response );
10    String message ;
11    if( answer == 3 )
12        message = " Good ! That 's correct !";
13    else
14        message = "Sorry, that 's incorrect.";
15    JOptionPane.showMessageDialog ( null, message, title,
16                                  JOptionPane.INFORMATION_MESSAGE );
17 }
18 }
```

Example 7.7: Input dialog with a list

In [Example 7.6](#) the user is required to enter a single value. To reduce input errors, we can restrict the user to picking from a predefined list. We can create this list by generating an array and supplying it as a parameter to the showInputDialog() method.

[Program 7.6](#) displays a list of chemical elements and asks the user to select the heaviest. [Line 9](#) passes an array of four String values to the showInputDialog() method. Note that the last parameter to this method is [null](#) indicating that no specific item on the list should be selected by default. (In this case, the first item in the list is initially selected.) The generated dialog is shown in [Figure 7.7\(b\)](#). The four elements are contained in a drop down list.

Atoms in Water



How many atoms are in a molecule of water?

3

OK

Cancel

(a)

Heaviest Element



Which is the heaviest element?

Iron



OK

Cancel

(b)

Figure 7.7: (a) A dialog requesting input as text. (b) A dialog requesting a selection from a list of choices. These dialogs are generated by Programs 7.5 and 7.6, respectively.

Program 7.6: Program to generate a dialog to input a choice from a list. (ChemistryQuizTwo.java)

```
1 import javax.swing.*;  
2  
3 public class ChemistryQuizTwo {  
4     public static void main ( String [] args ) {  
5         String title = " Heaviest Element ";  
6         String query = " Which is the heaviest element ?";  
7         String [] elements = {" Iron ", " Uranium ", " Copernicium ",  
8             " Nitrogen "};  
9         String response = ( String ) JOptionPane.showInputDialog ( null,  
10             query, title, JOptionPane.QUESTION_MESSAGE, null,
```

```
11     elements, null );
12
13     String message ;
14
15     if( response.equals (" Copernicium ") )
16         message = " You 're right !";
17     else
18         message = "Sorry, correct answer : Copernicium .";
19
20     JOptionPane.showMessageDialog (null, message, title,
21                                     JOptionPane.INFORMATION_MESSAGE );
22 }
```

Unlike [Example 7.6](#), the return value from `showInputDialog()` is now of type `Object`, not of type `String`. The type of the list required by the method is `Object` array. (You can pass a `String` array to a method that wants an `Object` array due to inheritance, which is further discussed in [Chapters 11](#) and [17](#).) The return value is the specific object from the array that was passed in. In our case, it **has** to be a `String`, but Java is not smart enough to figure that out. For this reason, we cast the object to a `String` before using the `equals()` method. ■

Example 7.8: Input dialog with a long list

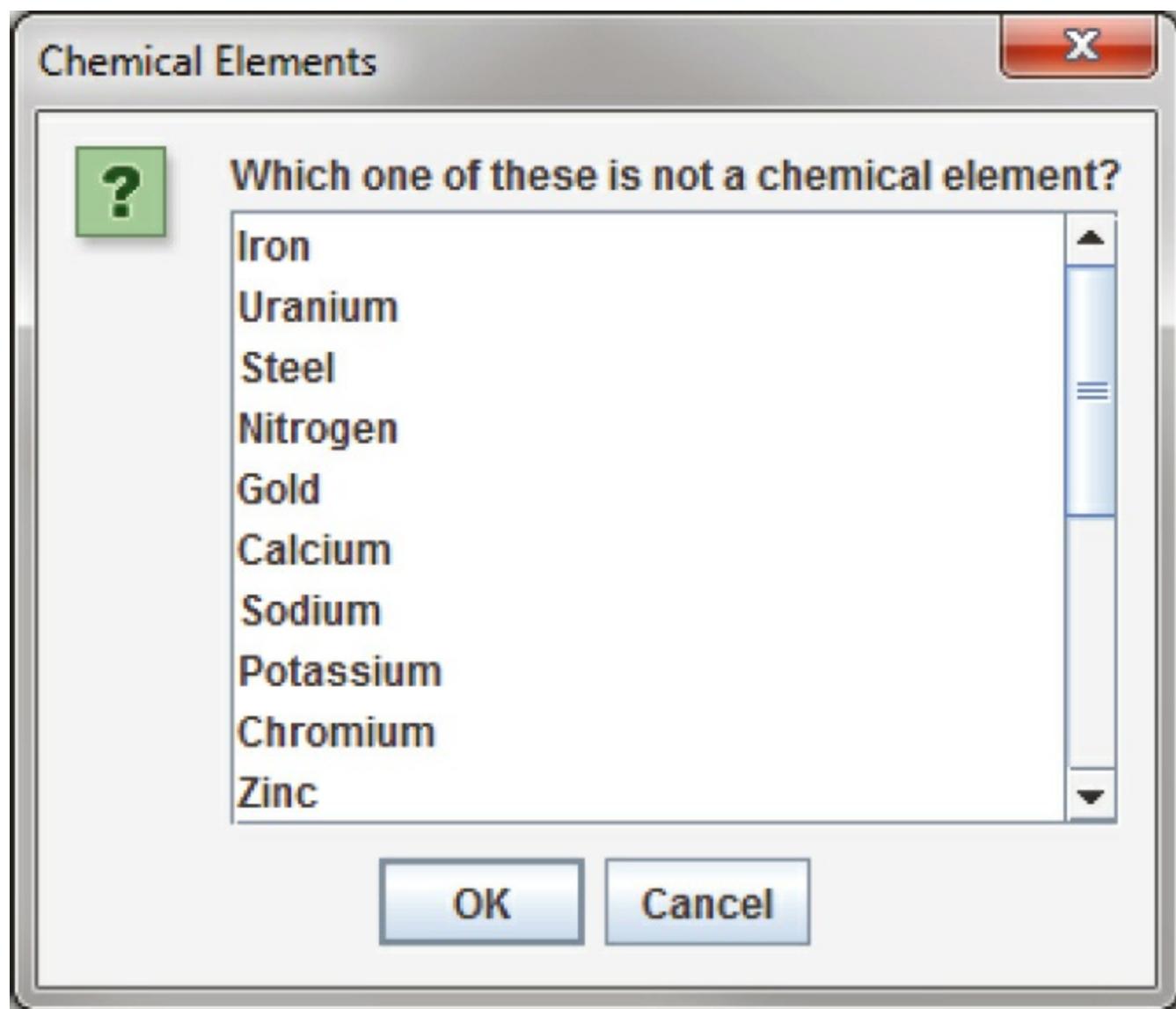


Figure 7.8: A dialog requesting user to select one item from a list of 20 data items. Note the use of a scroll down list to display the items.

When the number of elements in the list supplied to the showInputDialog() is 20 or more, a JList object is automatically used to display the items as shown in [Figure 7.8](#).

Other than a longer list, the code in this example is virtually identical to the code for [Example 7.7](#). ■

7.4 Solution: Codon extractor

Here we give the solution to the codon extractor problem posed at the beginning of the chapter. As we have done throughout this chapter, we start with the import needed for GUIs built on the Swing framework. Next we begin the CodonExtractor class and its main() method. For readability, the solution to this problem is divided into methods that each do a specific task. We hope that the way a method works is intuitively clear to you. If not, the next chapter explains them in detail.

```
1 import javax.swing.*;
2
3 public class CodonExtractor {
4     public static void main ( String [] args ) {
5         int continueProgram = JOptionPane.YES_OPTION ;
6         do {
7             // Read DNA sequence
8             String input = JOptionPane.showInputDialog (
9                 " Enter a DNA sequence ");
10            input = input.toUpperCase () // Make upper case
11            String message = "Do you want to continue ?";
12            if( isValid ( input ) ) // Check for validity
13                displayCodons ( input ); // Find codons
14            else
15                message = " Invalid DNA Sequence.\n" + message ;
16            continueProgram = JOptionPane.showConfirmDialog (
17                null, message, " Alert ", JOptionPane.YES_NO_OPTION );
18        } while ( continueProgram == JOptionPane.YES_OPTION );
19        JOptionPane.showMessageDialog (null,
20            " Thanks for using the Codon Extractor !");
21    }
```

The main() method contains a **do-while** loop that allows the user to enter sequences repeatedly. The showInputDialog() method makes an input dialog and returns the String the user enters. The toUpperCase() method converts the String to upper case, allowing us to read input in either case.

We then call the isValid() method to make sure that the user entered a valid DNA sequence. If it is valid, we use displayCodons() to display the codons in the sequence. Either way, we use a

showConfirmDialog() method to creating a confirm dialog, asking the user if he or she wants to continue entering sequences. The loop will continue as long as the return value is JOptionPane.YES_OPTION.

```
23 public static boolean isValid ( String DNA ) {  
24     String validBases = " ACGT ";  
25     for ( int i = 0; i < DNA. length (); i ++ ) {  
26         char base = DNA. charAt ( i );  
27         if( validBases.indexOf ( base ) == -1 )  
28             return false ; // base not in " ACGT "  
29     }  
30     return true ;  
31 }
```

The isValid() method checks to see if the DNA contains only the letters representing the four bases. To do this, we use the Java String library cleverly: We loop through the characters in our input, checking to see where they can be found in “ACGT”. If the index returned is -1, the character was not found, and the DNA is invalid.

```
33 public static void displayCodons ( String DNA ) {  
34     String message = "";  
35     // Get as many complete codons as possible  
36     for ( int i = 0; i < DNA. length () - 2; i += 3 )  
37         message += "\n" + DNA.substring ( i, i + 3 );  
38     // 1-2 bases might be left over  
39     int remaining = DNA.length () % 3;  
40     if( remaining == 1 )  
41         message += "\n" + DNA.substring ( DNA.length () - 1,  
42                                         DNA.length () ) + "***";  
43     else if( remaining == 2 )  
44         message += "\n" + DNA.substring ( DNA.length () - 2,  
45                                         DNA.length () ) + "*";  
46     message = "DNA length : " + DNA.length () +  
47             "\n\nCodons : " + message ;  
48     JOptionPane.showMessageDialog ( null, message,  
49                                     " Codons in DNA ", JOptionPane.INFORMATION_MESSAGE );  
50 }  
51 }
```

In the displayCodons() method, we display the individual codons to the user. We build a large String with newlines separating each codon. To do so, we loop through the input, jumping ahead three characters each time. If the input length is not a multiple of three, we pad with asterisks. Finally, we use the showMessageDialog() method to display an information dialog with the list of codons.

7.5 Concurrency: Simple GUIs

Many GUI frameworks (including Swing) are built on a multi-threaded model. Swing uses threads to redraw widgets and listen for user input while the main thread can continue processing other data.

In this chapter, the impact of these threads is minimal because we used only **modal** dialogs. Every time we called a JOptionPane method, the execution of the program's main thread had to wait until the method returned. As it turns out, several threads are created when showInputDialog() or any of the others dialog methods are called, but they do not interact with the main thread since it has been blocked.

The situation is more complicated with a non-modal dialog, which is one of the reasons we did not go into them. In a non-modal dialog, the threads that redraw the dialog and handle its events (like a user clicking on a button) are running at the same time as the thread that created the dialog. Since many threads are running, it is possible for them to write to the same data at the same time. Doing so can lead to inconsistencies such as the ones we will describe in [Chapter 14](#).

The GUIs we will create in [Chapter 15](#), however, will be more than dialogs. They will be fully functional windows, known as frames in Java. Like a non-modal dialog, the creation of a frame does not block the thread that created it.

Many applications launch a frame and then end their main thread. If no other threads are created, life is relatively easy. However, complex applications may create multiple frames or launch threads to work on tasks in the background. Another common problem is caused by performing complicated tasks in the event handler for a GUI. If a task takes too long, the GUI can freeze or become unresponsive, as you have probably experienced. The fact that this problem happens so frequently even in the latest operating systems should hint at the difficulty of managing GUI threads.

When we describe how to create fully featured GUIs in [Chapter 15](#), we will also give some techniques to help with avoiding unresponsive GUIs in a multi-threaded environment.

7.6 Summary

In this chapter we have introduced a way to create simple GUIs. These GUIs are created using various methods available in the JOptionPane class. While the interfaces created this way are simple in nature, they are often adequate for input and output in short Java programs. Construction of more complex GUIs is the subject of [Chapter 15](#).

Exercises

Conceptual Problems

- 7.1 In which situations would it be better to use a command-line interface instead of a GUI? When is it better to use a GUI over a command-line interface?
- 7.2 Explain the difference between a modal and a non-modal dialog. Give an example of when you would prefer a modal over non-modal dialog, and another example of when you would prefer a

non-modal to a modal dialog.

- 7.3 Give one example each when you would use the five different message type constants in `showMessageDialog()` method (see [page 206](#) for a listing of the five constants).
- 7.4 In [Program 7.2](#), we could have coded line 10 as follows without changing the program behavior.

```
if( ( response == 0 && x % 2 != 0 ) ||  
    ( response == 1 && x % 2 == 0 ) )
```

Yet another option is below.

```
if( response != x % 2 )
```

Which of these three implementations is best? Why?

Programming Practice

- 7.5 Modify the program in [Example 7.3](#) such that it tests the user several times, say 25 times, whether a randomly generated integer is odd or even. The program should keep a score indicating the number of correct answers. At the end of the test the score is displayed using a suitable dialog.
- 7.6 Modify the program in [Example 7.3](#) such that it displays a dialog that asks the user “Do you wish to continue?” and offers options “Yes” and “No.” The program exits the loop when the “No” option is selected and displays the score using a suitable dialog.
- 7.7 Rewrite [Program 7.2](#) so that the confirmatory dialog generated offers the “Yes,” “No,” and “Cancel” options to the user. The program exits with a message dialog saying “Thank You” when the user selects the “Cancel” option.
- 7.8 Modify [Program 7.3](#) to create and administer a test wherein the user is asked capitals of 10 countries in a sequence. The program must keep count of the score, i.e., the number of correct answers. Inform the user of the score at the end of the test using a suitable dialog.
- 7.9 Modify line 10 in [Program 7.3](#) so that the button labeled “Baku” has focus.
- 7.10 [Section 8.5](#) gives a method called `shuffle()` that is used to randomize an array representing a deck of cards. Adapt this code and modify [Program 7.3](#) so that the order of the capitals is randomized. Note that you will have to record which index contains the correct answer.
- 7.11 Re-implement the solution to the college cost calculator problem given in [Section 3.5](#) so that it uses GUIs constructed with `JOptionPane` for input and output.
- 7.12 Re-implement the solution to the Monty Hall problem given in [Section 4.4](#) so that it uses GUIs constructed with `JOptionPane` for input and output.
- 7.13 Re-implement the solution to the DNA searching problem given in [Section 5.4](#) so that it uses GUIs constructed with `JOptionPane` for input and output.
- 7.14 Write a program that creates an input dialog that prompts and reads a file name of an image

from the user. Then, create an information dialog that displays the file as a custom icon. In this way, you can construct a simple image viewer.

7.15 Note: You should attempt this exercise only if you are familiar with exceptions in Java. Exceptions are covered in [Chapter 12](#).

Use the **try-catch** block and modify [Program 7.5](#) so that it handles an exception generated when the user enters a string that cannot be converted to an integer. In the event such an exception is raised, pop up a message dialog box informing the user to try again and type an integer value. When the user responds by clicking the “OK” button on this message box, the input dialog box should appear once again and offer the user another chance at the answer. Write two versions of the modified program. In one version, your program should give only one chance for input after an incorrect string has been typed. In another version, your program should remain in a loop until the user enters a valid integer (note that a valid integer might not be the correct answer to the question asked).

Chapter 8

Methods

Polonius (Aside): Though this be madness, yet there is method in 't.

—William Shakespeare

8.1 Problem: Three card poker

Gambling has held a fascination for humankind since its invention. As long as there have been mathematicians, they have studied the underlying mechanisms of probability and statistics that drive games of chance. A classic game of both statistics and strategy is poker. The problem we want to solve is programming one of the many variations of poker, called three card poker. Instead of bluffing, a player competes only with the house according to fixed rules without any room for psychology. A player is dealt three cards. If the player's hand contains a pair or better, he or she wins a payoff greater than or equal to the money bet. If the player does not have a pair or better, he or she loses the money bet. Below is a table giving one possible set of payoffs for each possible hand.

Hand	Payoff
Straight Flush	40
Three of a Kind	30
Straight	6
Flush	2
Pair	1
Nothing	0

If you are unfamiliar with poker rules or card games in general, here is a quick explanation of the various hands. A traditional English or American deck of cards is made up of 52 cards, organized into four suits: spades, hearts, diamonds, and clubs. In each suit, there are 13 ranks: two, three, four, five, six, seven, eight, nine, ten, jack, queen, king, and ace.

In three card poker, a straight is when the three cards can be arranged so that their ranks (ignoring suit) are in order. For example, a hand consisting of a Four of Diamonds, a Five of Spades, and a Six of Clubs would constitute a straight. An ace can serve as either the highest rank card (coming after a king) or the lowest rank card (coming before a two) and can help form a straight in either role (but not both at the same time). A flush is when all three cards have the same suit. For example, a Three of Hearts, a Seven of Hearts, and a Jack of Hearts would constitute a flush in three card poker. A straight flush, the highest hand, is made up of cards that form both a straight (by rank ordering) and a flush (by uniformity of suit). A three of a kind occurs when all three cards have the same rank, and a pair occurs when two of the cards have the same rank. When none of these conditions hold, the hand has no special designation in three card poker, and the bet is lost.

Your task is to write a program that serves as a computerized version of the game. You must create

a deck of 52 cards, thoroughly randomize them to simulate shuffling, and select the first three cards as a hand. You must then determine the highest designation that applies to the hand of cards and the corresponding winnings (if any).

Any reasonable solution to this problem uses arrays, discussed in the previous chapter. Using only that knowledge, you should be able to create a three card poker game. However, the focus of this chapter is *methods*, which can be used to break the solution to a problem into logical pieces. By dividing the solution in this way, we will be able to solve the three card poker problem in a relatively short, elegant, and easy to read way.

8.2 Concepts: Dividing work into segments

You should notice that the solutions to problems we have been working on have become increasingly complex as the book progresses. Partly, this tendency is due to the fact that interesting problems tend to have complex solutions. Partly, it is due to the fact that more advanced features in Java are generally designed to tackle more complicated problems.

The tools we introduce in this chapter do not allow you to solve a problem you could not solve before. Instead, the tools in this chapter and in the next one make solving a problem easier and less susceptible to errors. A relatively long, complicated list of Java statements was necessary to solve problems like the statistics program or Conway's Game of Life. Instead of solving those problems as a single monolithic segment of code, we can break our solutions into units called *static methods*.

8.2.1 Reasons for static methods

A static method is a short, named segment of code that has been packaged up so that it can be *called* from other parts of the program. Whenever the task performed by this method is needed, the execution of the program jumps to the code in the method, does the work it is supposed to, and then returns to whatever it was doing before.

The reasons for using static methods can be boiled down to three essentials:

Modularity: Very little software development is done by individuals. Commercial software can have tens or even hundreds of programmers involved in designing, implementing, testing, and maintaining code. When the code is divided into individual methods, those methods can be written and tested by different people with minimal interference. It is much harder to work together on one giant block of code.

Readability: Methods are **named** segments of code. When a method is called, its name is used. If a meaningful name is chosen, the line in the code that calls the method helps document the code by explaining what is happening. For example, if a method called `sort()` is used, a reader instantly understands that something is being sorted. If, instead of a separate `sort()` method, the code for selection sort were pasted into the same location, a reader might have to devote a few moments to understanding the meaning of those lines of code, particularly without comments.

Reusability: The fact that a method can be called from anywhere in the code makes it reusable. To

use the `sort()` example again, we might have to sort several different arrays in one program. Without methods, we would have to keep copying and pasting code over and over again. With methods, the total amount of source code can be smaller. In fact, if you find yourself copying and pasting code, it is often an indication that a method would be useful.

This idea of reusability stretches beyond individual programs. A static method can be called from a different program entirely. If you create a method that is really useful, you are free to use it in other programs you may write. By doing so, you only have to make changes in one place if you discover errors or want to increase its functionality.

8.2.2 Parallel to mathematical functions

We have been discussing the usefulness of static methods without saying exactly what they are. One way to get insight into methods is by acknowledging their similarity to functions from mathematics. In procedural (non-object oriented) languages like C, the equivalent of static methods are usually referred to as functions, in fact. Like functions in mathematics, a static method takes some number of inputs (though, as few as zero) and usually produces some output.

You have already used several static methods from the `Math` class, such as `Math.sqrt()`. The expression $f(x) = \sqrt{x}$ defines what $f(x)$ does in the same way that the code inside of `Math.sqrt()` does. When you want to use $f(x)$ for a specific value, you can set $y = f(5)$ in the same way that you might type the following.

```
double y = Math.sqrt(5);
```

In the statement $f(5)$, the value of x is replaced by 5. In the same way, some variable inside of `Math.sqrt()` takes on the value 5 when the method is called.

It is important to understand this similarity between methods and mathematical functions because the designers of many programming languages have been directly influenced by the older notation. However, there are many differences between the two concepts as well. Mathematical functions can take more than one input, and so can Java methods. Methods can also take no input, while the whole point of a mathematical function is to map some input value(s) to some output value. Java methods do not necessarily even have output values. They may just **do** something.

8.2.3 Control flow

We have discussed selection statements such as `if` and `switch` as well as three different looping structures, `for`, `while`, and `do-while`. Each of these Java language features is used for control flow. The selection statements allow your program to choose which code to execute. The loops allow your code to repeatedly execute certain code. Methods also affect control flow. When a method is called, the execution of the program jumps into the method, does all the work it needs to there, and then returns to the code that called it. Recall the very simple example from above.

```
double y = Math.sqrt(5);
```

The JVM has allocated a variable of type `double` and is preparing to assign a value to it. Suddenly, the execution of the JVM jumps into the code inside of the `Math.sqrt()` method. It takes some non-

trivial amount of work to compute the square root of a number. After that computation is finished, the flow of execution returns to the assignment that has been waiting all that time. Just like a mathematical function, we can treat the method call `Math.sqrt(5)` as if it were “magically” replaced by a **double** approximating the square root of 5.

8.3 Syntax: Methods

By now, you should have a good feel for the concepts behind calling and even creating static methods and are probably getting impatient to use them. There are a number of issues of method syntax in Java you should be aware of. First, we describe how you can create your own static methods, then discuss the finer points of calling static methods, and finally explain how *class variables* can be used from many different methods.

8.3.1 Defining methods

A very simple method that the `Math` class provides is the `Math.max()` method. This method selects the larger of two values that you give it as input.

```
int maximum = Math.max (5, 10);
```

In this case, the value stored into `maximum` is 10. Despite its simplicity, we demonstrated how useful this method could be in our solution to Conway’s Game of Life from [Chapter 6](#). If we wanted to write this method ourselves, the code would be as follows.

```
public static int max ( int a, int b ) {  
    if( a >= b )  
        return a;  
    else  
        return b;  
}
```

Even in such a small method, there are a lot of pieces of syntax to worry about. The first line of this method is called the *method header*. The **public** keyword in this header is used to denote that any code, even code from a different class, can call this method. We discuss restricting access to methods and variables more in the later part of this chapter. For now, assume that every method is **public**.

The keyword **static** indicates that this method is static. Although we have used the term *static method* many times, we have not yet defined it. A static method is linked to a whole class, not to a specific object of that class—that is, a static method can be called without referencing an object of the class. Again, we discuss the finer points of objects and classes in the next chapter. For now, all methods are **static**.

The third keyword in the method header is the familiar **int**, giving the return type of the method. Wherever this method is called, it can be treated like an **int** value, because that is what it gives back. In this case, the return type is obvious: The maximum of two **int** values must also be an **int** value. Any type can be used as a return value including all the primitive types and any reference or array types.

The only limitation is that a method can only return a single item, but, since that item can be an array, this limitation is usually not important. It is also possible for a method to return nothing. In that case, the keyword **void** is used for the return type.

Next in the method header is the identifier `max`, which is the name of the method. Any legal identifier that you can use for a variable name is valid for a method name as well. It is important to pick a name that is readable and gives a reader a clear idea about what the method does. A common convention is to name a method using a verb phrase, indicating the operation that is being done by that method (e.g., `computeTax`). Like variable names, the Java standard is to use camel notation, starting with a lowercase letter and capitalizing the first letter of each new word in the name.

After the name of the method is the list of the *parameters*, separated by commas. In this case, each parameter has the `int` type. You are free to name your parameters whatever you want, though they should be meaningful. You can have as few as zero parameters, but there is an upper limit imposed by the JVM, usually 255. The body of the method follows the header of the method, surrounded by braces (`{ }`). Unlike `if` statements and loops, the braces for methods are required.

Inside the body of a method, the usual rules for Java control flow apply. Each line is executed line by line unless there are selection statements or loops. Calling methods inside of methods is allowed as well. In the `max()` method, we use an `if-else` construction to find the larger of `a` and `b`. A `return` statement immediately stops execution of the method, transfers execution back to the calling code, and gives back the value that comes after it. In this case, the value of `a` is returned if it is equal or larger, and the value of `b` is returned otherwise. Because a `return` statement immediately jumps out of a method, we could have written the method with one fewer line of code.

```
public static int max ( int a, int b ) {  
    if( a >= b )  
        return a;  
    return b;  
}
```

The only way that the line `return b;` can be reached is if `a` had not already been returned.

The `main()` method

If some of this syntax seems eerily familiar, remember that you have been coding static methods since your very first Java program. The `main()` method is just another static method, special only because the JVM chooses to start execution there. Let's look at the `main()` method from a standard Hello World! program.

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

Just like the `max()` method, the header for `main()` starts with `public static`. Then, the return type for `main()` is `void` because the JVM is not expecting to get any answer back. The `main()` method has a single parameter, an array of type `String`. In this program, we do not use the `args` parameter, but it is

available. For the `main()` method, this declaration is fine, because `main()` has to be uniform across all programs. However, when designing your own methods, you should not include unnecessary parameters.

The final executable line in this `main()` method is a `return` statement. Because `main()` has a `void` return type, the `return` statement has no value to return. For `void` methods, a `return` statement is optional. You can use it to leave a method early, if desired. For a value-returning method, execution must reach a `return` statement with a valid value no matter what the input of the method is. If Java finds a way that execution could reach the end of a value returning method without reaching a `return` statement, it causes a compiler error.

Overloaded methods

Since this declaration is in another class, it is OK to create a `max()` method even though there is already one in the `Math` class. However, it is possible to create more than one method with the same name in the same class, provided that their *signatures* are not the same. Two methods have the same *signature* if they have the same name and parameter types.

```
public static int max ( int a, int b, int c ) {  
    return max (max (a, b), c);  
}
```

In this example, we have created yet another `max()` method, but this one takes three parameters instead of two. This method even calls the two parameter version of `max()`. Creating more than one method with the same name is called *overloading* those methods. Overloading methods is useful because it allows you to use the same method name for similar functionality, even when there are some underlying differences in the implementation. For example, the `Math` class provides four different, overloaded versions of the `max()` method, specialized for `int`, `long`, `float`, and `double` values, respectively.

There are limitations on creating overloaded methods, of course. The compiler must be able to determine which method you intend to use. Thus, the signatures have to vary by type or number of parameters. A different return type is not enough.

8.3.2 Calling methods

After a method has been defined, it must be called before it does anything. You have plenty of experience calling static methods like `Math.sqrt()` and `Math.max()`. An example of the appropriate syntax was given earlier.

```
int maximum = Math.max (5, 10);
```

Formally, the call starts with the name of the class (`Math`), followed by a dot, followed by the name of the method (`max`), followed by the list of *arguments* inside parentheses. These arguments are the values you want to *pass* into the method. Some books use the term *formal parameters* to describe the variables defined in the method signature and *actual parameters* to describe the values passed into the methods, but we stick with the simpler terms *parameters* and *arguments*.

Of course, the number of arguments must match the number of parameters defined by the method, and the types must match as well. Java performs automatic casting when no precision is lost. Thus, you can always supply an `int` argument for a `double` parameter, but not the reverse. Arguments can be literal values, variables, or even other method calls that return the appropriate type.

Using the `max()` method defined before, we could rewrite our simple example without a class name.

```
int maximum = max(5, 10);
```

Whenever you call a static method from code that is inside the same class, you can leave out the class name.

Binding

Many new programmers are confused about the relationship between arguments and parameters. The process of supplying an argument to be used as a parameter is called *binding*. Through binding, a value or variable from the calling code is given a new name inside of a method. Consider the following method.

```
public static int add( int a, int b ) {  
    return a + b;  
}
```

This absurdly short method adds two numbers together and returns the result, approximating the functionality of the `+` operator. We could call the method in the following context.

```
int x = 3;  
int y = 5;  
int z = add( x, y );
```

Inside the method, the value of `x` is bound to the variable `a`, and the value of `y` is bound to the variable `b`. The `add()` method has its own *scope*. Scope means the area where a variable name is visible (or meaningful). Thus, `x` and `y` do not exist inside of the `add()` method, only the variables `a` and `b` do. Since methods have their own scope, variables in one method can have the same names as variables in another method without the compiler (or the programmer!) becoming confused. Consider the following example:

```
int a = 3;  
int b = 5;  
int c = add( b, a );
```

Here the variables `a` and `b` exist in both the calling code and inside the method, but the names are independent. The value of `a` in the calling code happens to be bound to a variable called `b` inside the method, but the JVM has no confusion about which `a` is which. Herein lies the value of methods: They are largely independent of whatever else is going on in the code, allowing the programmer to focus on a small, manageable task.

Another important feature of Java is that the process of binding variables is *pass by value*, meaning that only the **value** of the argument is bound to the parameter. Whenever a method is called, the method creates a new variable for each parameter and copies the value of its argument into it. In practice, this approach means that a method cannot directly change the value of an argument. Consider the following method:

```
public static void increment ( int counter ) {  
    counter ++;  
}
```

This method takes the value of its argument and copies it into the new variable counter. Then, it increments counter, but the original argument is unchanged. Thus, the following fragment is an infinite loop.

```
int i = 0;  
while ( i < 100 )  
    increment ( i );
```

The value of i remains fixed at 0 for the entire program. The copy of i bound to counter increases to 1 every time increment() is called, but i remains unaffected.

This is not to say that a method cannot affect the variables outside of itself. The primary way that it can do so is by using **return** statements. We can rewrite increment() to achieve this effect.

```
public static int increment ( int counter ) {  
    counter ++;  
    return counter ;  
}
```

Then, we need to adjust the loop so that it stores the returned value instead of dropping it on the floor.

```
int i = 0;  
while ( i < 100 )  
    i = increment ( i );
```

A second way that methods can affect the values of outside variables is more indirect. In Java, every argument is passed by value, even arrays and objects. Practically, this means that, if a reference to an array is passed into a method, you cannot change which array it is pointing at. Since references are not values but names pointing at a particular location in memory, you can directly change the contents of that memory with a method, even if you can't change which locations are being referenced. For example, the following method does **not** reverse the order of an array.

```
public static void badReverseArray ( int [ ] array ) {  
    int [ ] temp = new int [ array.length ];  
    for ( int i = 0; i < array.length ; i++ )  
        temp [i] = array [ array.length - i - 1];
```

```
array = temp ;
```

```
}
```

Although this code does store a reversed version of array in temp, the last line of the method is meaningless: The array passed into the method still points to the original location in memory. We can rewrite the method to do the reversal *in place*, meaning that the values of the array are shuffled around, but the array still occupies the same memory locations.

```
public static void goodReverseArray ( int [] array ) {  
    int temp ;  
    for ( int i = 0; i < array.length / 2; i++ ) {  
        temp = array [i];  
        array [i] = array [ array.length - i - 1];  
        array [ array.length - i - 1] = temp ;  
    }  
}
```

In this version of the method, we swap the first element of the array with the last, the second with the second to last, and so on. We only go up to the halfway point of the array, otherwise we undo the reversal process. The values of the array are reversed, but they still occupy the same chunk of memory. It is possible to write a correct method more in the style of badReverseArray() which creates a temporary array, copies the original values into it, and then copies them back to the original array in reverse order, but it is less efficient to create the extra array and perform two copies.

8.3.3 Class variables

According to the rules we have given so far, the only legal variables in the scope of a static method are the parameters and any other *local* variables declared inside the method. However, it is possible to create a variable that exists outside of static methods yet is visible inside all of them. These kinds of variables are called *class variables* (or sometimes *static fields* or *global variables*). These variables persist **between** method calls. The syntax for creating such a variable is to declare it outside of all methods (but inside the class) with an access modifier such as **public** or **private**, and the keyword **static**. For example, the following class includes a method called record() that increases the class variable counter every time it is called.

Program 8.1: A program that keeps track of the number of times the record() method is called.
(Bookkeeper.java)

```
1 public class Bookkeeper {  
2     public static int counter = 0;  
3  
4     public static void main ( String [] args ) {  
5         while ( Math.random () > 0.001 )  
6             record ();  
7  
8         System.out.println (" Record was called " + counter +
```

```

9      " times.");
10     }
11
12     public static void record() {
13         counter++;
14     }
15 }
```

When run, this program calls the record() method some random number of times, and the variable counter keeps track of the number. Because both main() and record() are static methods, the value of counter is accessible to each of them. Many programmers frown on the use of class variables precisely because they are visible to many different methods. The idea of a method is to isolate pieces of code so that the complexity of a program can be divided into simple units. In the case of a public class variable, even code in other classes can modify its value. So many different pieces of code can modify the value that it may be difficult to keep the variable from being changed in an unexpected way. If another method used the counter variable to record the number of times it was called, the final value of counter would be the sum of the number of times the two methods were called. There might be some reason to keep track of such information, but it would be impossible to reconstruct what fraction of the value in counter came from one method and what fraction came from the other.

Class variables have their uses, but they should generally be avoided. One exception to this rule is constants. Since a constant never changes, a class variable is a great place to store it, making the value available to any code that uses it. An example you have already used is Math.PI. As with static methods, a static field from another class can be accessed by using the class name, then a dot, then the name of the static field. Again, when the code using the field is in the same class, the class name can be dropped. A class constant is declared like a class variable, but with the addition of the **final** keyword. The following class allows a user to compute the one-dimensional force due to gravity, given by the equation $F = \frac{Gm_1m_2}{r^2}$, where m_1 is the mass of one object, m_2 is the mass of another, r is the distance between their centers, and G is the gravitational constant, $6.673 \times 10^{-11} \text{ N} \cdot \text{m}^2 \cdot \text{kg}^{-2}$.

Program 8.2: A program that calculates the attraction due to gravity between two masses.
(Gravity.java)

```

1 public class Gravity {
2     public static final double G = 6.673e -11;
3
4     public static void main ( String [] args ) {
5         double m1, m2, r;
6         Scanner in = new Scanner ( System.in );
7         System.out.println (" What is the first mass ?");
8         m1 = in. nextDouble ();
9         System.out.println (" What is the second mass ?");
10        m2 = in. nextDouble ();
11        System.out.println (" What is the distance between them ?");
```

```

12     r = in. nextDouble ();
13     System.out.println ("The force of gravity is " +
14         force ( m1, m2, r ) + " N");
15 }
16
17 public static double force ( double m1, double m2, double r ) {
18     return G*m1*m2 /(r*r);
19 }
20 }
```

Use named constants this way whenever you can. You can use the **public** modifier if you want all classes to have access to your constant (Gravity.G is a good example). You can use the **private** modifier if you want the constant to be accessible only inside your class, if it has no use outside, or if it contains secret information.

8.4 Examples: Defining methods

Any large problem should be broken down into methods. Because the technique is useful in so many circumstances, it is difficult to give a set of examples that covers all the bases. Instead, our examples are short, easy to understand methods, focusing on Euclidean distance, testing for palindromes, and converting a String representation of an **int** to an **int**.

Example 8.1: Euclidean distance

The Euclidean distance between two points is the length of a straight line connecting them. It plays an important role in 3D graphics and games and is the basis for many other practical applications involving spatial relationships. Unfortunately, the real world is complicated enough that, even if the shortest distance between two points is a straight line, we can seldom travel along it.

Given two points in 3D space (x_1, y_1, z_1) and (x_2, y_2, Z_2) , we can compute the Euclidean distance between them with the equation:

$$d = \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2 + (z_1 - z_2)^2}$$

The following method applies this equation directly.

```

public static double distance ( double x1, double y1, double z1,
    double x2, double y2, double z2 ) {
    double x = x1 - x2;
    double y = y1 - y2;
    double z = z1 - z2;
    return Math.sqrt ( x*x + y*y + z*z );
}
```

This calculation is a good choice for a static method since it might be necessary to do this

calculation many times and it does not depend on any other variables or program state. ■

Example 8.2: Palindrome testing

A palindrome is a word or phrase (or even a number) that is the same spelled forwards and backwards. “Racecar,” “Madam, I’m Adam,” and “Satan, oscillate my metallic sonatas” are examples in English. Typically, spaces and punctuation are ignored. We are going to write a function that, given a String, returns **true** if it is a palindrome and **false** otherwise. To simplify the problem, we are **not** going to ignore spaces and punctuation. Thus, with our method, “racecar” counts as a palindrome, but neither of the other two examples would.

Exercise 8.11

```
public static boolean isPalindrome ( String text ) {  
    text = text.toLowerCase ();  
    for ( int i = 0; i < text.length () / 2; i++ )  
        if( text.charAt ( i ) != text.charAt ( text.length () - i - 1 ))  
            return false ;  
    return true ;  
}
```

Because our method returns **true** or **false**, its return type must be **boolean**. Many methods that return a **boolean** value have a name starting with **is**, like our method. The first line of the body of our method changes text to lower case. The String method **toLowerCase()** creates a lower case copy of the String it is called on, in this case **text**. Then, we point the reference variable **text** at that new, lower case String. On the outside of this function, the String passed in does not change because the name **text** is passed by value.

The loop iterates through the first half of **text**, comparing it to the second half. This loop reflects the asymmetry of these kinds of tests: You can’t be sure that **text** is a palindrome until you have checked the entire thing, but you immediately know that it is not if even a single pair of characters does not match. If the test in that **if** statement ever shows that the two **char** values are not equal, the **return false** statement jumps out of the method without completing the loop. ■

Example 8.3: Parsing a number

When you read in a number using an object of the **Scanner** class, it converts (or *parses*) the text entered by the user into the appropriate type. For example, the **nextDouble()** method reads in some text and convert it into a **double**. When you use a **JOptionPane** method to read in input, it comes in as a String. If you want to use that data as a **double**, you must convert it using the **Double.parseDouble()** static method. Some Java programmer had to write this method. We are going to recreate a similar method to convert the String representation of a floating point number into a **double**. Our simple method ignores scientific notation.

```
public static double parseDouble ( String value ) {  
    int i = 0;  
    boolean negative = false ;
```

```

double temp = 0.0;
double fraction = 10;
if( value.charAt (i) == '-' ) {
    negative = true ;
    i++;
}
else if( value.charAt (i) == '+' )
    i++;
while ( i < value.length () && value.charAt (i) != '.' ) {
    temp *= 10;
    temp += value.charAt (i) - '0';
    i++;
}
i++; // move past decimal point (if there )
while ( i < value.length () ) {
    temp += ( value.charAt (i) - '0' ) / fraction ;
    fraction *= 10;
    i++;
}
if( negative )
    temp = -temp ;
return temp ;
}

```

After declaring a few variables, this method first checks index 0 in the input String value to see if it is a `'-'` or a `'+'`. If it is a `'-'`, it sets negative to `true` and moves on. If it is a `'+'`, it simply moves on. Then, the method loops through value until it reaches the end or reaches a decimal point. As it iterates, it multiplies the current value of temp by 10 and adds in the next digit from value (after subtracting `'0'` so that the range is from 0 to 9). This repetitive multiplication by 10 accounts for the increasing powers of 10 in the base 10 number system. Since temp starts with a value of 0.0, the first multiplication has no effect, as intended.

After the first `while` loop, the index i is incremented once, to skip the decimal point, if there is one. If there is no decimal point, the loop must have exited because the end of value had been reached. The second `while` loop runs to the end of value, this time adding in each digit value divided by fraction, which is increased by a factor of 10 each time. Doing so allows us to add smaller and smaller fractional digits to the total. We set temp to its opposite if the flag negative was set earlier and finally return temp.

You should note that the real `Double.parseDouble()` method not only accepts String values in scientific notation but also does a great deal of error checking. Our code either crashes or gives inaccurate results on an empty String, a String containing non-numerical characters, or a String with more than one decimal point. Furthermore, this code does not use the best approach for minimizing floating-point precision errors. ■

8.5 Solution: Three card poker

Here we present our solution to the three card poker problem. We explain each method individually.

```
1 public class ThreeCardPoker {  
2     public static final String [] SUITS = {" Spades ", " Hearts ",  
3         " Diamonds ", " Clubs "};  
4     public static final String [] RANKS = {"2", "3", "4", "5", "6",  
5         "7", "8", "9", "10", " Jack ", " Queen ", " King ", " Ace "};  
6     public static final int STRAIGHT_FLUSH = 40;  
7     public static final int THREE_OF_A_KIND = 30;  
8     public static final int STRAIGHT = 6;  
9     public static final int FLUSH = 2;  
10    public static final int PAIR = 1;  
11    public static final int NOTHING = 0;
```

Before our `main()` method even begins, we have declared a number of class constants. Two constant arrays of `String` values provide us with an easy way to represent suits and ranks. The remaining six `int` constants are used to allocate a winning payoff to each possible outcome. Note that these constants can be declared anywhere inside the class, provided that they are outside of all methods. However, it is typical (and good style) to declare them at the top of the class.

```
13    public static void main ( String [] args ) {  
14        int [] deck = new int [52];  
15        int [] hand = new int [3];  
16        for ( int i = 0; i < deck.length ; i++ )  
17            deck [i] = i;  
18        shuffle ( deck );  
19        for ( int i = 0; i < hand.length ; i++ )  
20            hand [i] = deck [i];  
21        int winnings = score ( hand );  
22        System.out.println (" Hand : ");  
23        print ( hand );  
24        if( winnings == 0 )  
25            System.out.println (" You win nothing. ");  
26        else  
27            System.out.println (" You win " + winnings +  
28                " times your bet. ");  
29    }
```

In the `main()` method, an array representing a deck of 52 cards is created first, followed by an array representing the 3 cards to be dealt. The deck is filled sequentially and then shuffled with a method. Next, the first 3 cards of the deck are copied into the array representing the hand of cards. The score of the hand is determined, and then the hand is printed out. We print the hand after determining the score because the hand is sorted in the process of determining the score, making the output easier to

read. Finally, we print the appropriate output, depending on the score.

```
31  public static void shuffle ( int [] deck ) {  
32      int index, temp ;  
33      for ( int i = 0; i < deck.length ; i++ ) {  
34          index = i + ( int )( ( deck.length - i ) * Math.random () );  
35          temp = deck [ index ];  
36          deck [ index ] = deck [i];  
37          deck [i] = temp ;  
38      }  
39  }
```

This method shuffles the deck. Its approach is to swap the first element in the array of cards with one of the elements that follow, chosen randomly. Then, it swaps the second element in the array with any of the elements that follow it, and so on. If `Math.random()` truly gives us a uniformly generated random number in the range $[0,1)$, the final shuffled deck should be any one of the $52!$ possible decks with equal probability.

```
41  public static void print ( int [] hand ) {  
42      for ( int i = 0; i < hand.length ; i++ )  
43          System.out.println ( RANKS [ getRank ( hand [i]) ] + " of "  
44          + SUITS [ getSuit ( hand [i]) ]);  
45  }  
46  
47  public static int getRank ( int value ) { return value % 13; }  
48  public static int getSuit ( int value ) { return value / 13; }
```

The first of these methods prints out a human readable version of each card in an array (instead of 0 - 51). It does so using the second and third methods as helper methods. Method `getRank()` computes the rank of a card from its number, and method `getSuit()` computes the suit of a card from its number. The indexes obtained from these methods are used to index into the `RANKS` and `SUITS` arrays.

In the C language, calling a method from a method defined earlier required a special declaration step called *prototyping* before both methods. Java does not have this complication, and the `getRank()` and `getSuit()` methods compile and function perfectly if they are written above `print()` or below it inside the class definition.

```
50  private static int score ( int [] hand ) {  
51      sortByRank ( hand );  
52      if( hasStraight ( hand ) && hasFlush ( hand ) )  
53          return STRAIGHT_FLUSH ;  
54      if( hasThree ( hand ) )  
55          return THREE_OF_A_KIND ;  
56      if( hasStraight ( hand ) )  
57          return STRAIGHT ;  
58      if( hasFlush ( hand ) )
```

```

59         return FLUSH ;
60     if( hasPair ( hand ) )
61         return PAIR ;
62     return NOTHING ;
63 }

```

This method computes the score by first sorting the hand and then testing progressively worse outcomes, starting with the best, a straight flush. As it moves down the list of outcomes, it calls appropriate methods to determine if a hand has a certain characteristic.

```

65     private static void sortByRank ( int [] hand ) {
66         int smallest, temp ;
67         for ( int i = 0; i < hand.length - 1; i++ ) {
68             smallest = i;
69             for ( int j = i + 1; j < hand.length ; j++ ) {
70                 if( getRank ( hand [j] ) < getRank ( hand [ smallest ] ) )
71                     smallest = j;
72             }
73             temp = hand [ smallest ];
74             hand [ smallest ] = hand [i];
75             hand [i] = temp ;
76         }
77     }

```

This code is an implementation of selection sort packaged into a method. Note that this method does actually change the values inside of the array hand even though it cannot change the array that hand points to. The array itself is passed by value, but its contents are effectively passed by reference.

```

79     private static boolean hasPair ( int [] hand ) {
80         return getRank ( hand [0] ) == getRank ( hand [1] ) ||
81             getRank ( hand [1] ) == getRank ( hand [2] );
82     }
83
84     private static boolean hasThree ( int [] hand ) {
85         return getRank ( hand [0] ) == getRank ( hand [1] ) &&
86             getRank ( hand [1] ) == getRank ( hand [2] );
87     }
88
89     private static boolean hasFlush ( int [] hand ) {
90         return getSuit ( hand [0] ) == getSuit ( hand [1] ) &&
91             getSuit ( hand [1] ) == getSuit ( hand [2] );
92     }
93
94     private static boolean hasStraight ( int [] hand ) {

```

```

95     return ( getRank ( hand [0] ) == 0 && getRank ( hand [1] ) == 1
96         && getRank ( hand [2] ) == 12 ) || // ace low
97     ( getRank ( hand [1] ) == getRank ( hand [0] ) + 1 &&
98     getRank ( hand [2] ) == getRank ( hand [1] ) + 1 );
99 }
100 }

```

These four methods do the actual work of determining the attributes of a hand. They are all similar and would be more complex for five- or seven-card poker hands. Methods `hasPair()` and `hasStraight()` depend on the array being sorted previously. The code in `hasPair()` works by checking to see if the first and second or second and third cards have the same rank. The code in `hasThree()` checks to see if all the ranks are the same. The code in `hasFlush()` is the same as `hasThree()` except that it checks for suit instead of rank. Finally, `hasStraight()` checks to see if the ranks are all in ascending order, with an extra case to deal with the possibility of the ace counting as low.

8.6 Concurrency: Methods

In Java, it is impossible to have concurrency without methods. Methods are the way we break a large program into manageable pieces but are also part of the syntax that Java uses to create threads of execution. Each thread of execution is associated with a `Thread` object, but creating the object is not enough to start a new thread of execution running. Only when the `start()` method is called on the `Thread` object does the new thread start running.

Hopefully, you have begun to visualize the execution of Java programs as an arrow that sits next to each line of code as it is executed. This arrow can jump to a choice and skip over other code using `if` and `switch` statements. Using loops, the arrow can jump backwards and repeatedly execute code it has just executed. As we have discussed in this chapter, it can jump into a method, execute the code in that method, and then return to its caller, going back right to where it left off before the call.

When the `start()` method is called on a `Thread` object, however, the arrow returns to the caller, but it also splits itself into a second arrow that then executes the corresponding `run()` method and any methods it calls. Note that we are talking about a method called on a `Thread` object, not a static method called on the class as a whole. Calling `start()` is an instance method, which we discuss in [Chapter 9](#). Unlike the static methods we have discussed in this chapter, an instance method is tied to a particular object, but most of what you have learned about methods still applies.

Methods are supposed to make programming easier by breaking programs into chunks small enough to think about. One of the only real dangers of methods is using class variables, as discussed in [Section 8.3.3](#). This problem becomes worse with multiple threads. With a single thread, two or more different methods can all affect the same class variable, perhaps in conflicting ways. With multiple threads, even the **same** method can interfere with itself.

Example 8.4: LCG without thread safety

A linear congruential generator (LCG) allows you to create a sequence of pseudorandom numbers using the equation $x_i = (ax_{i-1} + b) \bmod m$, deriving the next number from the previous one, and so on.

Program 8.3: This program implements an LCG similar to one sometimes used in the function rand() used in the C language. (UnsafeRandom.java)

```
1 public class UnsafeRandom {  
2     private static int next = 1;  
3     private final static int A = 1103515245;  
4     private final static int B = 12345;  
5     private final static int M = 32768;  
6  
7     public static int nextInt () {  
8         return next = (A* next + B) % M;  
9     }  
10 }
```

The UnsafeRandom program listed above always generates the same sequence of pseudorandom numbers, which can be very useful for debugging a program. However, if two or more threads are calling nextInt(), they probably are getting different sequences. One thread picks up some of the numbers, and the other picks up the missing numbers in between. If each thread wants to generate the same sequence of numbers, the method should be rewritten so that it takes in the previous number in the sequence. In that way, there is no shared state. Remember that using a (non-final) static field (class variable) should be avoided whenever possible.

Program 8.4: This program implements the same LCG safely by requiring the caller to supply the previous random number. (SafeRandom.java)

```
1 public class SafeRandom {  
2     private final static int A = 1103515245;  
3     private final static int B = 12345;  
4     private final static int M = 32768;  
5  
6     public static int nextInt ( int previous ) {  
7         return (A* previous + B) % M;  
8     }  
9 }
```



Example 8.5: Unpredictable methods

By forcing each thread to carry its own state, we fixed the previous problem. In [Chapter 14](#) we talk about the much nastier problem of two threads executing a method at exactly the same time. When that happens, very curious effects are possible. Consider the following program:

Program 8.5: The print() method always prints “Even” when run with a single thread but can sometimes print “Odd” if called repeatedly with multiple threads. (AlwaysEven.java)

```
1 public class AlwaysEven {
```

```
2     private static int value = 1;  
3  
4     public static void print() {  
5         value++;  
6         if( value % 2 == 0 )  
7             System.out.println (" Even ");  
8         else  
9             System.out.println (" Odd ");  
10        value++;  
11    }  
12 }
```

With a single thread running, value always goes up to an even number before printing and then increments to the next odd number afterwards. If two or more threads are calling the print() method, value could be changed by one right before the other executes the if statements. ■

Exercises

Conceptual Problems

- 8.1 Describe three advantages of dividing long segments of code into static methods.
- 8.2 Can you think of any disadvantages of dividing code into methods? Are there situations where using a method is unwise?
- 8.3 If you wanted to declare a static method that would compute the mean, median, and standard deviation of an input array of **double** values, how would you return those three answers?
- 8.4 Consider the following method definition.

```
public static void twice ( int i ) {  
    i = 2 * i;  
}
```

How many times does the following loop run, and why?

```
int x = 2;  
while ( x < 128 )  
    twice (x);
```

- 8.5 Consider the following signatures of two overloaded methods.

```
public static int magic ( int rabbit, double hat )  
public static int magic ( double wand, int spell )
```

Which method would be invoked by the following call?

```
int x = magic ( 3, 16 );
```

What about the following?

```
int y = magic( 3.2, 16.4 );
```

Use a compiler to check your answers.

- 8.6 The following class generates a sequence of even numbers. Each time the `next()` method is called, the next even number in the sequence is returned. What is the design problem with using a static field to keep track of the next value in the sequence?

```
public class EvenNumbers {  
    private static int counter = 0;  
    public static int next() {  
        counter += 2;  
        return counter;  
    }  
}
```

Programming Practice

- 8.7 Write a static method called `cube()` that takes a single `double` value as a parameter and returns its value cubed. Do not use the `Math.pow()` method.

- 8.8 Implement a static method that takes a single `int` value as a parameter and prints its digits in reverse. For example, if 103 was passed into this method, it would print 301 to the screen.

You can find out what digit is in the ones place of a number by taking its remainder modulus 10. Then, you can remove the digit in the ones place by dividing by 10. Do not convert the `int` value into a `String`.

- 8.9 Write a static method that takes an array of `int` values as a parameter and returns true if the array is in ascending order and false otherwise. Compare each element of the array to the next element of the array. If the current element is ever larger than the next element, the array is not sorted in ascending order. Note that you can only be sure that the array is in ascending order after you have checked all neighboring pairs.

- 8.10 Write a static method that finds the $\lfloor \log_2(n) \rfloor$ of an integer n . Note that if $\log_2 n = x$, it is also true that $n = 2^x$. In other words, the \log_2 operator tells you what power of 2 a number is. One way to define the $\log_2 n$ is the number of times you have to divide n by 2 to get 1. Use this definition to make a loop that finds the value without using any calls to the Math library.

Here are some examples of the return values your method should give for various input values of n .

<i>n</i>	Return Value
1	0
2	1
4	2

8	3
10	3
16	4
100	6
512	9
1000	9
1024	10

8.11 Write a method that tests palindromes like the method from [Example 8.2](#) but also ignores punctuation and spaces. Thus, “A man, a plan, a canal: Panama” should be counted as a palindrome by this new method.

GUI 8.12 Re-implement the solution from [Section 8.5](#) so that it uses a GUI constructed with JOptionPane to display the hand and the winnings.

8.13 Five card poker is a much more common version of poker than the three card version we discussed in [Section 8.1](#). Using static methods, implement a two-player game of poker in which the deck is shuffled and then dealt into two hands of five cards each. Then, state which player’s hand wins. With five cards, determining which hand wins is a more complicated process. The rankings of the various possible hands from best to worst are as follows.

Straight: All five cards belong to the same suit and have ranks in sequential order (with either ace high or low). If two people both have straight flushes, the higher ranked one wins. If they Flush: both have the same ranks, it is a tie.

Four of a Kind: Four of the five cards have the same rank. If two people have four of a kind, the higher rank a Kind: set of four wins.

Full House: Three of the five cards have the same rank and the other two share another rank. If two people have a full house, the higher ranked set of three wins.

Flush: All five cards have the same suit. If two people have flushes, the one with the highest card wins. If the highest card is a tie, the next highest is the tie breaker, and so on. If the two flushes have exactly the same ranks, the two flushes tie.

Straight: All five cards have ranks in sequential order (with either ace high or low). If two people both have straight flushes, the higher ranked one wins. If they both have the same ranks, it is a tie.

Three of a Kind: Three of the cards have the same rank. If two people have three of a kind, the higher ranked a Kind: set of three wins.

A pair of cards have the same rank and another pair of cards share another rank. If two people both have two pairs, the higher ranked pair is a tiebreaker. If the higher ranked pair Pair: is the same, the lower ranked pair is a tiebreaker. If the lower ranked pair is the same, the final unpaired card is the tiebreaker. If all the ranks of both hands match, it is a tie.

One Pair: A pair of cards has the same rank. If two people have pairs, the rank of the pair is a tiebreaker. If the pairs have the same rank, the remaining cards in each hand are tiebreakers, in descending rank order.

If none of the other cases hold, the high card determines the value of the hand. If two people

High have the same highest card, the remaining cards in each hand are used as tiebreakers, in Card: descending rank order.

Experiments

8.14 In terms of time, there is a small overhead associated with calling a method and returning a value, but it is very hard to measure. Write a program with two int variables, a and b, where a starts with a value of 1 and b starts with a value of 2. Run a for loop 100,000,000 times. On each iteration first increase the value of a by the value of b and then increase the value of b by a. Time this loop with System.nanoTime() and then print out the time taken and the value of a. The value of a is not important, but the compiler will optimize away the math done with a and b unless we output the value. We recommend that you run this program repeatedly to get a sense of the average running time. Now, instead of using the + operator to add a and b, use the following method.

```
public static int add ( int a, int b ) {  
    return a + b;  
}
```

Again, run your program repeatedly with this modification. What is the difference in running time between the version that uses a method and the version that does addition directly?

Depending on your JVM, it's quite possible that there's almost no difference. The JVM does a lot of optimizations including *inlining*, which replaces a call to a method with the actual code inside the method.

Chapter 9

Classes

Luminous beings are we, not this crude matter.

—Yoda

9.1 Problem: Nested expressions

How does the compiler check the Java code that you write and find errors? Syntax checking and type checking are involved processes that are key parts of compiler design. Compilers are some of the most complex programs of any kind, and building one is beyond the scope of the material covered in this book. However, we can get insight into some problems faced by compiler designers by considering the problem of correctly nested expressions.

There are many rules for forming correct Java code, but it is always the case that grouping symbols ((), [], {, }) are correctly nested. Ignoring other symbols, we may find a section of code that contains the sequence “() []”, but correctly written code never contains “([)]”.

To be correctly nested, left and right parentheses must be balanced, left and right square brackets must be balanced, and left and right curly braces must be balanced. Furthermore, a correctly balanced set of parentheses can be nested inside of a correctly balanced set of square brackets or curly braces (and vice versa), but they cannot intersect as they do at the end of the previous paragraph. [Table 9.1](#) shows more examples of correctly and incorrectly nested expressions.

But how can we examine an expression to see if it is correctly nested? The key to solving this problem is the idea of a *stack*. A stack is a simple data structure with three operations: *push*, *pop*, and *top*. The data structure is meant to behave like a stack of books or cups or any other physical objects. When you use the push operation, you are moving something to the top of the stack. When you use the pop operation, you are taking something off the top of the stack. The top operation is used to read what is at the top of the stack. In a stack, you can only read that very topmost item, as if all the other items were buried underneath it. A stack is known as a FILO (first in, last out) or a LIFO (last in, first out) data structure because, if you push a series of items onto the stack and then pop them all off the stack, they come off the stack in the reverse order from how they were added.

Correctly Nested	Incorrectly Nested
""	"("
"(){}"	
"((abc)){x}"	"(a { b) c }"
"({[z]})"	
"({xyz})({ijk}[123])"	

Table 9.1: Correctly and incorrectly nested expressions.

Armed with an understanding of the stack, it is straightforward to see if an expression is nested correctly. Scan through each character in the input and follow these steps.

1. If it is a left parenthesis, left square bracket, or left curly brace, put it on the stack.
2. If it is a right parenthesis, right square bracket, or right curly brace, check that the top of the stack is its matching left half. If it is, pop the left half off the stack. If it isn't (or if the stack is empty), the grouping symbols are either unbalanced or intersecting. Print an error and quit.
3. For any character that isn't a grouping symbol, ignore it and move on.

If you reach the end of input without an error, check to see if the stack is empty. If it is, then all of the left grouping symbols have been matched up with right grouping symbols. If not, there are left symbols left on the stack, and the expression isn't correctly nested.

9.2 Concepts: Object-oriented programming

To solve the nested expressions problem, we need to create a stack data structure. Each element of the stack should be able to hold a **char** value term from an expression, where a term is an operator, operand, or a parenthesis (bracket or curly brace). Although we could get by using only the static methods we introduced in the previous chapter, a better tool can make programming easier.

We are introducing these topics in a progression: First, all of our programs were a series of sequential instructions inside of a `main()` method. We added in selection statements and loops to solve more difficult problems. As our programs became more complicated, we started using additional static methods to divide the program into logical segments. Now, we are moving to fully object-oriented programming (OOP) in which data as well as code can be packaged into objects that interact with each other.

OOP has been a controversial topic, particularly in the computer science education community. The most important thing to remember is that we are not throwing away any of the ideas we used before. We are continuing on a path toward making code safer and more reusable. Several other chapters in this book touch on important ideas in OOP such as inheritance and polymorphism. Below,

we are going to focus on the basics, including the fundamentals of objects, encapsulation of data, and instance (non-static) methods.

9.2.1 Objects

You have already used objects, perhaps without realizing it. Every time you use a String, you are using an object. So far, we have created a class every time we have written a program. A class is a way to organize static methods, but a class is something more: a template for objects. Whenever you write a class, the potential to create specific objects exists.

For a conceptual example, you can think of Human as a class and Albert Einstein as an object (or an *instance*) of that class. The class defines certain characteristics that all human beings have: name, date of birth, height, and so on. Then, the object has specific values for each one of those characteristics, such as Albert Einstein, March 14, 1879, 175, and so on.

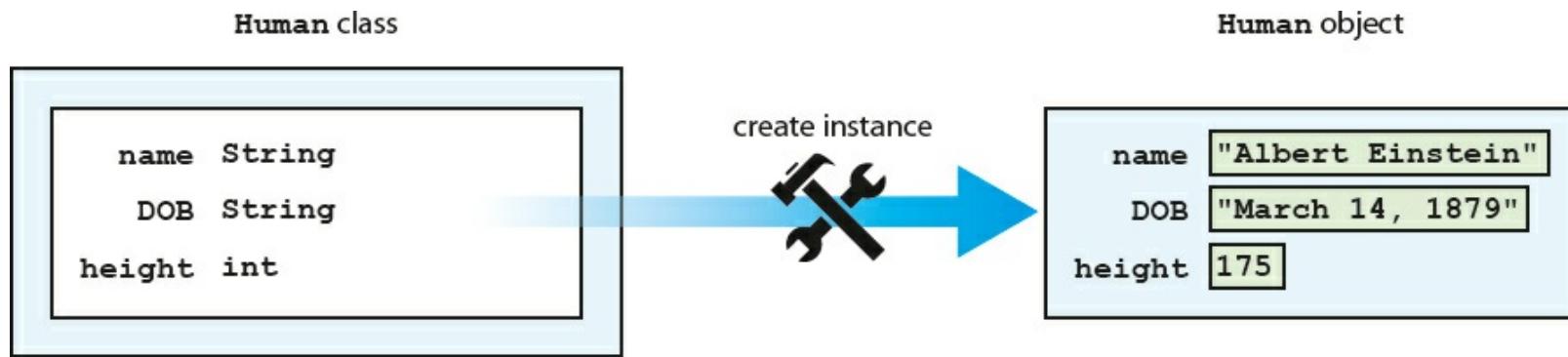


Figure 9.1: Example of a class serving as a template for an object.

This idea of a class as a template is key because it means that an object of a given type (from a given class) can be used anywhere that is appropriate for another object of that type. Later in the chapter, we create an object that performs the work of a stack, storing a number of other objects. If we design a library of code that can manipulate or use stack objects, we should be able to use this library countless times for countless different stack objects without changing the code. This kind of code reuse is one of the main goals of OOP.

9.2.2 Encapsulation

In order to guarantee that objects can be used in many different contexts safely, the data inside of the object must be protected. Java provides access modifiers so that code without appropriate permission cannot change or even read the data inside of an object.

This feature is called *encapsulation*. One programmer may write the class file defining a type of object while another or many others write code that uses those objects. The programmers who use objects written by others do not need to understand the inner workings of those objects. Instead, they can treat each object as a “black box” with a list of actions that the object can do. Each action has a certain specified input and a certain specified output, but the internal functioning of the object is hidden.

9.2.3 Instance methods

These “actions” are methods, but not static ones. Static methods did not need an object in order to be called. Regular (instance) methods should be thought of as an action performed on (or by) a specific object. This action could be asking a question, such as inquiring what the name of an object of type Human is. This action could be telling the object to change itself, such as the case of pushing something onto a stack.

One of the broadest definitions of an object is a collection of data and methods to access that data. We call the data inside of an object its *fields* or instance data and the methods to access them *instance methods*. Static methods are used primarily to modularize large blocks of code into smaller functional units. However, instance methods are tightly coupled to the fields of an object and perform tasks that change the object or get information from it.

9.3 Syntax: Classes in Java

OOP concepts such as encapsulation may seem esoteric until you see them in practice. Remember, we just want to create some private data and then define a few carefully controlled ways that the data can be manipulated. First, we’ll describe how to declare fields, then explain how to write instance methods to manipulate that data, and finally give more details about protecting its privacy.

9.3.1 Fields

Fields in an object must be declared like any other data in Java. The type of a piece of data can be a primitive or reference type. Fields are also sometimes called member variables. You declare fields just like you would class variables, except without the `static` keyword. Here’s an example with the Human class.

```
public class Human {  
    private String name ;  
    private String DOB ;  
    private int height ; // in cm  
}
```

With this definition, a Human object has three attributes: name, DOB, and height. Because the access modifier for each field is `private`, code outside of this class cannot change or even read the values. Note that this class cannot do anything yet. Also, note that this class does not contain a `main()` method. There is no way to run this class, but that’s fine. We can add a `main()` method, of course.

```
public class Human {  
    private String name ;  
    private String DOB ;  
    private int height ; // in cm  
    public static void main ( String [] args ) {  
        name = " Albert Einstein " ;  
        DOB = " March 14, 1879 " ;  
        height = 175 ;  
    }  
}
```

```
}
```

Now we have added a main() method, but our code does not compile. Since the main() method is a static method, it is not associated with any particular object. When we tell the main() method to change the fields, it does not know what object we are talking about. If we actually want to use an object, we'll have to create one.

Program 9.1: Example of a class encapsulating the attributes of a human being. (Human.java)

```
1 public class Human {  
2     private String name ;  
3     private String DOB ;  
4     private int height ; // in cm  
5  
6     public static void main ( String [] args ) {  
7         Human einstein = new Human ();  
8         einstein.name = " Albert Einstein ";  
9         einstein.DOB = " March 14, 1879 ";  
10        einstein.height = 175;  
11    }  
12 }
```

The above code compiles because we have used the `new` keyword to create an object of type Human saved in a reference variable called einstein. With a Human object, we can set its fields using dot notation. With static methods and static variables, we used the name of the **class** followed by a dot, for instance methods and instance variables, we use the name of the **object** followed by a dot. Even though each of these fields is private, we can access them from main() because main() is inside the Human class. Code inside of another class could create a new Human object, but it could not change its fields.

This juxtaposition of static and non-static fields and methods inside of a single class is confusing to many new Java programmers. The confusion seems to stem from the fact that the class (such as Human) is a template for objects but it is also a place to house other related code, such as static methods, including main().

Although the practice is discouraged, we mentioned in [Section 8.3.3](#) that class variables can be stored in the class itself. Every object has a distinct copy of each field, but there is only a single copy of each class variable that they all share. By using the keyword `static`, we could add a class variable called population to our Human class, since that is information connected to humans as a whole, not to any individual human being.

```
public class Human {  
    private String name ;  
    private String DOB ;  
    private int height ; // in cm
```

```
private static double population = 7.023 E9;  
}
```

We are using a **double** to represent the world's population since the value is too big to fit in an **int** and is easily expressed in scientific notation. If several Human objects were created, they would have their own name, DOB, and height values, but the value for population would only be stored in the class.

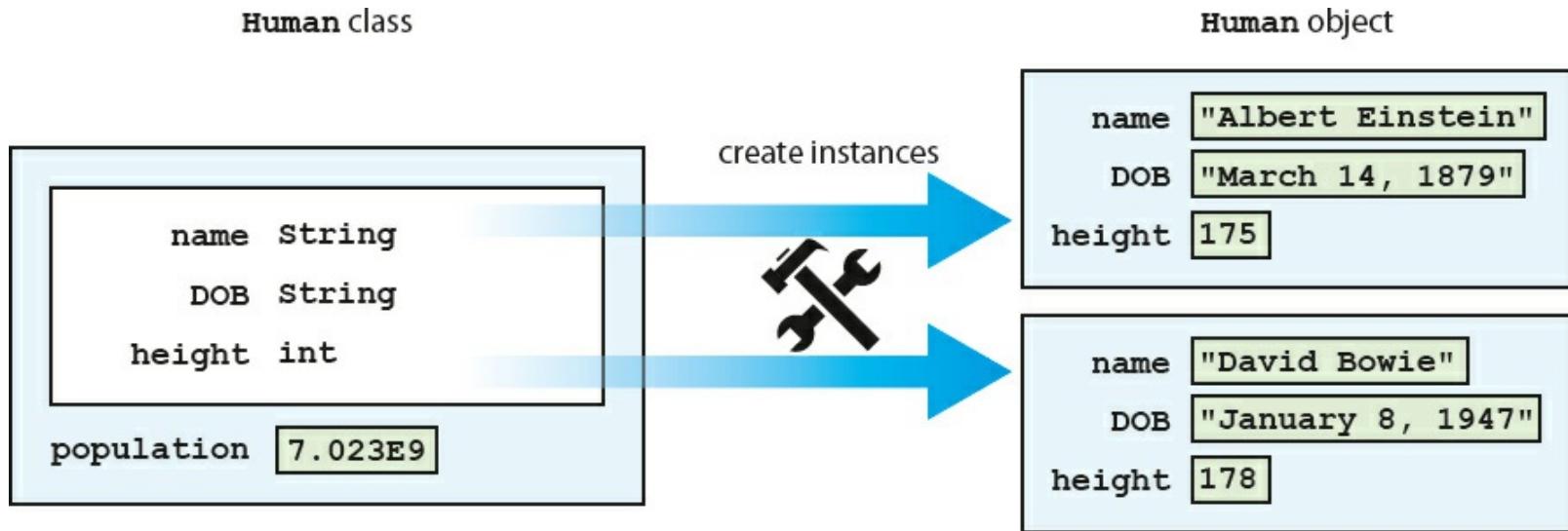


Figure 9.2: Class variables (static fields) are stored with the class, not with individual objects.

9.3.2 Constructors

To create a new object, you have to invoke a *constructor*, a special kind of method that can initialize the object. A constructor allows sets up the values inside an object when it is first created. Let's consider a simple Rectangle class with only two fields: length and width, both of type **int**.

```
public class Rectangle {  
    private int length;  
    private int width;
```

One possible constructor for the class is given below.

```
public Rectangle ( int l, int w ) {  
    length = l;  
    width = w;  
}
```

This constructor lets us set the width and length when the object is created. To do so, code could invoke the constructor using the **new** keyword.

```
Rectangle rectangle = new Rectangle (50, 20);
```

This code creates a new Rectangle object, with length 50 and width 20. Constructors are almost always **public**, otherwise it would be impossible for code outside of the Rectangle class to create a Rectangle object. Notice that the definition of the Rectangle constructor does not have a return type. A

constructors is the only kind of method that does not have a return type. It is possible to have more than one constructor as well, just as other methods can be overloaded. For more information about overloaded methods, refer back to [Section 8.3.1](#).

```
public Rectangle ( int value ) {  
    length = value ;  
    width = value ;  
}
```

In the very same class, we could have this second constructor, allowing us to quickly and easily create a square. All classes have constructors, but some are not written explicitly. If you don't type out a constructor for a class, a default one is automatically created. The default constructor takes no parameters and sets all the values inside the new object to defaults such as `null` and 0. Once you do create a constructor, the default one is no longer provided. Thus, since our definition of the Rectangle class already contains two constructors, the following line would cause a compiler error if someone tries to use it in their code.

```
Rectangle defaultRectangle = new Rectangle ();
```

Another important thing to consider with all instance methods is scope. Fields are visible inside of instance methods, but they can be hidden by parameters.

```
public Rectangle ( int length, int width ) {  
    length = length ;  
    width = width ;  
}
```

This version of the two parameter Rectangle constructor compiles, but it does not properly initialize the values of the fields length and width. Instead, the parameters length and width are copied back into themselves for no reason. The designers of Java anticipated that it would be useful to refer to fields even in the presence of other variables with the same name. To do so, the `this` keyword can be used. Any field (or method) can be referred to by its object name, followed by a dot, followed by the name of that field or method. Since you don't have a variable name to reference the object when you're inside of it, the `this` keyword acts as a reference to the object.

```
public Rectangle ( int length, int width ) {  
    this. length = length ;  
    this. width = width ;  
}
```

This version of the code functions correctly, since we have explicitly told Java to store the argument length into the field length inside the object pointed at by `this` and to do similarly for width.

9.3.3 Methods

Objects do not really come to life until you add instance methods. With the Rectangle class described above, any Rectangle objects created are not useful in other classes because it is impossible to access

their data. Instead, we want to create a clear and usable relationship between the fields and the methods.

There are many different kinds of methods, but two of the most important are accessors and mutators.

Accessors

We often want to read the data inside of various the objects. With our current definition of Rectangle, no code from an outside class can find out the length or width of the rectangle we are representing.

Accessor methods (or simply *accessors*) are designed for this task. By definition, an accessor allows us to read some data or get some information out of an object without making any changes to its fields. Accessors can be thought of as asking the object a question. The names of accessors often start with the word *get*.

```
public int getLength () {  
    return length ;  
}  
  
public int getWidth () {  
    return width ;  
}
```

Here are two accessors methods that we would expect in the Rectangle class. The first returns the value of length, and the second returns the value of width. These methods only report information. They do not change the value of either variable. Their syntax should be self-explanatory. Each is declared to be **public** so that anyone can read the length and width of a rectangle. Both methods have a return type of **int** because that is the type used to store length and width inside a Rectangle object. Neither method has any parameters. An accessor does not have to be so simple. An accessor could return a value that needs to be computed from the underlying field data.

```
public int getArea () {  
    return length * width ;  
}  
  
public int getPerimeter () {  
    return 2* length + 2* width ;  
}
```

These accessors compute the area and perimeter, respectively, of the rectangle in question, even though that data is not stored directly in the Rectangle object.

Mutators

Some objects, such as String values, are *immutable* objects, meaning that the data stored inside them cannot be changed after they have been created with a constructor. If you have ever thought that you were changing a String, you were actually creating a new String with the appropriate modifications. Most objects are mutable, however, and we use methods called *mutator methods* (or simply *mutators*) to change their fields.

Like accessors, mutators have no special syntax. The term is used to describe any methods that change the data inside of an object. For the Rectangle class, the only internal data we have is the length and width variables. Mutators for these might look as follows.

```
public void setLength( int length ) {  
    this.length = length;  
}  
  
public void setWidth( int width ) {  
    this.width = width;  
}
```

Just as the names for many accessors begin with get, the names for many mutators begin with set. Mutators often have a **void** return type because they are changing the object, not getting information back. Some mutators might have a return type that gives information about an error that occurred while trying to make a change. Note that we used the **this** keyword once again to distinguish each field from the method argument with the same name.

You may have noticed that we use the machinery of a method to both get and set the length field, for example. Perhaps doing so seems needlessly complex. After all, if the length variable had been declared with the **public** modifier instead of the **private** modifier, we could get and set its value directly, without using methods. In response, let's improve the mutators that set length and width.

```
public void setLength( int length ) {  
    if( length > 0 )  
        this.length = length;  
}  
  
public void setWidth( int width ) {  
    if( width > 0 )  
        this.width = width;  
}
```

With these better mutators, we can prevent a user from setting the values of length and width to negative numbers or zero, values that don't make sense for dimensions of a rectangle. For more complicated objects, it becomes even more important to protect the values of the fields from malicious or mistaken users.

9.3.4 Access modifiers

Hiding data is at the heart of the Java OOP model. There are four different levels of access that can be applied to fields and methods, whether static or not. They are **public**, **private**, **protected**, and **package-private**.

public modifier

The **public** access modifier states that a variable or method can be accessed by any code, no matter what class contains it. Most methods should be **public** so that they can be used freely to

interact with their object. Virtually no fields should be **public**. Constants (static or otherwise) are the most significant exception to this rule. Making constants **public** is usually not a problem since they cannot be changed by outside code anyway. In the Rectangle class, variables length and width are so simple that making them **public** is not unreasonable. If you have a field that can be changed at any time by any code to any value, you can leave that field **public**.

private modifier

This modifier states that a variable or method cannot be accessed by any code unless the code is contained in the same class. It is important to realize that the restriction is based on the **class**, not on the **object**. Code inside any Rectangle object can modify **private** values inside of any other Rectangle object or class. Most fields should be **private** so that outside code cannot modify them. Methods can be **private**, but these methods should be helper or utility methods used inside the class or object to divide up work.

protected modifier

This modifier states that a variable or method cannot be accessed by any code unless the code is contained in the same class, a subclass, or is in the same package. This level of access is more restrictive than **public** but less restrictive than **private** or default access. We discuss it further in the context of subclasses and inheritance in [Chapter 11](#).

Package-private (no explicit modifier)

If you do not type an access modifier when you declare a field or method, that field or method is not **public**. Instead, it has the default or package-private access modifier applied to it. Fields or methods with this modifier can be accessed by any code that is in the same *package* or directory. A package is yet another layer of organization that Java provides to group classes together. When you use an **import** statement, you can import an entire package of classes. There is no keyword for this access modifier. It is useful if you are designing a package containing classes that must be able to access each other's fields or methods. For now, you should always give your fields and methods an explicit **public** or **private** (or sometimes **protected**) modifier.

From least restrictive to most restrictive, the modifiers are **public**, **protected**, package-private, and **private**. Each additional level of restriction removes a single category of access. All fields and methods can be accessed by code from the same class. The following table gives the contexts outside the class that can access a field or method marked with each modifier.

9.4 Examples: Classes

Although large and complex programs are needed to see the real benefits of OOP in Java, here are two short examples showing, respectively, how objects can be used to make a roster of students and compute the value of an expression in postfix notation.

Modifier	Package	Subclass	Unrelated Classes
public	Yes	Yes	Yes
protected	Yes	Yes	No
Package-private	Yes	No	No
private	No	No	No

Example 9.1: Student roster

We are going to create a `Student` class so that we can store objects containing student roster information. Then, we're going to create a client program that reads data from a user to create `Student` objects, sort them by GPA, and then print them out.

```

1 public class Student {
2     public static final String [] YEARS =
3         {"Freshman", "Sophomore", "Junior", "Senior"};
4     private String name;
5     private int year;
6     private double GPA;

```

We start by defining the `Student` class. First, there is a constant array of type `String`, giving the names of each of the four years. Next, the fields in the `Student` class store the name, year, and GPA of the student.

```

8     public Student ( String name, int year, double GPA ) {
9         setName ( name );
10        setYear ( year );
11        setGPA ( GPA );
12    }

```

We have one constructor for this class, which takes in a `String`, an `int`, and a `double` corresponding to the name, year, and GPA of the student. The constructor then internally uses mutator methods to store the values into the fields. By doing so, we automatically take advantage of the error checking in the GPA mutator.

```

14    public void setName ( String name ) { this.name = name; }
15    public void setYear ( int year ) { this.year = year; }
16
17    public void setGPA ( double GPA ) {
18        if( GPA >= 0 && GPA <= 4.0 )
19            this.GPA = GPA;
20        else
21            System.out.println (" Invalid GPA : " + GPA );
22    }

```

These are the mutators corresponding to each of the three fields. The input for the name and year mutators are not checked, but the GPA mutator checks to make sure that the GPA value is in the proper range.

```
24 public String getName () { return name ; };
25 public int getYear () { return year ; };
26 public double getGPA () { return GPA ; };
27
28 public String toString () {
29     return name + "\t" + YEARS [ year ] +
30         "\t" + GPA ;
31 }
32 }
```

Finally, these accessors allow the user to find out the name, year, or GPA of a given student. Every class in Java automatically has a `toString()` method that is called whenever an object is being printed out directly. We have changed this method to return the information in `Student` formatted as a `String`.

Creating the `Student` class is only half the battle. We also must create client code to use it.

```
1 import java.util.*;
2
3 public class StudentRoster {
4     public static void main ( String [] args ) {
5         Scanner in = new Scanner ( System.in );
6         Student [] roster = new Student [in.nextInt () ];
7         for ( int i = 0; i < roster.length ; i++ ) {
8             in.nextLine ();
9             roster [i] = new Student ( in.nextLine (),
10                 in.nextInt (), in.nextDouble () );
11         }
12         sort ( roster );
13         for ( int i = 0; i < roster.length ; i++ )
14             System.out.println ( roster [i]);
15     }
}
```

The `main()` method in the `StudentRoster` class begins by reading in the total number of students. Next, it makes an array of type `Student` of that length. Then, it repeatedly reads in a name, year, and GPA, creates a new `Student` object with those values, and stores it into the array. After creating all the `Student` objects, it sorts them with a method call and prints them out.

One oddity in this code is the seemingly superfluous `in.nextLine()` in the first `for` loop. This line of code consumes a trailing newline character from previous input. Take it out and see how quickly the program malfunctions.

```
17     public static void sort ( Student [] roster ) {
18         for ( int i = 0; i < roster.length - 1; i++ ) {
```

```

19     int smallest = i;
20     for ( int j = i + 1; j < roster.length ; j++ )
21         if( roster [j]. getGPA () < roster [ smallest ]. getGPA () )
22             smallest = j;
23     Student temp = roster [ smallest ];
24     roster [ smallest ] = roster [i];
25     roster [i] = temp ;
26 }
27 }
28 }
```

This sort() method is similar to others that you have seen. It implements selection sort in ascending order based on GPA.

If you run this program, you will notice that it does not prompt the user for any input. This version of the code is designed for redirected input from a file. A more user friendly, interactive version should prompt the user clearly.

Using OOP is not necessary to solve this problem. Instead of objects, we could have used three separate arrays holding the name, year, and GPA of each student, respectively. However, coordinating these arrays together would become tedious, particularly when sorting. ■

9.5 Advanced Syntax: Nested classes

Inside of a class, you can define fields and methods, but what about other classes? Yes! Doing so creates a *nested class*. When you define a class inside of an outer class, it can access fields and methods in the outer class, even if they are marked **private**. Java allows a number of different ways to define a nested class. They are all useful, but each is subtly different. Some nested classes are tied to a specific object of the outer class while others are not.

9.5.1 Static nested classes

If you mark a nested class with the **static** keyword, you are creating a class whose objects are independent of any particular outer class object. Such a class is called a *static nested class*. Consider the following class definition.

```

public class Outer {
    private int x;
    private int y;
    public static class Nested {
        private int z;
    }
}
```

A static nested class is similar to a normal, top-level class with two differences. First, the full name of a nested class is the name of the outer class followed by a dot followed by the nested class

name. Second, when given an outer class object, code in a static nested class can access and modify **private** (and **protected**)) data in the outer class object.

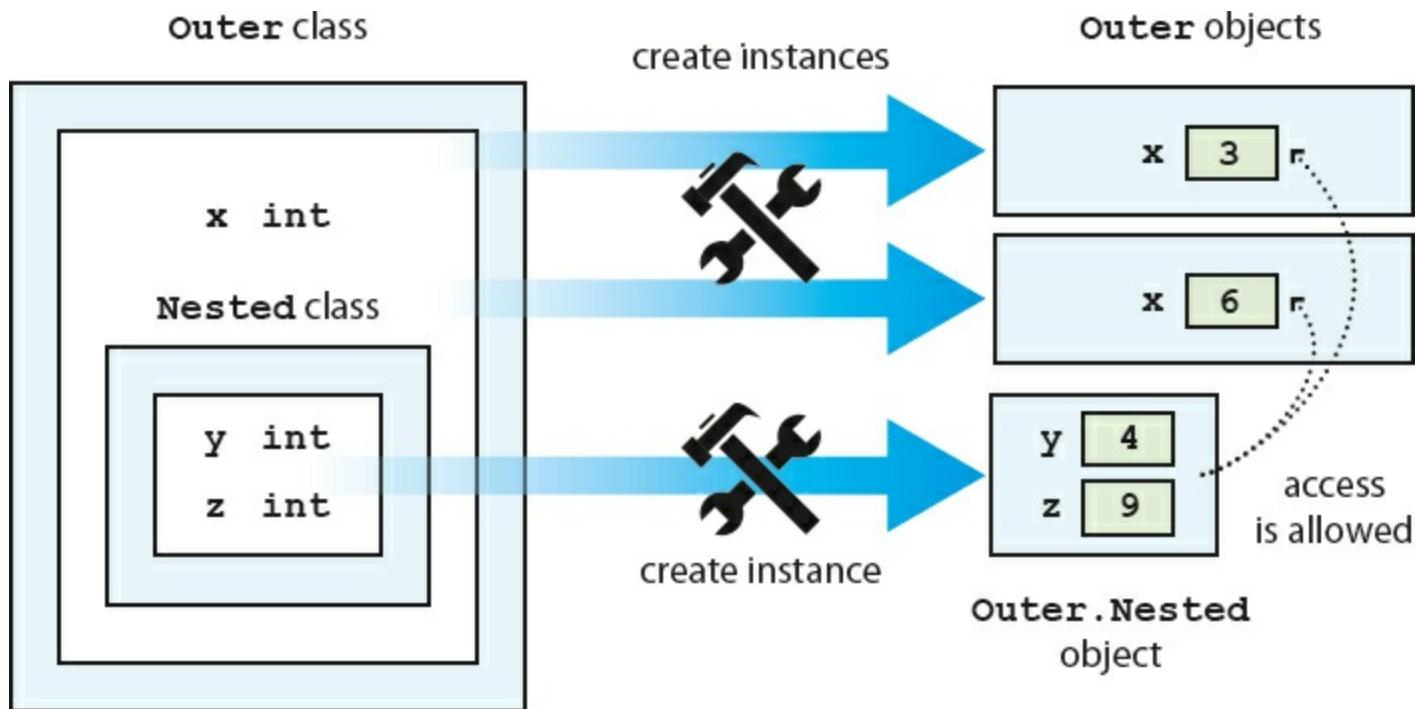


Figure 9.3: A static nested class object is allowed to access data from outer class objects even though there is no direct relationship between them.

Static nested classes can be used when the class you need is only useful in connection with the outer class. Thus, nesting the class groups it with its outer class. We can create an instance of the nested class above as follows.

```
Outer.Nested nested = new Outer.Nested();
```

Because it is a static nested class, we do not need an instance of type Outer to create an instance of type Outer.Nested. If you compile Outer.java, it will create two files, Outer.class and Outer\$Nested.class. The dollar sign (\$) separates the names of each level of nested class in the file name. It is possible to nest classes inside of nested classes, producing another .class file with another dollar sign and the new class name appended.

Note that static nested classes can be marked **public**, **private**, **protected**, or package-private (no explicit modifier). These access modifiers control which code can access or instantiate static nested classes using the same access rules for fields and methods.

Example 9.2: Static nested class for testing

One application for static nested classes is testing. You can write code that tests the functionality of your outer class, fiddling with its fields if needed. Then, because a separate .class file is created, you can deliver only the .class file for the outer class to your customer.

Consider the Square class, similar to the Rectangle class given earlier.

```
public class Square {
```

```

private int side ;
public Square ( int side ) {
    this. side = side ;
}
public int getArea () {
    return side * side ;
}
}

```

We could add a static nested class called Test to Square to test that its getArea() and getPerimeter() methods are working properly. The final code might be as follows.

```

public class Square {
    private int side ;

    public Square ( int side ) {
        this. side = side ;
    }

    public int getArea () {
        return side * side ;
    }

    public static class Test {
        public static void main ( String [] args ) {
            Square square = new Square (5) ;
            System.out.print (" Test 1: ");
            if( square.getArea () == 25 )
                System.out.println (" Passed ");
            else
                System.out.println (" Failed ");
            square.side = 7;
            System.out.print (" Test 2: ");
            if( square.getArea () == 49 )
                System.out.println (" Passed ");
            else
                System.out.println (" Failed ");
        }
    }
}

```

To run the tests, you would compile Square.java and then run the nested class by invoking java Square\$Test. It is unwise to use the nested class to change the private fields in square, but we did so to show that it is allowed by Java. A better test would create a second Square object with a side of

length 7. ■

9.5.2 Inner classes

Another kind of nested class is an *inner class*. Unlike static nested classes, the objects of inner classes are associated with a particular object of the outer class. You can think of an inner class object living **inside** an outer class object. It is impossible to instantiate an inner class object without having an outer class object first. Consider the following class definition.

```
public class Outer {  
    private int a;  
  
    public class Inner {  
        private int b;  
        private int c;  
    }  
}
```

Every instance of Inner must be associated with an instance of Outer. To instantiate an inner class, you use the name of an outer class object, followed by a dot, followed by the new keyword, and then the name of the inner class. We can create an instance of the inner class above as follows.

```
Outer outer = new Outer();  
Outer.Inner inner = outer. new Inner();
```

This syntax looks confusing, but it makes inner an object that exists inside of outer. Thus, if there were methods defined in Inner, they could refer to field a, because every instance of Inner would be inside of an instance of Outer with a copy of a.

The relationship between outer and inner objects is one to many. We can instantiate any number of inner class objects that all live inside of the same outer class object.

Another issue with inner classes (as opposed to static nested classes) is that they cannot contain static fields (except for constants) or methods. Since each instance of an inner class is tied to an instance of an outer class, the designers of Java thought that static fields and methods for an inner class really belong in the outer class.

It is even possible to define a class **inside** a method, if that class is only referred to in the method. Such a class is called a *local class*. It is possible to create an unnamed local class on the fly as well. Such a class is called an *anonymous class*. Both local and anonymous classes are special kinds of inner classes. Because of the way they are created and used, we discuss them in [Section 10.4](#).

Example 9.3: Inner class objects as iterators

If you create a data structure for other programmers to use, a useful feature is the ability to retrieve each item from the data structure in order. Different threads or methods might need to process these elements independently from each other. Each piece of code can be given an inner class object called an *iterator* that can repeatedly get the next item in the data structure. Since instances of an inner class

can read private data of the outer class, iterators can keep track of where they are inside the data structure. If outside code were allowed access to the data structure's internals, it would violate encapsulation. Iterators are a very common application of inner classes.

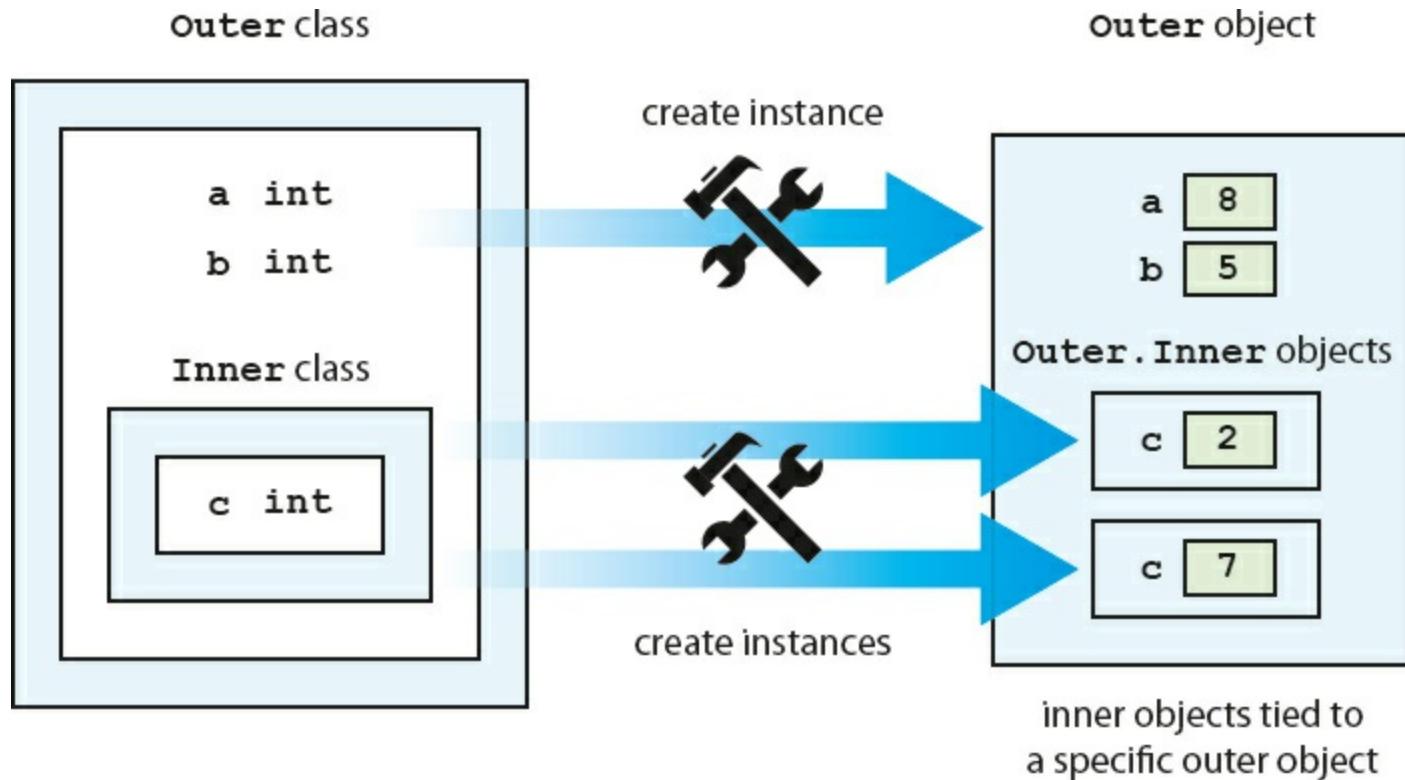


Figure 9.4: An inner class object is always associated with a specific outer class object.

We can create a `SafeArray` class that only allows data to be written to its internal array if it falls in the legal range of indexes.

```
public class SafeArray {  
    private double [] data ;  
    public SafeArray ( int size ) {  
        data = new double [ size ];  
    }  
  
    public int set ( int index, double value ) {  
        if( index >= 0 && index < data.length )  
            data [ index ] = value ;  
    }  
}
```

We could add an inner class called `Iterator` to `SafeArray` that allows us to process all the array values without knowing how many there are. This kind of behavior is useful for many dynamic data structures, as discussed in [Chapter 18](#).

```
public class SafeArray {  
    private double [] data ;
```

```

public SafeArray ( int size ) {
    data = new double [ size ];
}

public void set ( int index, double value ) {
    if( index >= 0 && index < data.length )
        data [ index ] = value ;
}

public class Iterator {
    private int index = 0;

    public boolean hasNext () {
        return ( index < data.length );
    }

    public double getNext () {
        if( index >= 0 && index < data.length )
            return data [ index ++];
        else
            return Double.NaN;
    }
}
}

```

The following method uses the iterator we have defined to find the sum of the values in a SafeArray object.

```

public static findSum ( SafeArray array ) {
    double sum = 0;
    SafeArray.Iterator iterator = array.new Iterator ();

    while ( iterator.hasNext () )
        sum += iterator.getNext ();
    return sum ;
}

```

9.6 Solution: Nested expressions

We now have enough knowledge to solve the nested expressions problem from the beginning of the chapter. Classes help us divide up the work of solving the problem. First, we need a stack class that can hold **char** values.

Program 9.2: Simple stack class to hold symbols from an input expression. (SymbolStack.java)

```

1 public class SymbolStack {
2     private char [] symbols ;
3     private int size ;
4
5     public SymbolStack ( int maxSize ) {
6         symbols = new char [ maxSize ];
7         size = 0;
8     }
9
10    public void push ( char symbol ) { symbols [ size ++] = symbol ; }
11    public void pop () { size --; }
12    public char top () { return symbols [ size - 1]; }
13    public boolean isEmpty () { return size == 0; }
14 }
```

The `SymbolStack` class allows us to perform the `push`, `pop`, and `top` stack operations with methods of the same names. Its constructor takes a `maxSize` for the stack and allocates an array of that size. It also sets the `size` field to 0 so that we can keep track of how many things are in the stack (and consequently where the top is).

The `push()` method stores an input `char` into the stack at location `size` and then increments `size`. The `pop()` method simply decrements `size`. It has no error checking to prevent a user from popping the stack once it is already empty. Finally, the `top()` method returns the value at the top of the stack, whose location is `size - 1`. `SymbolStack` also defines an `isEmpty()` method so that we can see if the stack is empty.

Now we need the client code that reads the input and interacts with the stack.

```

1 import java.util.*;
2
3 public class NestedExpressions {
4     public static void main ( String [] args ) {
5         Scanner in = new Scanner ( System.in );
6         String input = in. nextLine ();
7         SymbolStack stack = new SymbolStack ( input.length () );
8         char symbol ;
9         boolean correct = true ;
```

The `main()` method of this class reads in the input and creates a `SymbolStack` called `stack` with a maximum size of the input length. We know that the stack never needs to hold more than the total input. It also creates a `boolean` named `correct` to keep track of whether or not the input is correctly nested. We start by assuming that it is.

```

10    for ( int i = 0; i < input.length () && correct ; i++ ) {
11        symbol = input.charAt ( i );
12        switch ( symbol ) {
```

```

13     case '(':
14     case '[':
15     case '{':
16         stack.push ( symbol );
17         break ;
18     case ')':
19         if( stack.isEmpty () || stack.top () != '(' )
20             correct = false ;
21         else
22             stack.pop ();
23         break ;
24     case ']':
25         if( stack.isEmpty () || stack.top () != '[' )
26             correct = false ;
27         else
28             stack.pop ();
29         break ;
30     case '}':
31         if( stack.isEmpty () || stack.top () != '{' )
32             correct = false ;
33         else
34             stack.pop ();
35         break ;
36     }
37 }

```

This `for` loop runs through each `char` in the input. If it is a left parenthesis, left square bracket, or left curly brace, it pushes the symbol onto the stack. If it is a right parenthesis, right square bracket, or right curly brace, it checks to see if the stack is empty. Because of short-circuit evaluation, the code does not even look at the top of the stack if it is empty. However, if the stack is not empty, it checks to see if the top matches the current symbol. If the stack is empty or its top does not match, `correct` is set to `false`. For efficiency, the loop stops early if `correct` is no longer `true`.

```

39     if( ! stack.isEmpty () ) // unmatched left symbols
40         correct = false ;
41
42     if( correct )
43         System.out.println ("The input is correctly nested !");
44     else
45         System.out.println ("The input is incorrectly nested !");
46     }
47 }

```

After the input has been examined, we check to see if the stack is empty. If it isn't, there must be

some left symbols that were not matched with right symbols. In that case, we set correct to **false**. Finally, we print out whether the input is correctly or incorrectly nested based on the value of correct.

9.7 Concurrency: Objects

Nearly everything in Java is an object: arrays, lists, String values, colors, and even exceptions, which form Java's error-handling system and is discussed in [Chapter 12](#). Some critics of Java point out that **int**, **double**, and the other primitive types are not objects, forcing the programmer to adopt two different programming models. Regardless, threads are stored as objects as well. In [Chapter 13](#), we discuss how to create threads and the various methods that can be used to interact with them.

However, objects of type **Thread** are not the only ones you deal with when writing concurrent programs. As we have just noted, most data in Java is encapsulated in an object. One of the deep reasons for using OOP is safety: We want the private data inside of an object to stay in a consistent state. Due to their inexplicable ability to get out of tight situations, one tradition holds that cats have nine lives. Because of their inquisitive nature, another tradition holds that curiosity killed the cat. Consider the class below that keeps track of the lives a cat has, losing one every time it becomes curious.

Example 9.4: Thread safety

Program 9.3: Program to record the number of lives a cat has left. Each Cat object starts with 9, but loses one each time it uses its curiosity. If no more lives remain, an error message is output.
(Cat.java)

```
1 public class Cat {  
2     private int lives = 9;  
3  
4     public boolean useCuriosity () {  
5         if( lives > 1 ) {  
6             lives --;  
7             System.out.println ("Down to life " + lives );  
8             return true ;  
9         }  
10        else {  
11            System.out.println ("No more lives left !");  
12            return false ;  
13        }  
14    }  
15 }
```

If the relationship between curiosity and mortality is the only feature of a cat you are trying to model, this class appears to function well. If the **useCuriosity()** method is invoked, it removes a life or prints an error message if the cat has run out of lives. In a single-threaded situation, this object would work perfectly. No cat would be able to lose more than 9 lives.

In a multi-threaded situation, there is no telling when a thread might pause in executing the `useCuriosity()` method. If 100 threads all called `useCuriosity()`, each one might successfully pass the if statement on [line 5](#) before any had decremented lives. Once past the check, nothing would prevent them from continuing on and decrementing lives, resulting in a cat who lost 100 lives, resulting in a total of -91 lives. Such a scenario makes no sense.

In [Chapter 14](#), we discuss how to prevent this problem, using the `synchronized` keyword to allow only a single thread at a time to execute the body of a method. The goal is to make `useCuriosity()` *thread-safe*, meaning that its behavior is consistent and correct no matter how many threads try to execute it at the same time. ■

As you work through this book and begin to write your own concurrent programs, we discuss ways to make them thread-safe. However, you are also a consumer of code written by other people. In multi-threaded environments, you may need to use library classes that are thread-safe. For example, `AtomicInteger` is a thread-safe class designed to store and manipulate `int` values. In [Chapter 18](#), we talk about the `ArrayList` and `Vector` classes, which are both used to hold variable length lists of objects. One of the few differences between them is that `ArrayList` is not thread-safe while `Vector` is. There is even the `Collections.synchronizedCollection()` method (and other similar methods), which takes a collection that is not thread-safe and returns a version of it that is.

Java was intended to be multi-threaded from the very beginning, but concurrency was never the most important feature in the language. For that reason, the documentation does not clearly mark which methods are thread-safe. Usually, some of the paragraphs of description above the list of methods say that a class is “`synchronized`” if it is thread-safe. If it is not, the documentation may not mention anything. Careful attention is needed to be sure which classes and APIs are thread-safe.

You may wonder why all classes are not thread-safe, but everything comes with a price. If a class is thread-safe, its methods are usually marked with the `synchronized` keyword. The JVM is relatively efficient about how it enforces that keyword, but the computational expense is not zero. Learn the libraries well, and use the right tools for the right job.

Exercises

Conceptual Problems

- 9.1 Explain the relationship between a class and an object.
- 9.2 What is the difference between a static method and an instance method?
- 9.3 What is the purpose of a constructor? Why is it impossible for a constructor to return a value?
Why is it impossible for a constructor to be called multiple times on the same object?
- 9.4 A static method can be called directly from a instance method, but an instance method cannot be called directly from a static method. Why?
- 9.5 Describe the uses of accessor and mutator methods. Is it possible to create a method that is both an accessor and a mutator? Why or why not?
- 9.6 Why do we usually mark fields with the `private` keyword when it would be easier to make all

fields **public**?

9.7 What is the meaning of the **this** keyword? When is it necessary to use it? When can it be ignored?

9.8 Consider the following class definitions.

```
public class A {  
    private int a;  
  
    public int get() {  
        return a;  
    }  
  
    public static void increment() {  
        a++;  
    }  
}  
  
public class B {  
    private int b;  
  
    public B( int value ) {  
        b = value ;  
    }  
  
    public A generate() {  
        A object = new A();  
        object.a = b;  
    }  
}
```

The field **a** is used three times in the previous code. Which of these uses cause a compiler error and why?

9.9 In [Section 9.6](#), we gave a definition of **SymbolStack** that implements a simple stack using two fields, defined as followed:

```
private char [] symbols ;  
private int size ;
```

By calling the **top()** or **pop()** methods on an empty stack, it is possible to cause a program to crash. What additional problems could happen if **symbols** and **size** were declared **public** and malicious or poorly written code had access to the object?

9.10 Consider the following class definition.

```

public class GroceryItem {
    private String name ;
    private double price ;

    public GroceryItem ( String text, double money ) {
        String name = text ;
        double price = money ;
    }

    public String getName () { return name ; }
    public String getPrice () { return price ; }
}

```

This class compiles, but its constructor does not function properly. Why not?

Programming Practice

9.11 OOP is often used when the data inside the object must maintain special relationships. Consider a clock with hours, minutes, and seconds. When the number of seconds reaches 60, the number of minutes is increased by 1, and the number of seconds is reset to 0. When the number of minutes reaches 60, the number of hours is increased by 1, and the number of seconds is reset to 0. When the number of hours reaches 13, it is reset to 1. AM and PM switch whenever the number of hours reaches 12.

Define a Clock class with **private int** fields hours, minutes, and seconds and a **boolean** field PM. Write a constructor that initializes hours to 12, minutes and seconds to 0, and PM to **false**. Write a mutator increment() that adds 1 to seconds. This mutator should correctly handle all the clock behavior described above. Write an accessor called **toString()** that returns a nicely formatted version of the time as a String. For example, the initial time would be returned as “**12:00:00 AM**”. Make sure you pad the output for seconds and minutes with an extra “**0**” if they are less than 10.

9.12 Draw on any of your hobbies to come up with a collection of items, whether those items are books you like to read, athletes you follow, music you collect, or anything else that is easy to classify. First create a class that can describe one of these items with three to five attributes. For example, the important attributes of a book might be author, title, genre, and page count. Each of these attributes should be stored as a **private** field and manipulated with **public** accessor and mutator methods.

Using an array, create a database of these objects. Write methods that print out all objects that have a particular value for an attribute. Using the book example, your database program should let the user input that he or she is looking for all books whose author is Alexandre Dumas. You may wish to use input redirection so that you do not have to enter data about your objects repetitively.

9.13 The **java.awt** package defines a class called Point that can be used to manipulate an (x, y) pair in programs involving the Cartesian coordinate system. Create your own Point class with **int**

values x and y as fields.

Create one constructor that allows the user to specify values for x and y and a default constructor that takes no arguments and sets both x and y to 0. Create accessors and mutators for x and y.

Finally, create a method with the signature **public double** distance(Point p) that uses the distance formula $\sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}$ to find the distance between the current Point object and the Point object p passed in as an argument.

Write client code that allows you to create two Point objects and test if the distance() method gives the right answer.

- 9.14 Re-implement the solution from [Section 9.6](#) so that it performs its input and output with GUIs created using JOptionPane.

GUI

Experiments

- 9.15 Objects are great tools for solving problems, but there is some additional overhead associated with creating objects and calling methods.

Write a piece of code that allocates an array of 10,000,000 **int** values. Iterate through that array, storing the value i into index i, and time the process using an OS time command. As you know, the Integer wrapper class allows us to store an **int** value in object form. Repeat the experiment, but, instead of **int** values, allocate an array to hold 10,000,000 Integer objects. Iterate through the array again, storing an Integer object into each index of the array. For index i, store a new Integer object created by passing value i into its constructor. Compare the time taken to the previous time for int values. Do you think this is a reasonable way to estimate the time it takes to call a constructor and allocate a new object?

Chapter 10

Interfaces

Our work is the presentation of our capabilities.

—Edward Gibbon

10.1 Problem: Sort it out

Defining classes as we did in [Chapter 9](#) provides us with useful tools. First, we can group related data together. Then, we can encapsulate that data so that it can only be changed in carefully controlled ways. But these ideas are only a fraction of the true power of object orientation that becomes apparent when we exploit relationships between different classes. Eventually, in [Chapter 11](#), we will talk about hierarchical relationships, in which one class can be the parent of many other child classes. However, there is a simpler relationship that classes can share, the ability to do some specific action or answer some specific question. If we know that several different classes can all do the same things, we want to be able to treat them in a unified way. For example, imagine that you are writing code to aid laboratory analysis of the free radical content of a number of creatures and items. For this analysis it is necessary to sort the test subjects by weight and by age.

The subjects you are going to sort will be objects of type Dog, Cat, Person, and Cheese. All of these subjects will have some calendar age. Naturally, the age we are interested for a Dog is 7 times its calendar age. At the same time, because a cat has nine lives, we will use $\frac{1}{9}$ of its calendar age. The age we use for Person and Cheese objects is simply their calendar age.

As we discussed in [Example 6.2](#), sorting is one of the most fundamental things that computer scientists do. In that example, we introduced selection sort. Here we are going to use another sort called bubble sort. We are introducing bubble sort to expose you to another way to sort data. Although they are easy to understand, neither bubble sort nor selection sort are the fastest ways known to sort lists.

Bubble sort works by scanning through a list of items, a pair at a time, and swapping the elements of the pair if they are out of order. One scan through the list is called a *pass*. The algorithm keeps making passes until no consecutive pair of elements is out of order. Here is an implementation of bubble sort that sorts an array of `int` values in ascending order.

```
boolean swapped = true ;
int temp ;
while (swapped) {
    swapped = false ;
    for (int i = 0; i < array.length - 1; i++) {
        if (array[i] > array[i + 1]) {
            temp = array[i];
            array[i] = array[i + 1];
            array[i + 1] = temp;
            swapped = true;
        }
    }
}
```

```
array[i + 1] = temp ;  
swapped = true ;  
}  
}
```

Pitfall: Array out of bounds

Note that index variable `i` stops before `array.length - 1` since we don't want to get an `ArrayIndexOutOfBoundsException` when looking at `array[i + 1]`.

The only change needed to make a bubble sort work with lists of arbitrary objects (instead of just `int` values) is the `if` statement. Likewise, changing the greater than (`>`) to a less than (`<`) will change the sort from ascending to descending.

10.2 Concepts: Making a promise

Knowing how to sort a list of items is half of the problem we want to solve. The other half is designing the classes so that sorting by either age or weight is easy. And easy sorting is not enough: We want to follow good object oriented design so that sorting future classes will be easy as well.

For these kinds of situations, Java has a feature called *interfaces*. An interface is a set of methods that a class must have in order to *implement* that interface. If a class implements an interface, it is making a promise to have a version of each of the methods listed in the interface. It can choose to do anything it wants inside the body of each method, but it must have them to compile. Interfaces are also described as contracts, since a contract is a promise to do certain things. One reason that interfaces are so useful is because Java allows a class to implement any number of them.

The purpose of an interface is to guarantee that a class has the capability to do something. In the case of this sorting problem, we will need to sort by age and weight. Consequently, the ability to report age and weight are the two important capabilities that the objects we are sorting must have.

10.3 Syntax: Interfaces

Declaring an interface is similar to declaring a class except that interfaces can only hold public abstract methods and constants. An *abstract method* is one that does not have a body. Its return type, name, and parameters are given, but this information is concluded with a semicolon (`;`) instead of a method body. Because the purpose of an interface is to ensure a class has certain publicly available capabilities, all abstract methods and constants listed in an interface are assumed to be `public`. It is legal but needlessly cluttered to mark each method with `public`, and it is illegal to mark them with `private` or `protected`.

Here is an interface for a guitar player that defines two abstract methods.

```
2     void strumChord ( Chord chord );
3     void playMelody ( Melody notes );
4 }
```

Any class that implements this interface must have both of these methods, declared with the access modifier **public**, the same return types, and the same parameters. We could create a RockGuitarist class that implements Guitarist as follows.

```
1 public class RockGuitarist implements Guitarist {
2     public void strumChord ( Chord chord ) {
3         System.out.print (" Totally wails on that " + chord.getName ()
4             + " chord !");
5     }
6
7     public void playMelody ( Melody notes ) {
8         System.out.print ( " Burns through the notes " +
9             notes.toString () + " like Jimmy Page !" );
10    }
11 }
```

Or we can create a ClassicalGuitarist class. A classical guitarist is going to approach playing a chord or a melody differently from a rock guitarist. Java only requires that all the methods in the interface are implemented.

```
1 public class ClassicalGuitarist implements Guitarist {
2     public void strumChord ( Chord chord ) {
3         System.out.print ( " Delicately voices a " + chord.getName ()
4             + " chord." );
5     }
6
7     public void playMelody ( Melody notes ) {
8         System.out.print ( " Plucks the melodic line " +
9             notes.toString ()
10            + " with the skill of John Williams." );
11    }
12 }
```

Interfaces make our code more flexible because we can use a reference with an interface type to refer to any objects which implement that interface. In the following snippet of code we do this by creating 100 Guitarist references, each of which randomly points to either a RockGuitarist or a ClassicalGuitarist.

```
Guitarist [] guitarists = new Guitarist [100];
Random random = new Random ();
for ( int i = 0; i < guitarists.length ; i++ )
    if( random.nextBoolean ()
```

```
guitarists [i] = new RockGuitarist ();
else
guitarists [i] = new ClassicalGuitarist ();
```

Mistakes in the use of interfaces should generally be caught at compile time. If you implement an interface with a class you are writing but forget (or misspell) a required method, the compiler will fail to compile that class. If you remember all of the required methods but forget to specify that your class implements a particular interface, the compiler will fail when you try to use that class in a place where that interface is required.

If you have code that takes some arbitrary object as a parameter, you can determine if that object implements a particular interface by using the `instanceof` keyword. For example, the following method takes a Chord object and an object of type Object. The Object type is the most basic reference type there is. All reference types in Java can be treated as an Object type. Thus, a reference of type Object could point at any object of any type. In this case, if the Object turns out to implement the Guitarist interface, we can cast it to a Guitarist and use it to strum a chord.

```
void checkBeforeStrum ( Object unknown, Chord chord ) {
    if( unknown instanceof Guitarist ) {
        Guitarist player = ( Guitarist ) unknown ;
        player.strumChord ( chord );
    }
    else
        System.out.println ( " That 's not a guitarist ! " );
}
```

Pitfall: Interfaces cannot be instantiated

It is tempting to try to create an object of an interface type. If you think about this carefully, it should be clear why this is impossible. An interface is a promise to do things, but it is not a class. It has no data nor methods to manipulate data. Thus, the following code will fail to compile.

```
Guitarist guitarist = new Guitarist ();
```

We are permitted to make a reference to objects that implement Guitarist. We can make an array that holds such references. The interface itself, however, can never be instantiated.

10.3.1 Interfaces and static

When designing an interface, you cannot mark a method as `static`. Whether or not a method is `static` is considered by Java to be an implementation detail that is not suitable to specify in an interface. For example, even with Chord and Melody defined, the following interface fails to compile.

```
1 public interface StaticGuitarist {
2     void strumChord ( Chord chord );
```

```
3     void playMelody ( Melody notes );
4     static int getStrings ();
5 }
```

The designers of Java decided that they did not want their language to function this way. Some languages like PHP allow the equivalent of static methods in interfaces.

Neither fields nor class variables are allowed in interfaces either, but class constants are allowed. Thus, we could define **static final** values that might be useful to any class implementing an interface. With Chord and Melody defined, the following interface **will** compile.

```
1 public interface ConstantGuitarist {
2     static final int MAJOR = 1;
3     static final int NATURAL_MINOR = 2;
4     static final int HARMONIC_MINOR = 3;
5     static final int MELODIC_MINOR = 4;
6     static final int CHROMATIC = 5;
7     static final int PENTATONIC = 6;
8
9     void strumChord ( Chord chord );
10    void playMelody ( Melody notes );
11 }
```

Many modern Java users object to the use of constants in interfaces, since the purpose of an interface is to define a list of requirements for what a class does rather than dealing with data values. Nevertheless, constants are allowed in interfaces, and the Java API does so in many cases.

Example 10.1: Supermarket pricing

Interface names often include the suffix *-able*, for example, Runnable, Callable, and Comparable. This suffix is typical because it reminds us that a class implementing an interface has some specific ability. Let's consider an example in a supermarket in which the items could have very little in common with each other but they all have a price. We could define the interface Priceable as follows.

```
1 interface Priceable {
2     double getPrice ();
3 }
```

If bananas cost \$0.49 a pound, we can define the Bananas class as follows.

```
1 public class Bananas implements Priceable {
2     public static final double PRICE_PER_POUND = 0.49;
3     private double weight ;
4
5     public Bananas ( double weight ) { this.weight = weight ; }
6
7     public double getPrice () {
```

```
8     return weight * PRICE_PER_POUND ;
9 }
10 }
```

If eggs are \$1.50 for a dozen large eggs and \$1.75 for a dozen extra large eggs, we can define the Eggs class as follows.

```
1 public class Eggs implements Priceable {
2     public static final double PRICE_PER_DOZEN_LARGE = 1.5;
3     public static final double PRICE_PER_DOZEN_EXTRA_LARGE = 1.75;
4     private int dozens ;
5     private boolean extraLarge ;
6
7     public Eggs ( int dozens, boolean extraLarge ) {
8         this.dozens = dozens ;
9         this.extraLarge = extraLarge ;
10    }
11
12    public double getPrice () {
13        if( extraLarge )
14            return dozens * PRICE_PER_DOZEN_EXTRA_LARGE ;
15        else
16            return dozens * PRICE_PER_DOZEN_LARGE ;
17    }
18 }
```

Finally, if water is \$0.99 a gallon, we can define the Water class as follows.

```
1 public class Water implements Priceable {
2     public static final double PRICE_PER_GALLON = 0.99;
3     private int gallons ;
4
5     public Water ( int gallons ) { this.gallons = gallons ; }
6
7     public double getPrice () { return gallons * PRICE_PER_GALLON ; }
8 }
```

Each class could be much more complicated, but the code shown is all that is needed to implement the Priceable interface. Even though there is no clear relationship between bananas, eggs, and water, a shopping cart filled with these items (and any others implementing the Priceable interface) could easily be totaled at the register. If we represent the shopping cart as an array of Priceable items, we could write a simple method to total the values like so.

```
public static double getTotal ( Priceable [] cart ) {
    double total = 0.0;
    for ( int i = 0; i < cart.length ; i++ )
```

```
total += cart [i].getPrice ();
```

```
return total;
```

```
}
```

Note that we can pass in Bananas, Eggs, Water, and many other kinds of objects in a Priceable array as long as they all implement this interface. Even though it is impossible to create an object with an interface type, we can make as many references to it as we want. ■

10.4 Advanced Syntax: Local and anonymous classes

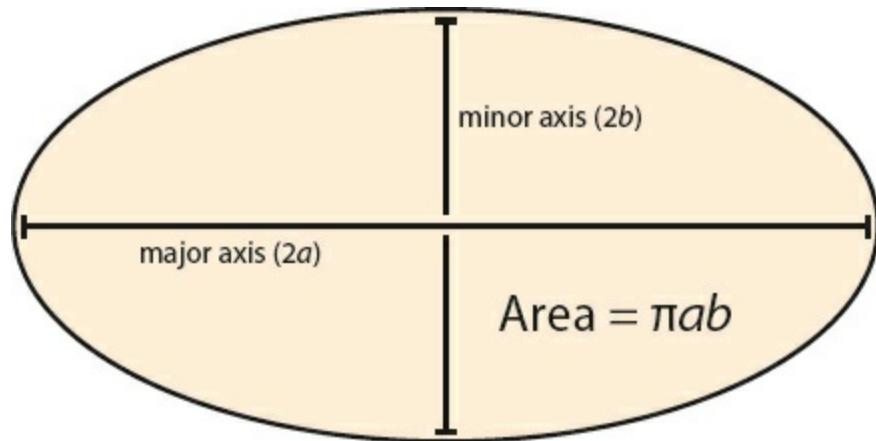
If you have not read [Section 9.5](#), you may want to look over that material to be sure that you understand what nested classes and inner classes are. Recall that a normal inner class is declared inside of another class, but it is also legal to declare a class inside of a method. Such a class is called a *local class*. Under some circumstances, it is useful to create another kind of inner class that has no name. That kind of class is called an *anonymous class*.

Both kinds of classes are inner classes. They can access fields and methods, even if they are marked **private**. Like other inner classes, they are not allowed to declare **static** variables other than constants. We bring up these kinds of classes in this chapter because they are commonly used to create a class with a narrow purpose that implements a required interface.

10.4.1 Local classes

A local class declaration looks like any other class declaration except that it occurs within a method. The name of a local class only has meaning inside the method where it is defined. Because the scope of the name is only the method, a local class cannot have access modifiers such as **public**, **private**, or **protected** applied to it.

Consider the following method in which an Ellipse class is defined locally. Recall that an ellipse (or oval) has a major (long) axis and a minor (short) axis. The area of an ellipse is half its major axis times half its minor axis times π . (Because the major and minor axes of a circle are its diameter, this formula becomes πr^2 in that case.)



```
public static void createEllipse ( double a1, double a2 ) {
```

```

class Ellipse {
    private double axis1 ;
    private double axis2 ;

    public Ellipse ( double axis1, double axis2 ) {
        this.axis1 = axis1 ;
        this.axis2 = axis2 ;
    }

    public double getArea () {
        return Math.PI *0.5* axis1 *0.5* axis2 ;
    }
}

Ellipse e = new Ellipse (a1, a2);
System.out.println ("The ellipse has area " + e.getArea ());
}

```

This Ellipse class cannot be referred to by any other methods. Since an Ellipse class might be useful in other code, a top-level class would make more sense than this local class. For this reason, local classes are not commonly used.

However, we can make local classes more useful if they implement interfaces. Consider the following interface which can be implemented by any shape that returns its area.

```

1 public interface AreaGettable {
2     double getArea ();
3 }

```

The method below takes an array of AreaGettable objects and sums their areas.

```

public static double sumAreas ( AreaGettable [] shapes ) {
    double sum = 0;
    for ( int i = 0; i < shapes.length ; i++ )
        sum += shapes [i].getArea ();

    return sum ;
}

```

If we create a local class that implements AreaGettable, we can use it in conjunction with the sumAreas() method. In the following method, we will expand the local Ellipse class in this way and fill an array with 100 Ellipse instances which can then be passed to sumAreas().

```

public static void createEllipses () {
    class Ellipse implements AreaGettable {
        private double axis1 ;
        private double axis2 ;

```

```

public Ellipse ( double axis1, double axis2 ) {
    this.axis1 = axis1 ;
    this.axis2 = axis2 ;
}

public double getArea () {
    return Math.PI *0.5* axis1 *0.5* axis2 ;
}

}

AreaGettable [] ellipses = new AreaGettable [100];

for ( int i = 0; i < 100; i++ )
    ellipses [i] = new Ellipse ( Math.random () * 25.0,
        Math.random () * 25.0 );

double sum = sumAreas ( ellipses );
System.out.println ("The total area is " + sum );
}

```

Even though the `Ellipse` class had the `getArea()` method before, the compiler would not have allowed us to store `Ellipse` references in an `AreaGettable` array until we marked the `Ellipse` class as implementing `AreaGettable`. As in [Example 10.1](#), we used an array of the interface type.

10.4.2 Anonymous classes

This second `Ellipse` class is more useful since objects with its type can be passed to other methods as an `AreaGettable` reference, but declaring the class locally provides few benefits over a top-level class. Indeed, local classes are seldom preferable to top-level classes. Although anonymous classes behave like local classes, they conveniently can be created at any point.

An anonymous class has no name. It is created on the fly from some interface or parent class and can be stored into a reference with that type. In the following example, we modify the `createEllipses()` method so that it creates an anonymous class which behaves exactly like the `Ellipse` class and implements the `AreaGettable` interface.

```

public static void createEllipses () {
    AreaGettable [] ellipses = new AreaGettable [100];

    for ( int i = 0; i < 100; i++ ) {
        final double value1 = Math.random ();
        final double value2 = Math.random ();

        ellipses [i] = new AreaGettable () {
            private double axis1 = value1 ;

```

```

private double axis2 = value2 ;

public double getArea () {
    return Math.PI *0.5* axis1 *0.5* axis2 ;
}

};

double sum = sumAreas ( ellipses );
System.out.println ("The total area is " + sum );
}

```

The syntax for creating an anonymous class is ugly. First, you use the `new` keyword followed by the name of the interface or parent class you want to create the anonymous class from. Next, you put the arguments to the parent class constructor inside of parentheses or leave empty parentheses for an interface. Finally, you open a set of braces and fill in the body for your anonymous class. When defining an anonymous class, the entire body is crammed into a single statement, and you will often need to complete that statement with a semicolon (`;`).

Anonymous classes do not have constructors. If you need a constructor, you will have to create a local class. Constructors are usually not necessary since both local and anonymous classes can see local variables and fields and use those to initialize values. Although any fields can be used, local variables must be marked `final` (as shown above) if their values will be used by local or anonymous classes. This restriction prevents local variables from being changed unpredictably by methods in the local class.

It may not be easy to see why anonymous classes are useful. Both the Java API and libraries written by other programmers have many methods that require parameters whose type implements a particular interface. Without anonymous classes, you would have to define a named class and instantiate it to supply such a method with an object with the required capabilities.

Using anonymous classes, you can create such an object in one step, right where you need it. This practice is commonly used for creating *listeners* for GUIs. A listener is an object that does the right action when a particular event happens. If you need many different listeners in one program, it can be convenient to create anonymous classes that can handle each event rather than defining many named classes which each have a single, narrow purpose. We will use this technique in [Chapter 15](#).

10.5 Solution: Sort it out

It is not difficult to move from totaling the value of items as we did in [Example 10.1](#) to sorting them. Refer to the following class diagram as we explain our solution to the sorting problem posed at the beginning of the chapter. Dotted lines are used to show the `implements` relationship.

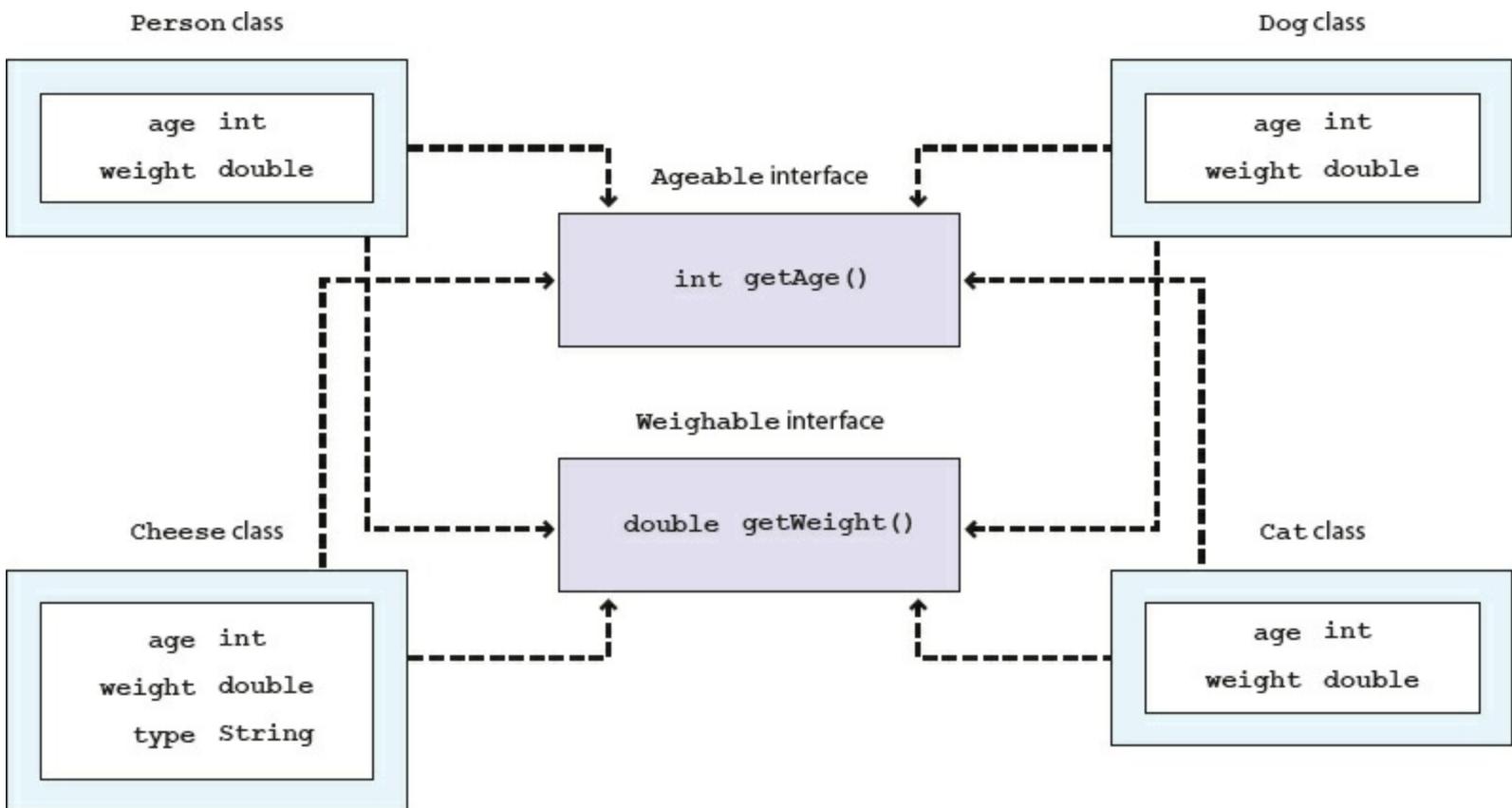


Figure 10.1: Sorting relationships

We will start with the definitions of the two interfaces we will use to compare objects.

```

1 public interface Ageable {
2     int getAge();
3 }
  
```

```

1 public interface Weighable {
2     double getWeight();
3 }
  
```

Classes implementing these two interfaces will be able to give their age and weight independently. The next step is to create the Dog, Cat, Person, and Cheese classes which can do so.

We'll see in [Chapter 11](#) that the Dog, Cat, and Person classes could inherit from a common ancestor (such as Creature or Mammal) which implements the Ageable and Weighable interfaces. That design could reduce the total amount of code needed. For now, each class will have to implement both interfaces directly.

```

1 public class Dog implements Ageable, Weighable {
2     private int age;
3     private double weight ;
4
5     public Dog( int age, double weight ) {
6         this.age = age ;
7         this.weight = weight ;
  
```

```
8      }
9
10     public int getAge () { return age *7; }
11
12     public int getWeight () { return weight ; }
13 }
```

```
1 public class Cat implements Ageable, Weighable {
2     private int age;
3     private double weight ;
4
5     public Cat ( int age, double weight ) {
6         this.age = age ;
7         this.weight = weight ;
8     }
9
10    public int getAge () { return age /9; }
11
12    public int getWeight () { return weight ; }
13 }
```

```
1 public class Person implements Ageable, Weighable {
2     private int age;
3     private double weight ;
4
5     public Person ( int age, double weight ) {
6         this.age = age ;
7         this.weight = weight ;
8     }
9
10    public int getAge () { return age ; }
11
12    public int getWeight () { return weight ; }
13 }
```

```
1 public class Cheese implements Ageable, Weighable {
2     private int age;
3     private double weight ;
4     private String type ;
5
6     public Cheese ( int age, double weight, String type ) {
7         this.age = age ;
8         this.weight = weight ;
9         this.type = type ;
```

```

10    }
11
12    public int getAge () { return age ; }
13
14    public int getWeight () { return weight ; }
15
16    public String getType () { return type ; }
17 }

```

With the classes in place, we can assume that client code will instantiate some objects and perform operations on them. All that is necessary is to write the method that will do the sorting. We can wrap the bubble code given earlier in a method body with only a few changes to generalize the sort beyond int values.

```

public void sort ( Object [] array, boolean age ) {
    boolean swapped = true ;
    Object temp ;
    while ( swapped ) {
        swapped = false ;
        for ( int i = 0; i < array.length - 1; i++ )
            if( outOfOrder ( array [i], array [i + 1], age ) {
                temp = array [i];
                array [i] = array [i + 1];
                array [i + 1] = temp ;
                swapped = true ;
            }
    }
}

```

In this method, the `boolean` `age` is `true` if we are sorting by age and `false` if we are sorting by weight. Note that the array and temp have the `Object` type. Recall that any object can be stored in a reference of type `Object`.

The only other change we needed was to replace the greater-than comparison (`>`) with the `outOfOrder()` method, which we define below.

```

public boolean outOfOrder ( Object o1, Object o2, boolean age ) {
    if( age ) {
        Ageable age1 = ( Ageable )o1;
        Ageable age2 = ( Ageable )o2;
        return age1.getAge () > age2.getAge ();
    }
    else {
        Weighable weight1 = ( Weighable )o1;
        Weighable weight2 = ( Weighable )o2;
        return weight1.getWeight () > weight2.getWeight ();
    }
}

```

```
}
```

Even though we have designed our program for objects that implement both the Ageable and Weighable interfaces, the compiler only sees Object references in the array. Thus, we must cast each object to the appropriate interface type to do the comparison. There is a danger that a user will pass in an array with objects which do not implement both Ageable and Weighable, causing a ClassCastException. To allow for universal sorting methods, the Java API defines a Comparable interface which can be implemented by any class which requires sorting. With Java 5 and higher, the Comparable interface uses generics to be more type-safe, so we will not discuss how to use this interface until we cover generics in [Chapter 18](#).

10.6 Concurrency: Interfaces

As we discussed in [Section 10.2](#), implementing an interface means making a promise to have public methods with the signatures specified in the interface definition. Making a promise seems only tangentially related to having multiple threads of execution. Indeed, interfaces and concurrency do not overlap a great deal, but there are two important areas where they affect one another.

The first is that a special interface called the Runnable interface can be used to create new threads of execution. Runnable is a very simple interface, containing the single signature `void run()`. This means that any object with a `run()` method that takes no arguments and returns no values can be used to create a thread of execution. This makes intuitive sense. A regular program has a single starting place, the `main()` method. If we want to run additional threads, some method needs to be marked as a starting place for each one. For more information about using the Runnable interface, refer to [Section 13.4.5](#).

The second connection between interfaces and concurrency is more philosophical. What can you specify in an interface? The rules for interfaces in Java are relatively limited: You can require a class to have a public instance method with specific parameters and a specific return type. Java interfaces do not allow you to require a static method.

In [Chapter 14](#), we will discuss a key way to make classes thread-safe by using the `synchronized` keyword. Like `static`, Java does not allow an interface to specify whether a method is synchronized. Thus, it is impossible to use an interface to guarantee that a method will be thread-safe.

As with all interface usage, this restriction cuts both ways: If you are designing an interface, there is no way that you can guarantee that implementing classes use synchronized methods. On the other hand, if you are implementing an interface, the designer may hope that your class uses synchronized (or otherwise thread-safe) methods, but the interface cannot force you to do so. Whenever thread-safety is an issue, make sure you read (or write) the documentation carefully. Since there is no way to force programmers to use the `synchronized` keyword, the documentation may be the only guide.

Exercises

Conceptual Problems

10.1 What is the purpose of an interface?

10.2 Why implement an interface when it puts additional requirements on a class yet adds no functionality?

10.3 Is it legal to have methods marked **private** or **protected** in an interface? Why did the designers of Java make this choice?

10.4 What is the **instanceof** keyword used for? Why is it useful in the context of interfaces?

10.5 What kind of programming error causes a **ClassCastException**?

10.6 Create an interface called **ColorWavelengths** that only contains constants storing the wavelengths in nanometers for each of the seven colors of light, as given below.

Color	Wavelength (nm)
Red	680
Orange	605
Yellow	580
Green	545
Blue	473
Indigo	430
Violet	415

10.7 Write an interface called **Clock** that specifies the functionality a clock should have. Remember that the classes that implement the clock may tell time in different ways (hourglass, water clock, mechanical movement, atomic clock), but they must share the basic functionality that you specify.

10.8 There are six compiler errors in the following interface. Name each one and explain why it is an error.

```
public interface Singable {  
    public int SOPRANO = 1;  
    public static int ALTO = 2;  
  
    public void sing();  
    private String chant();  
    public boolean hasDeepVoice() {  
        return false;  
    }  
    public static boolean hasPerfectPitch();  
    public synchronized void tune( int frequency );  
}
```

10.9 Consider the interface defined below.

```
public interface Explodable {  
    boolean explode ( double megatons );  
}
```

Which of the following classes properly implement Explodable?

```
public class Dynamite implements Explodable {  
    public boolean explode () {  
        System.out.println (" BOOM !");  
        return true ;  
    }  
}
```

```
public class AtomicBomb implements Explodable {  
    public boolean explode ( double size ) {  
        System.out.println ("A huge " + size +  
            " megaton blast shakes the earth !");  
        return true ;  
    }  
}  
public class Grenade {  
    public boolean explode ( double megatons ) {  
        return true ;  
    }  
}
```

```
public class Firecracker implements Explodable {  
    private boolean explode ( double megatons ) {  
        return ( megatons < 0.0000001) ;  
    }  
}
```

10.10 Write a single class that correctly implements the following three interfaces.

```
public interface Laughable {  
    boolean laugh ( int times );  
}
```

```
public interface Cryable {  
    void cry ( int tears, boolean moaning );  
}
```

```
public interface Shoutable {  
    void laugh ( double volume, String words );  
}
```

10.11 If you are sorting a list of items n elements long using bubble sort, what is the minimum number of passes you would need to be sure the list is sorted, assuming the worst possible ordering of items to start with? (Hint: Imagine the list is in backwards order.) What is the minimum number of passes if the list is already sorted?

Programming Practice

10.12 Add client code that randomly creates the objects needing sorting in the solution from [Section 10.5](#). Design and include additional classes Wine and Tortoise that both implement Ageable and Weighable. Add `toString()` methods to each class so that their contents can be easily output. Make sure that you print out the list of objects after sorting to test your implementation.

10.13 Refer to the sort given as a solution in [Section 10.5](#). Add another `boolean` to the parameters of the sort which specifies whether the sort is ascending or descending. Make the needed changes throughout the code to add this functionality.

Concurrency

10.14 After learning about threads in [Chapter 13](#), refer to the simple bubble sort from [Section 10.1](#). The goal is now to parallelize the sort. Write some code which will generate an array of random `int` values. Design your code so that you can spawn n threads. Partition the single array into n arrays and map one partition to each thread. Use your bubble sort implementation to sort each partition. Finally, merge the arrays back together, in sorted order, into one final array. For now, just use one thread (ideally the main thread) to do the merge.

The merge operation is a simple idea, but it is easy to make mistakes in its implementation. The idea is to have three indexes, one for each of the two arrays you are merging and one for the result array. Always take the smaller (or larger, if sorting in descending order) index value from the two arrays and put it in the result. Then increment the index from the array you took the data from as well as the index of the result array. Make sure that you are careful not to go beyond the end of the arrays which are being merged.

Experiments

Concurrency

10.15 Once you have implemented the sort in parallel from Exercise 10.14, time it against the sequential version. Try two, four, and eight different threads. Be sure to create one random array and use copies of the same array for both the parallel and sequential versions. Be careful not to sort an array that is already sorted! Try array sizes of 1,000, 100,000, and 1,000,000. Did the performance increase? Was it as much as you expected?

Chapter 11

Inheritance

Your children are not your children.

They are the sons and daughters of Life's longing for itself.

They come through you but not from you,

And though they are with you yet they belong not to you.

—Khalil Gibran

11.1 Problem: Boolean circuits

In [Chapter 4](#), we talked extensively about Boolean algebra and how it can be applied to if statements in order to control the flow of execution in your program. The commands that we give in software must be executed by hardware in order to have an effect. It should not be surprising that computer hardware is built out of digital circuits that behave according to the same rules as Boolean logic. Each component of these circuits is called a *logic gate*. There are logic gates corresponding to all the Boolean operations you are used to: AND, OR, XOR, NOT, and others.

The output of an AND gate is the result of performing a logical AND on its inputs. That is, the output is true if and only if both of the inputs are true. The same correlation exists between each gate and the Boolean operator with the same name. At the level of circuitry, a 1 (or “on”) is often used to represent true, and a 0 (or “off”) is often used to represent false. Modern computer circuitry is built almost entirely out of such gates, performing addition, subtraction, and all other basic operations as complicated combinations of logic gates, where each digit of every number is a 1 or 0 determined by the circuit.

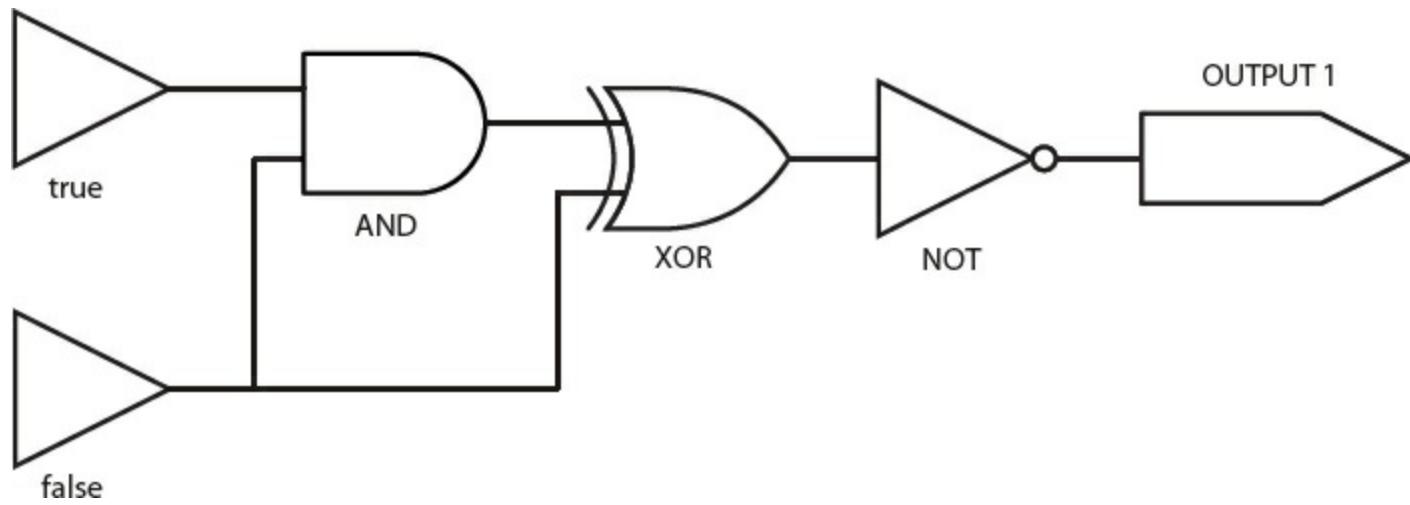
Because these circuits can become large and unwieldy, your problem is to write a Java program that will allow a user to specify the design of such a circuit and then see what its output is. The input to this program will be a text file redirected to standard input that gives the number of gates of the circuit, lists what each gate is, then gives a list of connections between them.

The following is an example of input to make a circuit with six components.

```
6
true
false
AND
XOR
NOT
OUTPUT 1
```

```
2 0 1  
3 2 1  
4 3  
5 4
```

The first line specifies the total number of components. The next two components give either true or false inputs, depending on their name. The AND and the XOR correspond to gates of the same name which each take two inputs. The NOT corresponds to a NOT gate with a single input. Finally OUTPUT 1 is the single output of this circuit that we are interested in. After the list of gates is a list of how they are connected. The line 2 0 1 specifies that gate at index 2, which happens to be an AND gate, has an input from the gate at index 0 and an input from the gate at index 1. In other words, the AND gate has one true input and one false input. The final circuit produced would look like the following.



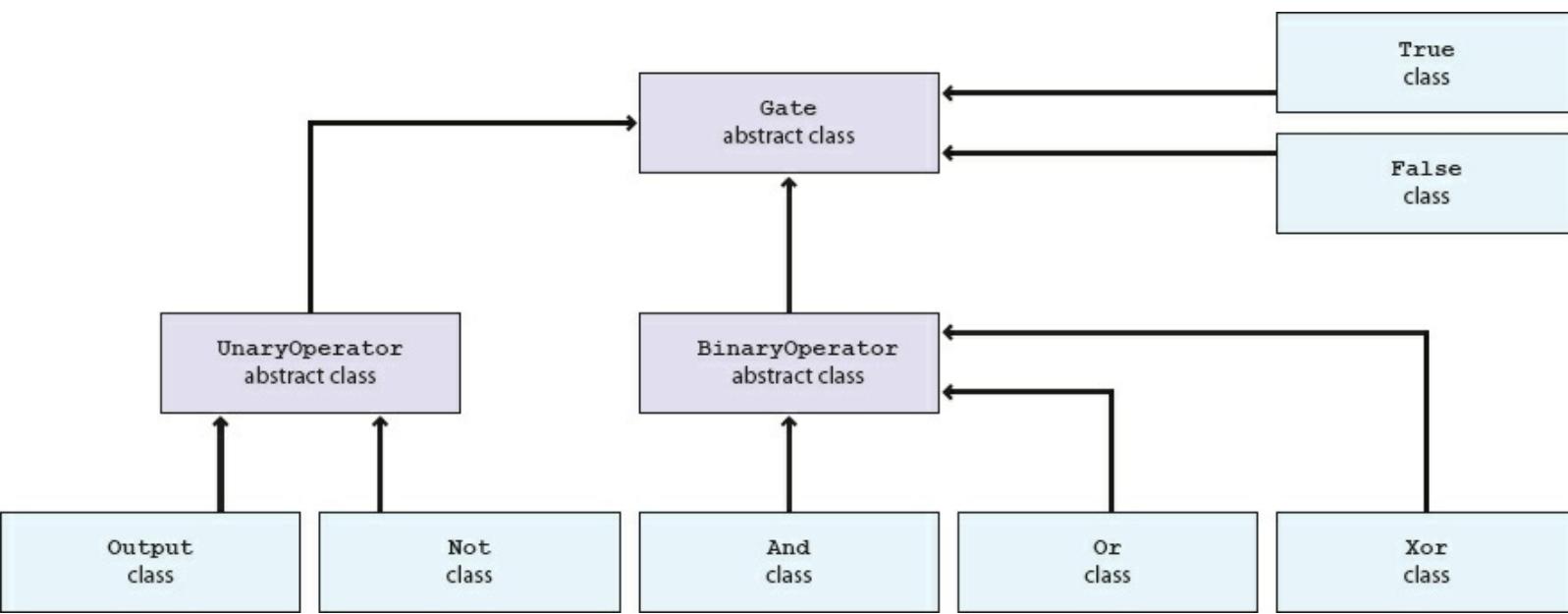
We are only interested in the value of any OUTPUT gates. Thus, the program that is simulating this circuit would print the following.

```
OUTPUT 1: true
```

There are many different ways to implement a solution to this problem. The total number of gates is given as the first input. Thus, you can make an array of gates. When you go to connect them, the indexes given in the input will map naturally onto the gates in the array. But, what type should the array be? You could create a Gate class that could do the work of any conceivable gate, but the implementation would be awkward. All gates would have to have two inputs even if they don't need any. Adding different kinds of gates later (like a NAND, for example) would mean rewriting the Gate class.

Instead, a cleaner approach to the solution is to use *inheritance*. In object oriented languages like Java, inheritance is a process that allows you to create a new specialized class from a more general, preexisting class.

We recommend the following inheritance hierarchy, in which the arrows point from each child class to its parent class.



As you can see, every class inherits from the **Gate** class. If your array can be of type **Gate**, it will be able to hold objects of any child of **Gate**. **UnaryOperator**, **BinaryOperator**, **True**, and **False** are children of the basic **Gate** class. Then, **Output** and **Not** are children of **UnaryOperator** since each only has a single input. Naturally, **And**, **Or**, and **Xor** are children of **BinaryOperator**, since they all have two inputs.

If this large collection of classes seems bewildering, do not be discouraged. Each is very short and easy to write. We'll explain what the inheritance relationship means and how to use it in the next few sections.

11.2 Concepts: Refining classes

Here we give a brief overview of inheritance that will give us enough information to continue onward. We cover some of the deeper areas of the subject in [Chapter 17](#).

11.2.1 Basic inheritance

The process of creating an inherited class out of an existing class is called *inheriting* a class, *deriving* a class, or simply *subclassing*. The class that already exists is called a *parent class*, *base class*, or *superclass* and the new class is called a *child class*, *derived class*, or *subclass*.

When you create a child class from a parent class, the child class inherits all of its fields and methods. Thus, you can use a child class object anywhere you would use the parent class object. This relationship explains the terms *superclass* and *subclass*: Since you can treat any subclass object as if it were a superclass object, the superclass type can be thought of as a superset including all subclass objects.

Others have noted that the names *superclass* and *subclass* sound misleading because the subclass can usually do more than the superclass. From that perspective, the child class has a superset of the fields and methods of its parent class. To avoid confusion, we favor the terminology of parent and child classes.

11.2.2 Adding functionality

When creating a child class, a programmer will normally add functionality above and beyond the original parent class. Otherwise, there is little point in creating a child class. For example, a simple Fish class might be able to do things like swim and feed. A child Flounder class has the additional ability to camouflage itself. Another child class, the Shark class, adds the ability to eat other Fish objects.

By adding a few methods, we can create a new class with special abilities without interfering with the basic functionality of the underlying class.

11.2.3 Code reuse

Of course, a programmer who wished to program a Shark class could simply copy and paste in all the code from the Fish class and then make the necessary additions. In many ways the evolution of modern programming languages has been specifically to minimize any copying and pasting.

Old mistakes are propagated with copying and pasting. When discovered, they must be fixed in several different locations. New mistakes can also be introduced by cutting and pasting. Instead, we wish to guarantee that working code from a parent class continues to work in a child class. Ideally, code from parent classes will not need to be debugged a second time when a child class is created.

Even without the issue of errors introduced by copying and pasting, the total amount of code increases. By minimizing the amount of code, issues of performance and storage can be improved, but not always. Object oriented languages have taken criticism for low speed and high memory use due to the additional complexities of objects and inheritance, but compiler optimization, good library design, and improved JVM performance have brought Java a long way in this area.

11.3 Syntax: Inheritance in Java

In this section we discuss the mechanism for creating a child class in Java using the `extends` keyword. Then, we discuss access restriction and visibility, constructor issues, the Object class, and overriding methods.

11.3.1 The `extends` keyword

In order to make a child class in Java, we use the `extends` keyword. Let's give an example using the Fish class defined below. This class creates a basic fish that can swim, feed, and die. We can check its color, location, and whether or not it is alive. When it runs out of energy, it dies.

```
1 import java.awt.*;
2
3 public class Fish {
4     protected Color color = Color.GRAY ;
5     private double location = 0.0;
6     private double energy = 100.0;
7     private boolean alive = true ;
```

```

8
9     public Color getColor () { return color ; }
10    public double getLocation () { return location ; }
11    public boolean isAlive () { return alive ; }
12
13    public void swim () {
14        if( alive ) {
15            location += 0.5;
16            energy -= 0.25;
17        }
18        if( energy <= 0.0 )
19            die ();
20    }
21
22    public void feed () { energy = 100.0; }
23    public void die () { alive = false ; }
24 }
```

From here we can create a child class called BoringFish that does exactly what Fish does. To do so, we use the `extends` keyword after the new class name, followed by the parent class name (in this case Fish), followed by the body of the class.

```

public class BoringFish extends Fish {

}
```

Just as we are allowed to make an empty class, we are allowed to make an inherited class and add nothing, but doing so is pointless. Instead, we can make a Flounder class that can change its color.

```

1  public class Flounder extends Fish {
2      public void setColor ( Color newColor ) { color = newColor ; }
3 }
```

The Flounder class can do everything a Fish can: It can swim, feed, and die. Now, we add the additional ability of changing color, since flounders are famous for their ability to mimic the ocean floor they swim over. Note that the color field in the Fish class has the `protected` access modifier, not `private`. We'll come back to this point.

Here is a Shark class that extends Fish in another ways, by adding the capability of eating other Fish.

```

1  public class Shark extends Fish {
2      public void eat( Fish fish ) {
3          fish.die ();
4          feed ();
5      }
```

Here we have added an eat() method that takes another Fish object as a parameter. First, the Fish parameter is killed; then the eat() method calls feed(), restoring the energy of the Shark object. Note that the Shark object is able to call the feed() method even though it isn't defined inside of Shark. Because it inherits from Fish, it has a version of feed().

Single inheritance only

Particularly if you have programmed in C++, you might be wondering if it is possible to have one class inherit from **multiple** classes in Java. In multiple inheritance, a single class can have many different parents. Since C++ supports multiple inheritance, you can have a SharkAlligatorMan class in that language that inherits from the Shark, Alligator, and Human classes. If you go back to the sorting problem from [Chapter 10](#), multiple inheritance would allow us to solve the problem with an Age class and a Weight class from which Dog, Cat, Person, and Cheese all inherit.

However, the designers of Java decided not to allow multiple inheritance, perhaps for this reason: Imagine a River class with a run() method and a Politician class with a run() method. It seems strange to create a class which is both a river and politician, but there is no rule in C++ which makes doing so impossible. If you did have a RiverPolitician class which inherits from both, what would happen when you call the run() method? How would the RiverPolitician class know which of its parents' methods to pick? Surely, the way that a politician runs for office is very different from the way a river runs along its banks.

If you find yourself in a situation where you want to use multiple inheritance in Java, try to reformulate your class hierarchy into one where your classes implement multiple interfaces. Recall that multiples interfaces **can** be implemented by a single class in Java, and like multiple inheritance, this practice allows a single class to be used in wildly different contexts.

Interfaces using extends

The **extends** keyword is not limited to classes. It is possible for an interface to extend another interface. In fact, an interface can extend any number of other interfaces. As when a class implements multiple interfaces, each interface in an extends list is separated by commas.

When an interface extends other interfaces, it includes all the methods (and constants) they define. If a class implements an interface that extends other interfaces, it must contain versions of all the methods specified by **all** the interfaces. Recall the Ageable and Weighable interfaces from [Chapter 10](#), which specified the getAge() and getWeight() methods, respectively. We could create an interface that required both of these methods by extending Ageable and Weighable.

```
public interface AgeableAndWeighable extends Ageable, Weighable {  
}
```

We could add additional methods to the AgeableAndWeighable interface, but even empty it will enforce the contracts defined by both Ageable and Weighable. It is generally not necessary to create an interface that extends other interfaces, since a class could implement each of the individual interfaces. Nevertheless, it can be used as a convenience to save typing or to create a reference type

with certain guaranteed abilities.

Note that a class can never extend an interface. Likewise, an interface cannot extend a class or implement another interface.

11.3.2 Access restriction and visibility

The Shark example above gives an example of inheritance in which the child class only calls methods of the parent class and does not interfere with the fields of the parent class. Generally, this is a good thing because it protects the state of the parent class from getting corrupted. However, it is not always possible. If we return to the earlier Flounder example, we had to change the color field directly, since there was no mutator to change it.

Perhaps the Fish class was poorly designed because it did not have a color mutator. On the other hand, most fish cannot change their color, so it might be good design to prevent outside code from changing the color field with such a mutator. There are no absolute rules for making these kinds of decisions.

We introduced access modifiers in [Section 9.3.4](#), but inheritance gives them new meaning. Recall that the access modifier for the color field of Fish was **protected**. A field or method with the **protected** modifier can be accessed by all child classes (as well as classes in the same package). If the modifier for color was **private**, the Flounder class would not be able to change it directly.

In the Shark class, it **must** use mutators to change the value of its own energy and the alive field of fish since they are both marked **private**. It is generally preferable to use mutator and accessor methods whenever possible, even within the same class, so that fields are not inadvertently corrupted.

11.3.3 Constructors

When you create a child class, you can think of it as if a copy of the parent class exists inside of the child. When you create an object from a child class, how do you properly initialize the fields inside the parent class?

As we discussed in [Chapter 9](#), every class has a constructor, even if it is a default one created for you. Whenever the constructor for a child class is invoked, the constructor for the parent class is invoked as well. If the parent class is also the child of some other class, that grandparent class will have its constructor invoked as well. This chain of constructors will continue, reaching all the way back to the ultimate ancestor, Object.

When writing the constructor for a child class, the first line of it should be the call to the parent constructor. If you don't explicitly call the parent constructor, its default (no parameter) constructor will be called. If the parent class does not have a default constructor, then leaving off an appropriate call to a parent constructor will result in a compiler error. Consider the following two classes.

```
1 public class Parent {  
2     private String name ;  
3     public Parent ( String name ) { this.name = name ; }  
4     public String getName () { return name ; }  
5 }
```

```
1 public class Child extends Parent {  
2     public Child ( String name ) {  
3         super ( " Baby " + name );  
4     }  
5 }
```

As shown above, the `super` keyword is used to call the constructor of a parent class. The `Child` constructor takes a name and prepends the String “`Baby`” to it before passing it on to the `Parent` constructor.

In a similar way, the `this` keyword can be used to call another constructor in the same class, provided that a constructor to the parent class is eventually reached. For example, we could add the following constructor to the `Child` class.

```
public Child () {  
    this ( " Unknown " );  
}
```

This second constructor will be called whenever a new `Child` object is instantiated without any arguments. It will supply the String “`Unknown`” to the other constructor, which will add “`Baby`” and pass it on to the `Parent` class.

11.3.4 Overriding methods and hiding fields

Sometimes a parent method does not provide all the power you want in the child class. It is possible to *override* a parent method in the child class. Then, when that method is called on child objects, the new method will be called. The new method has exactly the same name and parameters. The return type must be either exactly the same or a child class of the original return type.

We can return to the `Fish` class example and make a new kind of fish that never moves.

```
1 public class LazyFish extends Fish {  
2     public void swim () {  
3         System.out.println ("I think I'll just sit here.");  
4     }  
5 }
```

Whenever someone calls the `swim()` method on a `LazyFish` object, it will just announce that it is going to sit where it is. Its location is not updated and its energy does not change.

On the other hand, we could create another child class that swims twice as fast as the original `Fish`.

```
1 public class FastFish extends Fish {  
2     public void swim () {  
3         super.swim ();  
4         super.swim ();  
5     }  
6 }
```

Every time `swim()` is called on objects of type `FastFish`, those objects will call the `swim()` method from `Fish` twice. Thus, this fish will move twice as fast (and consume twice as much energy). Because the location and energy fields are `private`, we must use methods from `Fish` to affect them. Note the use of the keyword `super`, allowing us to specify that we want to call the `swim()` method from `Fish` and not just call the same method from `FastFish` again. Using the `super` keyword, we can call methods from the parent. If the parent did not override a method from an ancestor class, we can still use `super` to call a method from the last class that did implement the method. However, Java does not allow us to skip over a parent method to call a grandparent method if there is an implementation in the parent class. In other words, there is no way to call something like a `super.super.swim()` method.

Just as methods are overridden, fields are *hidden*. It is perfectly legal to declare a field with the same name as a field from a parent class, but the new field will then be used instead of the old one.

```
1 public class A {  
2     protected int a;  
3     public int getA () { return a; }  
4     public void setA ( int value ) { a = value ; }  
5 }
```

```
1 public class B extends A {  
2     protected int a;  
3     public void setA ( int value ) { a = value ; }  
4 }
```

Class `B` is a child of class `A` and declares a field called `a`, hiding a field of the same name from `A`. However, which `a` is which can cause some confusion. Consider the following fragment of code.

```
A objectA = new A();  
B objectB = new B();  
objectA.setA ( 5 );  
objectB.setA ( 10 );  
System.out.println ("A = " + objectA.getA () );  
System.out.println ("B = " + objectB.getA () );
```

The output of this code is:

`A = 5`

`B = 0`

Calling the `setA()` method on an `A` object sets the `a` field inside of `A`. Calling the overridden `setA()` method on a `B` object sets the `a` field inside of `B`, but since the `getA()` method has not been overridden, the `a` field from the `A` parent class part of `B` is returned. Since that a field in `B` has not been given a value, it still has the default value of 0. Both a fields exist inside of `B`, but the methods are poorly designed, leaving one field capable only of being set and the other capable only of being retrieved.

11.3.5 The Object class

You may not have realized it, but every class you created in Java used inheritance. To provide uniformity, the designers of Java made every class the child (or grandchild or great-grandchild ...) of a class called Object. When you omit the `extends` clause in a class definition, you are making that class a direct child of Object.

As a consequence, **all** classes in Java are guaranteed to have the following methods.

Method	Purpose
<code>clone()</code>	Make a separate copy of an object.
<code>equals()</code>	Determine if two objects are the same.
<code>finalize()</code>	Perform cleanup when an object is garbage collected. Similar to a destructor in C++. Rarely used.
<code>getClass()</code>	Find out what the class type of a given object is.
<code>hashCode()</code>	Get the hash code for an object, useful for making hash tables of objects.
<code>notify()</code>	Used for synchronization with threaded programs. More in Chapter 14.
<code>notifyAll()</code>	Same as previous.
<code>toString()</code>	Get a String representation of the given object.
<code>wait()</code>	Used with <code>notify()</code> and <code>notifyAll()</code> .

Java provides basic implementations for most of these, but if you want them to work well for your object, you will have to override some of them with appropriate methods. For example, the Object version of `toString()` returns the virtual address of the object in JVM memory.

Nevertheless, API classes usually have good `equals()` and `toString()` methods. Aside from making a few useful methods available, having a common ancestor for all classes means that you can store any object in an Object reference. An array of type Object can hold anything, provided that you know how to retrieve it. We discuss the finer points of inheritance and polymorphism in [Chapter 17](#) and how to build lists and other data structures using Object references in [Chapter 18](#).

11.4 Examples: Problem solving with inheritance

Here are two extended examples showing how we can use inheritance to solve problems. First, we revisit the student roster example from [Chapter 9](#) and then move onto an inheritance hierarchy of polygons.

Example 11.1: Graduate student roster

The Student class we create in [Chapter 9](#) is useful but works only for undergraduate students. With only a few additions, we can make it suitable for graduate students as well. First, lets take another look at the Student class.

Program 11.1: Basic student class, designed for undergraduates. (Student.java)

```

2  public static final String [] YEARS =
3      {"Freshman", "Sophomore", "Junior", "Senior"};
4
5  private String name;
6  private int year;
7  private double GPA;
8
9  public Student ( String name, int year, double GPA ) {
10     setName ( name );
11     setYear ( year );
12     setGPA ( GPA );
13 }
14
15     public void setName ( String name ) { this.name = name; }
16     public void setYear ( int year ) { this.year = year; }
17
18     public void setGPA ( double GPA ) {
19         if( GPA >= 0 && GPA <= 4.0 )
20             this.GPA = GPA;
21         else
22             System.out.println (" Invalid GPA : " + GPA );
23 }
24
25     public String getName () { return name; }
26     public int getYear () { return year; }
27     public double getGPA () { return GPA; }
28
29     public String toString () {
30         return name + "\t" + YEARS [ year ] +
31             "\t" + GPA;
32     }

```

We want to create a GraduateStudent class that inherits from Student. We need to add a thesis topic for each graduate student. Likewise, we need to update the `toString()` method so that outputs the appropriate data. We use 4 as the year value for graduate students.

Program 11.2: Class extending Student to add graduate student capabilities. (GraduateStudent.java)

```

1  public class GraduateStudent extends Student {
2      private String topic;
3
4      public GraduateStudent ( String name, double GPA, String topic )
5          {
6              super ( name, 4, GPA );

```

```

6     setTopic ( topic );
7 }
8
9     public void setTopic ( String topic ) { this.topic = topic ; }
10
11    public String toString () {
12        return getName () + "\t" + "Graduate " +
13            "\t" + getGPA () + "\tTopic :" + topic ;
14    }
15 }
```

Because we are inheriting most of the fields we need, we only need to declare the topic field. Then, in the GraduateStudent constructor, we call the parent constructor with the name, year, and GPA and then set topic to the input value.

Finally, we override the `toString()` method so that “Graduate” and the thesis topic are output. Note that we must use the `getName()` and `getGPA()` accessors since the actual values are `private`.

Most code that uses Student objects should be able to incorporate GraduateStudent objects easily. Code that creates Student objects from input will need slight modifications to handle the thesis topic. Also, old code that only expects values of 0, 1, 2, or 3 for year may need to be modified so that it doesn’t break. ■

Example 11.2: Polygons

Let’s examine a class hierarchy used to create several different polygons. Our base class needs to be general. It can represent any kind of closed polygon, using an array of Point objects. The Point class is a way to package up x and y values of type `int`. Each coordinate in the array gives the next vertex of the polygon.

```

1 import java.awt.*;
2
3 public class Polygon {
4     protected Point [] points ;
5
6     public Polygon ( Point [] points ) {
7         this.points = points ;
8     }
```

The `import` statement allows us to use the Point class as well as the Graphics class. Our array of type Point is declared `protected` so that the child classes we want to create can access it directly. The constructor takes an array of type Point and stores it.

```

10    public double getPerimeter () {
11        double perimeter = 0.0;
12        for ( int i = 0; i < points.length - 1; i++ )
13            perimeter += points [i].distance ( points [i + 1]) ;
```

```

14     perimeter += points [0].distance ( points [ points.length - 1] );
15     return perimeter ;
16 }
17
18 public void draw ( Graphics g ) {
19     for ( int i = 0; i < points.length - 1; i++ )
20         g.drawLine ( points [i].x, points [i].y,
21                     points [i+1].x, points [i+1].y );
22     g.drawLine ( points [0].x, points [0].y, points [ points.length
23                 - 1].x, points [ points.length - 1].y );
24 }
25 }
```

The number of things that can be done with this very general Polygon class are limited. The getPerimeter() method can determine the length of the perimeter by adding the lengths of the segments connecting the vertices. It is possible to determine the area enclosed by a list of vertices, but the algorithm is complex. The draw() method draws the polygon by drawing each line segment that connects adjacent vertices. We discuss the Graphics class in [Chapter 15](#). If you compile and run this code, please note that in Java graphics, like many computer graphics environments, the upper left hand corner of the screen or window is considered (0,0), and *y* values **increase** going downward, not upward.

With this basic parent class defined, we can design a Triangle class as a child of it.

```

1 import java.awt.*;
2
3 public class Triangle extends Polygon {
4     public Triangle ( int x1, int y1, int x2, int y2,
5                     int x3, int y3 ) {
6         super ( toPointArray ( x1, y1, x2, y2, x3, y3 ) );
7     }
8
9     protected static Point [] toPointArray ( int x1, int y1,
10                                            int x2, int y2, int x3, int y3 ) {
11         Point [] array = { new Point (x1, y1), new Point (x2, y2),
12                           new Point (x3, y3) };
13         return array ;
14     }
}
```

Again, the **import** statement is for the Point class. One reasonable constructor to make a triangle would take in six values, giving the *x* and *y* coordinates of the three vertices of the triangle. Of course, the Polygon class requires an array of type Point, but the **super** constructor must be the first line of the Triangle constructor. To solve this problem, we create a **static** method to package the values into an array. We could have done the same thing in the argument list of the **super** constructor, but it would have looked messier. The toPointArray() is **protected** because there is no reason to let external code

have access to it.

```
16 public String getType () {
17     double a = points [0].distance ( points [1]) ;
18     double b = points [1].distance ( points [2]) ;
19     double c = points [2].distance ( points [0]) ;
20     if( a == b && b == c )
21         return " Equilateral ";
22     if( a == b || b == c || a == c )
23         return " Isosceles ";
24     return " Scalene ";
25 }
26 }
```

Finally, the `getType()` method allows us to do something specific with triangles. We can use the `distance()` method from the `Point` class to find the length of each of the three sides. By comparing these lengths, we can determine whether the triangle represented is equilateral, isosceles, or scalene. Of course, computing the perimeter and drawing the triangle are already taken care of by the `Polygon` class.

We can easily make a `Rectangle` class along the same lines.

```
1 import java.awt.*;
2
3 public class Rectangle extends Polygon {
4     public Rectangle ( int x, int y, int length, int width ) {
5         super ( toArray ( x, y, length, width ) );
6     }
7
8     protected static Point [] toArray ( int x, int y, int length,
9             int width ) {
10        Point [] array = { new Point (x, y), new Point (x + length, y),
11            new Point (x + length, y + width ), new Point (x, y +
12                width )};
13        return array ;
14    }
15    public int getArea () {
16        int length = points [1].x - points [0].x;
17        int width = points [2].y - points [1].y;
18        return length * width ;
19    }
20 }
```

The constructor is similar to the `Triangle` constructor except that the upper left corner of the

rectangle is specified, along with the length and the width. From these values, the appropriate array of Point values is generated. The rectangle-specific code that we add is the getArea() method, which determines the length and width of the rectangle by examining the points array and then calculates area.

Using inheritance as form of specialization, we can go one step further and make a Square class.

```
1 public class Square extends Rectangle {  
2     public Square ( int x, int y, int size ) {  
3         super ( x, y, size, size );  
4     }  
5 }
```

This very short class uses everything available in Rectangle but simplifies the constructor slightly so that the user does not have to enter both length and width. ■

11.5 Solution: Boolean circuits

Here we present our solution to the Boolean Circuits problem. First, we define a parent class for all circuit components, called Gate.

```
1 public class Gate {  
2     private String name ;  
3  
4     public Gate ( String name ) { this.name = name ; }  
5     public String getName () { return name ; }  
6     public String toString () {  
7         return getName () + ": " + getValue ();  
8     }  
9     public boolean getValue () { return false ; }  
10 }
```

The Gate class doesn't do anything except set up ways to store a name and to get a value. From Gate, we can define the most basic circuit components: gates whose value is either always true or always false. It doesn't really matter what getValue() gives back for Gate, but we can say that it is false.

```
1 public class True extends Gate {  
2     public True () { super (" true "); }  
3     public boolean getValue () { return true ; }  
4 }  
  
1 public class False extends Gate {  
2     public False () { super (" false "); }  
3     public boolean getValue () { return false ; }  
4 }
```

To conform with the constructor for Gate, these new classes must pass a String giving their name to the **super** constructor. The values returned by the `getValue()` method are clear. Next, we want to create a class that can be used for general unary operators.

```
1 public class UnaryOperator extends Gate {  
2     private Gate input ;  
3  
4     public UnaryOperator ( String name ) { super ( name ); }  
5     public void setInput ( Gate input ) { this.input = input ; }  
6     public Gate getInput () { return input ; }  
7 }
```

The important addition in the `UnaryOperator` class is the `input` field. Any unary operator must have a single input gate that it operates on. This class provides a mutator and accessor for `input`, as well as an appropriate constructor. From `UnaryOperator`, we can derive two specific operators.

```
1 public class Output extends UnaryOperator {  
2     public Output ( int i ) { super ( " OUTPUT " + i ); }  
3     public boolean getValue () { return getInput ().getValue (); }  
4 }
```

```
1 public class Not extends UnaryOperator {  
2     public Not () { super ( " NOT " ); }  
3     public boolean getValue () { return !( getInput ().getValue () ); }  
4 }
```

The `Output` class takes in an `int` value and uses it to make a numbered name. Its `getValue()` method simply returns the value of its `input`. The `Output` class doesn't do anything except serve as a marker for circuit output. The `Not` class uses “NOT” as the name supplied to the `super` constructor and returns the logical NOT of the value of its `input`.

Just as we did for unary operators, we also need a base class for binary operators.

```
1 public class BinaryOperator extends Gate {  
2     private Gate operand1 ;  
3     private Gate operand2 ;  
4  
5     public BinaryOperator ( String name ) { super ( name ); }  
6     public Gate getOperand1 () { return operand1 ; }  
7     public Gate getOperand2 () { return operand2 ; }  
8     public void setOperand1 ( Gate operand ) { operand1 = operand ; }  
9     public void setOperand2 ( Gate operand ) { operand2 = operand ; }  
10 }
```

A `BinaryOperator` has two `Gate` fields, `operand1` and `operand2`, representing the inputs to the operator. The `BinaryOperator` class has an appropriate constructor and then accessors and mutators

for the operands. With `BinaryOperator` as a parent, only a few lines of code are necessary to define any logical binary operator.

```
1 public class And extends BinaryOperator {  
2     public And () { super ("AND"); }  
3  
4     public boolean getValue () {  
5         return getOperand1 ().getValue () &&  
6             getOperand2 ().getValue ();  
7     }  
8 }
```

```
1 public class Or extends BinaryOperator {  
2     public Or () { super ("OR"); }  
3  
4     public boolean getValue () {  
5         return getOperand1 ().getValue () ||  
6             getOperand2 ().getValue ();  
7     }  
8 }
```

```
1 public class Xor extends BinaryOperator {  
2     public Xor () { super ("XOR"); }  
3  
4     public boolean getValue () {  
5         return getOperand1 ().getValue () ^ getOperand2 ().getValue ();  
6     }  
7 }
```

In each case, a constructor passes the name of the gate to the `super` constructor. Then, each `getValue()` method gets the values from the two operands and combines them with AND, OR, or XOR, respectively. This design allows the programmer to focus only on the important element of each class. Adding new classes for NAND, NOR, or any other possible logical binary operator would be quick.

The client code that uses these classes to simulate a circuit follows.

```
1 import java.util.*;  
2  
3 public class BooleanCircuit {  
4     public static void main ( String [] args ) {  
5         Scanner in = new Scanner ( System.in );  
6         int count = in.nextInt ();  
7         Gate [] gates = new Gate [ count ];  
8         String name ;  
9         int value ;
```

First we have the import needed for Scanner. Next, we read in the total number of gates and create an array of type Gate of that length and declare a few useful temporary variables.

```
11 // Create gates
12 for ( int i = 0; i < count ; i++ ) {
13     name = in.next ().toUpperCase ();
14     if( name.equals ( " true " ) )
15         gates [i] = new True ();
16     else if( name.equals ( " false " ) )
17         gates [i] = new False ();
18     else if( name.equals ( " AND " ) )
19         gates [i] = new And ();
20     else if( name.equals ( "OR" ) )
21         gates [i] = new Or ();
22     else if( name.equals ( " XOR " ) )
23         gates [i] = new Xor ();
24     else if( name.equals ( " NOT " ) )
25         gates [i] = new Not ();
26     else if( name.equals ( " OUTPUT " ) ) {
27         value = in.nextInt ();
28         gates [i] = new Output ( value );
29     }
30 }
```

Then, we parse the input, creating an appropriate gate based on the name read in. In the case of an OUTPUT gate, we must also read in a number so that we can identify which OUTPUT gate is which later.

```
32 // connect gates
33 while ( in.hasNextInt () ) {
34     value = in.nextInt ();
35     name = gates [ value ].getName ();
36     if( name.equals ( " AND " ) || name.equals ( "OR" ) ||
37         name.equals ( " XOR " ) ) {
38         BinaryOperator operator =
39             ( BinaryOperator ) gates [ value ];
40         operator.setOperand1 ( gates [in.nextInt ()] );
41         operator.setOperand2 ( gates [in.nextInt ()] );
42     }
43     else if( name.equals ( " NOT " ) ||
44         name.startsWith ( " OUTPUT " ) ) {
45         UnaryOperator operator =
46             ( UnaryOperator ) gates [ value ];
47         operator.setInput ( gates [in.nextInt ()] );
```

```
48     }
49 }
```

As long as there is remaining input, we read in an index. Based on the name of the gate at that index in the array, we either read in two more indexes (for binary operators) or just a single additional index (for unary operators). In either case, we set the input or inputs of the operator to the gate or gates at those indexes.

```
51 // Compute output
52 for ( int i = 0; i < count ; i++ )
53     if( gates [i].getName () .startsWith (" OUTPUT ") )
54         System.out.println ( gates [i] );
55     }
56 }
```

Finally, the simulation of the circuit is absurdly simple. We look through array until we find a gate whose name starts with “**OUTPUT**”. Then, we print out its value. In order to determine its value, it will ask its input what its value is, which in turn will ask for the values from its input. The `toString()` in the `Gate` class will assure us that the final output is nicely formatted. This system accommodates any number of output gates connected arbitrarily, as long as the circuit has no loops inside of it, such as an AND gate whose output is also one of its inputs.

11.6 Concurrency: Inheritance

Like interfaces, inheritance in Java is not closely related to concurrency. However, two ways in which inheritance interacts with concurrency deserve attention.

The first is the `Thread` class. Each thread of execution in Java (except the main thread) is managed with a `Thread` object or an object whose type inherits from `Thread`. Creating such types is done by extending `Thread`, just as you would extend any other class. Further information about extending `Thread` for concurrency is given in [Section 13.4](#). Extending the `Thread` class to make your own customized threads of execution is an alternative to implementing the `Runnable` interface mentioned in [Section 10.6](#) and is discussed in greater detail in [Section 13.4.5](#).

The second interaction between inheritance and concurrency is again very similar to the problem with interfaces and concurrency: There is no way to specify that a method is thread-safe. Recall that it is not allowed to use the `synchronized` keyword on a method in an interface declaration. Likewise, there is no restriction on overriding a synchronized method with a non-synchronized method or vice versa.

The rules for overriding methods in Java guarantee that an object of a child class is usable anywhere that an object of the parent class is usable. Thus, you cannot override a public method with a private one, reducing the visibility of a method. We discuss a similar restriction with exceptions in [Section 17.3.4](#).

If it has these restrictions, why doesn’t Java prevent a synchronized method from being overridden by a non-synchronized method? In the first place, a non-synchronized method can be used anywhere a

synchronized one could. (Unlike a private method, which is not accessible everywhere a public one is.) In the second, the designers of Java have put thread safety in the category of implementation details left up to the programmer. Some classes need specific methods to be synchronized and others (even child classes) do not. However, if you override a class with a synchronized method, it is safest to mark your method synchronized as well.

Exercises

Conceptual Problems

11.1 Give three advantages of using inheritance instead of copying and pasting code from a parent class. Are there any disadvantages to using inheritance?

11.2 Consider classes Radish and Carrot which both extend class Vegetable and implement interface Crunchable. Which of the following sets of assignments are legal and why?

- a. Radish radish = **new** Radish();
- b. Radish radish = **new** Vegetable();
- c. Vegetable vegetable = **new** Radish();
- d. Crunchable crunchy = **new** Radish();
- e. Radish radish = **new** Carrot();

11.3 In the context of inheritance, the keyword **super** can be used for two different purposes. What are they?

11.4 Consider the following class definitions.

```
public class A {  
    private String value ;  
    public A( String s) { value = "A" + s + "A"; }  
    public String toString () { return value ; }  
}  
  
public class B extends A {  
    public B( String s) { super ( "B" + s + "B" ); }  
}  
  
public class C extends B {  
    public C( String s) { super ( "C" + s " "C" ); }  
}
```

What is output by the following code fragment?

```
C c = new C(" ABC");  
System.out.println (c);
```

11.5 Beginning Java programmers often confuse package-private access (no explicit specifier) with **public** access. How is this confusion possible when default access is more constrained than both **public** and **protected** access? (Hint: The file system plays a role.)

11.6 What are the similarities and differences between overloading a method and overriding a method?

11.7 What is field hiding? How can software bugs arise from this functionality in Java?

11.8 Give reasons why the designers of Java decided not to allow multiple inheritance. Would you have made the same decision? Why or why not?

11.9 Draw a class hierarchy establishing a sensible relationship between the Human, Soldier, Sailor, Marine, General, and Admiral classes. For this class hierarchy, refer to the U.S military structure in which the U.S. Marine Corps is a part of the U.S. Navy.

Programming Practice

11.10 Create an InternationalStudent class that extends Student. It should include String fields for country of origin and also visa status. It should include mutator and accessor methods for these two new fields.

11.11 Add Pentagon and Hexagon classes that extend the Polygon class. The constructor for each class should take an *x*, *y* and radius value, each of **int** type. Both classes should be implemented to create *regular* polygons, that is, polygons in which all 5 or 6 sides have the same length. The *x* and *y* values should give the center of the polygon, and each of the 5 or 6 points should be the radius distance away from that center.

Because the internal structure of Polygon keeps all vertices as Point values, the *x* and *y* coordinates of the points must be **int** values. This requirement will force you to round these *x* and *y* coordinates after using trigonometry to determine their locations. As a result, the final pentagons and hexagons stored and displayed will be slightly irregular.

11.12 The inheritance design of our solution to the Boolean circuits problem given in [Section 11.5](#) makes adding new gates easy. Add classes that implement a NAND gate and a NOR gate. Then rewrite the main() method of BooleanCircuit to accommodate these two extra classes.

11.13 Re-implement the object hierarchy in the solution to the sort it out problem from [Chapter 10](#). This time, let the Cat, Dog, and Person classes extend the Creature class defined below.

```
1 public abstract class Creature implements Ageable, Weighable {  
2     protected int age ;  
3     protected double weight ;  
4  
5     public Creature ( int age, double weight ) {  
6         this.age = age ;  
7         this.weight = weight ;  
8     }  
9 }
```

```

10     public int getAge () { return age ; }
11
12     public int getWeight () { return weight ; }
13 }
```

Refactor your code so that the Cat, Dog, and Person classes are as short as possible. How many lines of code do you save?

11.14 Design a celestial body simulator. You will need to create a class containing fields for the x , y , and z locations, x , y , and z velocities, radii, and masses of each object. For each time step of length t , you must do the following.

1. Compute the sum of forces exerted on each body by every other body. The equation for gravitational force on body b exerted by body a is $\mathbf{F}_{ab} = -G \frac{m_a m_b}{|\mathbf{r}_{ab}|^2} \hat{\mathbf{r}}_{ab}$ where $G = 6.673 \times 10^{-11} \text{ Nm}^2\text{kg}^{-2}$, $|\mathbf{r}_{ab}|$ is the distance between the centers of objects a and b , and $\hat{\mathbf{r}}_{ab} = \frac{\mathbf{r}_b - \mathbf{r}_a}{|\mathbf{r}_b - \mathbf{r}_a|}$, the unit vector between the centers of the two objects.
2. Compute the x , y , and z components of the acceleration vector \mathbf{a} for each object using the equation, $\mathbf{F} = m\mathbf{a}$, once the sum of forces has been calculated.
3. Update the x , y , and z components of the velocity vector \mathbf{v} for each object using the equation $\mathbf{v}_{new} = \mathbf{v}_{old} + \mathbf{a}t$.

Experiments

11.15 Inheritance is a powerful technique, but it comes with some overhead costs. Create a class called A with the following implementation.

```

public class A {
    protected int a;
}
```

Then, create 25 more classes named B through Z. Class B should extend A and add a **protected** int field called b. Continue in this manner, with each new class extending the previous one and adding an **int** field named the lowercase version of the class name. Thus, if you create an object of type Z, it will contain, through inheritance, 26 **int** fields named a through z. But, for a single Z object to be created, it must call 27 (Z back through A plus Object) constructors. You may wish to use the file I/O material in [Chapter 20](#) to write a program to create all these classes so that you do not have to do so by hand.

Finally, create a new class called All which contains 26 **protected** fields of **int** type named a through z. Now, the purpose of doing all this is to compare the time needed to instantiate an object of type Z with one of type All, though they both only contained 26 **int** fields named a through z.

Create an array of 100,000 elements of type Z and then populate it with 100,000 Z objects. Time this process. Create an array of 100,000 elements of type All and then populate that array with 100,000 All objects. You may wish to use the `System.nanoTime()` method described in

[Chapter 13](#) to accurately time these processes. Is there a significant difference in the times you found?

Chapter 12

Exceptions

The vulgar mind always mistakes the exceptional for the important.

—W. R. Inge

12.1 Problem: Bank burglary

Let's consider a problem in which various aspects of a bank burglary are modeled as objects in Java. You want to write some code that will accomplish the following steps:

1. Disable the burglar alarm
2. Break into the bank
3. Find the vault
4. Open the vault
5. Carry away the loot

But any number of things could go wrong! When trying to disable the burglar alarm, you might set it off. When you try to break into the bank, you might have thought that you'd disabled the burglar alarm but actually failed to do so. The door of the bank might be too difficult to open. The vault might be impossible to find, or the vault might be impossible to open. It could even be empty. The money might be made of enormous gold blocks that are too heavy to carry away. Finally, at any time during the heist, a night watchman might catch you in the act.

If you are the criminal mastermind who planned this deed, you need to know if (and preferably how) your henchman bungled the burglary. You need a simple system that can inform you of any errors that have occurred along the way. Likewise, you need to be able to react differently depending on what went wrong.

We are going to model each of these error conditions with Java *exceptions*. An exception is how Java indicates exceptional or incorrect situations inside of a program. These exceptions are *thrown* when the error happens. You must then write code to *catch* the error and deal with it appropriately. The bank burglar program will deal with three different classes, Bank, Vault, and Loot.

The Bank class has three methods, `disableAlarm()`, `breakIn()`, and `findVault()`, which returns a Vault object. The Vault class has an `open()` method and a `getLoot()` method, which returns a Loot object. The Loot class has a `carryAway()` method. Below is a table of the exceptions which can be thrown by each of the methods.

Class	Method	Exceptions
Bank	disableAlarm()	BurglarAlarmException WatchmanException
	breakIn()	BurglarAlarmException LockPickFailException WatchmanException
	findVault()	WatchmanException
Vault	open()	LockPickFailException WatchmanException
	getLoot()	WatchmanException
	carryAway()	LootTooHeavyException WatchmanException

In order to deal with each of these possible errors, you need some special Java syntax.

12.2 Concepts: Error handling

As a rule, computer programs are filled with errors. Writing a program is a difficult and complex process. Even if a segment of code is free from errors, it may call other code which contains mistakes. The user could be making mistakes and issuing commands to a program that are impossible to execute. Even hardware can produce errors, as in a hard drive crash or a DVD with a scratch on its surface.

12.2.1 Error codes

A robust program should deal with as many errors as possible. One strategy is to have every method give back a special error code corresponding to the error that has just occurred. Then, the code calling the method can react appropriately. Of course, many methods do not return a numerical type, limiting this kind of error handling. A solution that was very common in the C language, particularly in Unix system calls, was to set a globally visible **int** variable called `errno` to a value corresponding to the error that has just happened.

These solutions have a number of drawbacks. In the case of `errno`, if a number of different threads were running at the same time, different errors could occur simultaneously, but only one value could be kept in `errno`. For any system that relies on checking for an error condition after each method call, a large amount of error handling code must be mixed in with normal code. Doing so reduces code

readability and makes it difficult to centralize error handling. Likewise, a numerical value can be difficult to deal with. A user may have to look it up to find out what kind of error it is.

Exercise 12.1

12.2.2 Exceptions

Java adopts a different error handling strategy called exceptions. Whenever a specified error state or unusual situation is reached, an exception is thrown. When an exception is thrown, normal execution stops immediately. The exception looks for code that is designed to deal with that specific exception. The code that will handle the exception can be in the current method, in the calling method, in the caller of the calling method, or arbitrarily far back in the chain of method calls, all the way to main(). Each method will return, looking for code to handle this exception, until it is found. If no handling code is found, the exception will propagate all the way past main() to the JVM, and the program will end.

Exceptions give a unified and simple way to handle all errors. You can choose to deal with errors directly or delegate that responsibility to methods that call the code you are writing. Selection statements and loops are forms of local control flow, but exceptions give us the power of non-local control flow, able to jump back through any number of method calls.

12.3 Syntax: Exceptions in Java

In Java syntax, there are two important parts needed to use exceptions: throwing the exception when an error occurs and then handling that exception properly. Below we will explain both of these as well as the catch or specify requirement, the **finally** keyword, and the process of creating custom exceptions.

12.3.1 Throwing exceptions

By now you have probably experienced a NullPointerException in the process of coding. This exception happens when an object reference is **null** but we try to access one of its methods or fields.

```
String text = null;  
int x = text.length(); // NullPointerException
```

In this case, the exception is thrown by the JVM itself. It is possible to catch this exception and deal with it, but a NullPointerException generally means a mistake in the program, not an error that can be recovered from. Although many useful exceptions such as

NullPointerException, ArithmeticException, and ArrayIndexOutOfBoundsException are implicitly thrown by the JVM, we are permitted to throw them explicitly.

```
if( y < 14 )  
    throw new NullPointerException();
```

Like any other object, we use the **new** keyword to instantiate a NullPointerException using its default constructor. Once created, we use the **throw** keyword to cause the exception to go into effect.

Any exception you throw explicitly must use the `throw` keyword, but the majority of exceptions thrown by your programs will either be mistakes or exceptions thrown by library code you are calling. If you write a significant amount of library or API code, you may use `throw` more often.

12.3.2 Handling exceptions

Normal application programmers will find themselves writing code that handles exceptions much more often than code that throws them. In order to catch an exception, you must enclose the code that you think is going to throw an exception in a `try` block. Immediately after the `try` block, you can list one or more `catch` blocks. The code inside the first `catch` block that matches your exception will be executed.

```
try {
    String text = null;
    int x = text.length(); // NullPointerException
    System.out.println(" This will never be printed.");
}
catch ( NullPointerException e ) {
    System.out.println(" Surprise! A NullPointerException!");
}
```

In this case, trying to access the `length()` method of a `null` reference will still throw a `NullPointerException`, but now it will be caught by the `catch` block below. The message “`Surprise! A NullPointerException!`” will be printed to the screen, and execution will continue normally after the `catch` block. Once the exception is caught, it stops trying to propagate. Of course, whatever the code was doing when the exception was thrown was abandoned immediately because it might have depended on successful execution of the code that threw an exception. Thus, the call to the `System.out.println()` method in the `try` block will never be executed.

An exception will match the first `catch` block with the same class or any superclass. Since `Exception` is the parent of `RuntimeException` which is the parent of `NullPointerException`, we could write our example with `Exception` instead.

```
try {
    String text = null;
    int x = text.length(); // NullPointerException
    System.out.println(" This will never be printed.");
}
catch ( Exception e ) {
    System.out.println("Well, of course you got a
                      NullPointerException !");
}
```

In general, you should write the most specific exception class possible for your `catch` blocks. Otherwise, you might be catching a different exception than you plan for or preventing an exception from propagating up to an appropriate handler. For example, the following code has the potential to

throw either a NullPointerException or an ArithmeticException, because of a division by 0.

```
try {
    String text = null;
    int x;
    if( Math.random () > 0.5 )
        x = text.length (); // NullPointerException
    else
        x = 5 / 0; // ArithmeticException
}
catch ( Exception e ) {
    System.out.println (" You got some kind of exception !");
}
```

This code will catch either kind of exception, but it will not tell you which you got. Instead, the correct approach is to have one **catch** block for each possible kind of exception.

```
try {
    String text = null;
    int x;
    if( Math.random () > 0.5 )
        x = text.length (); // NullPointerException
    else
        x = 5 / 0; // ArithmeticException
}
catch ( NullPointerException e ) {
    System.out.println (" You got a null pointer !");
}
catch ( ArithmeticException e ) {
    System.out.println (" You divided by zero !");
}
```

The list of **catch** blocks can be arbitrarily long. You must always go from the most specific exceptions to the most general, like `Exception`, otherwise some exceptions could never be reached. The Java compiler enforces this requirement. The `e` is a reference to the exception itself, which behaves something like a parameter in a method. It is common to use `e` as the identifier but you are allowed to call it any legal variable name. Usually, the kind of exception is all you need to know, but every exception is an object and has fields and methods. Particularly useful is the `getMessage()` method which can give additional information about the exception.

Exercise 12.10

12.3.3 Catch or specify

Despite the examples given above, you will rarely write code to catch a `NullPointerException` or an `ArithmeticException`. Both of these exceptions are called *unchecked* exceptions. In [Chapter 6](#), we

used the `Thread.sleep()` method to put the execution of our program to sleep for a short period of time. We were forced to enclose this method call in a `try` block with a `catch` block for `InterruptedException`.

```
try {  
    Thread.sleep(100);  
}  
catch ( InterruptedException e ) {  
    System.out.println("Wake up!");  
}
```

An `InterruptedException` is thrown when another thread tells your thread of execution to wake up before it finishes sleeping or waiting. This exception is a *checked* exception, meaning that Java insists that you use a `try-catch` pair anytime there is even a chance of it being thrown. Otherwise your code will not compile.

Checked exceptions are those exceptions that your program should plan for. Library and API code often throw checked exceptions. For example, when trying to open a file with an API call, it's possible that no file with that name exists or that the user might not have permission to access it. A program should catch the corresponding exceptions and recover rather than crashing. Perhaps the program should prompt the user for a new name or explain that the required permission is not set.

Exercise 12.5

Exercise 12.8

In [Chapter 6](#), there were no executable statements in the `catch` block used with the `Thread.sleep()` method. You should never write an empty `catch` block. Doing so allows errors to fail silently.

Exercise 12.6

We are allowed to put code that can throw a checked exception into a `try-catch` block, but there is another possibility. Java has a catch or specify requirement, meaning that your code is required either to catch a checked exception or to specify that it has the potential for causing that exception. To specify that a method can throw certain exceptions, we use the `throws` keyword. Note that this is not the same as the `throw` keyword.

Exercise 12.3

Exercise 12.4

```
public static void sleepWithoutTry ( int milliseconds ) throws  
InterruptedException {  
    Thread.sleep ( milliseconds );  
}
```

In this case, there is no need for a `try-catch` block because the method announces that it has a risk of throwing an `InterruptedException`. Of course, any code that uses this method will have to have a `try-catch` block or specify that it also throws `InterruptedException`. A method can throw many different exceptions, and you can simply list them out after the `throws` keyword, separated by commas.

Almost every exception thrown in Java is a child class of Exception, RuntimeException, or Error. Any descendant of RuntimeException or Error is an unchecked exception and is exempt from the catch or specify requirement. Any direct descendant of Exception is a checked exception and must either be caught with a **try-catch** block or specified with the **throws** keyword. We say **direct** descendant because RuntimeException is a child of Exception, leading to the confusing situation where only those descendants of Exception which are not also descendants of RuntimeException are checked.

12.3.4 The **finally** keyword

To deal with the situation in which some important cleanup or finalizing task must be done no matter what, the designers of Java introduced the **finally** keyword. A **finally** block comes after all the **catch** blocks following a **try** block. The code inside the **finally** block will be executed **whether or not** any exception was thrown. A **finally** block is often used with file I/O to close the file, which should be closed whether or not something went wrong in the process of reading it as we demonstrate in [Chapter 20](#).

The **finally** keyword is unusually powerful. If an exception is not caught and propagates up another level, the **finally** block will be executed before propagating the exception. Even a **return** statement will wait for a **finally** block to be executed before returning, leading to the following bizarre possibility.

Exercise 12.2

Exercise 12.9

```
public static boolean neverTrue () {  
    try {  
        return true ;  
    }  
    finally {  
        return false ;  
    }  
}
```

This method attempts to return **true**, but before it can finish, the **finally** block returns **false**. Only one value can be returned, and the **finally** block wins. You should be aware of **finally** blocks and their unusual semantics. Use them sparingly and only for careful cleanup operations when necessary to guarantee that some event occurs.

Code in a **finally** block will execute **no matter what** unless the JVM exits or the thread in question terminates.

Exercise 12.11

12.3.5 Customized exceptions

Exceptions are incredibly useful when dealing with problems encountered by API code. In those cases, your code must merely catch exceptions defined by someone else; however, it is sometimes useful to define your own exceptions. For one thing, you may write an API yourself. Generally, you

will want to use the standard exceptions whenever possible, but your code may generate some unusual or very specific error condition that you want to communicate to a programmer using your own exception.

Defining a new exception is surprisingly simple. All you have to do is write a class that extends Exception. Theoretically, you could alternately extend RuntimeException or Error, but you typically will not. Children of RuntimeException are intended to indicate a bug in the program and children of Error are intended to indicate a system error. When creating your new exception, you don't even have to create any methods, but it is wise to implement a default constructor and one that takes a String as an additional message.

Exercise 12.12

Exercise 12.13

Exercise 12.14

```
public class EndOfWorldException extends Exception {  
    public EndOfWorldException () {}  
  
    public EndOfWorldException ( String message ) {  
        super ( message );  
    }  
}
```

As with all other classes, your exceptions should be named in a readable way. This exception is apparently thrown when the world ends. It is considered good style to end the name of any exception class with Exception. An exception class is a fully fledged class. If you need to add other fields or methods to give your exception the functionality it needs, go ahead. However, the main value of an exception is simply in its existence as a named error, not in any tricks it can perform.

Here we will give a few examples of exception handling, although exceptions are more useful in large systems with heavy API use. We will start with an example of a simple calculator that detects division by zero, then look at exceptions as a tool to detect array bounds problems, and end with a custom exception used with the Color class.

Example 12.1: Calculator with division by zero

Here we will implement a quick calculator that reads input from the user in the form of **integer operator integer**, where **operator** is one of the four basic arithmetic operators (+, -, *, /). Our code will perform the appropriate operation and output the answer, but we will use exception handling to avoid killing the program when a division by zero occurs.

```
1 import java.util.*;  
2  
3 public class QuickCalculator {  
4     public static void main ( String [] args ){  
5         Scanner in = new Scanner ( System.in );  
6         int a, b, answer = 0;
```

```

7     char operator ;
8     String [] terms ;
9     String line = in.nextLine ().trim ().toLowerCase ();
10    while ( ! line.equals (" quit ") ) {
11        terms = line.split (" ");
12        a = Integer.parseInt ( terms [0] );
13        operator = terms [1].charAt (0) ;
14        b = Integer.parseInt ( terms [2] );

```

By this point, the `main()` method has set up a system to read a line of input from a user, test to see if it is the sentinel value “quit”, and then parsing it into two `int` values and a `char` otherwise.

```

15    try {
16        switch ( operator ) {
17            case '+': answer = a + b; break ;
18            case '-': answer = a - b; break ;
19            case '*': answer = a * b; break ;
20            case '/': answer = a / b; break ;
21        }
22        System.out.println (" Answer : " + answer );
23    }
24    catch ( ArithmeticException e ) {
25        System.out.println (" You can 't divide by 0!");
26    }
27    line = in.nextLine ().trim ().toLowerCase ();
28}
29}
30}

```

Here we have a `try` block enclosing the code where the operations occur. Inside the `switch` statement, the code will blindly perform addition, subtraction, multiplication, or division, depending on the value of `operator`. Then, it will print the answer. However, if a division by zero occurred, the execution would jump to the `catch` block and print an appropriate message. This `try-catch` pair is situated inside the loop so that the input will continue even if there was a division by zero. We could achieve the same effect by using an `if` statement to test if the divisor is zero, but our solution allows easy extensions if there are other possible exceptions we want to catch. ■

Example 12.2: Array bounds

Exceptions provide a lot of power. If we want, we can use the `ArrayOutOfBoundsException` as a crutch when we don’t want to think about the bounds of our array. Although this makes for an interesting example, exceptions should not be used in Java to perform normal tasks. This method takes in an array of `int` values and prints them all out.

```

public static void exceptionalArrayPrint ( int [] array ) {
    try {

```

```

int i = 0;
while ( true )
    System.out.print ( array [i ++] + " " );
}
catch ( ArrayIndexOutOfBoundsException e ) {}
}

```

Although the `while` loop will run without stopping, the moment that `i` reaches `array.length`, it will throw an `ArrayIndexOutOfBoundsException` when it tries to access that element in `array`. Since we left the `catch` block empty, nothing will happen, the method will return, and everything will work fine. This example is a peculiar kind of laziness, indeed, since a `for` loop could achieve the same effect with fewer lines of code.

Programmers can be tempted to abuse exceptions in this way when a lot of calculations are needed to determine the correct bounds. Consider a game of Connect Four. To see if a player has won, the computer must examine all horizontal, vertical, and diagonal possibilities for four in a row. If the game board is represented as a 2D array, the programmer must be careful to make sure that checking for four in a row does not access any index greater than the last rows or column or smaller than 0.

The danger of using exceptions for these kinds of tasks has several sources. First, the programmer may not deeply understand the problem and may be careless about the solution. Second, there is a risk of hiding real exceptions that are generated because of real errors. Third, the code becomes difficult to read and unintuitive. Finally, excessive use of exceptions can negatively impact performance. ■

Exercise 12.15

Example 12.3: Color ranges

The `Color` class provided by Java allows us to represent a color as a triple of red, green, and blue values with each value in the range $[0, 255]$. Using these three components, we can produce $256^3 = 16,777,216$ colors. If we were programming some image manipulation software, we might want to be able to increase the red, green, or blue values separately. If changing a value makes it larger than 255, we could throw an exception. Likewise, if changing a value makes it less than 0, we could throw a different exception. Let's give two custom exceptions that could serve in these roles.

```

1 public class ColorUnderflowException extends Exception {
2     public ColorUnderflowException ( String message ) {
3         super ( message );
4     }
5     public ColorComponentTooSmallException () { super (); }
6 }

```

```

1 public class ColorOverflowException extends Exception {
2     public ColorOverflowException ( String message ) {
3         super ( message );
4     }
5     public ColorOverflowException () { super (); }

```

Now we can write six methods, each of which increases or decreases the red, green, or blue component of a Color object by 5. If the value of the component is out of range, an appropriate exception will be thrown.

```
public static Color increaseRed ( Color color )
    throws ColorOverflowException {
    if( color.getRed () + 5 > 255 )
        throw new ColorOverflowException (" Red : "
            + ( color.getRed () + 5));
    else
        return new Color ( color.getRed () + 5, color.getGreen (),
            color.getBlue () );
}
```

```
public static Color increaseGreen ( Color color )
    throws ColorOverflowException {
    if( color.getGreen () + 5 > 255 )
        throw new ColorOverflowException (" Green : "
            + ( color.getGreen () + 5));
    else
        return new Color ( color.getRed (), color.getGreen () + 5,
            color.getBlue () );
}
```

```
public static Color increaseBlue ( Color color )
    throws ColorOverflowException {
    if( color.getBlue () + 5 > 255 )
        throw new ColorOverflowException (" Blue : "
            + ( color.getBlue () + 5));
    else
        return new Color ( color.getRed (), color.getGreen (),
            color.getBlue () + 5 );
}
```

```
public static Color decreaseRed ( Color color )
    throws ColorUnderflowException {
    if( color.getRed () - 5 < 0 )
        throw new ColorUnderflowException (" Red : "
            + ( color.getRed () - 5));
    else
        return new Color ( color.getRed () - 5,
            color.getGreen (), color.getBlue () );
```

```

}
public static Color decreaseGreen ( Color color )
    throws ColorUnderflowException {
    if( color.getGreen () - 5 < 0 )
        throw new ColorUnderflowException (" Green : "
            + ( color.getGreen () - 5));
    else
        return new Color ( color.getRed (), color.getGreen () - 5,
            color.getBlue () );
}

```

```

public static Color decreaseBlue ( Color color )
    throws ColorUnderflowException {
    if( color.getBlue () - 5 < 0 )
        throw new ColorUnderflowException (" Blue : "
            + ( color.getBlue () - 5));
    else
        return new Color ( color.getRed (), color.getGreen (),
            color.getBlue () - 5 );
}

```

Finally, we can write a short method that changes a given color based on user input and deals with exceptions appropriately.

```

public static Color changeColor ( Color color ) {
    System.out.println (" Enter 'R ', 'G ', or 'B' to increase " +
        " the amount of red, green, or blue in your color." +
        " Enter 'r ', 'g ', or 'b' to decrease the amount of " +
        "red, green, or blue in your color.");
    Scanner in = new Scanner ( System.in );
    try {
        switch ( in.next ().trim ().charAt (0) ) {
            case 'R': color = increaseRed ( color ); break ;
            case 'G': color = increaseGreen ( color ); break ;
            case 'B': color = increaseBlue ( color ); break ;
            case 'r': color = decreaseRed ( color ); break ;
            case 'g': color = decreaseGreen ( color ); break ;
            case 'b': color = decreaseBlue ( color ); break ;
        }
    }
    catch ( ColorOverflowException e ) {
        System.out.println ( e );
    }
}

```

```

catch ( ColorUnderflowException e ) {
    System.out.println ( e );
}
return color ;
}

```

The code that uses these methods and exceptions is compact. One **try** block enclosing the method calls is needed so that the exceptions can be caught. Following the **try**, there is a **catch** block for the **ColorOverflowException** and one for the **ColorUnderflowException**. Each will print out its exception, including the customized message inside. If an exception occurred, the value of **color** would remain unchanged because execution would have jumped to a **catch** block before the assignment could happen. ■

12.4 Solution: Bank burglary

Here is our solution to the bank burglary problem. Although somewhat fanciful, the process could be expanded into a more serious simulation. We begin by defining each of the exceptions.

```

1 public class BurglarAlarmException extends Exception {
2     public BurglarAlarmException ( String message ) {
3         super ( message );
4     }
5     public BurglarAlarmException () { super (); }
6 }

```

```

1 public class WatchmanException extends Exception {
2     public WatchmanException ( String message ) {
3         super ( message );
4     }
5     public WatchmanException () { super (); }
6 }

```

```

1 public class LockPickFailException extends Exception {
2     public LockPickFailException ( String message ) {
3         super ( message );
4     }
5     public LockPickFailException () { super (); }
6 }

```

```

1 public class LootTooHeavyException extends Exception {
2     public LootTooHeavyException ( String message ) {
3         super ( message );
4     }
5     public LootTooHeavyException () { super (); }
6 }

```

Note that the default constructor for each exception is necessary, since constructors taking a String value are provided for each class. Although these default constructors do nothing other than call their parent constructor, they are needed so that it is possible to create each of these contructors **without** a customized message.

With the exceptions defined, we can assume that the Bank class and the Vault class throw the appropriate exceptions when something goes wrong. Thus, we can make a Henchman class who can try to do the heist and react appropriately if there is a problem.

```
1 public class Henchman {  
2     public void burgle ( Bank bank ) {  
3         try {  
4             bank.disableAlarm ();  
5             bank.breakIn ();  
6             Vault vault = bank.findVault ();  
7  
8             vault.open ();  
9             Loot loot = vault.getLoot ();  
10            loot.carryAway ();  
11  
12            System.out.println ("We got " + loot + "!");  
13        }  
14    }
```

To burgle a bank, one must create a Henchman object then pass a Bank object into its burgle() method. The method will try to disable the alarm, break into the bank, find the vault, open the vault, get the loot out of the vault, and carry it away. If all those steps happen successfully, the method will print out a String version of the loot. All of this code is inside of a try block. If an exception is thrown at any point, the following catch blocks will deal with it.

```
14     catch ( BurglarAlarmException e ) {  
15         System.out.println ("I set off the burglar because "  
16                     + e.getMessage ());  
17         System.out.println ("I had to run away.");  
18     }  
19     catch ( WatchmanException e ) {  
20         System.out.println ("A watchman caught me because "  
21                     + e.getMessage ());  
22         System.out.println (" Please bail me out of jail.");  
23     }  
24     catch ( LockPickFailException e ) {  
25         System.out.println ("I couldn't pick the vault lock.");  
26         System.out.println ("No loot for us.");  
27     }  
28     catch ( LootTooHeavyException e ) {  
29         System.out.println (" The loot was too heavy to carry.");
```

```

30         System.out.println ("No loot for us.");
31     }
32     catch ( NullPointerException e ) {
33         System.out.println (" The vault was hidden or empty.");
34         System.out.println ("No loot for us.");
35     }
36 }
37 }
```

If a BurglarAlarmException happens, the henchman is forced to run away. If a WatchmanException happens, the henchman is caught and must be bailed out of jail. If a LockPickFailException or LootTooHeavyException happens, the henchman is unable to carry the loot off.

The last catch block is a little unusual. In this case, a NullPointerException has occurred. Within the try block, two obvious sources of this exception are the vault and the loot variables. If either of them were null, in the case of a vault that could not be found or a vault that was empty, trying to call a method on that null reference would throw a NullPointerException. Although this code shows the power of exception handling, it is a little unwieldy since we do not know which variable was null. Also, we will hide any NullPointerException that might have for other reasons. A better solution would be to check for each of these null cases or create more specific exceptions thrown by findVault() and getLoot() if either returns null.

12.5 Concurrency: Exceptions

Any thread in Java can throw an exception. That thread might be the main thread or it might be an extra one that you spawned yourself. (Or even one spawned behind the scenes through a library call.)

What happens when a thread throws an exception? As we have been discussing in this chapter, the exception will either be caught or passed on to its caller. If the exception is caught, the catch block determines what happens. If the exception is passed up and up and up and never caught, then what? If you have coded some of the examples in this chapter, you might think the entire program crashes, but only the thread throwing the exception dies.

Example 12.4: Multiple threads with exceptions

In a program with a single thread, an exception thrown by the main() method will crash the program, completely halting execution. In a multi-threaded program, execution will continue on all threads that have not thrown exceptions. If even a single thread is executing, the program will run to completion before the JVM shuts down.

Program 12.1: This program spawns 10 threads. 9 out of 10 spawned threads as well as the main thread throw an exception and die. The remaining thread outputs the sum of the sines of 1 through 1,000,000. (CrazyThread.java)

```

1 public class CrazyThread extends Thread {
2     private int value ;
```

```

3  public static void main ( String [] args ) {
4      for ( int i = 0; i < 10; i++ )
5          new CrazyThread ( i ).start ();
6      throw new RuntimeException ();
7  }
8
9  public CrazyThread ( int value ) {
10     this.value = value ;
11 }
12
13 public void run () {
14     if( value == 7 ) {
15         double sum = 0;
16         for ( int i = 1; i <= 1000000; i++ )
17             sum += Math.sin (i);
18         System.out.println (" Sum : " + sum );
19     }
20     else
21         throw new RuntimeException ();
22 }
23 }
```

In the program given above, all of the threads except one will die because of the `RuntimeException` that they throw. Note that we use the unchecked `RuntimeException` so that Java does not complain about the lack of catch blocks. The thread with a value of 7 will complete its calculation and print it to the screen even though the main thread has died. For more information on how to spawn threads, refer to [Chapter 13](#). ■

This behavior can cause a program that never seems to finish. You might write a program that spawns a number of threads and does some work. Even if the `main()` method has completed and all the important data has been output, the program will not terminate if any threads are still alive. This problem can also be caused by creating a GUI (such as a `JFrame`), which spawns one or more threads indirectly, if the GUI is not properly disposed.

12.5.1 InterruptedException

In conjunction with concurrency, one exception deserves special attention: `InterruptedException`. This exception can happen when a thread calls `wait()`, `join()`, or `sleep()`. It is a checked exception, requiring either a catch block or a `throws` declaration.

This exception is used in cases where the executing thread must wait for some event to occur or some time to pass. In extreme circumstances, another thread can interrupt the waiting thread, forcing it to continue executing before it's done waiting. If that happens, the code in the catch block determines how the thread should recover from being woken prematurely.

Programmers who are new to concurrency in Java are often confused or annoyed by `InterruptedException`, particularly since it never seems to be thrown. Although it is thrown rarely,

situations such as a system shutting down may be best dealt with by calling `interrupt()` on a waiting thread, causing such an exception. Although we will generally leave the `InterruptedException` catch block empty in this book, threads written for production code should always handle interruptions gracefully.

Exercises

Conceptual Problems

- 12.1 What are the advantages of using exceptions instead of returning error codes?
- 12.2 The keywords `final` and `finally`, as well as the `Object` method `finalize()`, are sometimes confused. What is the purpose of each one?
- 12.3 What is the difference between the `throw` keyword and the `throws` keyword?
- 12.4 Explain the catch or specify requirement of Java.
- 12.5 What must be done differently when using methods that throw checked exceptions as compared to unchecked exceptions? How do the classes `Exception`, `RuntimeException`, and `Error` play a role?
- 12.6 For every program you write, you could choose to put the entire body of your `main()` method in a large `try` block with a `catch` block at the end that catches `Exception`. In this way, no exception would cause your program to crash. Why is this approach a bad programming decision?
- 12.7 Why did the designers of Java choose to make `NullPointerException` and `ArithmaticException` unchecked exceptions even though this choice means that a program that unintentionally dereferences a null pointer or divides by zero will often crash.
- 12.8 Consider the following two classes.

```
public class Trouble {  
    public makeTrouble () {  
        throw new ArithmaticException ();  
    }  
}  
  
public class Hazard {  
    public makeHazard () {  
        throw new InterruptedException ();  
    }  
}
```

Class `Trouble` will compile, but class `Hazard` will not. Explain why and what could be done to make `Hazard` compile.

- 12.9 What value will the following method always return and why?

```

public static int magic ( String value ) {
    try {
        int x = Integer.parseInt ( value );
        return x;
    }
    catch ( Exception e ) {
        System.out.println (" Some exception occurred.");
        return 0;
    }
    finally {
        return -1;
    }
}

```

12.10 Why will the following segment of code fail to compile?

```

try {
    Thread.sleep (1000);
}
catch ( Exception e ) {
    System.out.println (" Exception occurred !");
}
catch ( InterruptedException e ) {
    System.out.println (" Woke up early !");
}

```

12.11 Consider the following fragment of Java.

```

try {
    throw new NullPointerException ();
}
finally {
    throw new ArrayIndexOutOfBoundsException ();
}

```

This code is legal Java. It is possible to have a finally block after a try block without any catch blocks between them. However, only a single exception can be active at once. Which exception will propagate up from this code and why?

Programming Practice

12.12 The NumberFormatException exception is thrown whenever the Integer.parseInt() method receives a poorly formatted String representation of an integer. Re-implement QuickCalculator to catch any NumberFormatException and give an appropriate message to the user.

12.13 Refer to Exercise 11.14 from [Chapter 11](#). Add to the basic mechanics of the simulation by

designing two custom exceptions, `CollisionException` and `LightSpeedException`. These exceptions should be thrown, respectively, if two bodies collide or if the total magnitude of a body's velocity exceeds the speed of light.

12.14 Users often log onto systems by entering their user name and a password. Unfortunately, human beings are notoriously bad at picking passwords. In computer security, a tool called a *proactive password checker* allows a user to pick a password but rejects the choice if it doesn't meet certain criteria.

Common criteria for a password are that it must be at least a certain length, must contain uppercase and lowercase letters, must contain numerical digits, must contain symbols, cannot be the same as a list of words from a dictionary, and others.

Write a short program with a `check()` method that takes a single `String` parameter giving a possible password. This method should throw an exception if the password does not meet the matching criteria listed below.

Your `main()` method should prompt the user to select a password and then pass it to the `check()` method. If the method throws an exception, you should catch it and print an appropriate error message. Otherwise, you should report to the user that the password is acceptable. Note that you will need to define each of the four exceptions as well.

Experiments

12.15 Throwing and catching exceptions is a useful tool for making robust programs in Java. However, the JVM machinery needed to implement such a powerful tool is complex. Create an array containing 100,000 random `int` values. First, sum all these variables up using a `for` loop and time how long it takes. Then, do the same thing, but, inside of the `for` loop, put a `try` block containing a simple division by zero instruction such as `x = 5 / 0;`. After the `try` block, put a `catch` block catching an `ArithmaticException`. Time this version of the code. Again, you may wish to use `System.nanoTime()` to measure the time accurately. Was there a large difference in the time taken? Do your findings have any implications for code that routinely throws thousands of exceptions?

Password criteria	Exception
At least 8 characters in length	<code>TooShortException</code>
Contains both upper- and lowercase letters	<code>NoMixedCaseException</code>
Contains at least one numerical digit	<code>NoDigitException</code>
Contains at least one symbol	<code>NoSymbolException</code>

Chapter 13

Concurrent Programming

Time is the substance from which I am made. Time is a river which carries me along, but I am the river; it is a tiger that devours me, but I am the tiger; it is a fire that consumes me, but I am the fire.

—Jorge Luis Borges

13.1 Introduction

So far we have focused mostly on writing sequential programs. Such programs are executed sequentially by a computer. Sequential execution implies that program statements are executed one at a time in a sequence determined by program logic and input data. While it is common for programmers to write sequential programs, the widespread availability of multicore processors in a single computer has led to an increase in the demand for programmers who can write *concurrent* programs. More than one program statement can be executed independently by a multicore processor.

A concurrent program is one in which several statements can be executed simultaneously by two or more cores. In this chapter we show how to write simple concurrent programs in Java that exploit the power of a multicore computer. We begin with an example in which the fate of the planet is in grave danger!

13.2 Problem: Deadly virus

A deadly virus capable of wiping out the world's population is about to be released by an evil mastermind. Only he knows the security code that can stop the countdown. The world is doomed. The single hope for salvation lies with you and your Java programming skills. Through the investigations of a top secret, government-funded spy ring, it has been revealed that the security code is tied to the number 59,984,005,171,248,659. This large number is the product of two prime numbers, and the security code is their sum. All you need to do is factor the number 59,984,005,171,248,659 into its two prime factors.

Of course, there is a catch. The deadly virus is going to be released soon, so soon that there might not be enough time for your computer to search through all the numbers one by one. Instead, you must use concurrency to check more than one number at a time.

Does this problem sound contrived? To keep information sent over the Internet private, some kinds of public key cryptography rely on the difficulty of factoring large numbers. Although factoring the number in this problem is not difficult, the numbers used for public key cryptography, typically more than 300 decimal digits long, have resisted factorization by even the fastest computers.

13.3 Concepts: Splitting up work

The deadly virus problem has one large task (factoring a number) to perform. How should we split up this task so that we can do it more quickly? Splitting up the work to be done is at the heart of any concurrent solution to a problem.

In a multicore processor, each core is an independent worker. It takes some care to coordinate these workers. First of all, we still need to get the right answer. A concurrent solution is worthless if it is incorrect, and by reading and writing to the same shared memory, answers found by one core can corrupt answers found by other cores. Preventing that problem will be addressed in [Chapter 14](#). If we are sure the concurrent solution is correct, we also want to improve performance in some way. Perhaps we want the task to finish more quickly. Perhaps we have an interactive system that should continue to handle user requests even though it is working on a solution in the background. Again, if the overhead of coordinating our workers takes more time than a sequential solution or makes the system less responsive, it is not useful.

Exercise 13.10

There are two main ways to split up work. The first is called *task decomposition*. In this approach, each worker is given a different job to do. The second is called *domain decomposition*. In this approach, the workers do the same job, but to different data.

It is possible to use both task and domain decomposition together to solve the same problem. With both kinds of decomposition, it is usually necessary to coordinate the workers so that they can share information. In the next two subsections, we describe task decomposition and domain decomposition in greater depth. Then we discuss mapping tasks to threads of execution and the different memory architectures that can be used for concurrent programming.

13.3.1 Task decomposition

The idea of breaking a task down into smaller subtasks is a natural one. Imagine you are planning a dinner party. You need to buy supplies, cook the dinner, clean your house, and set the table. If four of you were planning the party, each could perform a separate activity. The preparations could go much faster than if a single person was doing the work, but coordination is still important. Perhaps the person cooking the dinner couldn't finish until certain supplies had been bought.

Task decomposition is often easier than domain decomposition because many tasks have natural divisions. Unfortunately, it is not always an effective way to use multiple cores on a computer. If one task finishes long before the others, a core might sit idle.

Exercise 13.9

The next two examples give simple illustrations of the process of splitting a task into smaller subtasks in the context of multicore programming.

Example 13.1: Video game tasks

Consider a simple video game that consists of the following tasks.

A: start game

B: process move

C: update score

D: repaint screen

E: end game

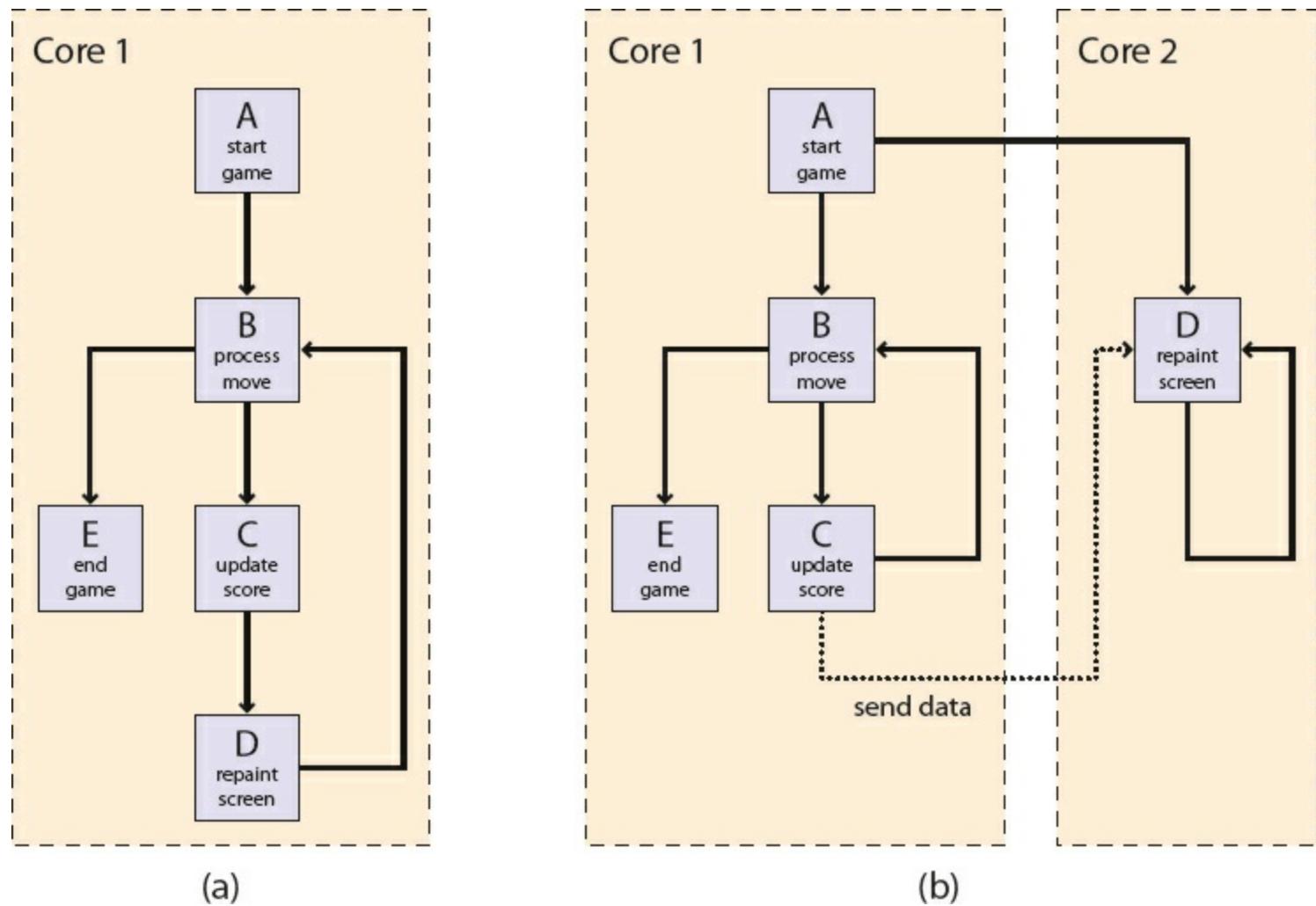


Figure 13.1: Execution of tasks in a video game. (a) Sequential execution on a single core. (b) Concurrent execution on two cores. Arrows show the flow of execution and data transfer.

Suppose that tasks B and D are independent and can be executed concurrently if two cores are available. Task D continually updates the screen with the old data until task C updates the information.

Figure 13.1(a) and (b) show how the tasks in this video game can be sequenced, respectively, on a single core or on two cores. All tasks are executed sequentially on a single-core processor. In a dual-core processor tasks B and C can execute on one core while task D is executing concurrently on another. Note from the figure that task C sends the score and any other data to task D which is continuously updating the screen. Having two cores can allow a faster refresh of the display as the processor does not have to wait for tasks B or C to complete. ■

Example 13.2: Math expression tasks

Suppose that the following mathematical expression is to be evaluated for parameters a and K at a

given value of t .

$$2 \cdot K \cdot a \cdot t \cdot e^{-a \cdot t^2}$$

We can divide the expression into two terms: $2 \cdot K \cdot a \cdot t$ and $e^{-a \cdot t^2}$. Each of these terms can be assigned to a different task for evaluation. On a dual-core processor, these two tasks can be executed on separate cores and the results from each combined to generate the value of the expression by the main task.

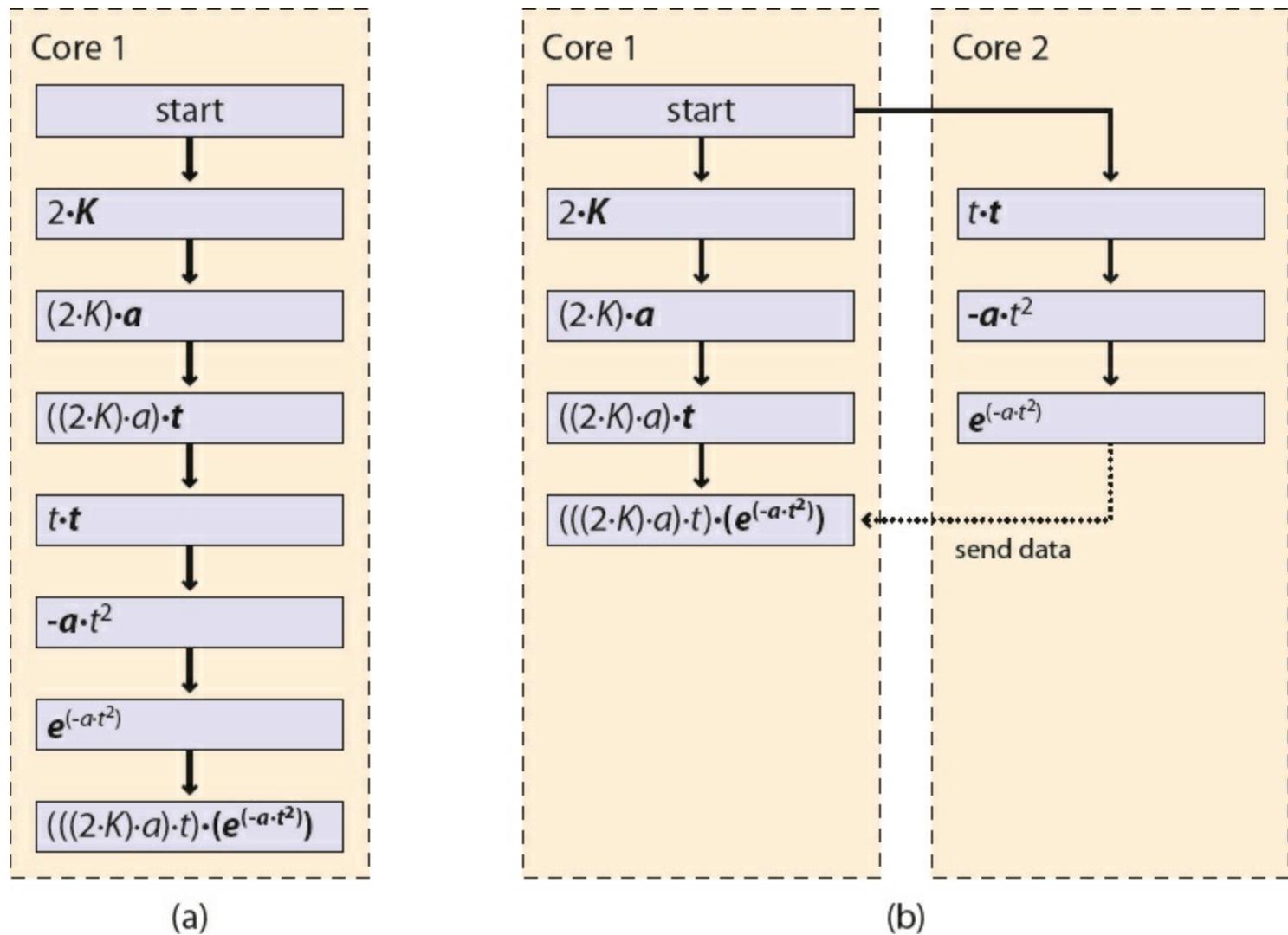


Figure 13.2: Evaluation of a mathematical expression (a) sequentially on a single core and (b) concurrently on two cores. Arrows show the flow of execution and data transfer. Bold typeface indicates the operation being performed.

Figure 13.2 shows how this expression can be evaluated on single core and dual-core processors. Sometimes, using multiple cores to evaluate an expression like this will take less time than a single core. However, there is no guarantee that using multiple cores will always be faster, because tasks take time to set up and to communicate with each other. ■

Exercise 13.4

These examples illustrate how a task can be divided into two or more subtasks executed by different cores of a processor. We use a dual-core processor for our examples, but the same ideas can

be expanded to a larger number of cores.

Exercise 13.5

13.3.2 Domain decomposition

In a computer program, every task performs operations on data. This data is called the *domain* of that task. In domain decomposition, the data is divided into smaller chunks where each chunk is assigned to a different core, instead of dividing a task into subtasks. Each core executes the same task but on different data.

In the example of the dinner party, we could have used domain decomposition instead of (or in addition to) task decomposition. If you want to cook a massive batch of mashed potatoes, you could peel 24 potatoes yourself. However, if there are four of you (and you each have a potato peeler), each person would only need to peel 6 potatoes.

The strategy of domain decomposition is very useful and is one of the major focuses of concurrency in this book. Problems in modern computing often use massive data, comprising millions of numbers or thousands of database records. Writing programs that can chop up data so that multiple cores can process smaller sections of it can greatly speed up the time it takes to finish computing a solution.

Domain decomposition can be more difficult than task decomposition. The data must be divided evenly and fairly. Once each section of data has been processed, the results must be combined. Companies like Google that process massive amounts of information have developed terminology to describe this process. Dividing up the data and assigning it to workers is called the *map* step. Combining the partial answers into the final answer is called the *reduce* step.

We illustrate the domain decomposition strategy in the next two examples.

Example 13.3: Array summation preview

Suppose that we want to apply function $f()$ to each element of an array A and sum the results. Mathematically, we want to compute the following sum.

$$S = \sum_{i=1}^N f(a(i))$$

In this formula, $a(i)$ is the i^{th} element of array A . We want to perform the task of applying function $f()$ to each element in the array and summing the result. Let's assume that we have a dual-core processor available to compute the sum. We split up the array so that each core performs the task on half of the array. Let S_1 and S_2 denote the sums computed by core 1 and core 2, respectively.

$$S_1 = \sum_{i=1}^{\lfloor \frac{N}{2} \rfloor} f(a(i)) \quad S_2 = \sum_{i=\lfloor \frac{N}{2} \rfloor + 1}^N f(a(i))$$

Assuming that N is even, both cores process exactly the same amount of data. For odd N , one of the cores processes one more data item than the other.

After S_1 and S_2 have been computed, one of the cores can add these two numbers together to get S . This strategy is illustrated in [Figure 13.3](#). After the two cores have completed their work on each half of the array, the individual sums are added together to produce the final sum. ■

Example 13.4: Matrix multiplication preview

The need to multiply matrices arises in many mathematical, scientific, and engineering applications. Suppose we are asked to write a program to multiply two square matrices A and B , which are both $n \times n$ matrices. The product matrix C will also be $n \times n$. A sequential program will compute each element of matrix C one at a time. However, a concurrent program can compute more than one element of C simultaneously using multiple cores.

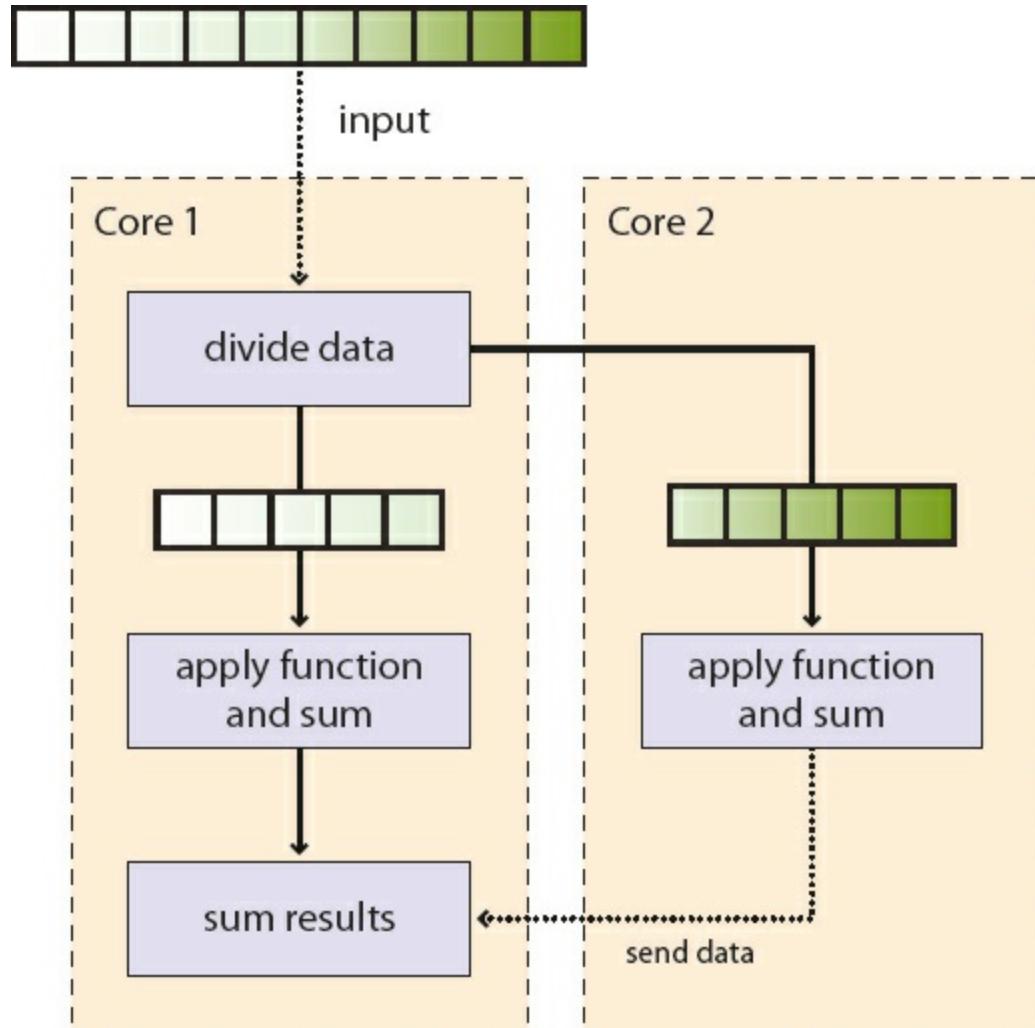


Figure 13.3: Computing the sum of a function of each element of an array.

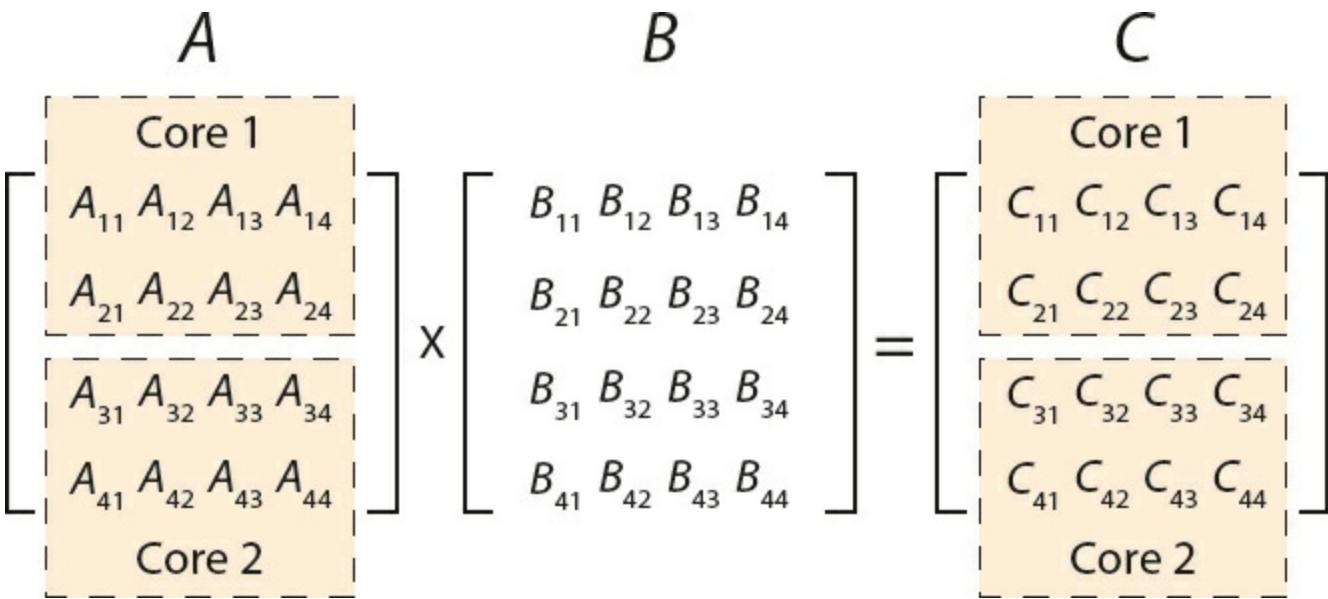


Figure 13.4: Data decomposition to multiply two 4×4 matrices. The two cores perform the same multiplication tasks but on different data from matrix A . The two cores compute the top and bottom two rows of C , respectively.

In this problem, the task is to multiply matrices A and B . Through domain decomposition, we can replicate this task on each core. As shown in Figure 13.4, each core computes only a portion of C . For example, if A and B are 4×4 matrices, we can ask one core to compute the product of the first two rows of A with all four columns of B to generate the first two rows of C . The second core computes the remaining two rows of C . Both cores can access matrices A and B . ■

13.3.3 Tasks and threads

It is the programmer's responsibility to divide his or her solution into a number of tasks and subtasks which will run on one or more cores on a processor. In previous sections, we described concurrent programs as if specific tasks could be assigned specific cores, but Java does not provide a direct way to do so.

Instead, a Java programmer must group together a set of tasks and subtasks into a *thread*. A thread is very much like a sequential program. In fact, all sequential programs have only one thread. A thread is a segment of executing code that runs through its instructions step by step. Each thread can run independently. If you have a single core processor, only one thread can run at a time, and all the threads will take turns. If you have a multicore processor, as many threads as there are cores can execute at the same time. You cannot pick which core a given thread will run on. In most cases, you will not even be able to tell which core a given thread is using.

Even though you cannot control which core a thread will use to execute, it takes care to package up the right set of tasks into a single thread of execution. Recall the previous examples of concurrent programming in this chapter.

Consider dividing the tasks in Example 13.1 into two threads. Tasks B and C in can be packaged into thread 1, and task D can be packaged into thread 2. This division is shown in Figure 13.5(a).

Tasks to evaluate different subexpressions in Example 13.2 can also be divided into two threads as shown in Figure 13.5(b). In many problems there are several reasonable ways of dividing a set of

subtasks into threads.

Note that these figures look exactly like the earlier figures, except that the tasks are grouped as threads instead of cores. This grouping is matches reality better, since we can control how the tasks are packaged into threads but not how they are assigned to cores.

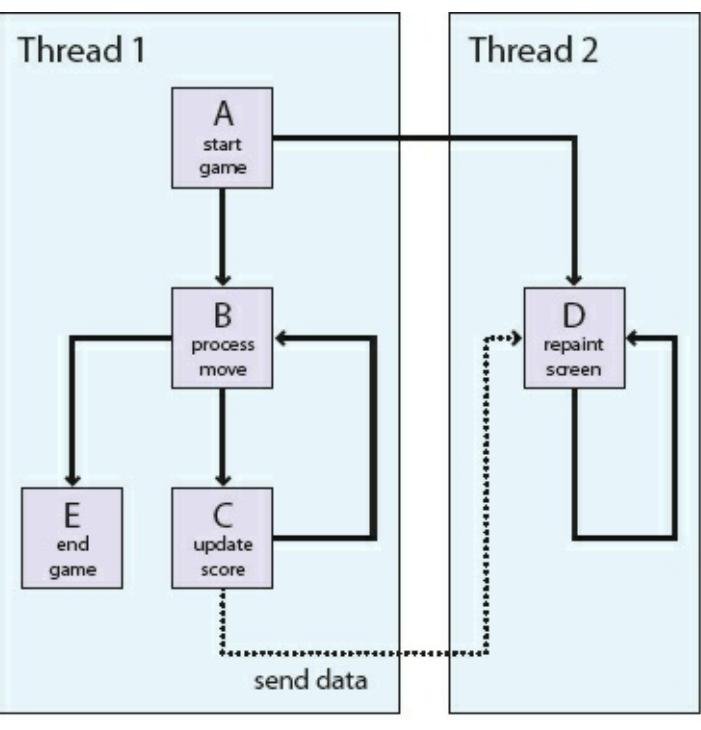
In both examples, we have two threads. It is possible that some other thread started these threads running. Every Java program, concurrent or sequential, starts with one thread. We will refer to this thread as the *main* thread since it contains the `main()` method.

[Examples 13.3](#) and [13.4](#) use multiple identical tasks. But these tasks operate on different data. Nevertheless, in [Example 13.3](#), the two tasks can be assigned two threads that operate on different portions of the input array. The task of summing the results from the two threads can either be a separate thread or a subtask included in one of the other threads. In [Example 13.4](#), the two tasks can again be assigned to two distinct threads that operate on different parts of the input matrix A to generate the corresponding portions of the output matrix C .

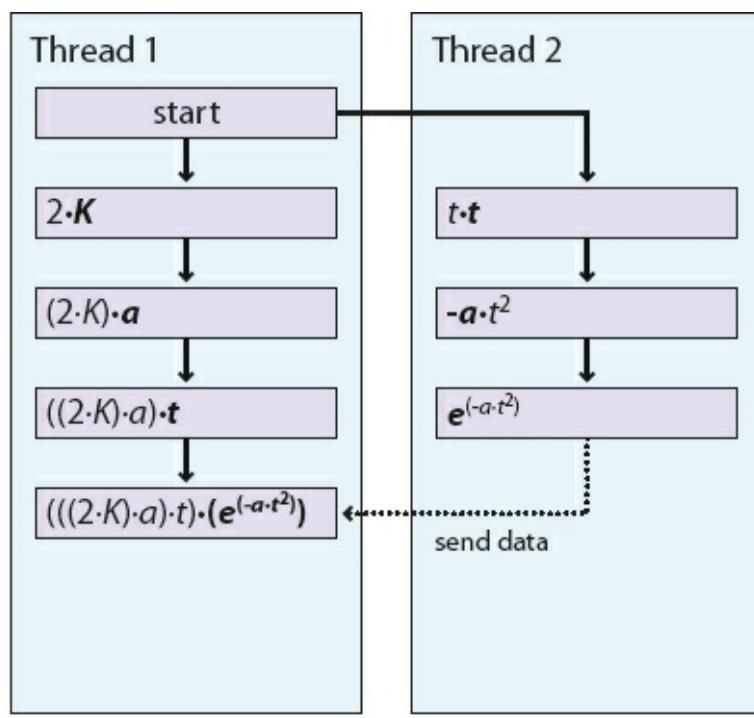
There can be many ways to package tasks into threads. There can also be many ways to decompose data into smaller chunks. The best ways to perform these subdivisions of tasks or data depend on the problem at hand and the processor architecture on which the program will be executed.

13.3.4 Memory architectures and concurrency

The two most important paradigms for concurrent programming are message passing and shared memory systems. Each paradigm handles communication between the various pieces of code running in parallel in a different way. Message passing systems such as MPI and PVM approach this problem by sending messages between otherwise independent pieces of code called processes. A process which is executing a task may have to wait until it receives a message from another process before it knows how to proceed. Messages can be sent from a single process to a single other or broadcast to many. Message passing systems are especially useful when the processors doing the work do not share memory.



(a)



(b)

Figure 13.5: (a) Tasks in a video game shown packaged into two threads. (b) Tasks to evaluate a mathematical expression shown packaged into two threads. Each thread may or may not run on the same core as the other.

In contrast, Java uses the shared memory paradigm. In Java, a programmer can create a number of threads which share the same memory space. Each thread is an object which can perform work. We described threads as a way to package up a group of tasks, and processes are another. People use the term *processes* to describe executing sections of code with separate memory and *threads* to describe executing sections of code with shared memory.

When you first learned to program, one of the biggest challenges was probably learning to solve a problem step by step. Each line of the program had to be executed one at a time, logically and deterministically. Human beings do not naturally think that way. We tend to jump from one thing to another, making inferences and guesses, thinking about two unrelated things at once, and so on. As you know well by now, it is only possible to write and debug programs because of the methodical way they work.

You can imagine the execution of a program as an arrow that points to one line of code, then the next, then the next, and so on. We can think of the movement of this arrow as the thread of execution of the program. The code does the actual work, but the arrow keeps track of where execution in the program currently is. The code can move the arrow forward, it can do basic arithmetic, it can decide between choices with if statements, it can do things repeatedly with loops, it can jump into a method and then come back. A single thread of execution can do all of these things, but it cannot be two places at once. It cannot both be dividing two numbers in one part of the program and evaluating an if statement in another. However, there is a way to split this thread of execution so that two or more threads are executing different parts of the program, and the next section will show you how it is done in Java.

13.4 Syntax: Threads in Java

13.4.1 The Thread class

Java, like many programming languages, provides the necessary features to package tasks and subtasks into threads. The Thread class and its subclasses provide the tools for creating and managing threads. For example, the following class definition allows objects of type ThreadedTask to be created. Such an object can be executed as a separate thread.

```
public class ThreadedTask extends Thread {  
    // Add constructor and body of class  
}
```

The constructor is written just like any other constructor, but there is a special run() method in Thread that can be overridden by any of its subclasses. This method is the starting point for the thread of execution associated with an instance of the class. Most Java applications begin with a single main thread which starts in a main() method. Additional threads must start somewhere, and that place is the run() method. A Java application will continue to run as long as at least one thread is active. The following example shows two threads, each evaluating a separate subexpression as in [Figure 13.5\(b\)](#).

Example 13.5: Thread samples

We will create Thread1 and Thread2 classes. The threads of execution created by instances of these classes compute, respectively, the two subexpressions in [Figure 13.5\(b\)](#) and save the computed values.

```
1 public class Thread1 extends Thread {  
2     private double K, a, t, value ;  
3  
4     public Thread1 ( double K, double a, double t) {  
5         this.K = K;  
6         this.a = a;  
7         this.t = t;  
8     }  
9     public void run () { value = 2*K*a*t; }  
10    public double getValue () { return value ; }  
11 }
```

```
1 public class Thread2 extends Thread {  
2     private double a, t, value ;  
3  
4     public Thread2 ( double a, double t) {  
5         this.a = a;  
6         this.t = t;  
7     }  
8     public void run () { value = Math.exp (-a*t*t); }
```

```
9     public double getValue () { return value ; }  
10 }
```

The run() method in each thread above computes a subexpression and saves its value. We show how these threads can be executed to solve the math expression problem in [Example 13.6](#). ■

13.4.2 Creating a thread object

Creating an object from a subclass of Thread is the same as creating any other object in Java. For example, we can instantiate the Thread1 class above to create an object called thread1.

```
Thread1 thread1 = new Thread1 ( 15.1 , 2.8 , 7.53 );
```

Using the `new` keyword to invoke the constructor creates a Thread1 object, but it does not start executing it as a new thread. As with all other classes, the constructor initializes the values inside of the new object. A subclass of Thread can have many different constructors with whatever parameters its designer thinks appropriate.

13.4.3 Starting a thread

To start the thread object executing, its `start()` method must be called. For example, the `thread1` object created above can be started as follows.

```
thread1.start ();
```

Once started, a thread executes independently. When a thread needs to share data with another thread, it might have to wait.

13.4.4 Waiting for a thread

Often some thread, main or otherwise, needs to wait for another thread before proceeding further with its execution. The `join()` method is used to wait for a thread to finish executing. For example, whichever thread executes the following code will wait for `thread1` to complete.

```
thread1.join ();
```

Calling `join()` is a *blocking* call, meaning that the code calling this method will wait until it returns. Since it can throw a checked `InterruptedException`, while the code is waiting, the `join()` method is generally used within a `try-catch` block. We can add a `try-catch` block to the `thread1` example so that we can recover from being interrupted while waiting for `thread1` to finish.

```
try {  
    System.out.println (" Waiting for thread 1...");  
    thread1.join ();  
    System.out.println (" Thread 1 finished !");  
}  
  
catch ( InterruptedException e ) {  
    System.out.println (" Thread 1 didn 't finish !");  
}
```

Note that the `InterruptedException` is thrown because the main thread was interrupted while waiting for `thread1` to finish. If the `join()` call returns, then `thread1` must have finished, and we inform the user. If an `InterruptedException` is thrown, some outside thread must have interrupted the main thread, forcing it to stop waiting for `thread1`.

In earlier versions of Java, there was a `stop()` method which would stop an executing thread. Although this method still exists, it has been deprecated and should not be used.

Exercise 13.1

Example 13.6: Math expression threads

Now that we have the syntax to start threads and wait for them to finish, we can use the threads defined in [Example 13.5](#) with a main thread to make our first complete concurrent program. The main thread in class `MathExpression` creates and starts the worker threads `thread1` and `thread2` and waits for their completion. When the two threads complete their execution, we can ask each for its computed value. The main thread then prints the product of these values, which is the result of the expression we want to evaluate.

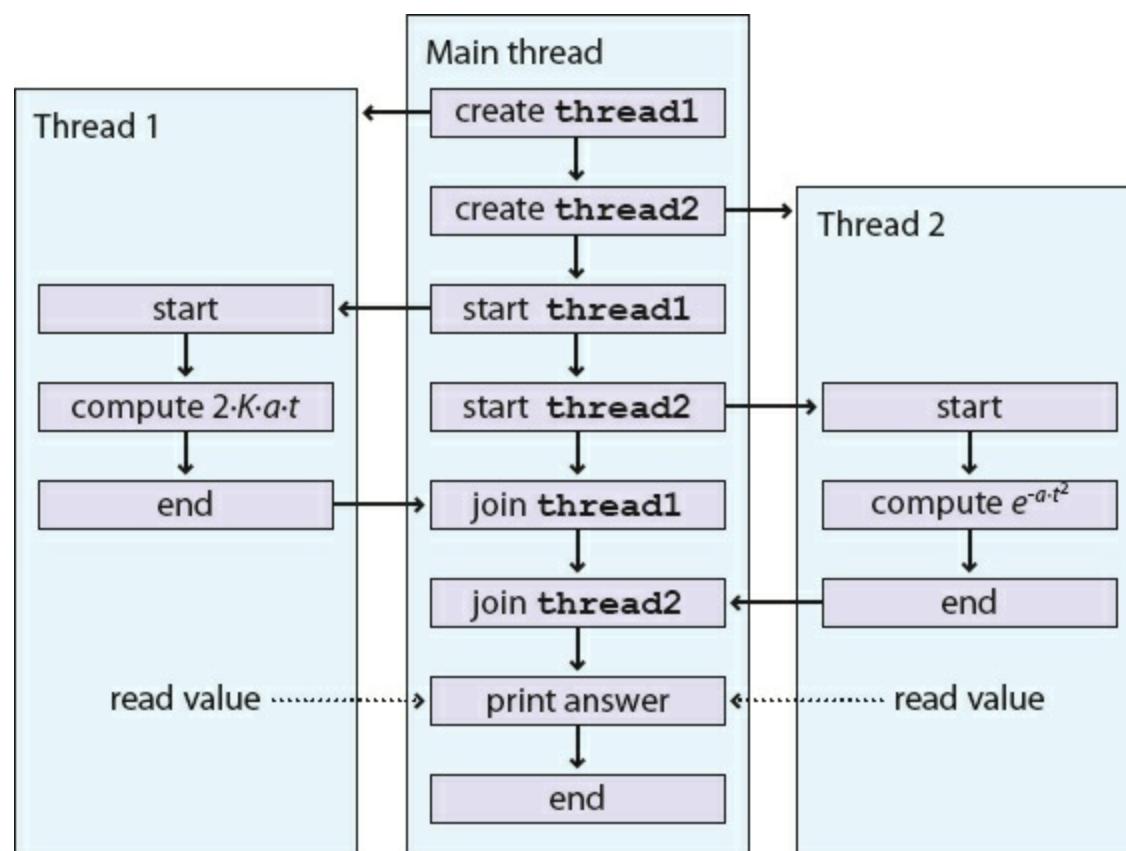


Figure 13.6: Creation, starting, and joining of threads in `MathExpression`, `Thread1`, and `Thread2`.

```

1 public class MathExpression {
2     public static void main ( String [] args ) {
3         double K = 120 , a = 1.2 , t = 2;
4         Thread1 thread1 = new Thread1 ( K, a, t );
5         Thread2 thread2 = new Thread2 ( a, t );
6         thread1.start () // start thread1
    
```

```

7     thread2.start(); // start thread2
8     try { // wait for both threads to complete
9         thread1.join();
10        thread2.join();
11        System.out.println (" Value of expression : " +
12             thread1.getValue ()* thread2.getValue ());
13    }
14    catch ( InterruptedException e) {
15        System.out.println ("A thread didn 't finish !");
16    }
17 }
18 }
```

We want to make it absolutely clear when threads are created, start executing, and finish. These details are crucial for the finer points of concurrent Java programming. In [Figure 13.5](#), it appears as if execution of the concurrent math expression evaluation begins with Thread 1 which spawns Thread 2. Although that figure explains the basics of task decomposition well, the details are messier for real Java code.

In the code above, execution starts with the `main()` method in `MathExpression`. It creates `Thread1` and `Thread2` objects and waits for them to finish. Then, it reads the values from the objects after they have stopped executing. We could have put the `main()` method in the `Thread1` class, omitting the `MathExpression` class entirely. Doing so would make the execution match [Figure 13.5](#) more closely, but it would make the two `Thread` subclasses less symmetrical: The main thread and `thread1` would both independently execute code inside the `Thread1` class while only `thread2` would execute code inside the `Thread2` class.

[Figure 13.6](#) shows the execution of `thread1` and `thread2` and the main thread. Note that the creation and start of the main thread is done implicitly by the JVM while `thread1` and `thread2` are created explicitly and started by the main thread. Even after the threads associated with `thread1` and `thread2` have stopped running, the objects continue to exist. Methods and fields can continue to be accessed. ■

13.4.5 The Runnable interface

Although it is possible to create Java threads by inheriting from the `Thread` class directly, the Java API allows the programmer to use an interface instead.

As an example, the `Summer` class takes an array of `int` values and sums them up within a given range. If multiple instances of this class are executed as separate threads, each one can sum up different parts of an array.

```

1 public class Summer implements Runnable {
2     int [] array ;
3     int lower ;
4     int upper ;
5     int sum = 0;
6 }
```

```

7  public Summer ( int [] array , int lower , int upper ) {
8      this.array = array ;
9      this.lower = lower ;
10     this.upper = upper ;
11 }
12
13     public void run () {
14         for ( int i = lower ; i < upper ; i++ )
15             sum += array [i];
16     }
17
18     public int getSum () { return sum ; }
19 }
```

This class is very similar to one that inherits from Thread. Imagine for a moment that the code following Summer is `extends` Thread instead of `implements` Runnable. The key thing a class derived from Thread needs is an overridden `run()` method. Since only the `run()` method is important, the designers of Java provided a way to create a thread using the Runnable interface. To implement this interface, only a `public void run()` method is required.

When creating a new thread, there are some differences in syntax between the two styles. The familiar way of creating and running a thread from a Thread subclass is as follows.

```
Summer summer = new Summer ( array , lower , upper );
summer.start ();
```

Since Summer does not inherit from Thread, it does not have a `start()` method, and this code will not compile. When a class only implements Runnable, it is still necessary to create a Thread object and call its `start()` method. Thus, an extra step is needed.

```
Summer summer = new Summer ( array , lower , upper );
Thread thread = new Thread ( summer );
thread.start ();
```

This alternate way of implementing the Runnable interface seems more cumbersome than inheriting directly from Thread, since you have to instantiate a separate Thread object. However, most developers prefer to design classes that implement Runnable instead of inheriting from Thread. Why? Java only allows for single inheritance. If your class implements Runnable, it is free to inherit from another parent class with the features you want.

Exercise 13.2

Example 13.7: Array of threads

In domain decomposition, we often need to create multiple threads, all from the same class. As an example, consider the following thread declaration.

```
1  public class NumberedThread extends Thread {
```

```

2     private int value ;
3
4     public NumberedThread ( int input ) { value = input ; }
5
6     public void run () {
7         System.out.println (" Thread " + value );
8     }
9 }
```

Now suppose that we want to create 10 thread objects of type NumberedThread, start them, and then wait for them to complete.

```

NumberedThread [] threads = new NumberedThread [10];
for ( int i = 0; i < threads.length ; i++ ) {
    threads [i] = new NumberedThread (i);
    threads [i].start ();
}
try {
    for ( int i = 0; i < threads.length ; i++ )
        threads [i].join ();
}
catch ( InterruptedException e ) {
    System.out.println ("A thread didn 't finish !");
}
```

First, we declare an array to hold references to NumberedThread objects. Like any other type, we can make an array to hold objects that inherit from Thread. The first line of the **for** loop instantiates a new NumberedThread objects, invoking the constructor which stores the current iteration of the loop into the value field. The reference to each NumberedThread object is stored in the array. Remember that the constructor does not start a new thread running. The second line of the **for** loop does that.

We are also interested in when the threads stop. Calling the **join()** method forces the main thread to wait for each thread to finish.

The entire second **for** loop is nested inside of a **try** block. If the main thread is interrupted while waiting for any of the threads to finish, an **InterruptedException** will be caught. As before, we warn the user that a thread didn't finish. For production-quality code, the **catch** block should handle the exception in such a way that the thread can recover and do useful work even though it didn't get what it was waiting for. ■

13.5 Examples: Concurrency and speedup

Speedup is one of the classic motivations for writing concurrent programs. To understand speedup, let's assume we have a problem to solve. We write two programs to solve this problem, one that is sequential and another that is concurrent and, hence, able to exploit the multiple cores. Let t_s be the

average time to execute the sequential program and t_c the average time to execute the concurrent program. Sometimes we will use the notation t_c^k to refer to the execution time of a concurrent program with k threads. So that the comparison is meaningful, assume that both programs are executed on the same computer. The speedup obtained from concurrent programming is defined as follows.

$$\text{speedup} = \frac{t_s}{t_c}$$

Speedup measures how much faster the concurrent program executes relative to the sequential program. Ideally, we expect $t_c < t_s$, making the speedup greater than 1. However, simply writing a concurrent program does not necessarily make it faster than the sequential version.

Exercise 13.6

To determine speedup, we need to measure t_s and t_c . Time in a Java program can easily be measured with the following two static methods in the System class.

Exercise 13.8

```
public static long currentTimeMillis()  
public static long nanoTime()
```

The first of these methods returns the current time in milliseconds (ms). A *millisecond* is 0.001 seconds. This method gives the difference between the current time on your computer's clock and midnight of January 1, 1970 coordinated universal time (UTC). This point in time is used for many timing features on many computer platforms and is called the *Unix epoch*. The second method returns the current time in nanoseconds (ns). A *nanosecond* is 0.000001 seconds. This method also gives the difference between the current time and some fixed time, which is system dependent and not necessarily the Unix epoch. The System.nanoTime() method can be used when you want timing precision finer than milliseconds; however, the level of accuracy it returns is again system dependent. The next example show how to use these methods to measure execution time.

Example 13.8: Measuring execution time

Suppose we want to measure the execution time of a piece of Java code. For convenience, we can assume this code is contained in the work() method. The following code snippet measures the time to execute work().

```
long start = System.currentTimeMillis();  
work();  
long end = System.currentTimeMillis();  
System.out.println("Elapsed time : " + (end - start) + " ms");
```

The output will give the execution time for work() measured in milliseconds. To get the execution time in nanoseconds, use the System.nanoTime() method instead. ■

Exercise 13.15

Now that we have the tools to measure execution time, we can measure speedup. The next few

examples show the speedup (or lack of it) that we can achieve using a concurrent solution to a few simple problems.

Exercise 13.16

Example 13.9: Math expression speedup

Recall the concurrent program in [Example 13.6](#) to evaluate a simple mathematical expression. This program uses two threads. We executed this multi-threaded program on an iMac computer with an Intel Core 2 Duo running at 2.16 Ghz. The execution time was measured at 1,660,000 nanoseconds. We also wrote a simple sequential program to evaluate the same expression. It took 4,100 nanoseconds to execute this program on the same computer. Plugging in these values for t_c and t_s , we can find the speedup.

$$\text{speedup} = \frac{t_s}{t_c} = \frac{4,100}{1,660,000} = 0.00246$$

This speedup is much less than 1. Although the result may be surprising, the concurrent program with two threads executes much slower than the sequential program. In this example, the cost of creating, running, and joining threads outweighed the benefits of concurrent calculation on two cores.

■

Exercise 13.7

Example 13.10: Array summation

In [Example 13.3](#), we introduced the problem of applying a function to every value in an array and then summing the results. Let's say that we want to apply the sine function to each value. To solve this problem concurrently, we partition the array evenly among a number of threads, using the domain decomposition strategy. Each thread finds the sum of the sines of the values in its part of the array. One factor that determines whether or not we achieve speedup is the complexity of the function, in this case sine, that we apply. Although we may achieve speedup with sine, a simpler function such as doubling the value might not create enough work to justify the overhead of using threads.

We create class SumThread whose run() method sums the sines of those elements of the array in its assigned partition.

```
1 import java.util.Random;  
2  
3 public class SumThread extends Thread {  
4     private static double [] data ;  
5     private double sum = 0.0;  
6     private int lower ;  
7     private int upper ;  
8     public static final int SIZE = 1000000;  
9     public static final int THREADS = 8;  
10  
11    public SumThread ( int lower , int upper ) {
```

```
12     this.lower = lower ;
13     this.upper = upper ;
14 }
```

First, we set up all the fields that the class will need. We fix the array size at 1,000,000 and the number of threads at 8, but these values could easily be changed or read as input instead. In its constructor, a SumThread takes the lower and upper bounds of its partition. Like most ranges we discuss, the lower bound is inclusive though the upper bound is exclusive.

```
16 public void run() {
17     for ( int i = lower ; i < upper ; i++ )
18         sum += Math.sin ( data [i]);
19 }
20
21 public double getSum () { return sum ; }
```

In the `for` loop of the `run()` method, the `SumThread` finds the sine of each number in its array partition and adds that value to its running sum. The `getSum()` method is an accessor that allows the running sum to be retrieved.

```
23 public static void main ( String [] args ) {
24     data = new double [ SIZE ];
25     Random random = new Random ();
26     int start = 0;
27     for ( int i = 0; i < SIZE ; i++ )
28         data [i] = random.nextDouble ();
29     SumThread threads = new SumThread [ THREADS ];
30     int quotient = data.length / THREADS ;
31     int remainder = data.length % THREADS ;
32     for ( int i = 0; i < THREADS ; i++ ) {
33         int work = quotient ;
34         if( i < remainder )
35             work++;
36         threads [i] = new SumThread ( start , start + work );
37         threads [i].start ();
38         start += work ;
39     }
```

The `main()` method begins by instantiating the array and filling it with random values. Note that the array is a static field so that it can be shared by all instances of `SumThread`. Then each thread is created with lower and upper bounds that mark its array partition. If the process using the array length and the number of threads to determine upper and lower bounds doesn't make sense, refer to [Section 6.11](#) which describes the fair division of work to threads. If the length of the array is not divisible by the number of threads, simple division isn't enough. After creating each thread, its `start()` method is called.

```

41     double sum = 0.0;
42     try {
43         for ( int i = 0; i < THREADS ; i++ ) {
44             thread [i].join ();
45             sum += thread [i].getSum ();
46         }
47         System.out.println (" Sum : " + threads [0].getSum ());
48     }
49     catch ( InterruptedException e ) {
50         e.printStackTrace ();
51     }
52 }
53 }
```

After the threads have started working, the main thread creates its own running total and iterates through each thread waiting for it to complete. When each thread is done, the main thread adds its value to the running total. If the main thread is interrupted while waiting for a thread to complete, the stack trace is printed. Otherwise, the final sum is printed out. ■

Exercise 13.12

Exercise 13.19

Exercise 13.20

Example 13.11: Matrix multiplication

In [Example 13.4](#), we discussed the importance of matrix operations in many applications. Now that we know the necessary Java syntax, we can write a concurrent program to multiply two square matrices A and B and compute the resultant matrix C . If these matrices have n rows and n columns, the value at the i^{th} row and j^{th} column of C is

$$C_{ij} = \sum_{k=1}^n A_{ik}B_{kj} = A_{i1}B_{1j} + A_{i2}B_{2j} + \dots + A_{in}B_{nj}$$

In Java, it is natural for us to store matrices as 2-dimensional arrays. To do this multiplication sequentially, the simplest approach uses three nested `for` loops. The code below is a direct translation of the mathematical notation, but we do have to be careful about bookkeeping. Note that mathematical notation often uses uppercase letters to represent matrices though the Java convention is to start all variable names with lowercase letters.

```

for ( int i = 0; i < c.length ; i++ )
    for ( int j = 0; j < c[i].length ; j++ )
        for ( int k = 0; k < b.length ; k++ )
            c[i][j] += a[i][k] * b[k][j];
```

The first step in making a concurrent solution to this problem is to create a Thread subclass which

will do some part of the matrix multiplication. Below is the MatrixThread class which will compute a number of rows in the answer matrix c.

```
1 public class MatrixThread extends Thread {  
2     private double [][] a;  
3     private double [][] b;  
4     private double [][] c;  
5     private int lower ;  
6     private int upper ;  
7  
8     public MatrixThread ( double [][] a, double [][] b,  
9             double [][] c, int lower , int upper ) {  
10         this.a = a;  
11         this.b = b;  
12         this.c = c;  
13         this.lower = lower ;  
14         this.upper = upper ;  
15     }  
16  
17     public void run () {  
18         for ( int i = lower ; i < upper ; i++ )  
19             for ( int j = 0; j < c[i].length ; j++ )  
20                 for ( int k = 0; k < b.length ; k++ )  
21                     c[i][j] += a[i][k] * b[k][j];  
22     }  
23 }
```

The constructor for MatrixThread stores references to the arrays corresponding to matrices A , B , and C as well as lower and upper bounds on the rows of C to compute. The body of the run() method is identical to the sequential solution except that its outermost loop runs only from lower to upper instead of through all the rows of the result. It is critical that each thread is assigned a set of rows that does not overlap with the rows another thread has. Not only would having multiple threads compute the same row be inefficient, it could very likely lead to an incorrect result, as we will see in [Chapter 14](#).

The following client code uses an array of MatrixThread objects to perform a matrix multiplication. We assume that an `int` constant named THREADS has been defined which gives the number of threads we want to create.

```
MatrixThread [] threads = new MatrixThread [ THREADS ];  
int quotient = c.length / THREADS ;  
int remainder = c.length % THREADS ;  
int start = 0;  
for ( int i = 0; i < THREADS ; i++ ) {  
    int rows = quotient ;
```

```

if( i < remainder )
    rows++;
threads [i] = new MatrixThread ( a, b, c, start , start + rows );
threads [i].start ();
start += rows ;
}
try {
    for ( int i = 0; i < THREADS ; i++ )
        threads [i].join ();
}
catch ( InterruptedException e ) {
    e.printStackTrace ();
}

```

We loop through the array, creating a `MatrixThread` object for each location. As in the previous example, we use the approach described in [Section 6.11](#) to allocate rows to each thread fairly. Each new `MatrixThread` object is given a reference to each of the three matrices as well as an inclusive starting and an exclusive ending row. After the `MatrixThread` objects are created, we start them running with the next line of code.

Next, there is a familiar `for` loop with the `join()` calls that force the main thread to wait for the other threads to finish. Presumably, code following this snippet will print the values of the result matrix or use it for some other calculations. If we don't use the `join()` calls to be sure the threads have finished, we might print out a result matrix that has only been partially filled in.

We completed the code for threaded matrix multiplication and executed it on an iMac computer running on an Intel running at 2.16 Ghz. The program was executed for matrices of different sizes ($n \times n$). For each size, the sequential and concurrent execution times in seconds and the corresponding speedup are listed in the following table.

Size (n)	t_s (s)	t_c (s)	Speedup
100	0.013	0.9	0.014
500	1.75	4.5	0.39
1000	15.6	10.7	1.45*

Only with 1000×1000 matrices did we see improved performance when using two threads. In that case, we achieved a speedup of 1.45, marked with an asterisk. In the other two cases, performance became worse. ■

Exercise 13.17

Exercise 13.14

Now that we have seen how multiple threads can be used together, a number of questions should be coming to the forefront. Who decides when these threads run? How is processor time shared between threads? Can we make any assumptions about the order in which the threads will run? Can we affect this order?

These questions focus on thread scheduling. Because different concurrent systems handle scheduling differently, we will only describe scheduling in Java. Although sequential programming is all about precise control over what happens **next**, concurrency takes much of this control away from the programmer. When threads are scheduled and which processor they run on is handled by a combination of the JVM and the OS. With normal JVMs, there is no explicit way to access the scheduling and alter it to your liking.

Of course, there are a number of implicit ways a programmer can affect scheduling. In Java, as in several other languages and programming systems, threads have *priorities*. Higher priority threads run more often than lower priority threads. Some threads are performing mission-critical operations which must be carried out as quickly as possible, and some threads are just doing periodic tasks in the background. A programmer can set the priorities accordingly.

Setting priorities gives only a very general way of controlling which thread will run. The threads themselves might have more specific information about when they will and will not need processor time. A thread may need to wait for a specific event and will not need to run until then. Java allows threads to interact with the scheduler through `Thread.sleep()` and `Thread.yield()`, which we will discuss in [Section 13.7](#), and through the `wait()`, method which we will discuss in [Chapter 14](#).

13.6.1 Nondeterminism

In Java, the mapping of a thread inside the JVM to a thread in the OS varies. Some implementations give each Java thread an OS thread, some put all Java threads on a single OS thread (with the side effect of preventing concurrency), and some allow for the possibility of changing which OS thread a Java thread uses. Thus, the performance and, in some cases, the correctness of your program might vary, depending on which system you are running. This is, yet again, one of those times when Java is platform independent... but not entirely.

Unfortunately, the situation is even more complicated. Making threads part of your program means that the same program could run differently on the **same** system. The JVM and the OS have to cooperate to schedule threads, and both programs are complex mountains of code which try to balance many factors. If you create three threads, there is no guarantee that the first will run first, the second second, and the third third, not even if it happens that way the first 10 times you run the program. Exercise 13.18 shows that the pattern of thread execution can vary a lot.

Exercise 13.18

In all the programs before this chapter, the same sequence of input would always produce the same sequence of output. Perhaps the biggest hurdle created by this nondeterminism is that programmers must shift their paradigm considerably. The processor can switch between executions of threads at any time, even in the middle of operations. Every possible interleaving of thread execution can crop up at some point. Unless you can be sure that your program behaves properly for all of them, you may never be able to debug your code completely. What is so insidious about some nondeterministic bugs

is that they can occur rarely and be almost impossible to reproduce. In this chapter, we introduce how to create and run threads, but making these threads interact properly is a major problem we tackle in subsequent chapters.

After those dire words of warning, we'd like to remind you that nondeterminism is not in itself a bad thing. Many threaded applications with a lot of input and output, such as server applications, necessarily exist in a nondeterministic world. For these programs, many different sequences of thread execution may be perfectly valid. Each individual program may have a different definition of correctness. For example, if a stock market server receives two requests to buy the last share of a particular stock at almost the same time from two threads corresponding to two different clients, it might be correct for either one of them to get that last share. However, it would never be correct for **both** of them to get it.

13.6.2 Polling

So far the only mechanism we have introduced for coordinating different threads is using the `join()` method to wait for a thread to end. Another technique is *polling*, or *busy waiting*. The idea is to keep checking the state of one thread until it changes.

There are a number of problems with this approach. The first is that it wastes CPU cycles. Those cycles spent by the waiting thread continually checking could have been used productively by some other thread in the system. The second problem is that we have to be certain that the state of the thread we are waiting for won't change back to the original state or to some other state. Because of the unpredictability of scheduling, there is no guarantee that the waiting thread will read the state of the other thread when it has the correct value.

We bring up polling partly because it has a historical importance to parallel programming, partly because it can be useful in solving some problems in this chapter, and partly because we want you to understand the reasons why we need better techniques for thread communication.

Exercise 13.11

13.7 Syntax: Thread states

A widely used Java tool for manipulating scheduling is the `Thread.sleep()` method. This method can be called any time you want a thread to do nothing for a set period of time. Until the sleep timer expires, the thread will not be scheduled for any CPU time, unless it is interrupted. To make a thread of execution sleep, call `Thread.sleep()` in that thread of execution with a number of milliseconds as a parameter. For example, calling `Thread.sleep(2000)` will make the calling thread sleep for two full seconds.

Another useful tool is the `Thread.yield()` method. It gives up use of the CPU so that the next waiting thread can run. To use it, a thread calls `Thread.yield()`. This method is very useful in practice, but according to official documentation, the JVM does not **have** to do anything when a `Thread.yield()` call happens. The Java specification does not demand a particular implementation. A JVM could ignore a `Thread.yield()` call completely, but most JVMs will move on to the next thread in the schedule.

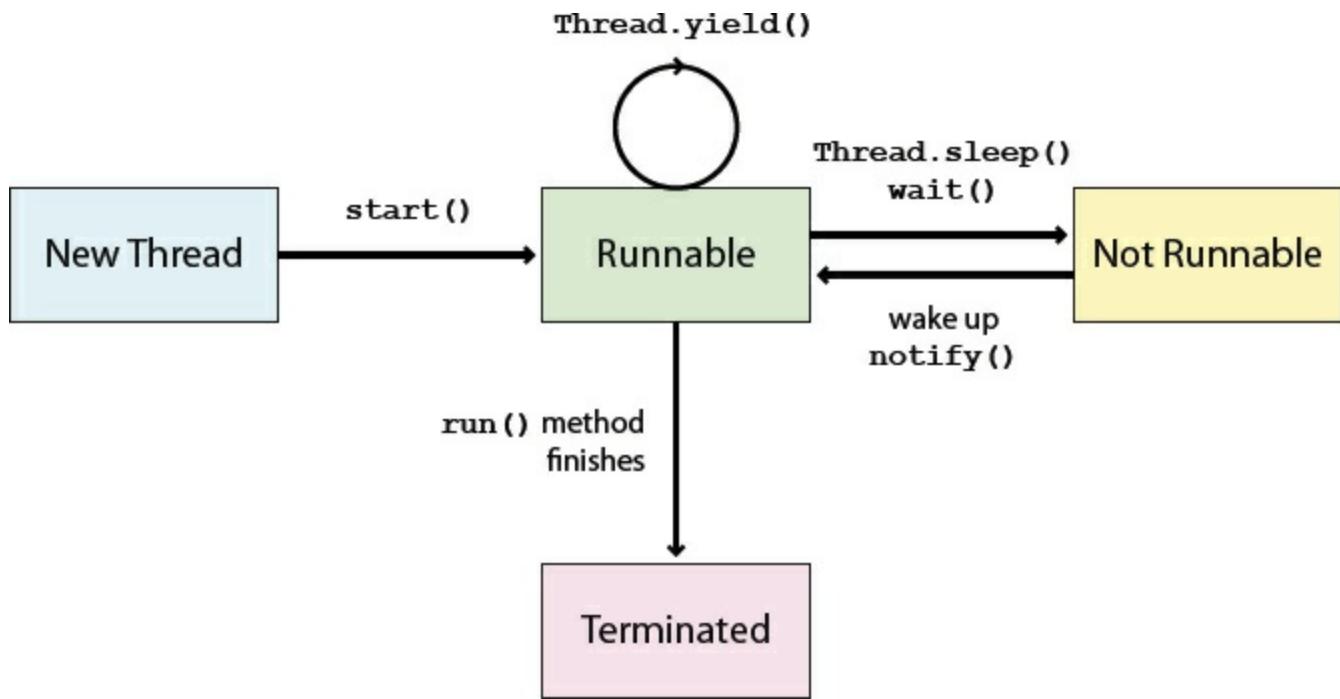


Figure 13.7: Thread states and transitions.

Figure 13.7 shows the lifecycle of a thread. A thread begins its life in the New Thread state, after the constructor is called. When the `start()` method is called, the thread begins to run and transitions to the Runnable state. Being Runnable doesn't necessarily mean that the thread is executing at any given moment but that it is ready to run at any time. When in the Runnable state, a thread may call `Thread.yield()`, relinquishing use of the processor, but it will still remain Runnable.

However, if a thread goes to sleep with a `Thread.sleep()` call, waits for a condition to be true using a `wait()` call, or performs a blocking I/O operation, the thread will transition to the Not Runnable state. Not Runnable threads cannot be scheduled for processor time until they wake up, finish waiting, or complete their I/O. The final state is Terminated. A thread becomes Terminated when its `run()` method finishes. A Terminated thread cannot become Runnable again and is no longer a separate thread of execution.

Any object with a type that is a subclass of `Thread` can tell you its current state using the `getState()` method. This method returns an *enum* type, whose value must come from a fixed list of constant objects. These objects are `Thread.State.NEW`, `Thread.State.RUNNABLE`, `Thread.State.BLOCKED`, `Thread.State.WAITING`, `Thread.State.TIMED_WAITING`, and `Thread.State.TERMINATED`. Although the others are self explanatory, we lump the `Thread.State.BLOCKED`, `Thread.State.WAITING`, and `Thread.State.TIMED_WAITING` values into the Not Runnable state since the distinction between the three is not important for us.

Threads also have priorities in Java. When an object that is a subclass of `Thread` is created in Java, its priority is initially the same as the thread that creates it. Usually, this priority is `Thread.NORM_PRIORITY`, but there are some special cases when it is a good idea to raise or lower this priority. Avoid changing thread priorities because it increases platform dependence and because

the effects are not always predictable. Be aware that priorities exist, but don't use them unless and until you have a good reason.

Example 13.12: Military marching

Let's apply the ideas discussed above to a lighthearted example. You might be familiar with sound of soldiers marching: "Left, Left, Left, Right, Left!" We can design a thread that prints Left and another thread that prints Right. We can combine the two to print the correct sequence for marching and loop the whole thing 10 times so that we can see how accurately place the words. We want to use the scheduling tools discussed above to get the timing right. Let's try Thread.sleep() first.

```
1 public class LeftThread extends Thread {  
2     public void run () {  
3         for ( int i = 0; i < 10; i++ ) {  
4             System.out.print (" Left ");  
5             System.out.print (" Left ");  
6             System.out.print (" Left ");  
7             try { Thread.sleep (10) ; }  
8             catch ( InterruptedException e ) {  
9                 e.printStackTrace ();  
10            }  
11            System.out.println (" Left ");  
12        }  
13    }  
14 }
```

Class LeftThread has a **for** loop which prints out Left three times, waits for 50 milliseconds, prints out Left again, then repeats.

```
1 public class RightThread extends Thread {  
2     public void run () {  
3         try { Thread.sleep (5); }  
4         catch ( InterruptedException e ) {  
5             e.printStackTrace ();  
6         }  
7         for ( int i = 0; i < 10; i++ ) {  
8             System.out.print (" Right ");  
9             try { Thread.sleep (10) ; }  
10            catch ( InterruptedException e ) {  
11                e.printStackTrace ();  
12            }  
13        }  
14    }  
15 }
```

Class RightThread waits for 5 milliseconds to get synchronized, then has a **for** loop which prints out Right, waits for 10 milliseconds, and repeats. The driver program below creates a thread for each of these classes and then starts them. If you run this program, you should see 10 lines of Left Left Left Right Left, but there are a few problems.

```
1 public class MilitaryMarching {  
2     public static void main ( String [] args ) {  
3         LeftThread left = new LeftThread ();  
4         RightThread right = new RightThread ();  
5         left.start ();  
6         right.start ();  
7         try {  
8             left.join ();  
9             right.join ();  
10        }  
11        catch ( InterruptedException e ) {  
12            e.printStackTrace ();  
13        }  
14    }  
15 }
```

The first problem is that we have to wait some amount of time between calls. We could shorten the Thread.sleep() calls, but there are limits on the resolution of the timer. The bigger problem is that the two threads can sometimes get out of sync. If you run the program many times, you may see a Right out of place once in a while. If you increase the repetitions of the **for** loops to a larger number, the errors are more likely. Whether or not you see errors is somewhat system dependent. We can try Thread.yield() instead of Thread.sleep().

```
1 public class LeftYieldThread extends Thread {  
2     public void run () {  
3         for ( int i = 0; i < 10; i++ ) {  
4             System.out.print (" Left ");  
5             System.out.print (" Left ");  
6             System.out.print (" Left ");  
7             Thread.yield ();  
8             System.out.println (" Left ");  
9         }  
10    }  
11 }
```

```
1 public class RightYieldThread extends Thread {  
2     public void run () {  
3         for ( int i = 0; i < 10; i++ ) {  
4             System.out.print (" Right ");  
5             Thread.yield ();
```

```
6      }
7  }
8 }
```

These new versions of the two classes have essentially replaced calls to `Thread.sleep()` with calls to `Thread.yield()`. Without the need for exception handling, the code is simpler, but we have traded one set of problems for another. If there are other threads operating in the same application, they will be scheduled in ways that will interfere with the pattern of yielding. Also, if you are running this code on a machine with a single processor and a single core, you have a good chance of seeing something which matches the expected output. If you are running this on multiple cores, everything will be jumbled. It is likely that the `LeftYieldThread` will be running on one processor with the `RightYieldThread` on another. In that case, neither has any competition to yield to.

Finally, let us look at a polling solution which still falls short of the mark. To do this, we need state variables inside of each class to keep track of whether or not it is done. Each thread needs a reference to the other thread to make queries, and the driver program must be updated to add these in before starting the threads.

```
1 public class LeftPollingThread extends Thread {
2     private RightThread right ;
3     private boolean done = false ;
4
5     public void setRight ( RightPollingThread right ) {
6         this.right = right ;
7     }
8
9     public void run () {
10        for ( int i = 0; i < 10; i++ ) {
11            System.out.print (" Left ");
12            System.out.print (" Left ");
13            System.out.print (" Left ");
14            done = true ;
15            while ( ! right.isDone () );
16            right.setDone ( false );
17            System.out.println (" Left ");
18        }
19    }
20
21    public boolean isDone () { return done ; }
22    public void setDone ( boolean value ) { done = value ; }
23 }
```

```
1 public class RightPollingThread extends Thread {
2     private LeftThread left ;
3     private boolean done = false ;
```

```

4
5     public void setLeft ( LeftPollingThread left ) {
6         this.left = left ;
7     }
8
9     public void run () {
10        for ( int i = 0; i < 10; i++ ) {
11            while ( ! left.isDone () );
12            left.setDone ( false );
13            System.out.print (" Right ");
14            done = true ;
15        }
16    }
17
18    public boolean isDone () { return done ; }
19    public void setDone ( boolean value ) { done = value ; }
20 }
```

Whether single core or multicore, this solution will always give the right output. Or it should. Java experts will point out that we are violating a technicality of the Java Memory Model. Because we are not using synchronization tools, we have no guarantee that the change of the done variable will even be **visible** from one thread to another. In practice, this problem should affect you rarely, but to be safe, both of the done variables should be declared with the keyword **volatile**. This keyword makes Java aware that the value may be accessed at any time from arbitrary threads.

Another issue is that there is **no** concurrency. Each thread must wait for the other to complete. Of course, this problem does not benefit from a concurrent solution, but applying this solution to problems which can benefit from concurrency might cause performance problems. Each thread wastes time busy waiting in a **while** loop for the other to be done, consuming CPU cycles while it does so. You will notice that the code must still be carefully written. Each thread must set the other thread's done value to **false**. If threads were responsible for setting their own done values to **false**, one thread might print its information and go back to the top of the **for** loop before the other thread had reset its own done to **false**.

In short, coordinating two or more threads together is a difficult problem. None of the solutions we give here are fully acceptable. We introduce better tools for coordination and synchronization in [Chapter 14](#). ■

13.8 Solution: Deadly virus

Finally, we give the solution to the deadly virus problem. By this point, the threaded part of this problem should not seem very difficult. It is simpler than some of the examples, such as matrix multiplication. We begin with the worker class FactorThread that can be spawned as a thread.

Program 13.1: Thread class used to find the sum of the two factors of a large odd composite.

(FactorThread.java)

```
1 public class FactorThread extends Thread {  
2     private long lower ;  
3     private long upper ;  
4  
5     public FactorThread ( long lower , long upper ) {  
6         this.lower = lower ;  
7         this.upper = upper ;  
8     }  
9  
10    public void run () {  
11        if( lower % 2 == 0 ) // only check odd numbers  
12            lower ++;  
13        while ( lower < upper ) {  
14            if( Factor.NUMBER % lower == 0 ) {  
15                System.out.println (" Security code : " +  
16                                ( lower + Factor.NUMBER / lower ));  
17                return ;  
18            }  
19            lower += 2;  
20        }  
21    }  
22 }
```

The constructor for FactorThread takes an upper and lower bound, similar to MatrixThread. Once a FactorThread object has those bounds, it can search between them. The number to factor is stored in the Factor class. If any value divides that number evenly, it must be one of the factors, making the other factor easy to find, sum, and print out. We have to add a couple of extra lines of code to make sure that we only search the odd numbers in the range. This solution is tuned for efficiency for this specific security problem. A program to find general prime factors would have to be more flexible. Next let us examine the driver program Factor.

Program 13.2: Driver class which creates threads to lower the average search time for the factors of a large odd composite. (Factor.java)

```
1 public class Factor {  
2     public static final int THREADS = 4;  
3     public static final long NUMBER = 59984005171248659 L;  
4  
5     public static void main ( String [] args ) {  
6         FactorThread [] threads = new FactorThread [ THREADS ];  
7         long root = ( long ) Math.sqrt ( NUMBER ); // go to square root  
8         long start = 3; // no need to test 2  
9         long quotient = root / THREADS ;
```

```

10    long remainder = root % THREADS ;
11
12    for ( int i = 0; i < THREADS ; i++ ) {
13        long work = quotient ;
14        if( i < remainder )
15            work++;
16        threads [i] = new FactorThread ( start , start + work );
17        threads [i].start ();
18        start += work ;
19    }
20    try {
21        for ( int i = 0; i < THREADS ; i++ )
22            threads [i].join ();
23    }
24    catch ( InterruptedException e ) {
25        e.printStackTrace ();
26    }
27}
28}

```

Static constants hold both the number to be factored and the number of threads. In the main() method, we create an array of threads for storage. Then, we create each FactorThread object, assigning upper and lower bounds at the same time, using the standard technique from [Section 6.11](#) to divide the work fairly. Because we know the number we're dividing isn't even, we start with 3. By only going up to the square root of the number, we know that we will only find the smaller of the two factors. In that way we can avoid having one thread find the smaller while another is finds the larger.

Afterwards, we have the usual join() calls to make sure that all the threads are done. In this problem, these calls are unnecessary. One thread will print out the correct security code, and the others will search fruitlessly. If the program went on to do other work, we might need to let the other threads finish or even interrupt them. Don't forget join() calls since they are usually very important.

13.9 Summary

In this chapter we have examined tasks and domains. We have explained two strategies to obtain a concurrent solution to a programming problem. One strategy, task decomposition, splits a task into two or more subtasks. These subtasks can then be packaged as Java threads and executed on different cores of a multicore processor. Another strategy, domain decomposition, partitions input data into smaller chunks and allows different threads to work concurrently on each chunk of data.

A concurrent solution to a programming problem can execute more quickly than a sequential solution. Speedup measure how effective a concurrent solution is at exploiting the architecture of a multicore processor. Note that not all concurrent programs lead to speedup as some run slower than their sequential counterparts. Writing a concurrent program is a challenge that forces us to discover

solutions that best exploit a given processor and OS.

Java provides a rich set of primitives and syntactic elements to write concurrent programs. Only a few of these have been introduced in this chapter. Subsequent chapters give additional tools to code more complex concurrent programs.

Exercises

Conceptual Problems

- 13.1 The `start()`, `run()`, and `join()` methods are essential parts of the process of using threads in Java. Explain the purpose of each method.
- 13.2 What is the difference between extending the `Thread` class and implementing the `Runnable` interface? When should you use one over the other?
- 13.3 How do the `Thread.sleep()` method and the `Thread.yield()` method each affect thread scheduling?
- 13.4 Consider the expression in [Example 13.2](#). Suppose that the multiply and exponentiation operations require 1 and 10 time units, respectively. Compute the number of time units required to evaluate the expression as in [Figure 13.2\(a\)](#) and [\(b\)](#).
- 13.5 Suppose that a computer has one quadcore processor. Can the tasks in [Examples 13.1](#) and [13.2](#) be further subdivided to improve performance on four cores? Why or why not?
- 13.6 Consider the definition of speedup from [Section 13.5](#). Let's assume you have a job 1,000,000 units in size. A thread can process 10,000 units of work every second. It takes an additional 100 units of work to create a new thread. What is the speedup if you have a dual-core processor and create 2 threads? What if you have a quadcore processor and create 4 threads? Or an 8-core processor and create 8 threads? You may assume that a thread does not need to communicate after it has been created.
- 13.7 In which situations can speedup be smaller than the number of processors? Is it ever possible for speedup to be greater than the number of processors?
- 13.8 Amdahl's Law is a mathematical description of the maximum amount you can improve a system by only improving a part of it. One form of it states that the maximum speedup attainable in a parallel program is $\frac{1}{1-P}$ where P is the fraction of the program which can be parallelized to an arbitrary degree. If 30% of the work in a program can be fully parallelized but the rest is completely serial, what is the speedup with 2 processors? 4? 8? What implications does Amdahl's Law have?
- 13.9 Consider the following table of tasks:

Task	Time	Concurrency	Dependency
Washing Dishes	30	3	-
Cooking Dinner	45	3	Washing Dishes
Cleaning Bedroom	10	2	-
Cleaning Bathroom	30	2	-
Doing Homework	30	1	Cleaning Bedroom

In this table, the **Time** column gives the number of minutes a task takes to perform with a single person, the **Concurrency** column gives the maximum number of people who can be assigned to a task, and the **Dependency** column shows which tasks cannot start until other tasks have been finished. Assume that people assigned to a given task can perfectly divide the work. In other words, the time a task takes is the single person time divided by the number of people assigned. What is the minimum amount of time needed to perform all tasks with only a single person? What is the minimum amount of time needed to perform all tasks with an unlimited number of people? What is the smallest number of people needed to achieve this minimum time?

13.10 Consider the following code snippet.

```
x = 13;
x = x * 10;
```

Consider this snippet as well.

```
x = 7;
x = x + x;
```

If we assume that these two snippets of code are running on separate threads but that `x` is a shared variable, what are the possible values `x` could have after both snippets have run? Remember that the execution of these snippets can be interleaved in any way.

Programming Practice

13.11 Re-implement the array summing problem from [Example 13.10](#) using polling instead of `join()` calls. Your program should not use a single call to `join()`. Polling is not an ideal way to solve this problem, but it is worth experimenting with the technique.

13.12 Composers often work with multiple tracks of music. One track might contain solo vocals, another drums, a third one violins, and so on. After recording the entire take, a mix engineer might want to apply special effects such as an echo to one or more tracks.

To understand how to add echo to a track, suppose that the track consists of a list of audio samples. Each sample in a mono (not stereo) track can be stored as a `double` in an array. To create an echo effect, we combine the current value of an audio sample with a sample from a fixed time earlier. This time is called the *delay* parameter. Varying the delay can produce long

and short echoes.

If the samples are stored in array in and the delay parameter is stored in variable delay, the following code snippet can be used to create array out which contains the sound with an echo.

```
double [] out = new double [in.length + delay];  
// sound before echo starts  
for ( int i = 0; i < delay ; i++ )  
    out [i] = in[i];  
// sound with echo  
for ( int i = delay ; i < in.length ; i++ )  
    out [i] = a*in[i] + b*in[i - delay ];  
// echo after sound is over  
for ( int i = in.length ; i < out.length ; i++ )  
    out [i] = b*in[i - delay ];
```

Parameters a and b are used to control the nature of the echo. When a is 1 and b is 0, there is no echo. When a is 0 and b is 1, there is no mixing. Audio engineers will control the values of a and b to create the desired echo effect.

Write a threaded program that computes the values in out in parallel for an arbitrary number of threads.

13.13 Write a program which takes a number of minutes and seconds as input. In this program, implement a timer using Thread.sleep() calls. Each second, print the remaining time to the screen. How accurate is your timer?

13.14 As you know, $\pi \approx 3.1416$. A more precise value can be found by writing a program which approximates the area of a circle. The area of a circle can be approximated by summing up the area of rectangles filling curve of the arc of the circle. As the width of the rectangle goes to zero, the approximation becomes closer and closer to the true area. Recall that that height y of a circle centered at the origin at any distance x is given by $y = \sqrt{r^2 - x^2}$ where r is the radius of the circle.

Write a parallel implementation of this problem which divides up portions of the arc of the circle among several threads and then sums the results after they all finish. By setting $r = 2$, you need only sum one quadrant of a circle to get π . You will need to use a very small rectangle width to get an accurate answer. When your program finishes running, you can compare your value against Math.PI for accuracy.

Experiments

13.15 Use the currentTimeMillis() method to measure the time taken to execute a relatively long-running piece of Java code you have written. Execute your program several times and compare the execution time you obtain during different executions. Why do you think the execution times are different?

13.16 Thread creation overhead is an important consideration in writing efficient parallel

programs. Write a program which creates a large number of threads which do nothing. Test how long it takes to create and join various numbers of threads. See if you can determine how long a single thread creation operation takes on your system, on average.

13.17 Create serial and concurrent implementations of matrix multiplication like those described in [Example 13.11](#).

- (a) Experiment with different matrix sizes and thread counts to see how the speedup performance changes. If possible, run your tests on machines with different numbers of cores or processors.
- (b) Given a machine with $k > 1$ cores, what is the maximum speedup you can expect to obtain?

13.18 Repeatedly run the code in [Example 13.7](#) which creates several NumberedThread objects. Can you discover any patterns in the order that the threads print? Add a loop and some additional instrumentation to the NumberedThread class which will allow you to measure how long each thread runs before the next thread has a turn.

13.19 Create serial and parallel implementations of the array summing problem solved in [Example 13.10](#). Experiment with different array sizes and thread counts to see how performance changes. How does the speedup differ from matrix multiply? What happens if you simply sum the numbers instead of taking the sine first?

13.20 The solution to the array summing problem in [Example 13.10](#) seems to use concurrency half-heartedly. After all the threads have computed their sums, the main thread sums up the partial sums sequentially.

An alternative approach is to sum up the partial sums concurrently. Once a thread has computed the sum of the sines of each partition, the sums of each pair of neighboring partitions should be merged into a single sum. The process can be repeated until the final sum has been computed. At each step, half of the remaining threads will have nothing left to do and will stop. The pattern of summing is like a tree which starts with k threads working at the first stage, $\frac{k}{2}$ working at the second stage, $\frac{k}{4}$ working at the third, and so on, until a single thread completes the summing process.

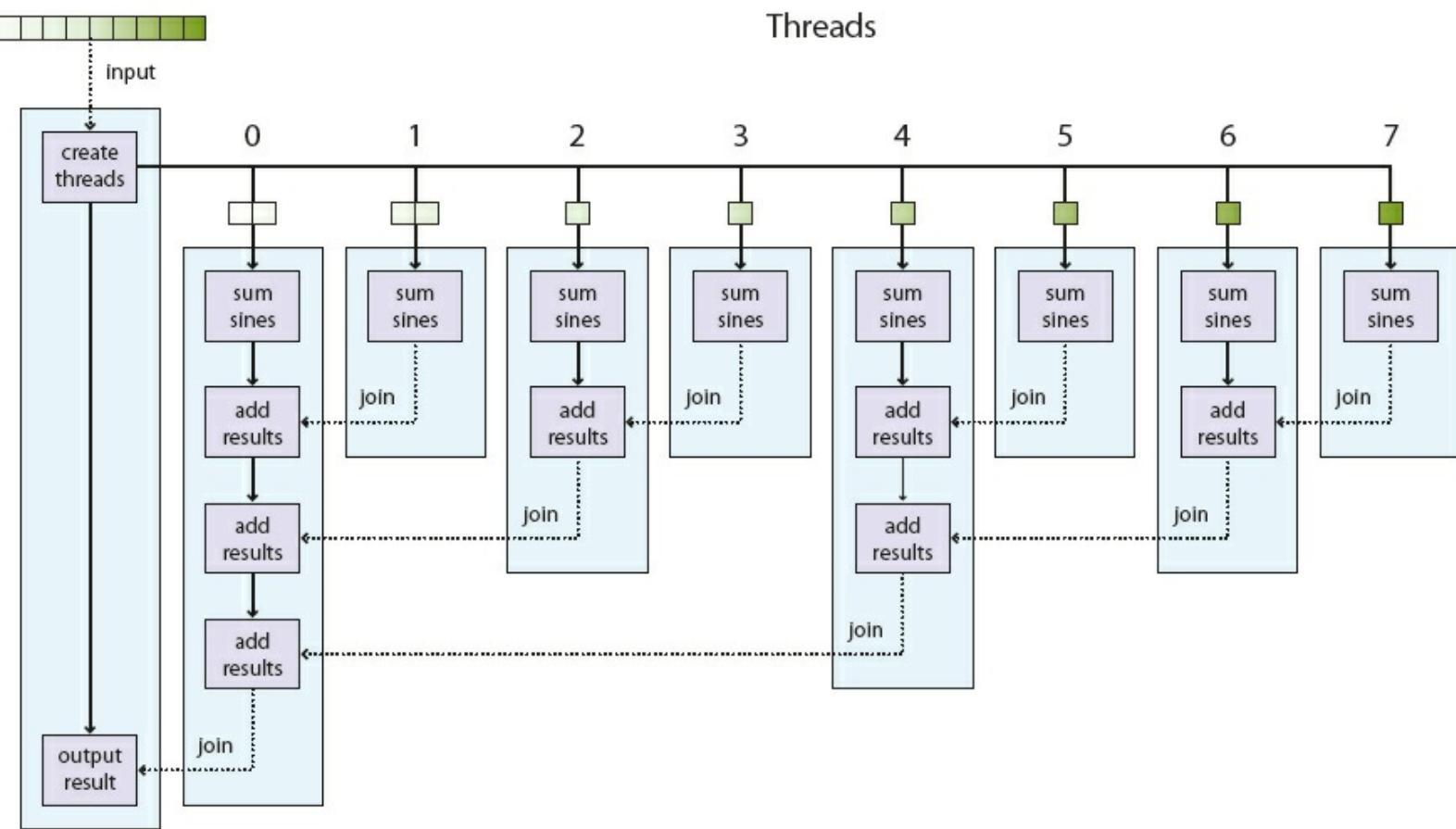


Figure 13.8: Example of concurrent tree-style summation with 8 threads.

Update the run() method in the SumThread class so that it adds its assigned elements as before and then adds its neighbors sum to its own. To do so, it must use the join() method to wait for the neighboring thread. It should perform this process repeatedly. After summing their own values, each even numbered thread should add in the partial sum from its neighbor. At the next step, each thread with a number divisible by 4 should add the partial sum from its neighbor. At the next step, each thread with a number divisible by 8 should add the partial sum from its neighbor, and so on. Thread 0 will perform the final summation. Consequently, the main thread only needs to wait for thread 0. So that each thread can wait for other threads, the threads array will need to be a static field. [Figure 13.8](#) illustrates this process.

Once you have implemented this design, test it against the original SumThread class to see how it performs. Restrict the number of threads you create to a power of 2 to make it easier to determine which threads wait and which threads terminate.

Chapter 14

Synchronization

Sharing is sometimes more demanding than giving.

—Mary Catherine Bateson

14.1 Introduction

Concurrent programs allow multiple threads to be scheduled and executed, but the programmer does not have a great deal of control over when threads execute. As explained in [Section 13.6](#), the JVM and the underlying OS are responsible for scheduling threads onto processor cores.

While writing a concurrent program, you have to ensure that the program will work correctly even though different executions of the same program will likely lead to different sequences of thread execution. The problem we introduce next illustrates why one thread execution sequence might be perfectly fine while another might lead to unexpected and incorrect behavior (and even people starving!)

14.2 Problem: Dining philosophers

Concurrency gives us the potential to make our programs faster but introduces a number of other problems. The way that threads interact can be unpredictable. Because they share memory, one thread can corrupt the value in another thread's variables. We introduce synchronization tools in this chapter that can prevent threads from corrupting data, but these tools create new pitfalls. To explore these pitfalls, we give you another problem to solve.

Imagine a number of philosophers sitting at a round table with plates that are periodically filled with rice. Between adjacent philosophers are single chopsticks so that there are exactly the same number of chopsticks as there are philosophers. These philosophers only think and eat. In order to eat, a philosopher must pick up both the chopstick on her left and the chopstick on her right. [Figure 14.1](#) shows an example of the situation.

Your goal is to write a class called `DiningPhilosopher` which extends `Thread`. Each thread created in the `main()` method should be a philosopher who thinks for some random amount of time, then acquires the two necessary chopsticks and eats. No philosopher should starve. No philosophers should be stuck indefinitely fighting over chopsticks.



Figure 14.1: Table for five dining philosophers.

Although this problem sounds simple, the solution is somewhat complex. Make sure that you understand the concepts and Java syntax in this chapter thoroughly before trying to implement your solution. It is important that no two philosophers try to use the same chopstick at the same time. Likewise, we need to avoid a situation where every philosopher is waiting for every other philosopher to give up a chopstick.

14.3 Concepts: Thread interaction

This dining philosopher problem highlights some difficulties which were emerging toward the end of the last chapter. In Exercise 13.10 two snippets of code could run concurrently and modify the same shared variable, potentially producing incorrect output. Because of the nondeterministic nature of scheduling, we have to assume that the code executing in two or more threads can be interleaved in any possible way. When the **result** of computation changes depending on the order of thread execution, it is called a *race condition*. Below is a simple example of a race condition in Java.

Exercise 14.6

Exercise 14.14

Program 14.1: A short example of a race condition. (RaceCondition.java)

```
1 public class RaceCondition extends Thread {  
2     private static int counter = 0;
```

```

3   public static final int THREADS = 4;
4   public static final int COUNT = 1000000;
5
6   public static void main ( String [] args ) {
7       RaceCondition [] threads = new RaceCondition [ THREADS ];
8       for ( int i = 0; i < THREADS ; i++ ) {
9           threads [i] = new RaceCondition ();
10          threads [i].start ();
11      }
12      try {
13          for ( int i = 0; i < THREADS ; i++ )
14              threads [i].join ();
15      }
16      catch ( InterruptedException e ) {
17          e.printStackTrace ();
18      }
19      System.out.println ("Counter \n" + counter );
20  }
21
22  public void run () {
23      for ( int i = 0; i < COUNT / THREADS ; i++ )
24          counter++;
25  }
26 }
```

This short (and pointless) class attempts to increment the variable counter until it reaches 1000000. To illustrate the race condition, we have divided the work of incrementing counter evenly among a number of threads. If you run this program, the final value of counter will often not be 1000000. Depending on which JVM, OS, and how many cores you have, you may never get 1000000, and the answer you do get will vary a lot. On all systems, if you change the value of THREADS to 1, the answer should always be correct.

Looking at the code, the problem might not be obvious. Everything centers on the statement counter++ in the `for` loop inside the `run()` method. But, this statement appears to execute in a single step! Each thread should increase the value of counter a total of COUNT/THREADS times, adding $\frac{1}{4}$ to 1000000. The trouble is that counter++ is not a single step. Recall that counter++ is shorthand for `counter = counter + 1`. To be even more explicit we could write it as follows.

```
temp = counter ;
counter = temp + 1;
```

One thread may get as far as storing counter into a temporary location, but then it runs out of time and the next thread in the schedule runs. If that is the case, this next thread may do a series of increments to count which are all overwritten when the first thread runs again. Because the first thread had an old value of counter stored in `temp`, adding 1 to `temp` has the effect of ignoring many

increments. This situation can happen on a single processor with threads switching back and forth, but it is even more dangerous on a multicore system.

The primary lesson here is that threads can switch between each other at any time with unpredictable effects. The secondary lesson is that the source code is too coarse-grained to show *atomic* operations. An atomic operation is one which cannot be interrupted by a context switch to another thread. The actual code that the JVM runs is much lower level than source code.

We can't easily force a non-atomic operation to be atomic, but there are ways to restrict access to certain pieces of code under certain conditions. The name we give to a piece of code which should not be accessed by more than one thread at a time is a *critical section*. In the example above, the single line of code which increments counter is a critical section, and the error in the program would be removed if only one thread were able to run that line of code at a time.

Protecting a critical section is done with *mutual exclusion* tools. They are called mutual exclusion tools because they enforce the requirement that one thread executing a critical section excludes the possibility of another. There are many different techniques, algorithms, and language features in computer science which can be used to create mutual exclusion. Java relies heavily on a tool called a *monitor* which hides some of the details of enforcing mutual exclusion from the user. Mutual exclusion is a deeply researched topic with many approaches other than monitors. If you plan to write concurrent programs in another language, you may need to brush up on the features provided by that language.

Exercise 14.5

14.4 Syntax: Thread synchronization

14.4.1 The synchronized keyword

In Java, the language feature which allows you to enforce mutual exclusion is the **synchronized** keyword. There are two ways to use this keyword: with a method or with an arbitrary block of code. In the method version, you add the **synchronized** modifier before the return type. Let's imagine a class with a private String field called message which is set to be “Will Robinson !” by the constructor. Now, we define the following method.

Exercise 14.1

```
public synchronized void danger () {  
    message = " Danger , " + message ;  
}
```

If danger() is called five times from different threads, message will contain “Danger, Danger, Danger, Danger, Danger, Will Robinson!” Without the **synchronized** keyword, danger() would suffer from a race condition similar to the one in RaceCondition. Some of the String concatenations might be overwritten by other calls to danger(). You would never have more than give copies of “Danger,” appended to the beginning of message, but you might have fewer.

Any time a thread enters a piece of code protected by the **synchronized** keyword, it implicitly acquires a lock which only a single thread can hold. If another thread tries to access the code, it is

forced to wait until the lock is released. This lock is *re-entrant*. Re-entrant means that, when a thread currently holds a lock and tries to get it again, it succeeds. This situation frequently occurs with synchronized methods which call other synchronized methods.

Exercise 14.13

Consider method safety() which does the “opposite” of danger(), by removing occurrences of “Danger,” from the beginning of message.

```
public synchronized void safety () {  
    if( message.startsWith (" Danger , ") )  
        message = message.substring ( 8 );  
}
```

Will the danger() and safety() methods play nicely together? In other words, will a thread be blocked from entering safety() if another thread is already in danger()? Yes! The locks in Java are connected to objects. When you use the **synchronized** keyword on a method, the object the method is being called on (whichever object **this** refers to inside the method) serves as the lock. Thus, only one thread can be inside of either of these methods. If you have 10 synchronized methods in an object, only one of them can execute at a time in a given object.

Exercise 14.2

Exercise 14.7

Perhaps this level of control is too restrictive. You may have six methods which conflict with each other and four others which conflict with each other but not the first six. Using **synchronized** in each method declaration would unnecessarily limit the amount of concurrency your program could have.

Although it takes a little more work, using **synchronized** with a block of code allows more fine-grained control. The following version of danger() is equivalent to the earlier one.

```
public void danger () {  
    synchronized ( this ) {  
        message = " Danger , " + message ;  
    }  
}
```

Using **synchronized** on a block of code gives us more flexibility in two ways. First, we can choose exactly how much code we want to control, instead of the whole method. Second, we can choose which object we want to use for synchronization. For the block style, any arbitrary object can be used as a lock. Objects keep a list of threads which are waiting to get the lock and do all the other management needed to make the **synchronized** keyword work.

If you have two critical sections which are unrelated to each other, you can use the fine-grained control the block style provides. First, you’ll need some objects to use as locks, probably declared so that they can easily be shared, perhaps as static fields of a class.

```
private static Object lock1 = new Object ();  
private static Object lock2 = new Object ();
```

Then, wherever you need control over concurrency, you use them as locks.

```
synchronized ( lock1 ){  
    // do dangerous thing 1  
}  
  
// do safe things  
synchronized ( lock2 ){  
    // do dangerous thing 2, unrelated to dangerous thing 1  
}
```

Since declaring a method with **synchronized** is equivalent to having its body enclosed in a block beginning with **synchronized(this)**, what about **static** methods? Can they be **synchronized**? Yes, they can. Whenever a class is loaded, Java creates an object of type **Class** which corresponds to that class. This object is what synchronized static methods inside the class will use as a lock. For example, a synchronized static method inside of the **Eggplant** class will lock on the object **Eggplant.class**.

14.4.2 The **wait()** and **notify()** methods

Protecting critical sections with the **synchronized** keyword is a powerful technique, and many other synchronization tools can be built using just this tool. However, efficiency demands a few more options.

Sometimes a thread is waiting for another thread to finish a task so that it can process the results. Imagine one thread collecting votes while another one is waiting to count them. In this example, the counting thread must wait for all votes to be cast before it can begin counting. We could use a synchronized block and an indicator **boolean** called **votingComplete** to allow the collector thread to signal to the counting thread.

```
while ( true ) {  
    synchronized ( this ) {  
        if( votingComplete )  
            break ;  
    }  
}  
countVotes ();
```

What's the problem with this design? Notice that the counting thread is running through the **while** loop over and over waiting for **votingComplete** to become **true**. On a single processor, the counting thread would slow down the job of the collecting thread which is trying to process all the votes. On a multicore system, the counting thread is still wasting CPU cycles that some other thread could use. This phenomenon is known as *busy waiting*, for obvious reasons.

To combat this problem, Java provides the **wait()** method. When a thread is executing synchronized code, it can call **wait()**. Instead of busy waiting, a thread which has called **wait()** will be removed from the list of running threads. It will wait in a dormant state until someone comes along and notifies the thread that its waiting is done. If you recall the threadstate diagram from [Chapter 13](#), there is a Not

Runnable state which threads enter by calling sleep(), calling wait(), or performing blocking I/O. Using wait(), we can rewrite the vote counting thread.

```
synchronized ( this ) {  
    while ( ! votingComplete ) {  
        wait ();  
    }  
}  
countVotes ();
```

Notice that the **while** loop has moved inside the synchronized block. Doing so before might have kept our program from terminating: As long as the vote counting thread held the lock, the vote collecting thread would not be allowed to modify votingComplete. When a thread calls wait(), however, it gives up the corresponding lock it is holding until it wakes up and runs again. Why use the **while** loop at all, now? There is no guarantee that the condition you are waiting for is **true**. Many threads may be waiting on this particular lock. We use the **while** loop to check that votingComplete is **true** and wait again if it isn't.

In order to notify a waiting thread, the other thread calls the notify() method. Like wait(), notify() must be called within a synchronized block or method. Here is corresponding code the vote collecting thread would use to notify the counting thread that voting is complete.

```
// finish collecting votes  
synchronized ( this ) {  
    votingComplete = true ;  
    notifyAll ();  
}
```

A call to notify() will wake up one thread waiting on the lock object. If there are many threads waiting, the method notifyAll() used above can be called to wake them all up. In practice, it is usually safer to call notifyAll(). If a particular condition changes and a single waiting thread is notified, that thread may need to notify the next waiting thread when it is done. If your code is not very carefully designed, some thread may end up waiting forever and never be notified if you only rely on notify().

Exercise 14.3

Exercise 14.4

Example 14.1: Producer/consumer

To illustrate the use of wait() and notify() calls inside of synchronized code, we give a simple solution to the producer/consumer problem below. This problem is a classic example in the concurrent programming world. Many times one thread (or a group of threads) is producing data, perhaps from some input operation. At the same time, one thread (or, again, potentially many threads) are taking these chunks of data and consuming them by performing some computational or output task.

Every resource inside of a computer is finite. Producer/consumer problems often assume a bounded buffer which stores items from the producer until the consumer can take them away. Our

solution does all synchronization on the buffer. Many different threads can share this buffer, but all accesses will be controlled.

Program 14.2: An example of a synchronized buffer. (Buffer.java)

```
1 public class Buffer {  
2     public final static int SIZE = 10;  
3     private Object [] objects = new Object [ SIZE ];  
4     private int count = 0;  
5  
6     public synchronized void addItem ( Object object )  
7         throws InterruptedException {  
8         while ( count == SIZE )  
9             wait ();  
10        objects [ count ] = object ;  
11        count ++;  
12        notifyAll ();  
13    }  
14  
15    public synchronized Object removeItem ()  
16        throws InterruptedException {  
17        while ( count == 0 )  
18            wait ();  
19        count --;  
20        Object object = objects [ count ];  
21        notifyAll ();  
22        return object ;  
23    }  
24 }
```

When adding an item, producers enter the synchronized addItem() method. If count shows that the buffer is full, the producer must wait until the buffer has at least one open space. After adding an item to the buffer, the producer then notifies all waiting threads. The consumer does the mirror image in removeItem(). A consumer thread cannot consume anything if the buffer is empty and must then wait. After there is an object to consume, the consumer removes it and notifies all other threads.

Both methods are synchronized, making access to the buffer completely sequential. Although it seems undesirable, sequential behavior is precisely what is needed for the producer/consumer problem. All synchronized code is a protection against unsafe concurrency. The goal is to minimize the amount of time spent in synchronized code and get threads back to concurrent execution as quickly as possible. ■

Exercise 14.10

Example 14.2: Bank account

Although producer/consumer is a good model to keep in mind, there are other ways that reading and writing threads might interact. Consider the following programming problem, similar to one you might find in real life.

As a rising star in a bank's IT department, you have been given the job of creating a new bank account class called `SynchronizedAccount`. This class must have methods to support the following operations: deposit, withdraw, and check balance. Each method should print a status message to the screen on completion. Also, the method for withdraw should return `false` and do nothing if there are insufficient funds. Because the latest system is multi-threaded, these methods must be designed so that the bookkeeping is consistent even if many threads are accessing a single account. No money should magically appear or disappear.

There is an additional challenge. To maximize concurrency, `SynchronizedAccount` should be synchronized differently for read and write accesses. Any number of threads should simultaneously be able to check the balance on an account, but only one thread can deposit or withdraw at a time.

To solve this problem, our implementation of the class has a `balance` variable to record the balance, but it also has a `readers` variable to keep track of the number of threads which are reading from the account at any given time.

```
1 public class SynchronizedAccount {  
2     private double balance = 0.0;  
3     private int readers = 0;
```

Next, the `getBalance()` method is called by threads which wish to read the balance. Access to the `readers` variable is synchronized. But, after passing that first `synchronized` block, the code which stores the balance is not synchronized. In this way, multiple readers can access the data at the same time. For this example, the concurrency controls we have are overkill. The command `amount = balance` does not take a great deal of time. If it did, however, it would make sense for readers to execute it concurrently as we do. After reading the balance, this method decrements `readers`. If `readers` reaches 0, a call to `notifyAll()` is made, signaling that threads trying to deposit to or withdraw from the account can continue.

```
5     public double getBalance () throws InterruptedException {  
6         double amount ;  
7         synchronized ( this ) {  
8             readers ++;  
9         }  
10        amount = balance ;  
11        synchronized ( this ) {  
12            if( -- readers == 0 )  
13                notifyAll ();  
14        }  
15        return amount ;  
16    }
```

The `deposit()` and `withdraw()` methods are wrappers for the `changeBalance()` method, which has

all the interesting concurrency controls.

```
18     public void deposit ( double amount )
19         throws InterruptedException {
20             changeBalance ( amount );
21             System.out.println (" Deposited $" + amount + ".");
22     }
23
24     public boolean withdraw ( double amount )
25         throws InterruptedException {
26             boolean success = changeBalance ( -amount );
27             if( success )
28                 System.out.println (" Withdrew $" + amount + ".");
29             else
30                 System.out.println (" Failed to withdraw $" +
31                         amount + ": insufficient funds.");
32             return success ;
33     }
```

The `changeBalance()` method is synchronized so that it can have exclusive access to the `readers` variable. As long as `readers` is greater than 0, this method will wait. Eventually, the readers should finish their job and notify the waiting writer which can finish changing the balance of the account.

```
35     private synchronized boolean changeBalance ( double amount )
36         throws InterruptedException {
37             boolean success ;
38             while ( readers > 0 )
39                 wait ();
40             if( success = ( balance + amount > 0 ) )
41                 balance += amount ;
42             return success ;
43     }
44 }
```

14.5 Pitfalls: Synchronization challenges

As you can see from the dining philosophers problem, synchronization tools help us get the right answer but also create other difficulties.

14.5.1 Deadlock

Deadlock is the situation when two or more threads are both waiting for the others to complete, forever. Some combination of locks or other synchronization tools has forced a blocking dependence onto a group of threads which will never be resolved.

In the past, people have described four conditions which must exist for deadlock to happen.

1. Mutual Exclusion: Only one thread can access the resource (often a lock) at a time.
2. Hold and Wait: A thread holding a resource can ask for additional resources.
3. No Preemption: A thread holding a resource cannot be forced to release it by another thread.
4. Circular Wait: Two or more threads hold resources which make up a circular chain of dependency.

Example 14.3: Deadlock philosophers

We illustrate deadlock with an example of how **not** to solve the dining philosophers problem. What if all the philosophers decided to pick up the chopstick on her right and then the chopstick on her left? If the timing was just right, each philosopher would be holding one chopstick in her right hand and be waiting forever for her neighbor on the left to give up a chopstick. No philosopher would ever be able to eat. Here is that scenario illustrated in code.

```
1 public class DeadlockPhilosopher extends Thread {  
2     public static final int SEATS = 5;  
3     private static boolean [] chopsticks = new boolean [ SEATS ];  
4     private int seat ;  
5  
6     public DeadlockPhilosopher ( int seat ) {  
7         this.seat = seat ;  
8     }  
9 }
```

After setting up the class and the constructor, things get interesting in the `run()` method. First a philosopher tries to get her left chopstick, then sleeps for 50 milliseconds, then tries to get her right chopstick. Without sleeping, this code would usually run just fine. Every once in a while, the philosophers would become deadlocked, but it would be hard to predict when. By introducing the sleep, we can all but guarantee that the philosophers will deadlock every time.

```
10    public void run () {  
11        try {  
12            getChopstick ( seat );  
13            Thread.sleep ( 50 );  
14            getChopstick ( seat - 1 );  
15        }  
16        catch ( InterruptedException e ) {  
17            e.printStackTrace ();  
18        }  
19        eat ();  
20    }
```

The remaining two methods are worth examining to see how the synchronization is done, but by getting the two chopsticks separately above, we have already gotten ourselves into trouble.

```
22     private void getChopstick ( int location )
23         throws InterruptedException {
24             if( location < 0 )
25                 location += SEATS ;
26             synchronized ( chopsticks ) {
27                 while ( chopsticks [ location ] )
28                     chopsticks.wait ();
29                 chopsticks [ location ] = true ;
30             }
31             System.out.println (" Philosopher " + seat +
32                 " picked up chopstick " + location + ".");
33         }
34
35     private void eat () {
36         // done eating , put back chopsticks
37         synchronized ( chopsticks ) {
38             chopsticks [ seat ] = false ;
39             if( seat == 0 )
40                 chopsticks [ SEATS - 1] = false ;
41             else
42                 chopsticks [ seat - 1] = false ;
43             chopsticks.notifyAll ();
44         }
45     }
46 }
```



Example 14.4: Deadlock sum

Here is another example of deadlock. We emphasize deadlock because it is one of the most common and problematic issues with using synchronization carelessly.

Consider two threads which both need access to two separate resources. In our example, the two resources are random number generators. The goal of each of these threads is to acquire locks for the two shared random number generators, generate two random numbers each, and sum the numbers generated. (Note that locks are unnecessary for this problem anyway, because access to Random objects is synchronized.)

```
1 import java.util.Random ;
2
3 public class Deadlock extends Thread {
4     private static Random random1 = new Random ();
```

```
5     private static Random random2 = new Random ();
6     private boolean reverse ;
7     private int sum;
```

The class begins as expected, creating shared **static** Random objects random1 and random2. Then, in the main() method, the main thread spawns two new threads, passing true to one and false to the other.

```
9     public static void main ( String [] args ) {
10         Thread thread1 = new Deadlock ( true );
11         Thread thread2 = new Deadlock ( false );
12         thread1.start ();
13         thread2.start ();
14         try {
15             thread1.join ();
16             thread2.join ();
17         }
18         catch ( InterruptedException e ) {
19             e.printStackTrace ();
20         }
21     }
```

Next, the mischief begins to unfold. One of the two threads stores true in its reverse field.

```
23     public Deadlock ( boolean reverse ) {
24         this.reverse = reverse ;
25     }
```

Finally, we have the run() method where all the action happens. If the two running threads both acquire locks for random1 and random2 in the same order, everything would work out fine. However, the reversed thread locks on random2 and then random1, with a sleep() in between. The non-reversed thread tries to lock on random1 and then random2.

```
27     public void run () {
28         if( reverse ) {
29             synchronized ( random2 ) {
30                 System.out.println (
31                     " Reversed Thread : synchronized on random2 ");
32                 try { Thread.sleep (50) ; }
33                 catch ( InterruptedException e ) {
34                     e.printStackTrace ();
35                 }
36                 synchronized ( random1 ) {
37                     System.out.println (
38                         " Reversed Thread : synchronized on random1 ");
39                     sum = random1.nextInt () + random2.nextInt ();
```

```

40         }
41     }
42 }
43 else {
44     synchronized ( random1 ) {
45         System.out.println (
46             " Normal Thread : synchronized on random1 ");
47         try { Thread.sleep ( 50 ) ; }
48         catch ( InterruptedException e ) {
49             e.printStackTrace ();
50         }
51     synchronized ( random2 ) {
52         System.out.println (
53             " Normal Thread : synchronized on random2 ");
54         sum = random1.nextInt () + random2.nextInt ();
55     }
56 }
57 }
58 }
59 }

```

If you run this code, it should invariably deadlock with thread1 locked on random2 and thread2 locked on random1. No sane programmer would intentionally code the threads like this. In fact, the extra work we did to acquire the locks in opposite orders is exactly what causes the deadlock. For more complicated programs, there may be many different kinds of threads and many different resources. If two different threads (perhaps written by different programmers, even) need both resource A and resource B at the same time but try to acquire them in reverse order, this kind of deadlock can occur without such an obvious cause.

For deadlock of this type, the circular wait condition can be broken by ordering the resources and always locking the resources in ascending order. Of course, this solution only works if there is some universal way of ordering the resources and the ordering is always followed by all code in the program.

Ignoring the deadlock problems with the example above, it gives a nice example of the way Java intended synchronization to be done: when possible, use the resource you need as its own lock. Many other languages require programmers to create additional locks or semaphores to protect a given resource, but this approach causes problems if the same lock is not consistently used. Using the resource itself as a lock elegantly avoids this problem. ■

Exercise 14.8

14.5.2 Starvation and livelock

Starvation is another problem which can occur with careless use of synchronization tools. Starvation is a general term which covers any situation in which some thread never gets access to the resources

it needs. Deadlock can be viewed as a special case of starvation since none of the threads which are deadlocking make progress.

The dining philosophers problem was framed around the idea of eating with humorous intent. If a philosopher is never able to acquire chopsticks, that philosopher will quite literally starve.

Starvation does not necessarily mean deadlock, however. Examine the implementation in [Example 14.2](#) for the bank account. That solution is correct in the sense that it preserves mutual exclusion. No combination of balance checks, deposits, or withdrawals will cause the balance to be incorrect. Money will neither be created nor destroyed. A closer inspection reveals that the solution is not entirely fair. If a single thread is checking the balance, no other thread can make a deposit or a withdrawal. Balance checking threads could be coming and going constantly, incrementing and decrementing the readers variable, but if readers never goes down to zero, threads waiting to make deposits and withdrawals will wait forever.

Exercise 14.11

Another kind of starvation is *livelock*. In deadlock, two or more threads get stuck and wait forever, doing nothing. Livelock is similar except that the two threads keep executing code and waiting for some condition that never arrives. A classic example of livelock is two polite (but oddly predictable) people who are speaking with each other: Both happen to start talking at exactly the same moment and then stop to hear what the other has to say. After exactly one second, they both begin again and immediately stop. Lather, rinse, repeat.

Example 14.5: Livelock party preparations

Imagine three friends who are going to a party. Each of them starts getting ready at different times. They follow the pattern of getting ready for a while, waiting for their friends to get ready, and then calling their friends to see if the other two are ready. If all three are ready, then the friends will leave. Unfortunately, if a friend calls and either of the other two aren't ready, he'll become frustrated and stop being ready. Perhaps he'll realize that he's got time to take a shower or get involved in some other activity for a while. After finishing that activity, he'll become ready again and wait for his friends to become ready.

If the timing is just right, the three friends will keep becoming ready, waiting for a while, and then becoming frustrated when they realize that their friends aren't ready. Here is a rough simulation of this process in code.

```
1 public class Livelock extends Thread {  
2     private static int totalReady = 0;  
3     private static Object lock = new Object();  
4  
5     public static void main ( String [] args ) {  
6         Livelock friend1 = new Livelock ();  
7         Livelock friend2 = new Livelock ();  
8         Livelock friend3 = new Livelock ();
```

The first few lines create a shared variable called totalReady which keeps track of the total number

of friends ready. To avoid race conditions, a shared Object called lock will be used to control access to totalReady. Then, the main() method creates Livelock objects representing each of the friends.

```
10     try {
11         friend1.start ();
12         Thread.sleep (100) ;
13         friend2.start ();
14         Thread.sleep (100) ;
15         friend3.start ();
16
17         friend1.join ();
18         friend2.join ();
19         friend3.join ();
20     }
21     catch ( InterruptedException e ) {
22         e.printStackTrace ();
23     }
24     System.out.println ("All ready !");
25 }
```

The rest of the main() method starts each of the threads representing the friends running, with a 100 millisecond delay before the next thread starts. Then, the main() method waits for them all to finish. If successful, it will print All ready! to the screen.

```
27 public void run () {
28     boolean done = false ;
29
30     try {
31         while ( ! done ) {
32             Thread.sleep (75) ; // prepare for party
33             synchronized ( lock ) {
34                 totalReady++;
35             }
36             Thread.sleep (75) ; // wait for friends
37             synchronized ( lock ) {
38                 if( totalReady >= 3 )
39                     done = true ;
40                 else
41                     totalReady--;
42             }
43         }
44     }
45     catch ( InterruptedException e ) {
46         e.printStackTrace ();
```

```
47      }
48  }
49 }
```

In the run() method, each friend goes through a loop until the done variable is **true**. In this loop, an initial call to Thread.sleep() for 75 milliseconds represents preparing for the party. After that, totalReady is incremented by one. Then, the friend waits for another 75 milliseconds. Finally, he checks to see if everyone else is ready by testing whether totalReady is 3. If not, he decrements totalReady and repeats the process.

At roughly 75 milliseconds into the simulation, the first friend becomes ready, but he doesn't check with his friends until 150 milliseconds. Unfortunately, the second friend doesn't become ready until 175 milliseconds. He then checks with his friends at 225 milliseconds, around which time the first friend is becoming ready a second time. However, the third friend isn't ready until 275 milliseconds. When he then checks at 350 milliseconds, the first friend isn't ready anymore. On some systems the timing might drift such that the friends all become ready at the same time, but it could take a long, long while.

In reality, human beings would not put off going to a party indefinitely. Some people would decide that it was too late to go. Others would go alone. Others would go over to their friends' houses and demand to know what was taking so long. Computers are not nearly as sensible and must obey instructions, even if they cause useless repetitive patterns. Realistic examples of livelock are hard to show in a short amount of code, but they do crop up in real systems and can be very difficult to predict. ■

14.5.3 Sequential execution

When designing a parallel program, you may notice that synchronization tools are necessary to get a correct answer. Then, when you run this parallel version and compare it to the sequential version, it runs no faster or, worse, runs slower than the sequential version. Too much zeal with synchronization tools may produce a program which gives the right answer but does not exploit any parallelism.

For example, we can take the run() method from the parallel implementation of matrix multiply given in [Example 13.11](#) and use the **synchronized** keyword to lock on the matrix itself.

```
public void run () {
    synchronized { c } {
        for ( int i = lower ; i < upper ; i++ )
            for ( int j = 0; j < c[i].length ; j++ )
                for ( int k = 0; k < b.length ; k++ )
                    c[i][j] += a[i][k] * b[k][j];
    }
}
```

In this case, only a single thread would have access to the matrix at any given time, and all speedup would be lost.

For this version of matrix multiply, no synchronization is needed. In the case of the

producer/consumer problem, synchronization is necessary, and the only way to manage the buffer properly is to enforce sequential execution. Sometimes sequential execution cannot be avoided, but you should always know which pieces of code are truly executing in parallel and which are not if you hope to get the maximum amount of speedup. The **synchronized** keyword should be used whenever it is needed, but no more.

14.5.4 Priority inversion

In [Chapter 13](#) we suggest that you use thread priorities rarely. Even good reasons to use priorities can be thwarted by *priority inversion*. In priority inversion, a lower priority thread holds a lock needed by a higher priority thread, potentially for a long time. Because the high priority thread cannot continue, the lower priority thread gets more CPU time, as if it were a high priority thread.

Worse, if there are some medium priority threads in the system, the low priority thread may hold the lock needed by the high priority thread for even longer because those medium priority threads reduce the amount of CPU time the low priority thread has to finish its task.

Exercise 14.9

Exercise 14.15

14.6 Solution: Dining philosophers

Now we give our solution to the dining philosophers problem. Although deadlock was the key pitfall we were trying to avoid, many other issues can crop up in solutions to this problem. A single philosopher may be forced into starvation, or all philosophers may experience livelock through some pattern of picking up and putting down chopsticks which never quite works out. A very simple solution could allow the philosophers to eat, one by one, in order. Then, the philosophers would often and unnecessarily be waiting to eat, and the program would approach sequential execution.

The key element that makes our solution work is that we force a philosopher to pick up two chopsticks atomically. The philosopher will either pick up both chopsticks or neither.

```
1 public class DiningPhilosopher extends Thread {  
2     public static final int SEATS = 5;  
3     private static boolean [] chopsticks = new boolean [ SEATS ];  
4     private int seat ;  
5  
6     public DiningPhilosopher ( int seat ) {  
7         this.seat = seat ;  
8     }  
9 }
```

We begin with the same setup as the deadlocking version given in [Section 14.5.1](#). Unlike that example, we include a `main()` method here for completeness. In `main()`, we set all the chopsticks to `false` (unused), create and start a thread for each philosopher, and then wait for them to finish.

```
10    public static void main ( String args [] ) {  
11        for ( int i = 0; i < SEATS ; i++ ) {
```

```

12     chopsticks [i] = false ;
13     DiningPhilosopher [] philosophers =
14         new DiningPhilosopher [ SEATS ];
15     for ( int i = 0; i < SEATS ; i++ ) {
16         philosophers [i] = new DiningPhilosopher ( i );
17         philosophers [i].start ();
18     }
19     try {
20         for ( int i = 0; i < SEATS ; i++ )
21             philosophers [i].join ();
22     }
23     catch ( InterruptedException e ) {
24         e.printStackTrace ();
25     }
26     System.out.println ("All philosophers done.");
27 }
```

This run() method is different from the deadlocking version but not in a way that prevents deadlock. We added the **for** loop so that you could see the philosophers eat and think many different times without problems. We also added the think() method to randomize the amount of time between eating so that each run of the program is less deterministic.

```

29     public void run () {
30         for ( int i = 0; i < 100; i++ ) {
31             think ();
32             getChopsticks ();
33             eat ();
34         }
35     }
36
37     private void think () {
38         Random random = new Random ();
39         try {
40             sleep ( random.nextInt (20) + 10 );
41         }
42         catch ( InterruptedException e ) {
43             e.printStackTrace ();
44         }
45     }
```

The real place where deadlock is prevented is in the getChopsticks() method. The philosopher acquires the chopsticks lock and then picks up the two chopsticks she needs only if both are available. Otherwise, she waits.

```
47     private void getChopsticks () {
```

```

48     int location1 = seat ;
49     int location2 ;
50     if( seat == 0 )
51         location2 = SEATS - 1;
52     else
53         location2 = seat - 1;
54     synchronized ( chopsticks ) {
55         while ( chopsticks [ location1 ]
56             || chopsticks [ location2 ] ) {
57             try {
58                 chopsticks.wait ();
59             }
60             catch ( InterruptedException e ) {
61                 e.printStackTrace ();
62             }
63         }
64         chopsticks [ location1 ] = true ;
65         chopsticks [ location2 ] = true ;
66     }
67     System.out.println (" Philosopher " + seat +
68         " picked up chopsticks " + location1 + " and "
69         + location2 + ".");
70 }

```

Finally, in the eat() method, the philosopher eats the rice. We would assume that some other computation would be done here in a realistic problem **before** entering the **synchronized** block. The eating itself does not require a lock. After eating is done, the lock is acquired to give back the chopsticks (hopefully after some cleaning), and then all waiting philosophers are notified that some chopsticks may have become available.

```

72     private void eat () {
73         // eat rice first
74         synchronized ( chopsticks ) {
75             chopsticks [ seat ] = false ;
76             if( seat == 0 )
77                 chopsticks [ SEATS - 1] = false ;
78             else
79                 chopsticks [ seat - 1] = false ;
80             chopsticks.notifyAll ();
81         }
82     }
83 }

```

Our solution prevents deadlock and livelock because some philosopher will get control of two

chopsticks eventually, yet there are still issues. Note that each philosopher only eats and thinks 100 times. If, instead of philosophers sharing chopsticks, each thread were a server sharing network storage units, the program could run for an unspecified amount of time: days, weeks, even years. If starvation is happening to a particular philosopher in our program, the other philosophers will finish after 100 rounds, and the starved philosopher can catch up. If there were no limitation on the loop, a starving philosopher might never catch up.

Even if you increase the number of iterations of the loop quite a lot, you would probably not see starvation of an individual thread because we are cheating in another way. Some unlucky sequence of chopstick accesses by two neighboring philosophers could starve the philosopher between them. By making the `think()` method wait a random amount of time, such a sequence will probably be interrupted. If all philosophers thought for exactly the same amount of time each turn, an unlucky pattern could repeat over and over. It is not unreasonable to believe that the amount of thinking a philosopher (or a server) will do at any given time will vary, but this kind of behavior will vary from system to system.

Exercise 14.12

It is very difficult to come up with a perfect answer to some synchronization problems. Such problems have been studied for many years, and research continues to find better solutions.

Exercises

Conceptual Problems

- 14.1 What is the purpose of the `synchronized` keyword? How does it work?
- 14.2 The language specification for Java makes it illegal to use the `synchronized` keyword on constructors. During the creation of an object, it is possible to *leak* data to the outside world by adding a reference to the object under construction to some shared data structure. What's the danger of leaking data in this way?
- 14.3 If you call `wait()` or `notify()` on an object, it must be inside of a block synchronized on the same object. If not, the code will compile, but an `IllegalMonitorStateException` may be thrown at run time. Why is it necessary to own the lock on an object before calling `wait()` or `notify()` on it?
- 14.4 Why is it safer to call `notifyAll()` than `notify()`? If it is generally safer to call `notifyAll()`, are there any scenarios in which there are good reasons to call `notify()`?
- 14.5 Imagine a simulation of a restaurant with many waiter and chef objects. The waiters must submit orders to the kitchen staff, and the chefs must divide the work among themselves. How would you design this system? How would information and food be passed from waiter to chef and chef to waiter? How would you synchronize the process?
- 14.6 What is a race condition? Give a real life example of one.
- 14.7 Let's reexamine the code that increments a variable with several threads from [Section 14.3](#). We can rewrite the `run()` method as follows.

```
public synchronized void run () {  
    for ( int i = 0; i < COUNT / THREADS ; i++ )  
        counter ++;  
}
```

Will this change fix the race condition? Why or why not?

14.8 Examine our deadlock example from [Example 14.4](#). Explain why this example fulfills all four conditions for deadlock. Be specific about which threads and which resources are needed to show each condition.

14.9 What is priority inversion? Why can a low priority thread holding a lock be particularly problematic?

Programming Practice

14.10 In [Example 14.1](#) the Buffer class used to implement a solution to the producer/consumer problem only has a single lock. When the buffer is empty and a producer puts an item in it, both producers and consumers are woken up. A similar situation happens whenever the buffer is full and a consumer removes an item. Re-implement this solution with two locks so that a producer putting an item into an empty buffer only wakes up consumers and a consumer removing an item from a full buffer only wakes up producers.

14.11 In [Example 14.2](#) we used the class SynchronizedAccount to solve a bank account problem. As we mention in [Section 14.5.2](#), depositing and withdrawing threads can be starved out by a steady supply of balance checking threads. Add additional synchronization tools to SynchronizedAccount so that balance checking threads will take turns with depositing and withdrawing threads. If there are no depositing or withdrawing threads, make your implementation continue to allow an unlimited number of balance checking threads to read concurrently.

14.12 The solution to the dining philosophers problem given in [Section 14.6](#) suffers from the problem that a philosopher could be starved by the two philosophers on either side of her, if she happened to get unlucky. Add variables to each philosopher which indicate hunger and the last time a philosopher has eaten. If a given philosopher is hungry and has not eaten for longer than her neighbor, her neighbor should not pick up the chopstick they share. Add synchronization tools to enforce this principle of fairness. Note that this solution should not cause deadlock. Although one philosopher may be waiting on another who is waiting on another and so on, **some** philosopher in the circle must have gone hungry the longest, breaking circular wait.

Experiments

14.13 Critical sections can slow down your program by preventing parallel computation. However, the locks used to enforce critical sections can add extra delays on top of that. Design a simple experiment which repeatedly acquires a lock and does some simple operation. Test the running time with and without the lock. See if you can estimate the time needed to acquire a

lock in Java on your system.

14.14 Design a program which experimentally determines how much time a thread is scheduled to spend running on a CPU before switching to the next thread. To do this, first create a tight loop which runs a large number of iterations, perhaps 1,000,000 or more. Determine how much time it takes to run a single run of those iterations. Then, write an outer loop which runs the tight loop several times. Each iteration of the outer loop, test to see how much time has passed. When you encounter a large jump in time, typically at least 10 times the amount of time the tight loop usually takes to run to completion, record that time. If you run these loops in multiple threads and average the unusually long times together for each thread, you should be able to find out about how long each thread waits between runs. Using this information, you can estimate how much time each thread is allotted. Bear in mind that this is only an estimation. Some JVMs will change the amount of CPU time allotted to threads for various reasons. If you are on a multicore machine, it will be more difficult to interpret your data since some threads will be running concurrently.

14.15 Create an experiment to investigate priority inversion in the following way.

- (a) Create two threads, setting the priority of the first to MIN_PRIORITY and the priority of the second to MAX_PRIORITY. Start the first thread running but wait 100 millisecond before starting the second thread. The first thread should acquire a shared lock and then perform some lengthy process such as finding the sum of the sines of the first million integers. After it finishes its computation, it should release the lock, and print a message. The second thread should try to acquire the lock, print a message, and then release the lock. Time the process. Because the lock is held by the lower priority thread, the higher priority thread will have to wait until the other thread is done for it to finish.
- (b) Once you have a feel for the time it takes for these two threads to finish alone, create 10 more threads that must also perform a lot of computation. However, these threads do not try to acquire the lock. How much do they delay completion of the task? How does this delay relate to the number of cores in your processor? How much does the delay change if you set the priorities of these new threads to MAX_PRIORITY or MIN_PRIORITY?

Chapter 15

Constructing Graphical User Interfaces

A good sketch is better than a long speech.

—Napoleon Bonaparte

15.1 Problem: Math applet

An *applet* is a graphical Java program that runs inside of a special environment, usually a web browser. The beauty of an applet is that anyone with a Java-enabled web browser can run the program without any special knowledge or computer skills. Most users don't even realize they are running a program.

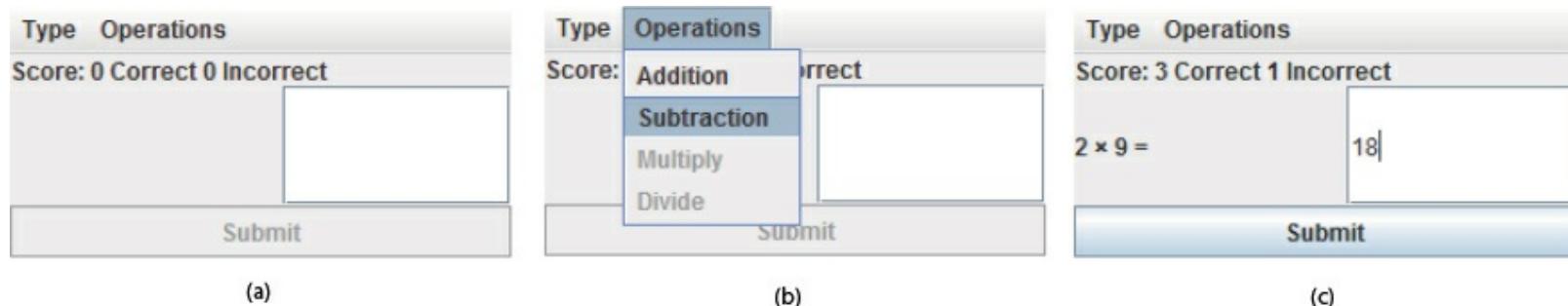


Figure 15.1: The MathTutorApplet as seen through the Google Chrome browser. (a) Initial applet. (b) Selecting an operation to practice. (c) After some practice.

Let's write an applet that will allow young math students to practice their arithmetic. Specifically, we are interested in addition, subtraction, multiplication, and division with small, positive integers. For this program, we will consider addition and subtraction basic and multiplication and division advanced. Our applet should allow the user to select a check box in a menu setting the mode to advanced or basic. The user should then be able to select one of the four operations from another menu. Once the operation is selected, the applet should generate a random problem testing that operation. The problem should be displayed as a label on the applet with a text field to one side. The user should be able to enter an answer in the text field and hit a button to submit it. The applet should check the answer and display the updated number of correct and incorrect answers.

Figure 15.1 shows the final applet in two different states. The screen to the left appears when the applet is first initialized. The applet is set in basic mode, and the Submit button is disabled until a problem is generated.

15.2 Concepts: Graphical user interfaces

The applet shown above with its menus, labels, buttons, and other interactive components is called a graphical user interface, or GUI. A GUI is a means of communication between a computer program

and a (usually human) user. (Although it is possible for some programs such as scripts to interact with a GUI, most program to program communication is done in other ways.) While communication through input and output on the console is one way to communicate with a program, GUIs offer a user-friendly alternative that has become extremely commonplace. In fact, GUIs are so common that many people have never used anything else to interact with programs and may not even suspect that other kinds of interaction are possible.

This chapter will teach you how to write programs with GUIs. Although GUIs can make input and output easier for the user, the programmer has to shoulder the burden of arranging the layout and appearance of the GUI and making it function properly. [Chapter 7](#) introduced a way to make simple GUIs, but those GUIs came in preset flavors designed for displaying a message, getting a range of responses in the form of buttons, or reading a short piece of text as input. In this chapter we will explore ways to make GUIs of arbitrary complexity with no limitations on the size or shape of the GUI or the components it contains.

A typical GUI consists of a *frame* (also known as a window) on which are displayed one or more components (known as widgets), such as panels, buttons, and text boxes. *Panels* are used to organize the contents within the frame. A frame contains at least one panel, but additional ones can be added. Each panel can also contain widgets: buttons, labels, text boxes, and even other panels. Using code to create a GUI with all the widgets laid out exactly where you want them is half the work of making a GUI-driven program in Java.

Some components like labels are read-only to the user. They display information such as status messages. Other components are more dynamic in the sense that the user can use these to give input to the program. Such components are associated with events. Handling events inside of a program is the other half of writing a GUI program in Java. Layout is concerned with appearance, but event handling is concerned with functionality.

There are IDEs such as NetBeans have graphical tools that automatically generate the GUI code for you. We focus on how to write this code yourself because doing so gives you more control and helps you understand Java GUIs better.

15.2.1 Swing and AWT

Most of the components you will use to create GUIs are defined in classes that belong to the Java Swing library. This library contains many interfaces and classes. Several components of the Swing library are built upon another library known as the Abstract Window Toolkit that is commonly referred to as AWT. In earlier chapters, you have seen, and probably used, one class in the Swing library, the JOptionPane class.

The AWT is an older library which provides direct access to the OS widgets. Thus, an AWT Button object in Microsoft Windows creates a Windows button. Swing, however, draws its own button. AWT GUIs look exactly like other GUIs from the same OS. Swing GUIs can be configured to look similar using *look and feel* settings, or developers can choose to use a default Java look and feel that will look almost the same across all platforms.

We will discuss Swing widgets such as JButton and JTextField. Swing components usually have a J at the beginning of their names to distinguish them from similar AWT components. (The AWT

contains Button and TextField. If you see examples from other sources using widgets that do not start with J, they are probably using AWT.) Although Swing is built on top of the older AWT library, it is not a good idea to mix Swing and AWT components in a single GUI, though it is possible.

The Swing library is far too large for us to cover in its entirety. Instead, our goal is to show you how to construct GUIs using some of the most common Java widgets. Once you have grasped the material in this chapter, you will be able to read and understand how to use many other interesting Swing components, as well as other Java libraries for constructing GUIs. Since a frame is the basis for most windowed GUIs, we will begin by showing you how to create a frame and display it on the screen.

15.3 Syntax: GUIs in Java

15.3.1 Creating a frame

A frame is the Java terminology for a window. Except when programming an applet, GUI components in Java will usually be found on a frame. In Swing, a frame is an object whose type is derived from the JFrame class. Here is a line of code that creates a JFrame object.

```
JFrame soundCheck = new JFrame (" Sound Check ");
```

The above statement declares and creates a JFrame object named soundCheck. The title of the frame is “Sound Check” and is given as an argument to the JFrame constructor. Although calling the constructor creates the frame, you need to make it visible for it to show up on the screen.

```
soundCheck.setVisible ( true );
```

The setVisible() method causes the soundCheck frame to be visible on the screen as a window. You can specify its size as in the following code.

```
soundCheck.setSize (300 ,200) ;
```

The setSize() method sets the width and height of the frame specified in pixels. In the above example, the width of soundCheck is set to 300 pixels and its height to 200 pixels. The window created by the above code is resizable. You could use the following code if you do not want the user to be able to resize the window.

```
soundCheck.setResizable ( false );
```

Your entire GUI will often be a frame you create and the components inside that frame. You might want your application to end when you close the frame by clicking on the close button towards the top. The actual location of this button depends on the operating system and the look and feel managers you are using. Regardless, the following statement can be used to set the behavior of the application when you close the frame window.

```
soundCheck.setDefaultCloseOperation ( JFrame.DISPOSE_ON_CLOSE );
```

Pitfall: Closing frames

Some books and online tutorials suggest setting `JFrame.EXIT_ON_CLOSE` as the default close operation for a `JFrame` instead of `JFrame.DISPOSE_ON_CLOSE`. In our opinion, this option should never be used.

Using `JFrame.EXIT_ON_CLOSE` is equivalent to calling `System.exit(0)`, which shuts down the JVM. In a single-threaded application, the difference between disposing and exiting is small. However, in a multi-threaded application, shutting down the JVM means killing off all the other threads, no matter what task they are performing. In general, the `System.exit()` should never be called, whether the program has a GUI or not.

By using `JFrame.DISPOSE_ON_CLOSE`, the frame releases all the resources it has been using and terminates any threads it uses to redraw itself and check for events. If there are no other threads running, the application will then shut down.

It is still necessary to select some default closing operation. By default, the operation is set to `JFrame.HIDE_ON_CLOSE`, which hides the frame but does not end its threads. Unfortunately, if the frame is the only way you have of interacting with your application, you can no longer use it! At that point, you may have to use a process or task manager to shut down the JVM by hand.

Example 15.1: Empty frame

[Program 15.1](#) creates and displays an empty frame with the title “Sound Check.” Note that we have used the `import` statement to let the compiler know that we are using the Swing library. The first line inside the `main()` method declares and creates a `JFrame` object and assigns a title to it. The following line sets its size. The third line sets the default close operation and the following line makes the frame visible. The frame so created is shown in [Figure 15.2](#). Note that the frame is resizable.

Program 15.1: Program to create an empty frame. (`EmptyFrame.java`)

```
1 import javax.swing.*;  
2  
3 public class EmptyFrame {  
4     public static void main ( String [] args ){  
5         // Create frame  
6         JFrame soundCheck = new JFrame (" Sound Check ");  
7         soundCheck.setSize (350 ,150) ; // Set size in pixels  
8         soundCheck.setDefaultCloseOperation (  
9                 JFrame.DISPOSE_ON_CLOSE );  
10        soundCheck.setVisible ( true ); // Display it  
11    }  
12 }
```

You may resize the frame at any point in the program even after the frame has been created and made visible. The initial size may or may not be set prior to making the frame visible. Similarly, the frame title can be set, and reset, at any point in the program. ■

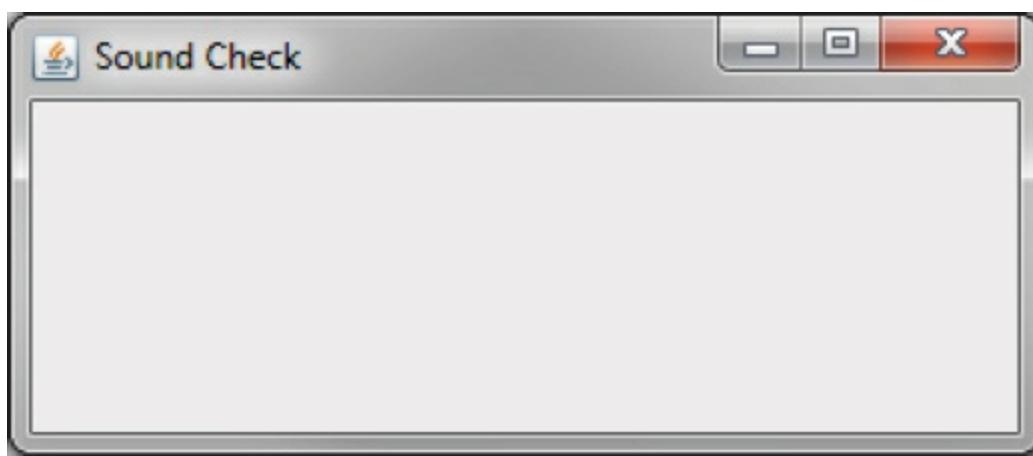


Figure 15.2: An empty frame titled “Sound Check.”

15.3.2 Widgets

A widget is a component of a GUI. Java provides a large variety of widgets including panels, buttons, text boxes, check boxes, and menus. While laying out a GUI, one or more of the widgets are created and then placed on a frame. A widget is declared and created like any other object.

```
Widget w = new Widget( arguments );
```

Here we use class Widget to represent some Java widget class like JButton or, and we use reference w to represent the name of a widget like startButton. Arguments supplied while constructing a widget allow you to set attributes such as its icon, color, size, or text to display. A widget can be added to a frame and removed when not needed. Its attributes can also be changed after creation.

Although widgets can be added directly to a frame, it is often convenient to lay out a GUI by placing panels on a frame and then placing widgets on the panels. Each panel can hold zero or more widgets. A panel is also referred to as a *container* object. Next we show you how to create a panel, populate it with widgets, place it on a frame, and display the completed frame.

A panel is an instance of the JPanel class and can be created as follows.

```
JPanel soundPanel = new JPanel();
```

This statement creates a panel named soundPanel. Thus far, the panel is empty. Let’s create two buttons and add these to soundPanel.

```
JButton chirp = new JButton("Chirp");
JButton bark = new JButton("Bark");
soundPanel.add(chirp);
soundPanel.add(bark);
```

The first two lines above create two buttons named chirp and bark. These buttons are labeled using the String values “Chirp” and “Bark”, but their labels could be any String values. The last two statements add the two buttons one by one to the soundPanel. Then we add the panel to a frame object.

```
soundCheck.add(soundPanel);
```

This statement adds the soundPanel to soundCheck frame that we created in [Example 15.1](#).

Another useful widget is JTextField. It creates a text field that can be used by a program for both input and output of String data.

```
JTextField message = new JTextField (" This is not a pipe. ");
```

This statement creates an object named message as an instance of the JTextField class. When displayed, it will show the message “This is not a pipe.” The user can replace this text by typing, but we will need to talk about event handling before we can read the text and act on it from within a program. The following example combines various concepts and widgets already introduced into one program.

Example 15.2: GUI with buttons and text field

We will now write an application with a GUI that contains three buttons labeled “Chirp,” “Bark,” and “Exit.” In addition it contains a text field that initially displays the text, “Listen to nature!”

In this example the buttons are only for display. You can click each one, but the program will not do anything useful. Similarly, the text field will not be changed by the program after it is initialized. In the next subsection we add actions to each button and make the program more useful.

Program 15.2: Program to create a frame with a panel containing three buttons.
(FrameWithPanel.java)

```
1 import javax.swing.*;
2
3 public class FrameWithPanel {
4     public static void main ( String [] args ){
5         // Create frame
6         JFrame soundCheck = new JFrame (" Sound Check ");
7         JPanel soundPanel = new JPanel ();
8         // Create three buttons
9         JButton chirp = new JButton (" Chirp ");
10        JButton bark = new JButton (" Bark ");
11        JButton exit = new JButton (" Exit ");
12        JTextField message = new JTextField (" Listen to nature !");
13        soundPanel.add( chirp ); // Add chirp to panel
14        soundPanel.add( bark ); // Add bark to panel
15        soundPanel.add( message ); // Add message box
16        soundPanel.add( exit ); // Add exit button
17        soundCheck.add( soundPanel ); // Add the panel to the frame
18        soundCheck.setSize (350 ,150) ; // Set size in pixels
19        soundCheck.setDefaultCloseOperation (
20            JFrame.DISPOSE_ON_CLOSE );
21        soundCheck.setVisible ( true ); // Display frame
22    }
```

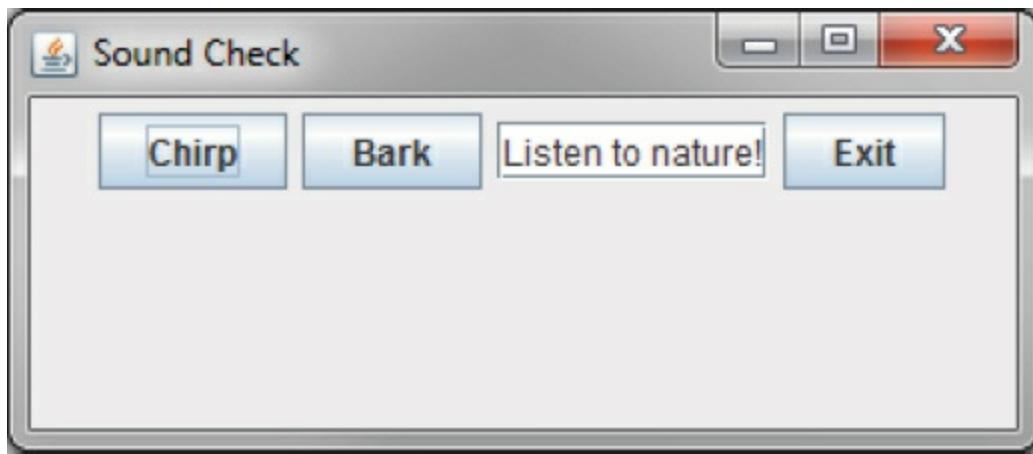


Figure 15.3: GUI consisting of a frame, a panel, three buttons, and a text field.

In the `main()` method in [Program 15.3](#), we first create a frame named `soundCheck` and a panel named `soundPanel`. Next, we create three buttons named `chirp`, `bark`, and `exit`, labeled “Chirp,” “Bark,” and “Exit,” respectively. Lines 13 until 16 add the three buttons and the text field to `soundPanel`. At [line 17](#), `soundPanel` is added to `soundCheck`. After setting the closing operation, the frame is made visible. The final GUI is shown in [Figure 15.3](#). Note that the same GUI may look different on different platforms. Later we will see how to add multiple panels to a frame. ■

Exercise 15.17

15.3.3 Adding actions to widgets

Many widgets in Java can be used to cause an action. For example, a user could click a button labeled “Chirp,” causing the program to play a bird chirp sound. Clicking a button generates an *event*. In Java, an event is processed by one or more *listeners*. Java provides various types of listeners, some of which are introduced here and a few others in [Section 15.3.7](#). Next we show you how to handle action events generated by a few different kinds of widgets.

The ActionListener interface

Java provides an `ActionListener` interface. This interface has a single method named `actionPerformed()`. This method takes an `ActionEvent` as input and performs a suitable action based on the event. A `JButton` object generates `ActionEvent` when it is pressed. Any class that implements the `ActionListener` interface can be registered as an action listener on a `JButton` or any other widget that generates an `ActionEvent`.

As discussed in [Chapter 10](#), an interface is a set of method signatures. If you implement an interface, you promise to have all of the methods in the interface. If a class implements `ActionListener`, it’s saying that it knows what to do when an action is performed. The following statements show how to add an `ActionListener` to a button and implement its `actionPerformed()` method.

```
JButton chirp = new JButton ("Chirp");
```

```
chirp.addActionListener ( new ActionListener () {  
    public void actionPerformed ( ActionEvent e ) {  
        // Code to perform an action goes here  
    }  
});
```

The first line above creates a button named chirp. The second line adds an action listener to the button. The process of adding an action listener to an object is also known as *registering* a listener on the object. Note that the sole argument to this `addActionListener()` method is a newly created `ActionListener` object. Inside this newly created and anonymous `ActionListener` object, we implement the `actionPerformed()` method. Whatever code we want to execute in response to the clicking of the chirp button goes inside the `actionPerformed()` method.

This syntax may look strange to you. `ActionListener` is an interface, which cannot be instantiated. What is that `new` keyword doing? It's doing something pretty amazing by creating an instance of an *anonymous class*. On the fly, we're creating a class that has never existed before. It doesn't even have a name. All we know about it is that it implements the interface `ActionListener`.

Note that there are braces after the constructor call, defining what's inside of this class. Inside, we have only created an `actionPerformed()` method, but we could have created fields as well as other methods. It's a little ugly to create a whole new class and instantiate it in the middle of calling the `addActionListener()` method, but it's also very convenient. We need to supply an object that reacts to the event exactly the way we want it to. Since one doesn't exist yet, we have to create it. Of course, it is possible to supply any object that implements the `ActionListener` interface, not just instances of anonymous classes. For more information about nested classes, inner classes, and anonymous classes, refer to [Sections 9.5](#) and [10.4](#).

Example 15.3: GUI with actions

We now modify [Program 15.2](#) to respond to button clicks. When the chirp button is clicked, the program will display the message "Chirp requested." in the text field. Similarly, when the bark button is clicked, the program will display "Bark requested."

[Program 15.3](#) is largely the same as [Program 15.2](#). It adds the same buttons and text field but then adds an action listener to each button. The action performed when the chirp and bark buttons are clicked is to display a message in the text box. When the exit button is clicked, the listener displays a message on the console and exits the program.

Program 15.3: Program to demonstrate handling of action events. (`FrameWithPanelAndActions.java`)

```
1 import javax.swing.*;  
2 import java.awt.event.*;  
3  
4 public class FrameWithPanelAndActions {  
5     public static void main ( String [] args ){  
6         final JFrame soundCheck = new JFrame (" Sound Check ");  
7         JPanel soundPanel = new JPanel();
```

```

8     JButton chirp = new JButton ("Chirp ");
9     JButton bark = new JButton ("Bark ");
10    JButton exit = new JButton ("Exit ");
11    final JTextField message = new JTextField (
12        " Listen to nature !");
13    soundPanel.add( chirp );
14    soundPanel.add( bark );
15    soundPanel.add( message );
16    soundPanel.add( exit );
17    // Add action listeners to various buttons
18    chirp.addActionListener ( new ActionListener () {
19        public void actionPerformed ( ActionEvent e){
20            message.setText (" Chirp requested.");
21        }
22    });
23    bark.addActionListener ( new ActionListener (){
24        public void actionPerformed ( ActionEvent e){
25            message.setText (" Bark requested.");
26        }
27    });
28    exit.addActionListener ( new ActionListener (){
29        public void actionPerformed ( ActionEvent e){
30            System.out.println (" Exit ");
31            soundCheck.dispose ();
32        }
33    });
34    soundCheck.add( soundPanel );
35    soundCheck.setSize (350 ,150 );
36    soundCheck.setDefaultCloseOperation (
37        JFrame.DISPOSE_ON_CLOSE );
38    soundCheck.setVisible ( true );
39    }
40 }

```

Note that the sequence in which you add the buttons to the panel determines the appearance of the GUI. The action listeners can be added either before or after the panel has been set up but should be added before the frame is made visible.

Exercise 15.16

Do not be overly concerned with why the **final** keyword is used when declaring soundCheck and message. You might be surprised that you are even allowed to reference these local variables inside of the third ActionListener class. An inner class has access to all the fields inside of its outer class and access to local variables declared in the same scope as the inner class. For technical reasons, any local variable used in an anonymous inner class must be declared **final**. This restriction does not

apply to class variables. ■

Example 15.4: GUI with alternate action listener style

In the previous example we added an ActionListener object to each button and implemented its actionPerformed() method with anonymous inner classes. An alternate way to use ActionListener is to implement ActionListener on the surrounding class and include an actionPerformed() method exactly once instead of creating several individual anonymous inner classes which each handle an event. Let's examine one such implementation in [Program 15.4](#) and contrast it with [Program 15.3](#). Note that both programs generate exactly the same GUI and exhibit identical behavior.

Program 15.4: Program to demonstrate handling of action events by implementing ActionListener at the class level. (AlternateActionListener.java)

```
1 import javax.swing.*;
2 import java.awt.event.*;
3
4 public class AlternateActionListener implements ActionListener {
5     JFrame soundCheck = new JFrame (" Sound Check ");
6     JPanel soundPanel = new JPanel ();
7     JButton chirpButton = new JButton (" Bird ");
8     JButton barkButton = new JButton (" Dog ");
9     JButton exitButton = new JButton (" Exit ");
10    JTextField message = new JTextField (" Listen to nature !!");
11
12    public AlternateActionListener (){
13        chirpButton.addActionListener ( this );
14        barkButton.addActionListener ( this );
15        exitButton.addActionListener ( this );
16        soundPanel.add ( chirpButton );
17        soundPanel.add ( barkButton );
18        soundPanel.add ( message );
19        soundPanel.add ( exitButton );
20        soundCheck.add ( soundPanel );
21        soundCheck.setSize ( 200 ,125 );
22        soundCheck.setDefaultCloseOperation (
23            JFrame.DISPOSE_ON_CLOSE );
24        soundCheck.setVisible ( true );
25    }
26
27    public void actionPerformed ( ActionEvent e){
28        Object button = e.getSource ();
29        if( button == chirpButton )
30            message.setText (" Chirp requested.");
31        else if( button == barkButton )
```

```

32     message.setText (" Bark requested.");
33     else {
34         System.out.println (" Exit ");
35         soundCheck.dispose ();
36     }
37 }
38
39 public static void main ( String [] args ){
40     new AlternateActionListener ();
41 }
42 }
```

In [Program 15.4](#) class `AlternateActionListener` implements `ActionListener`. Doing so requires the class to include `actionPerformed()`. The constructor, starting at [line 12](#), adds an `ActionListener` to each button. The listener added is `this`, specifying that the `AlternateActionListener` object is the one that will process any action event generated by the buttons. The remainder of the code for the constructor is essentially the same as that from the `main()` method in [Program 15.3](#).

The `actionPerformed()` method, starting at [line 27](#), has exactly one parameter, an `ActionEvent` object. Whenever an action event occurs, its attributes are bundled into an `ActionEvent` object and passed into the `actionPerformed()` method. At [line 28](#) the `getSource()` method is used to determine which object is responsible for the event. Variable `button` holds the object returned by `getSource()`. The next `if` statement compares `button` with `chirp` and `bark` to determine if either of these generated the event. Then a suitable message is displayed in the message box. If neither button generated the event, it must have been `exit`, and the frame disposes itself after displaying “Exit” on the console.

The `main()` method simply creates an instance of `AlternateActionListener` and terminates. The program does not end because the threads for the GUI are still running. ■

It is instructive to note the differences between [Programs 15.3](#) and [15.4](#). In [Program 15.4](#), most of the code has been moved from the `main()` method to the constructor. Various objects, namely the `soundCheck` frame, the `soundPanel` panel, all three buttons, and the text box are now fields of the object instead of local variables in the `main()` method. This removes the need to mark `soundCheck` and `message` as `final`.

We have examined two ways of adding an `ActionListener` to a Java program. The choice of which style to use depends on your needs. Adding an anonymous `ActionListener` to each object can require you to use the `final` keyword to reference local variables, but it is otherwise quick and easy. Using a named class (often the main program class or a subclass of `JFrame`) as the `ActionListener` allows you to handle many events in a centralized location. It can be easier to find errors when all events are handled in one `actionPerformed()` method, but the method can become long and complex as well.

Exercise 15.6

The MouseListener interface

Clicking a button is great, but a mouse can be used to generate other events too. For example, in a screen full of pictures, you might want to highlight a picture when the cursor hovers over it. Or you

might want to create a drawing program which uses a mouse as a pen. To process general mouse events, we need an object that implements the `MouseListener` interface, which defines the following methods.

- `mouseClicked()`
- `mouseEntered()`
- `mouseExited()`
- `mousePressed()`
- `mouseReleased()`

The function of each method is implied by its name. The `mouseEntered()` event fires when the mouse cursor moves from outside of the area covered by a widget into the area above it. Conversely, the `mouseExited()` event fires when a mouse cursor was over a widget and has just moved away. The `mousePressed()` event fires when a mouse button is pressed over a widget. The `mouseReleased()` event fires when a mouse button is released over a widget. The `mouseClicked()` event is a combination of both the `mousePressed()` and `mouseReleased()` events, occurring only if a mouse button was pressed and then released while the cursor was over a widget. As you can see, a widget only fires events when the cursor is over the component (or has just left). Thus, a widget only reports events that have to do with it, not the general state of the mouse.

Each method in `MouseListener` receives a `MouseEvent` object as its argument. To handle mouse events, a class must implement the `MouseListener` interface. This is similar to the implementation of the `ActionListener` interface from the previous section, but implementing `MouseListener` requires a definition for **each** of the five methods listed above. The next example illustrates `MouseListener` in use.

Example 15.5: Mouse listener

We write a program that displays a GUI containing two buttons labeled “One” and “Two.” In addition a text box displays a suitable message when the cursor enters a button. When a button is clicked, the status box should display the total number of times that button has been clicked so far.

Program 15.5 generates the GUI shown in Figure 15.4. At line 5 class `SimpleMouseEvents` is declared, implementing the `MouseListener` interface. The following few lines declare frame `frame`, buttons `one` and `two`, and a text box `status`. Two integers `oneClicks` and `twoClicks` are initialized to 0 and are used to keep track of the number of times each button has been clicked.

Program 15.5: Program to demonstrate the handling of mouse generated events using the `MouseListener` interface. (`SimpleMouseEvents.java`)

```
1 import javax.swing.*;  
2 import java.awt.*;  
3 import java.awt.event.*;
```

```
4
5 public class SimpleMouseEvents implements MouseListener {
6     JFrame frame = new JFrame (" Mouse Events ");
7     JTextField status = new JTextField (
8         " Mouse status comes here .");
9     JButton one = new JButton (" One");
10    JButton two = new JButton (" Two");
11    int oneClicks = 0, twoClicks = 0; // Number of clicks
12
13    public SimpleMouseEvents () {
14        JPanel panel = new JPanel ();
15        one.addMouseListener ( this );
16        two.addMouseListener ( this );
17        panel.add ( one );
18        panel.add ( two );
19        panel.add ( status );
20        frame.add ( panel );
21        frame.setSize ( 200 ,100 );
22        frame.setDefaultCloseOperation (
23            JFrame.DISPOSE_ON_CLOSE );
24        frame.setVisible ( true );
25    }
26
27 // Implement all abstract methods in MouseListener
28    public void mouseEntered ( MouseEvent e ) {
29        if (e.getSource () == one )
30            status.setText (" Mouse enters One .");
31        else
32            status.setText (" Mouse enters Two .");
33    }
34
35    public void mouseClicked ( MouseEvent e ) {
36        if (e.getSource () == one ) {
37            oneClicks++;
38            status.setText (" One clicked "+ oneClicks +
39                " times .");
40        }
41        else {
42            twoClicks++;
43            status.setText (" Two clicked "+ twoClicks +
44                " times .");
45        }
46    }

```

```

47
48     public void mouseExited ( MouseEvent e ) {}
49     public void mousePressed ( MouseEvent e ) {}
50     public void mouseReleased ( MouseEvent e ) {}
51     public static void main ( String [] args ){
52         new SimpleMouseEvents ();
53     }
54 }
```

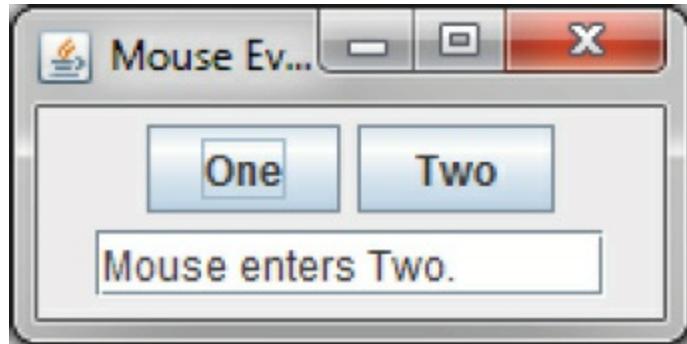


Figure 15.4: GUI consisting of two buttons and a text box. Button clicks and entry of the cursor into a button are reported by the text box.

The first line of the SimpleMouseEvents constructor creates the panel panel. It does not need to be a field, and it is always preferable to keep a variable local if it can be. The next two lines add a MouseListener to the two buttons. Note the use of `this` in the argument to `addMouseListener()` which refers to the object being created by the constructor. Next, the panel is set up and added to the frame. Finally the frame size and its default close operations are set, and the frame is made visible.

Implementation of the methods in the MouseListener interface begins at [line 28](#). The `mouseEntered()` method is invoked when the cursor enters either of the two buttons. First we retrieve the source of the event using the `getSource()` method and identify which object generated the event. A suitable message is displayed in the status box using the `setText()` method.

The `mouseClicked()` method is invoked when the mouse cursor is placed over a button and clicked. As before, we retrieve the source of the event using the `getSource()` method. A suitable message, including the number of clicks, is displayed in the text box. Of course, recording the button clicks could have been done with an ActionListener instead.

The only job of the `main()` method is to create an instance of `SimpleMouseEvents`. You may wish to compile and execute the program and test whether or not the program behaves as described. ■

Exercise 15.8

Mouse adapter

Creating a MouseListener requires all five methods in the interface to be implemented. In some cases as in [Example 15.5](#), there is no need to implement all the methods because we are not interested in all the corresponding events. In such situations we are forced to include the methods without any statement in the method body. However, you might want to include the methods only when they are

needed. The MouseAdapter abstract class helps us avoid implementing methods we do not need.

MouseListener is an abstract class, unlike the MouseListener interface. The advantage of using MouseAdapter is that it already provides a skeletal implementation of each method needed to process mouse events. We can override these implementations as needed, and we do not need to provide an implementation of a method that is not used.

Example 15.6: Mouse adapter

Program 15.6 is a revised version of Program 15.5. At line 5 class SimpleMouseAdapter extends the abstract class MouseAdapter. Thus, it inherits all the empty methods defined in MouseAdapter. We then override only the implementations of the methods we want to use, mouseEntered() and mouseClicked() in this example. Remember that an abstract class is **extended** whereas an interface is **implemented**.

Exercise 15.3

Program 15.6: Program to handle mouse generated events using the MouseAdapter abstract class.
(SimpleMouseAdapter.java)

```
1 import javax.swing.*;
2 import java.awt.*;
3 import java.awt.event.*;
4
5 public class SimpleMouseAdapter extends MouseAdapter {
6     JFrame frame = new JFrame (" Mouse Events ");
7     JTextField status = new JTextField (
8         " Mouse status comes here. ");
9     JButton one = new JButton (" One ");
10    JButton two = new JButton (" Two ");
11    int oneClicks = 0, twoClicks = 0;
12
13    public SimpleMouseAdapter () {
14        JPanel panel = new JPanel ();
15        one.addMouseListener ( this );
16        two.addMouseListener ( this );
17        panel.add ( one );
18        panel.add ( two );
19        panel.add ( status );
20        frame.add ( panel );
21        frame.setSize ( 200 ,100 );
22        frame.setDefaultCloseOperation (
23            JFrame.DISPOSE_ON_CLOSE );
24        frame.setVisible ( true );
25    }
26}
```

```

27 // Override only those methods we wants
28 public void mouseEntered ( MouseEvent e) {
29     if(e. getSource () == one )
30         status.setText (" Mouse enters One.");
31     else
32         status.setText (" Mouse enters Two.");
33 }
34
35 public void mouseClicked ( MouseEvent e) {
36     if(e. getSource () == one ) {
37         oneClicks++;
38         status.setText (" One clicked "+ oneClicks +
39             " times.");
40     }
41     else {
42         twoClicks++;
43         status.setText (" Two clicked "+ twoClicks +
44             " times.");
45     }
46 }
47
48 public static void main ( String [] args ){
49     new SimpleMouseAdapter ();
50 }
51 }
```

Other event listeners

In this chapter we describe two types of listeners in Java, ActionListener and MouseListener. You may have noticed that none of the mouse events we discussed involved the movement of the mouse inside of the component, only whether it was entering or exiting the component. Because tracking mouse movement is more computationally expensive than tracking presses, releases, enters, and exits, Java uses yet another listener to handle mouse movement, MouseMotionListener. It contains the methods mouseDragged() and mouseMoved(), which are used to handle mouse movement with or without the button pressed.

An ItemListener can be attached to a widget such as a check box or a radio button to listen for a check box to be selected. This listener is illustrated in [Section 15.3.7](#).

Java provides several other listeners to handle a variety of events. For example, the DocumentListener can be attached to a JTextField or a JTextArea object to listen to document events which include the insertUpdate() event that is fired when a character is inserted the text box. A KeyListener can also be attached to text boxes to listen to key events such as the return key being typed, which can have similar functionality. These events could be useful while writing a text editor application, for example.

After you have mastered the contents of this chapter, you may plan to write more complex GUIs than the ones we discuss. For further information, you may wish to follow the Java tutorial on writing event listeners provided by Oracle at <http://download.oracle.com/javase/tutorial/uiswing/events/index.html>.

15.3.4 Adding sounds and images

Sounds and images can also be added to a Java GUI application. While Java offers a rich set of sound APIs, we restrict our examples to playing sound clips from audio files that come in au or wav formats. We also introduce the ImageIcon class to create icons from image files.

Sounds

First, let's see how to define an audio clip.

```
AudioClip chirpClip = Applet.newAudioClip( chirpURL );
```

This statement declares chirpClip with type AudioClip. It instantiates chirpClip using the newAudioClip() method found in the Applet class in the java.applet package. Note that the argument to the newAudioClip() method is an object of type URL. URL stands for *universal resource locator*. It serves as a web address from which files can be retrieved. A URL object can be defined as follows.

```
URL chirpURL =  
    new URL (" http://users.etown.edu/w/ wittmanb / chirp.wav");
```

Above we specify the full URL to access an audio file named chirp.wav. Alternately you can give the file name from a local directory.

```
URL chirpURL = new URL(" file : sounds / chirp.wav ");
```

The prefix file: indicates that the chirp.wav file is in a local directory. Once a clip has been declared and loaded, it can be played as follows.

```
chirpClip.play();
```

This command will play the clip in chirpClip loaded from the specified URL, exactly once. If you want to play the clip in a loop, use the loop() method.

```
chirpClip.loop();
```

To stop a clip from playing, use the stop() method.

```
chirpClip.stop();
```

Now that we have seen how to declare, load, play, and stop an audio clip, we are ready to write a more interesting version of [Program 15.3](#).

Example 15.7: Sound game

We can rewrite [Program 15.3](#) so that the new program plays the sound clips in a loop when the corresponding button is clicked. Think of the new program as a Java version of the pull-string toys that children use to play animal sounds. Our program only has two sounds, but more could be added.

Exercise 15.11

We will add a new button labeled “Stop Sound” that stops the playback of sounds when clicked. Let’s assume that this program is only allowed to play one sound at a time. When it starts, the GUI will look like [Figure 15.5\(a\)](#). Note that the stop button is gray, indicating that it is disabled.

The complete program for the game is shown in [Program 15.7](#). Because most of this program is similar to [Program 15.2](#), we will only look at the differences. The action listeners for the chirp and bark buttons are called when these two buttons are clicked. On [line 30](#) we disable the bark button when the bird chirp sound is playing. Also, on [line 32](#) we enable the stop button so that the user can stop the chirp sound. Note that the stop button begins disabled and remains that way until a sound is played. The GUI now looks like [Figure 15.5\(b\)](#).



Figure 15.5: GUI for the sound game application. (a) On program start. (b) After the “Chirp” button has been clicked and the clip is playing.

Program 15.7: An animal sound game. (SoundGame.java)

```
1 import javax.swing.*;
2 import java.awt.event.*;
3 import java.applet.*;
4 import java.net.URL ;
5 
6 public class SoundGame {
7     public static void main ( String [] args ) throws Exception {
8         final JFrame soundGame = new JFrame (" Sound Game ");
9         JPanel soundPanel = new JPanel ();
```

```
10 final JButton chirp = new JButton (" Chirp ");
11 final JButton bark = new JButton (" Bark ");
12 final JButton stop = new JButton (" Stop Sound ");
13 final JButton exit = new JButton (" Exit ");
14 final JTextField message =
15     new JTextField (" Click Chirp or Bark. ");
16 URL chirpURL = new URL(" file : sounds / chirp.wav ");
17 URL barkURL = new URL (" file : sounds / bark.wav ");
18 final AudioClip chirpClip = Applet.newAudioClip ( chirpURL );
19 final AudioClip barkClip = Applet.newAudioClip ( barkURL );
20 soundPanel.add ( chirp );
21 soundPanel.add ( bark );
22 soundPanel.add ( stop );
23 soundPanel.add ( exit );
24 soundPanel.add ( message );
25 soundGame.add ( soundPanel );
26 stop.setEnabled ( false );
27 chirp.addActionListener ( new ActionListener () {
28     public void actionPerformed ( ActionEvent e){
29         message.setText (" Chirp sound playing. ");
30         bark.setEnabled ( false );
31         chirpClip.loop ();
32         stop.setEnabled ( true );
33     }
34 });
35 bark.addActionListener ( new ActionListener (){
36     public void actionPerformed ( ActionEvent e){
37         message.setText (" Bark sound playing. ");
38         chirp.setEnabled ( false );
39         barkClip.loop ();
40         stop.setEnabled ( true );
41     }
42 });
43 stop.addActionListener ( new ActionListener (){
44     public void actionPerformed ( ActionEvent e){
45         message.setText (" Click Chirp or Bark. ");
46         chirpClip.stop ();
47         bark.setEnabled ( true );
48         barkClip.stop ();
49         chirp.setEnabled ( true );
50         stop.setEnabled ( false );
51     }
52 });
```

```

53     exit.addActionListener ( new ActionListener (){
54         public void actionPerformed ( ActionEvent e){
55             System.out.println (" Exit ");
56             soundGame.dispose ();
57         }
58     });
59     soundGame.setSize (175 ,175) ; // Set size in pixels
60     soundGame.setDefaultCloseOperation (
61         JFrame.DISPOSE_ON_CLOSE );
62     soundGame.setVisible ( true ); // Display it
63 }
64 }
```

The actions for the bark button are very similar to the actions for the chirp button, but the actions for stop are different. When stop is clicked, the action listener does not know which sound is playing. For that reason, it stops both sounds and then disables the stop button. An exercise asks you to modify the program so that the action listener associated with the stop button knows which sound is playing.

Exercise 15.12

Images and icons

Images are often useful when creating GUIs. In this section we show you how to use images to create icons and then use the icon to decorate buttons and labels. First, let's see how an icon object can be created. Suppose we have a picture file named smile.jpg in a directory named pictures. Note that the pictures directory should be located in same directory as the class files for the program. The following statement creates an object of type ImageIcon from this picture.

```
ImageIcon smileIcon = new ImageIcon (" pictures / smile.jpg ");
```

Note that the file name, along with its path, is passed to the ImageIcon constructor as a string. Now we can add the image to a button.

```

JButton smile = new JButton ();
smile.add ( smileIcon );
```

Similarly you can add an image icon to a label. The next example gives a simple program Exercise 15.10 that creates a button with an image.

Example 15.8: Icon example

[Figure 15.6](#) shows a GUI with a button decorated with a picture. [Program 15.8](#) gives the code to create this GUI using the ImageIcon class.

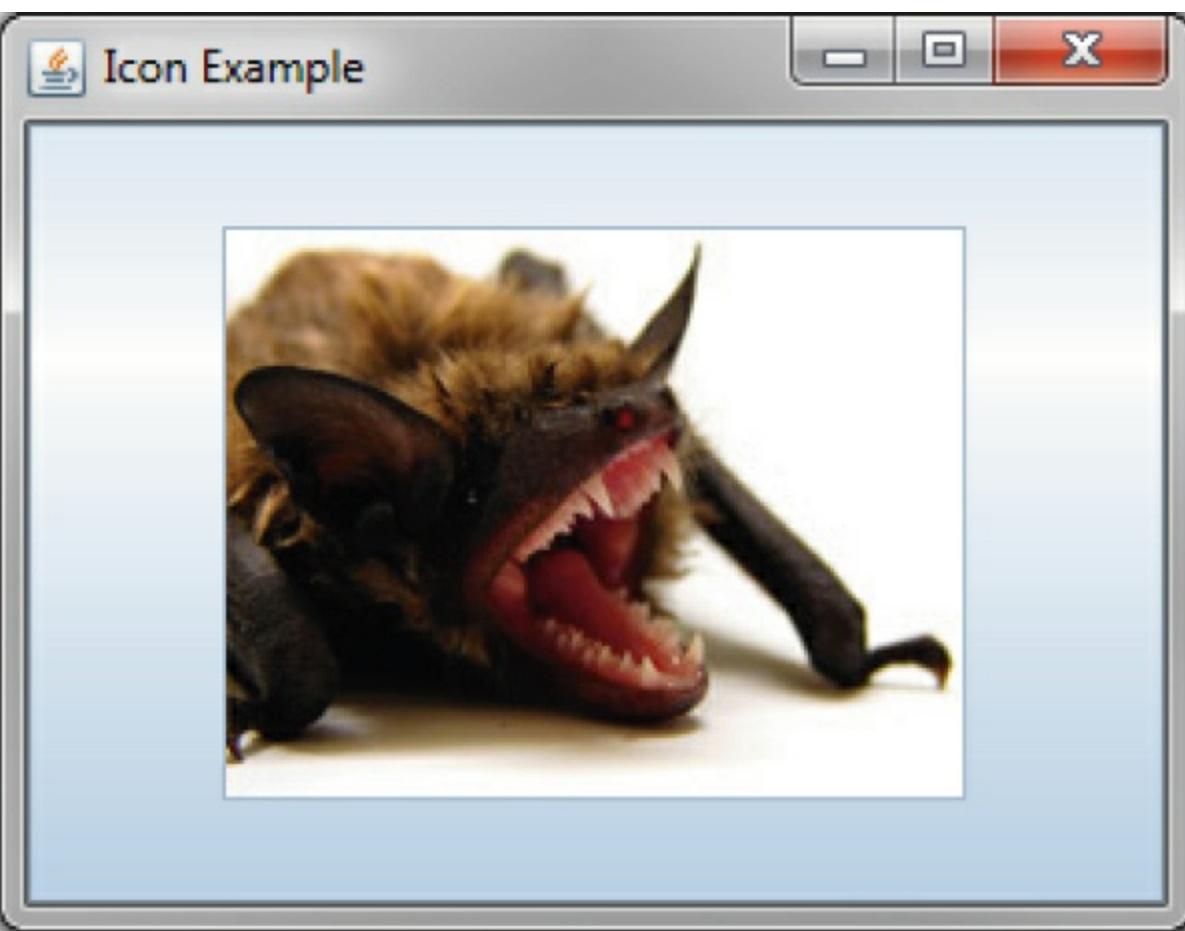


Figure 15.6: A GUI with a button decorated by an image icon.

As explained earlier, an image icon is created at [line 7](#) from a JPEG file named smile.jpg located in the pictures directory. The following line creates a button named smile and decorates it with the icon by supplying it as an argument to the JButton constructor. Note that, if the file cannot be found, the program will fail *quietly*. This means that no exception is thrown. Instead, the button will appear with no image. Subsequent lines add the button to the frame, set the frame size and its default close operation, and make the frame visible.

Exercise 15.9

Program 15.8: A program to create a GUI with a button decorated by a picture. (IconExample.java)

```
1 import javax.swing.*;
2 import java.awt.*;
3
4 public class IconExample {
5     public static void main ( String [] args ) {
6         JFrame iconFrame = new JFrame ("Icon Example");
7         ImageIcon smileIcon = new ImageIcon (
8             "pictures / smile.jpg ");
9         JButton smile = new JButton ( smileIcon );
10        iconFrame.add ( smile );
11        iconFrame.setSize ( 325 ,250 );
12        iconFrame.setDefaultCloseOperation (
```

```
13         JFrame.DISPOSE_ON_CLOSE );
14     iconFrame.setVisible ( true );
15 }
16 }
```



Labels, icons, and text

In some applications you might want to show a picture with an attached text label. For example, in a shopping cart for an online clothing store, you might have seen pictures of clothes, each labeled with a name and a price. The JLabel class is flexible, able to display text alone, an image alone, or both. Note that a JLabel is only for displaying information and is incapable of firing events or reading user input.

Here are three different ways to create a label.

```
ImageIcon hibiscus = new ImageIcon (" pictures / hibiscus.jpg ")
JLabel textOnly = new JLabel (" Text only ");
JLabel flower = new JLabel ( hibiscus );
JLabel labeledflower = new JLabel (" Red Hibiscus ", hibiscus, JLabel.CENTER );
```

The first JLabel constructor above creates a label displaying only text, namely, “Text only.” The second constructor creates a label decorated with an icon created from a picture. The third constructor creates a label with the same icon and additional text. Note the last argument in this third case. JLabel.CENTER is a constant that specifies that the content of the label (both the image and the text) should be placed horizontally in the center of the label. A horizontal alignment of left or right can also be specified using the constants JLabel.LEFT or JLabel.RIGHT, respectively. These three horizontal alignment indicators are some of the many constants found in Swing.

Sometimes you might want to place the text below the icon that decorates the label. This can be done as follows by setting the horizontal and vertical positions of the text.

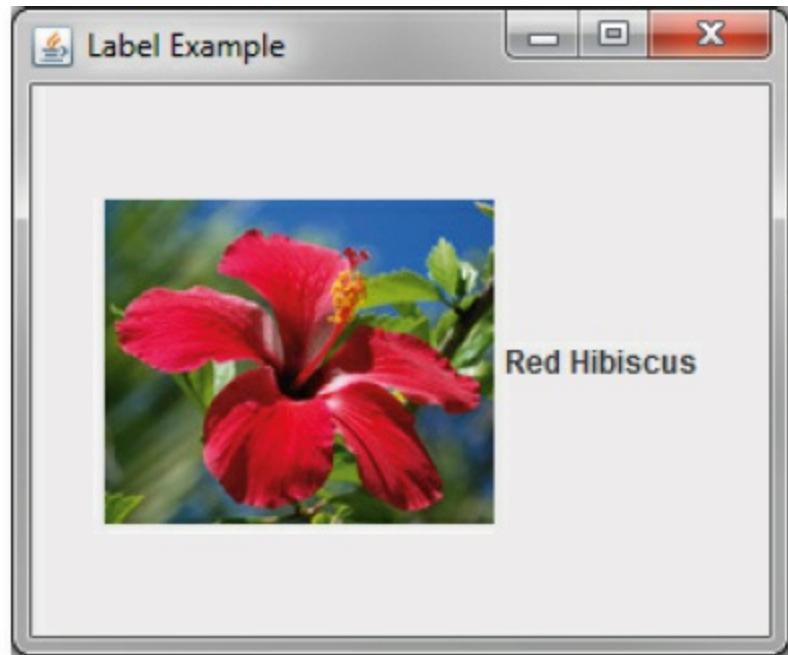
```
flower.setVerticalTextPosition ( JLabel.BOTTOM );
flower.setHorizontalTextPosition ( JLabel.CENTER );
```

Example 15.9: Label example

Figure 15.7(a) is generated by executing Program 15.9. You will see the GUI shown in Figure 15.7(b) if the alignment instructions at lines 11 and 12 are omitted.



(a)



(b)

Figure 15.7: (a) A GUI with a label decorated by an image icon and a title beneath it. (b) The GUI with unaligned text.

Program 15.9: A program to create a GUI with a label decorated by a picture and text. (LabelExample.java)

```
1 import javax.swing.*;
2 import java.awt.*;
3
4 public class LabelExample {
5     public static void main ( String [] args ) {
6         JFrame frame = new JFrame ("Label Example");
7         ImageIcon hibiscus = new ImageIcon (
8             "pictures / hibiscus.jpg");
9         JLabel flower = new JLabel("Red Hibiscus",
10             hibiscus, JLabel.CENTER );
11         flower.setVerticalTextPosition ( JLabel.BOTTOM );
12         flower.setHorizontalTextPosition ( JLabel.CENTER );
13         frame.add ( flower );
14         frame.setSize (300 ,250) ;
15         frame.setDefaultCloseOperation (
16             JFrame.DISPOSE_ON_CLOSE );
17         frame.setVisible ( true );
18     }
19 }
```



15.3.5 Layout managers

Java provides a number of *layout managers* to assist with the design of GUIs. A layout manager is an object that controls the placement of widgets on a frame or panel. Every container has a default layout manager, but it is possible to set it to other ones. In this section we will introduce the three layout managers FlowLayout, GridLayout, and BorderLayout. Java also provides several other layout managers, each designed for different layouts.

FlowLayout

The FlowLayout manager is one of simplest layout managers. When a container is using the FlowLayout manager, widgets will be added in order from left to right. When there is no more space, subsequent widgets will be added starting on the next row. In addition, each row of components is centered within the container. The JPanel container uses FlowLayout by default, but it is possible to set it explicitly as well.

```
JPanel panel = new JPanel(new FlowLayout());
```

When we have added more than one widget to a JFrame in previous examples, we have first added them to a JPanel. The reason we have done so is because FlowLayout is the default layout manager for JPanel containers. Although every JFrame has a container, it uses the BorderLayout manager by default, which would have complicated our examples. The next example illustrates FlowLayout further.

Example 15.10: FlowLayout

Program 15.10 creates a GUI with several buttons. It first creates a frame. At line 7 it creates a panel and set its layout manager to FlowLayout. Then, it uses a for loop to create a new button, label it appropriately, and add it to the panel.

The FlowLayout manager neatly places the buttons along a number of rows depending on the width of the frame. The GUI created is shown in Figure 15.8. Run the program and resize the frame to see the effect on the layout of the buttons.

Exercise 15.18

Exercise 15.19

Program 15.10: Adding several buttons using FlowLayout. (FlowLayoutExample.java)

```
1 import javax.swing.*;
2 import java.awt.*;
3 
4 public class FlowLayoutExample {
5     public static void main ( String [] args ) {
6         JFrame frame = new JFrame ("FlowLayout Example");
7         JPanel panel = new JPanel ( new FlowLayout ());
8         final int MAX_BUTTONS = 6;
9         for( int i = 0; i < MAX_BUTTONS ; i ++)
10             panel.add ( new JButton (" " + i + " ") );
```

```
11     frame.add ( panel );
12     frame.setSize ( 250 ,100 ) ;
13     frame.setResizable ( false );
14     frame.setDefaultCloseOperation ( JFrame.DISPOSE_ON_CLOSE );
15     frame.setVisible ( true );
16 }
17 }
```

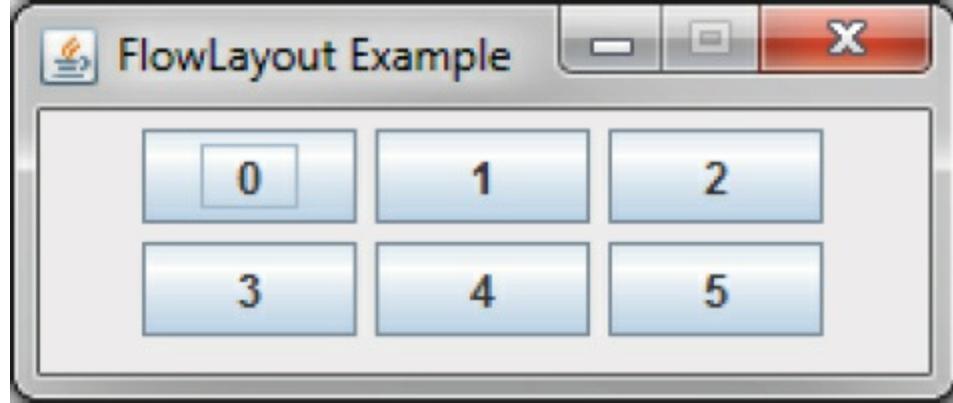


Figure 15.8: Using the FlowLayout manager to generate a GUI containing six buttons.

■

GridLayout

The GridLayout manager lays out components in a grid with a set number of rows and columns. As with other layout managers, GridLayout can be applied to frames and panels.

```
JFrame frame = new JFrame (" Grid Layout ");
frame.setLayout ( new GridLayout (3, 2, 5, 5));
```

This snippet creates a frame named frame and sets its layout manager to GridLayout. The first two arguments to GridLayout give the number of rows and columns, respectively. The last two arguments give the horizontal and vertical gaps between the neighboring cells in the grid. In this example the frame will contain a total of six cells organized into three rows with two columns.

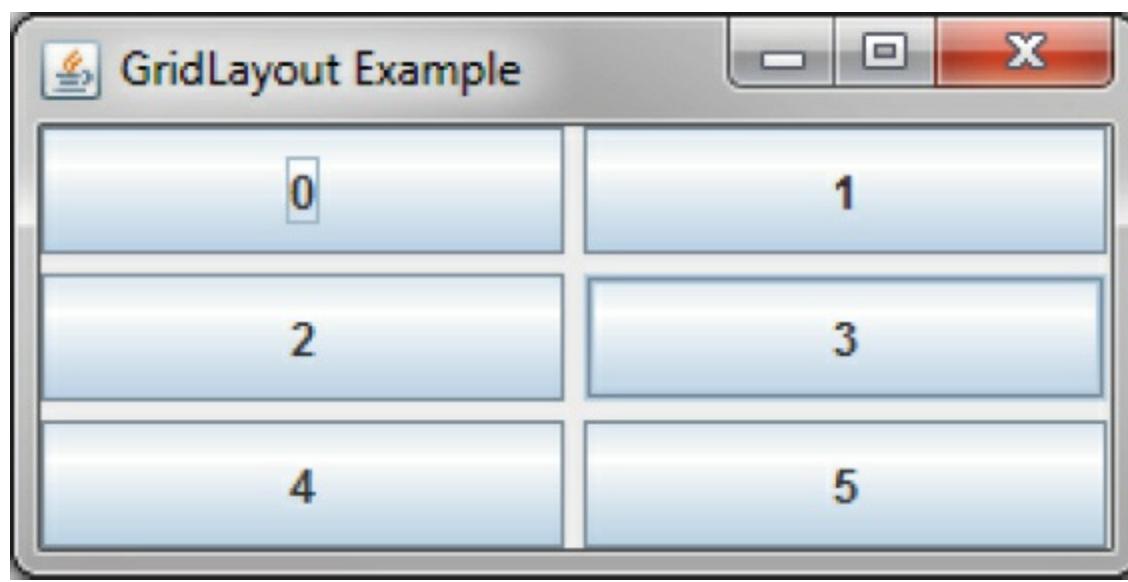


Figure 15.9: A 3×2 grid layout containing six buttons using left to right orientation.

[Figure 15.9](#) shows how frame will look after six buttons, labeled 0 through 5, have been added to it in that order. A key feature of using a GridLayout is that all cells in the grid will be the same size and will stretch to fill the entire container. Also note the equal spacing between the neighboring cells. It is possible to add more or fewer cells than specified in the GridLayout constructor, but the layout manager will be forced to guess at your intentions.

Exercise 15.20

Example 15.11: Animal identifier

We can write a program to displays pictures of animals and identify which animal the mouse is current hovering over. The animal's name will be displayed in the title of the frame. [Figure 15.10](#) shows this GUI.

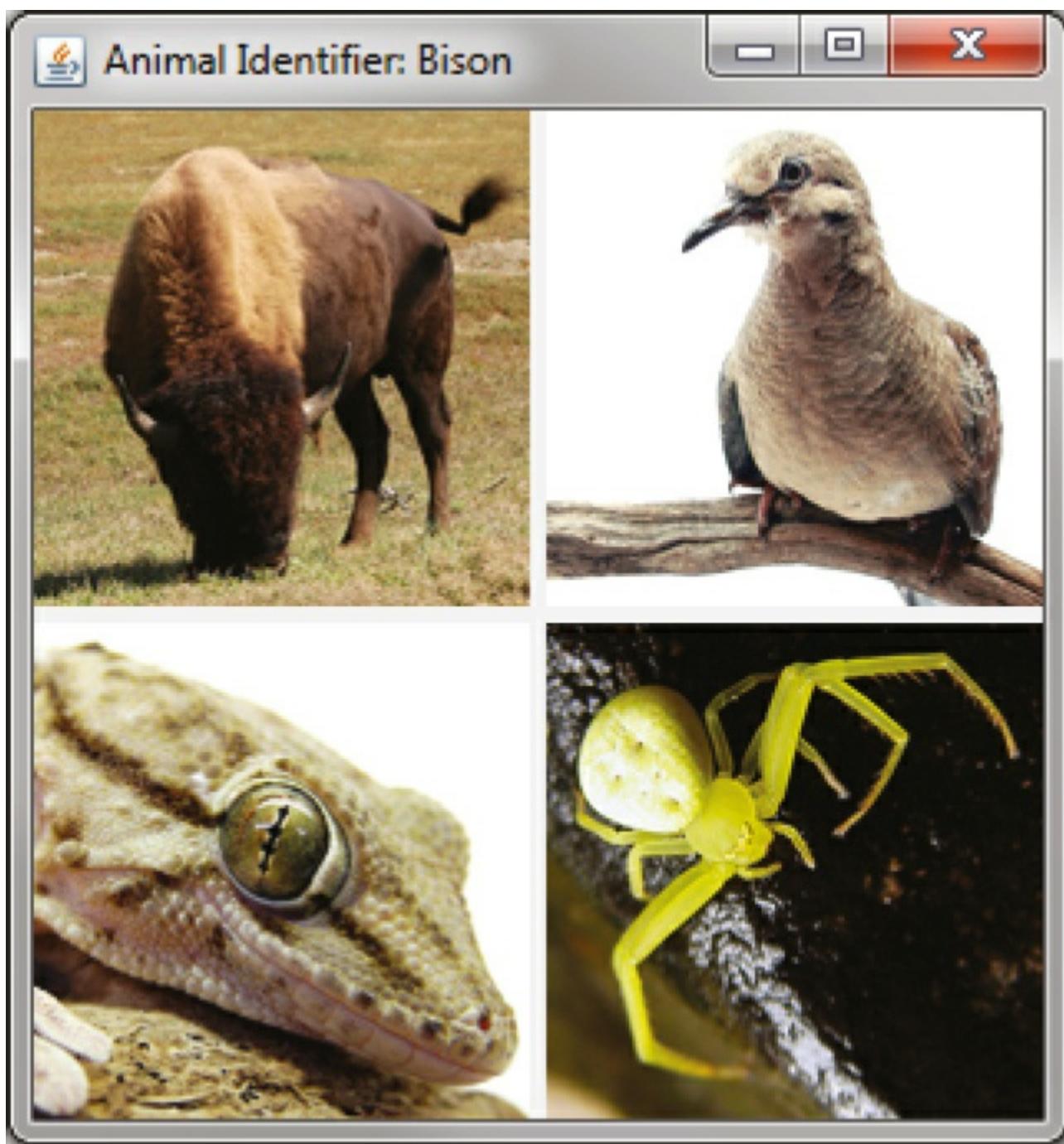


Figure 15.10: A GUI for an animal identifier. Note that the bison is identified according to the frame title.

Program 15.11 creates the GUI shown in Figure 15.10. Class AnimalIdentifier declares four labels and a frame as fields. The labels are named bison, dove, gecko, and spider.

Program 15.11: A program that identifies which animal is in various images when the mouse hovers over the image. Images are laid out using a GridLayout manager. (AnimalIdentifier.java)

```
1 import javax.swing.*;  
2 import java.awt.*;  
3 import java.awt.event.*;  
4  
5 public class AnimalIdentifier implements MouseListener {
```

```
6  JLabel bison, dove, gecko, spider ;
7  JFrame frame = new JFrame (" Animal Identifier : Bison ");
8
9  public AnimalIdentifier () {
10    JPanel panel = new JPanel ( new GridLayout (2 ,2 ,5 ,5));
11    ImageIcon bisonIcon = new ImageIcon (" pictures / bison.jpg ");
12    ImageIcon doveIcon = new ImageIcon (" pictures / dove.jpg ");
13    ImageIcon geckoIcon = new ImageIcon (" pictures / gecko.jpg ");
14    ImageIcon spiderIcon = new ImageIcon (" pictures / spider.jpg ");
15    bison = new JLabel ( bisonIcon );
16    bison.addMouseListener ( this );
17    dove = new JLabel ( doveIcon );
18    dove.addMouseListener ( this );
19    gecko = new JLabel ( geckoIcon );
20    gecko.addMouseListener ( this );
21    spider = new JLabel ( spiderIcon );
22    spider.addMouseListener ( this );
23    // Add labels
24    panel.add ( bison );
25    panel.add ( dove );
26    panel.add ( gecko );
27    panel.add ( spider );
28    frame.add ( panel );
29    frame.setSize (400 ,400) ;
30    frame.pack ();
31    frame.setDefaultCloseOperation (
32      JFrame.DISPOSE_ON_CLOSE );
33    frame.setVisible ( true );
34  }
35  // Implement all abstract methods in MouseListener
36  public void mouseEntered ( MouseEvent e ) {
37    Object label = e.getSource ();
38    if ( label == bison )
39      frame.setTitle (" Animal Identifier : Bison ");
40    else if ( label == dove )
41      frame.setTitle (" Animal Identifier : Dove ");
42    else if ( label == gecko )
43      frame.setTitle (" Animal Identifier : Gecko ");
44    else if ( label == spider )
45      frame.setTitle (" Animal Identifier : Spider ");
46  }
47
48  public void mouseExited ( MouseEvent e ) {
```

```

49     System.out.println (" Mouse exited.");
50 }
51
52 public void mousePressed ( MouseEvent e) {
53     System.out.println (" Mouse pressed.");
54 }
55
56 public void mouseReleased ( MouseEvent e) {
57     System.out.println (" Mouse released ");
58 }
59
60 public void mouseClicked ( MouseEvent e) {
61     System.out.println (" Mouse clicked.");
62 }
63
64 public static void main ( String [] args ) {
65     new AnimalIdentifier ();
66 }
67 }
```

Inside the constructor, we create a panel with a 2×2 GridLayout. Then, we create four image icons, one to decorate each label. Next, starting at [line 16](#), each of the four labels is created with its respective icon. We also add a MouseListener to each label. Starting at [line 24](#), the buttons are added to the panel, and the panel is added to the frame. The last few lines set the size of the frame, pack it, set its close operation, and make it visible.

Since AnimalIdentifier implements MouseListener, we need to define all the methods in the MouseListener interface. These are implemented starting at [line 36](#). As we are only interested in displaying the name of a picture when the mouse moves over it, the mouseEntered() method is where most of our even handling code will be. At [line 37](#) we get the label the mouse entered. We compare this label to the four labels and change the frame title correspondingly.

We implement the remaining methods from MouseListener to report the event on the console. The main() method creates a new instance of class AnimalIdentifier, launching the GUI.

Exercise 15.13

Play with [Program 15.11](#). What happens when you resize the window? ■

BorderLayout

The BorderLayout manager is the default one for a JFrame. It allows components to be laid out spatially in regions of a container. These regions are north, south, east, west, and center. This layout is intuitively easy to understand, but it is difficult to describe precisely.

You can only add one component to each region of the layout, and adding a component to any region is optional. The regions will stretch or shrink to accommodate the components inside. The north and south regions will only be as tall as needed to hold their contents, but their width will

stretch as wide as the entire container. The east and west regions will only be as wide as needed to hold their contents, but their height will stretch as tall as needed to fit the remaining height of the container. Both the height and the width of the center region will stretch as big as it needs to fill the container.

Example 15.12: BorderLayout

Here is an example of a frame using BorderLayout. Five buttons have been added, one to each region using the program shown below.

Program 15.12: Program showing buttons laid out in each of the five regions of a BorderLayout.
(BorderLayoutExample.java)

```
1 import javax.swing.*;
2 import java.awt.*;
3
4 public class BorderLayoutExample {
5     public static void main ( String [] args ) {
6         JFrame frame = new JFrame (" BorderLayout Example ");
7         JPanel panel = new JPanel ( new BorderLayout () );
8         panel.add ( new JButton (" North "), BorderLayout.NORTH );
9         panel.add ( new JButton (" South "), BorderLayout.SOUTH );
10        panel.add ( new JButton (" East "), BorderLayout.EAST );
11        panel.add ( new JButton (" West "), BorderLayout.WEST );
12        panel.add ( new JButton (" Center "), BorderLayout.CENTER );
13        frame.add ( panel );
14        frame.setSize (300 ,200) ;
15        frame.setDefaultCloseOperation ( JFrame.DISPOSE_ON_CLOSE );
16        frame.setVisible ( true );
17    }
18 }
```

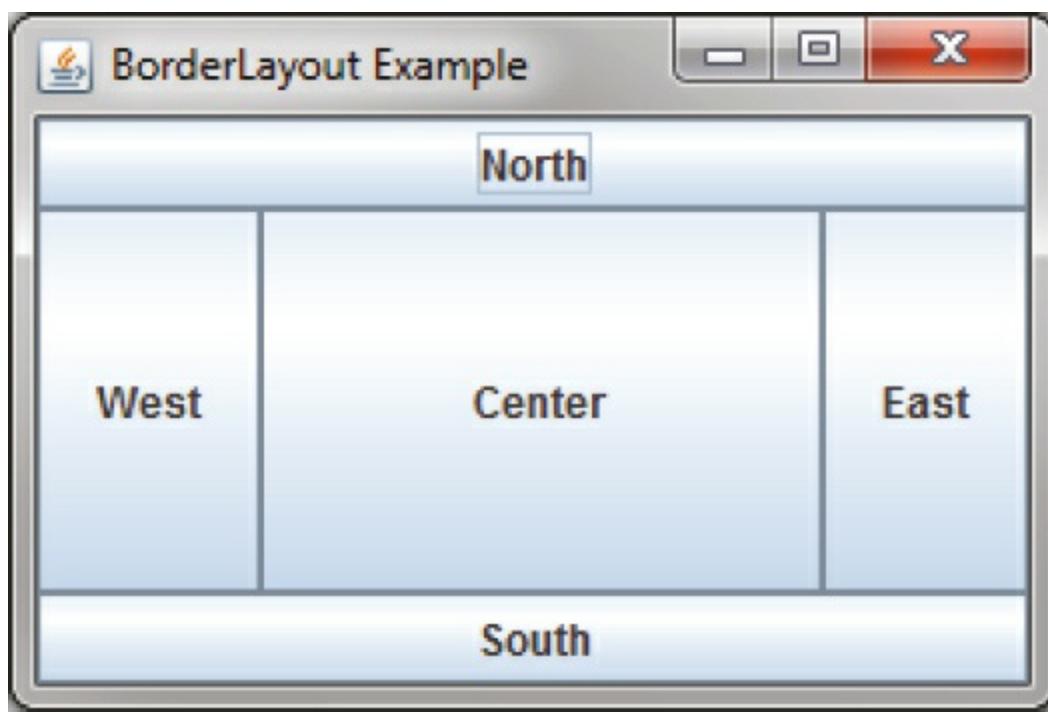


Figure 15.11: A GUI demonstrating BorderLayout generated by [Program 15.12](#).

Unlike FlowLayout or GridLayout, the location must be specified to add a component to a BorderLayout. (There is an overloaded add() method to add a component to specific grid cell in a GridLayout, but it is not necessary to use it.) With BorderLayout, the add() method uses a second parameter, which is BorderLayout.NORTH, BorderLayout.SOUTH, BorderLayout.EAST, BorderLayout.WEST, or BorderLayout.CENTER, depending on where you want to add the component. If you do not specify a second parameter, the component will be added to the center region. Since only one component can be in each region, adding a component to a region that is already occupied will replace the old component with the new one. ■

At first glance, BorderLayout seems that it would rarely be useful. However, this layout is commonly used because it can be used to establish a spatial relationship between different parts of a GUI. A container with a BorderLayout generally has other containers with their own layouts added to its regions, as shown in the following example.

Example 15.13: Calculator layout

We can make a GUI application that functions as a simple calculator. The calculator has the ten digits 0-9, a plus button, a minus button, and an enter button. At the top is a display that shows the current value.

We create ten JButton objects for the digits and three more JButton objects for plus, minus, and enter. The display is a JLabel. The code is given below.

Program 15.13: Program to layout a simple calculator. ([CalculatorLayout.java](#))

```
1 import javax.swing.*;  
2 import java.awt.*;  
3  
4 public class CalculatorLayout {
```

```

5 public static void main ( String [] args ) {
6     JFrame frame = new JFrame (" Calculator Layout ");
7     JPanel panel = new JPanel ( new BorderLayout () );
8     JPanel numbers = new JPanel (new GridLayout (4 ,3));
9     numbers.add ( new JButton ("7"));
10    numbers.add ( new JButton ("8"));
11    numbers.add ( new JButton ("9"));
12    numbers.add ( new JButton ("4"));
13    numbers.add ( new JButton ("5"));
14    numbers.add ( new JButton ("6"));
15    numbers.add ( new JButton ("1"));
16    numbers.add ( new JButton ("2"));
17    numbers.add ( new JButton ("3"));
18    numbers.add ( new JButton ("0"));
19    numbers.add ( new JButton ("+"));
20    numbers.add ( new JButton ("-"));
21    panel.add ( numbers, BorderLayout.CENTER );
22    JButton enter = new JButton (" Enter ");
23    panel.add (enter, BorderLayout.SOUTH );
24    JLabel display = new JLabel ("0");
25    panel.add ( display, BorderLayout.NORTH );
26    frame.add ( panel );
27    frame.setSize (250 ,300) ;
28    frame.setDefaultCloseOperation ( JFrame.DISPOSE_ON_CLOSE );
29    frame.setVisible ( true );
30 }
31 }
```

This program uses a BorderLayout to put a container with a GridLayout in a spatial arrangement with other components. First, we put the ten digit buttons in a panel with a GridLayout having 4 rows and 3 columns. We put the plus button and the minus button in the remaining two cells of the grid. We add this grid panel to the larger panel in the center region. We put the enter button in the south region and the display in the north region. ■



Figure 15.12: GUI generated by [Program 15.13](#).

There is no limit to how deeply you can nest containers within each other. Sometimes you must create a very complex GUI using many BorderLayout managers to achieve the appearance you want.

The three layout managers discussed in this section are the simplest, but there are others. The BoxLayout manager is a useful tool for laying out components in a stack or in a row. The GridBagLayout manager can be used to create complex layouts in a single container using a grid-based framework that is much more flexible than GridLayout, but the complexity of programming GridBagLayout is significant. The SpringLayout and GroupLayout managers are also powerful, but they are designed for use with a GUI builder utility.

15.3.6 Menus

Menus provide a useful form of interface that is expected from most GUI applications. In this section we show how to create menus and respond to the selection of menu items. Menus are placed on a menu bar. Each menu usually consists of several menu items that could be selected by the user. In addition to simple text, a menu item can be a button, a radio button, check box, or an icon. A menu can have one or more sub-menus opening out of a menu item.

Creating menus

First we have to create a menu bar.

```
JMenuBar bar = new JMenuBar();
```

This statement creates an object of type JMenuBar named bar which can hold menus. A JFrame has only one menu bar. We can create several menus and add them to the menu bar.

```
JMenu type = new JMenu (" Type ");
JMenu operations = new JMenu (" Operations ");
bar.add ( type );
bar.add ( operations );
```

These statements create two menus named type and operations labeled “Type” and “Operations,” respectively. The two menus are added to the existing menu bar using the add() method. Menus can be populated with menu items as follows.

```
JMenuItem addition = new JMenuItem (" Addition ");
JMenuItem subtraction = new JMenuItem (" Subtraction ")
operation.add ( addition );
operation.add ( subtraction );
```

These statements create two menu items named addition and subtraction. These menu items are then added to the menu operations. After having created a menu bar together with its menus and their respective menu items, we need to add it to a frame.

```
JFrame frame = new JFrame (" Menu Example ");
frame.setJMenuBar ( bar );
```

These statements create a frame and set its menu bar to bar. It is possible to use the add() method instead of the setJMenuBar() method to add a JMenuBar to a JFrame. However, doing so will add the JMenuBar to the regular content area, not to the menu area.

Exercise 15.4

Exercise 15.21

Sometimes you might need to disable a menu item and enable it only under certain conditions.

```
JMenuItem subtraction = new JMenuItem (" Subtraction ");
subtraction.setEnabled ( false );
```

These statements create a menu item named subtraction and disable it. A disabled menu item shows as a gray item and does not respond to attempts to select it. Note that JButton objects and many other widgets can be disabled in the same way.

Adding events to menus

An action listener can be added to each menu item, just like a JButton. Then, when a menu item is

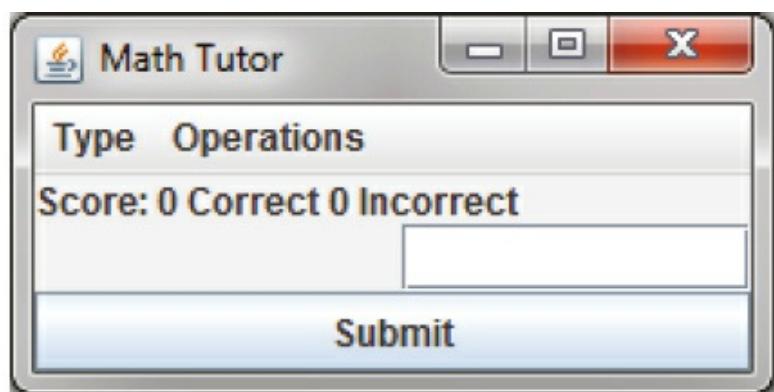
clicked by the user, an action event is generated. A JCheckBoxMenuItem object can be added to a JMenu as well. This object will have a check box which can be selected or unselected. A regular JMenuItem object generates an ActionEvent which is handled by an ActionListener. A JCheckBoxMenuItem, however, generates an ItemEvent handled by an ItemListener. Here are examples of both situations.

```
JMenuItem subtraction = new JMenuItem (" Subtraction ");
JCheckBoxMenuItem checkBox = new JCheckBoxMenuItem (" Check yourself " );
subtraction.addActionListener ( this );
checkbox.addItemListener ( this );
```

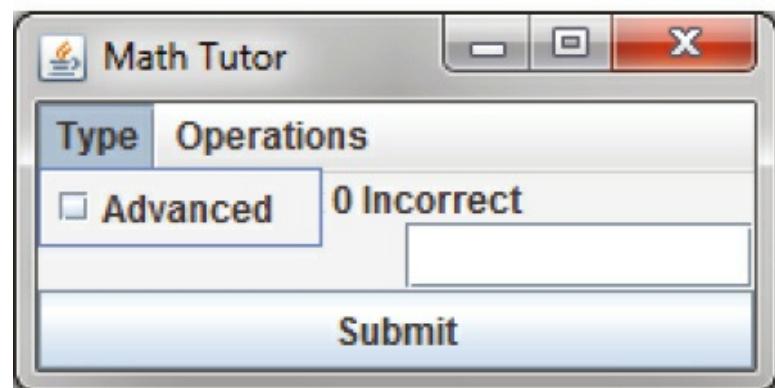
A JMenuItem works just like a JButton. In fact, the same action listener code could handle events for both buttons and menu items. However, a JCheckBoxMenuItem generates an ItemEvent (like many other check box and list widgets that we do not discuss) when the state of its check box changes. Thus, when you select a check box, a sign appears to its left and an ItemEvent is generated. When you select an already checked check box, the sign disappears and another ItemEvent is generated. Although an ActionEvent and an ItemEvent are very similar, Java differentiates between them because an ItemEvent has more information: By using the getStateChange() method, it is possible to tell whether the widget that fired the ItemEvent is now selected or deselected.

Example 15.14: Math tutor

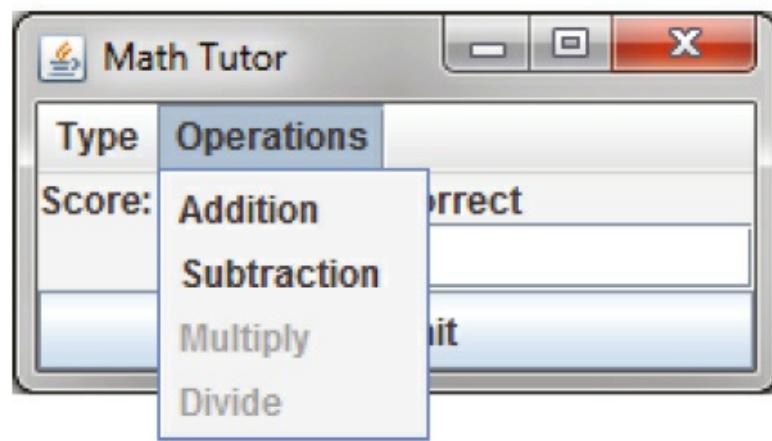
Here we give the code to generate the GUI shown in [Figure 15.13](#) which displays basic (addition and subtraction) as well as advanced (multiplication and division) problems. In [Section 15.3.7](#) we transform this GUI into an applet like the one described in [Section 15.1](#).



(a)



(b)



(c)

Figure 15.13: GUI for the MathTutor. (a) No menu selected. (b) Type selected. (c) Operations selected.

As shown in Figure 15.13, this GUI has a menu bar consisting of two menus labeled “Type” and “Operations.” The “Type” menu contains a check box labeled “Advanced” while the “Operations” menu contains four menu items labeled “Add,” “Subtract,” “Multiply,” and “Divide.” Note that the “Multiply” and “Divide” menu items are disabled. They will be enabled when the user selects the “Advanced” check box.

Before we introduce the program that creates this GUI, we need a helper class called ProblemGenerator that can randomly generate arithmetic problems. The class is designed so that the answers are always positive integers.

Program 15.14: Utility class to generate random addition, subtraction, multiplication, and division problems. (ProblemGenerator.java)

```
1 import java.util.Random;  
2 import javax.swing.*;  
3  
4 public class ProblemGenerator {  
5     public static Random random = new Random();  
6 }
```

```

7  public static int addPractice ( JLabel label ) {
8      int a = random.nextInt ( 12 ) + 1;
9      int b = random.nextInt ( 12 ) + 1;
10     label.setText ( a + " + " + b + " = " );
11     return a + b;
12 }
13
14 public static int subtractPractice ( JLabel label ) {
15     int a = random.nextInt ( 12 ) + 1;
16     int b = a + random.nextInt ( 12 ) + 1;
17     label.setText ( b + " - " + a + " = " );
18     return b - a;
19 }
20
21 public static int multiplyPractice ( JLabel label ) {
22     int a = random.nextInt ( 12 ) + 1;
23     int b = random.nextInt ( 12 ) + 1;
24     label.setText ( a + "\u00D7" + b + " = " );
25     return a * b;
26 }
27
28 public static int dividePractice ( JLabel label ) {
29     int a = random.nextInt ( 12 ) + 1;
30     int b = a*( random.nextInt ( 12 ) + 1 );
31     label.setText ( b + "\u00F7" + a + " = " );
32     return b / a;
33 }
34 }
```

The code listed above has four static methods `addPractice()`, `subtractPractice()`, `multiplyPractice()`, and `dividePractice()`. Each method generates an appropriate math problem, sets an input `JLabel` to display the problem, and returns the solution. Note that `\u00D7` and `\u00F7` are the Unicode values for the multiplication and division symbols.

[Program 15.15](#) generates the GUI in [Figure 15.13](#). Class `MathTutor` begins by creating only the objects that need to interact with event handlers: four menu items, a label, and a text field.

Program 15.15: Program that uses menus to generate math problems. (`MathTutor.java`)

```

1 import javax.swing.*;
2 import java.awt.*;
3 import java.awt.event.*;
4
5 public class MathTutor implements ActionListener, ItemListener {
6     private JMenuItem add = new JMenuItem (" Addition ");
7
8     public void actionPerformed ( ActionEvent e ) {
9         if ( e.getSource () == add ) {
10            label.setText ( addPractice ( label ) );
11        }
12    }
13
14     public void itemStateChanged ( ItemEvent e ) {
15         if ( e.getSource () == multiply ) {
16             label.setText ( multiplyPractice ( label ) );
17         }
18     }
19
20     public void divide ( ) {
21         label.setText ( dividePractice ( label ) );
22     }
23
24     public void subtract ( ) {
25         label.setText ( subtractPractice ( label ) );
26     }
27
28     public void add ( ) {
29         label.setText ( addPractice ( label ) );
30     }
31 }
```

```
7  private JMenuItem subtract = new JMenuItem (" Subtraction ");
8  private JMenuItem multiply = new JMenuItem (" Multiply ");
9  private JMenuItem divide = new JMenuItem (" Divide ");
10 private JLabel score =
11     new JLabel (" Score : 0 Correct 0 Incorrect ");
12 private JLabel label = new JLabel ();
13 private JTextField field = new JTextField (10) ;
14 private JButton submit = new JButton (" Submit ");
15
16 public MathTutor () {
17     JFrame frame = new JFrame (" Math Tutor ");
18     JMenuBar menuBar = new JMenuBar ();
19     JMenu type = new JMenu (" Type ");
20     JMenu operations = new JMenu (" Operations ");
21     JCheckBoxMenuItem advanced =
22         new JCheckBoxMenuItem (" Advanced ");
23     // Add ActionListener to menu items
24     add.addActionListener ( this );
25     subtract.addActionListener ( this );
26     multiply.addActionListener ( this );
27     divide.addActionListener ( this );
28     // Add ItemListener to checkbox menu item
29     advanced.addItemListener ( this );
30     // Fill menu for problem type
31     type.add ( advanced );
32     // Fill menu for operations
33     operations.add( add );
34     operations.add( subtract );
35     operations.add( multiply );
36     operations.add( divide );
37     // Disable advanced operations
38     multiply.setEnabled ( false );
39     divide.setEnabled ( false );
40     // Fill menu bar set on frame
41     menuBar.add ( type );
42     menuBar.add ( operations );
43     frame.setJMenuBar ( menuBar );
44     // Add widgets to frame and display GUI
45     frame.add ( score, BorderLayout.NORTH );
46     frame.add ( label, BorderLayout.WEST );
47     frame.add ( field, BorderLayout.EAST );
48     frame.add ( submit, BorderLayout.SOUTH );
49     frame.setSize (250, 125) ;
```

```

50     frame.setDefaultCloseOperation ( JFrame.DISPOSE_ON_CLOSE );
51     frame.setVisible ( true );
52 }
53
54 public void itemStateChanged ( ItemEvent e) {
55     if(e.getStateChange () == ItemEvent.SELECTED ) {
56         add.setEnabled ( false );
57         subtract.setEnabled ( false );
58         multiply.setEnabled ( true );
59         divide.setEnabled ( true );
60     }
61     else {
62         add.setEnabled ( true );
63         subtract.setEnabled ( true );
64         multiply.setEnabled ( false );
65         divide.setEnabled ( false );
66     }
67 }
68
69 public void actionPerformed ( ActionEvent e){
70     Object menuItem = e.getSource ();
71     if( menuItem == add )
72         ProblemGenerator.addPractice ( label );
73     else if( menuItem == subtract )
74         ProblemGenerator.subtractPractice ( label );
75     else if( menuItem == multiply )
76         ProblemGenerator.multiplyPractice ( label );
77     else if( menuItem == divide )
78         ProblemGenerator.dividePractice ( label );
79     field.setText ("");
80 }
81
82 public static void main ( String [] args ){
83     new MathTutor ();
84 }
85 }
```

Inside the MathTutor constructor, the frame and remaining widgets are created. Action listeners are added to the four operations menu items. An item listener is added to the “Advanced” check box menu item, because it requires a different kind of event handler. Note that MathTutor implements both the ActionListener and ItemListener interfaces, allowing it to handle both kinds of events.

Starting at [line 31](#), the two menus are populated with their respective menu items. At [line 38](#) the multiply and divide menu items are disabled because the application starts in basic mode. The menus

are then added to the menu bar, and the menu bar is set on the frame. Finally, the label and text field are added to the frame, which is made visible. Note that these two widgets are added directly to the frame with the parameters `BorderLayout.NORTH` and `BorderLayout.SOUTH`. `JFrame` objects use the `BorderLayout` manager by default.

The `itemStateChanged()` method enables the multiply and divide menus and disables the add and subtract menus if the advanced check box is selected and does the reverse if it has been unselected.

The `actionPerformed()` method is similar to earlier examples. Depending on which menu item fired the event, the appropriate static methods from the `ProblemGenerator` class are used to display a math problem on `label`.

The `main()` method creates an instance of `MathTutor`, initiating GUI construction. ■

15.3.7 Applets

An applet is a Java program that is written to run inside a web browser. A normal Java application is stored on disk and runs on the command line or as a window on your desktop (or both), but an applet exists inside of a webpage. This means that the code that launches an applet is usually embedded in HTML code.

Almost all of the GUI tools we discuss are useful for writing an applet. For example, you can write an applet that has menus, buttons, text fields, and any other widget that Swing offers. You can also add listeners to various objects for the applet to respond to events.

A Swing applet is derived from the `JApplet` class, which is a subclass of the older AWT class `Applet`.

An applet differs from a stand-alone Java application in several important ways. The first is that applets are sent across the Internet. Unless you have increased security settings on your browser, they will generally run as soon as you visit a page with an applet embedded in it, without even asking your permission. This poses a huge security risk, but the designers of Java have worked to protect us. Unsigned applets (which are the ones we will talk about creating) cannot read, write, or execute files, make network connections to servers other than the one they come from, or interact with most other local system settings. However, applets loaded from a local directory (instead of downloaded over the Internet) do not have these restrictions. Be sure to remember these restrictions when you design applets for distribution over the Internet.

Another important and concrete difference from normal Java classes is that an applet should not define a constructor. Instead, an applet uses the `init()` method. When the applet is executed, often through a browser, its `init()` method is called first. You can use this method to set up the applet GUI by adding various widgets and event listeners.

Rainbows

Have the colors ROYGBIV.

Figure 15.14: A simple applet containing one button.

Example 15.15: RainbowApplet

We can create a simple applet with a button and a label. Clicking this button sets the text on the label to contain some information about the colors in a rainbow. This applet is shown in [Figure 15.14](#). [Program 15.16](#) defines the applet.

Program 15.16: An applet with a button and a label. (RainbowApplet.java)

```
1 import javax.swing.*;
2 import java.awt.*;
3 import java.awt.event.*;
4
5 public class RainbowApplet extends JApplet {
6     private JLabel label = new JLabel();
7
8     public void init(){
9         JButton button = new JButton(" Rainbows ");
10        add(button, BorderLayout.NORTH);
11        add(label, BorderLayout.SOUTH);
12        button.addActionListener(new ActionListener(){
13            public void actionPerformed(ActionEvent e){
14                label.setText(" Have the colors ROYGBIV.");
15            }
16        });
17    }
18 }
```

Note that class RainbowApplet extends JApplet, as will most applets you write. The label is declared as a field, so that the event handler can access it. The init() method creates a JButton with the label “Rainbows,” adds both the label and the button to the applet, and adds an action listener to the button with an anonymous inner class event handler.

Note that the button and the label were added to the applet just as it would have been done inside of a JFrame constructor. They were added with the BorderLayout.WEST and BorderLayout.EAST constants because a JApplet uses a BorderLayout manager by default. ■

As you can see, creating an applet is very similar to creating a GUI based on a JFrame. In fact, it may be easier. However, we need to create an HTML file for the applet to be accessible via a browser. HTML stands for *hypertext markup language*. HTML is the core language for writing webpages. HTML is designed for marking up text and images to be displayed within a browser. It is not as complicated as Java, but we do not have time or space to describe the language deeply.

HTML is a language made up of tags that mark parts of a document with formatting instructions. All of the tags we will need to embed an applet will start with the name of the tag in angle brackets and then end with the same name in angle brackets but preceded by a slash. For example, to mark text as bold in HTML, you can use the tag as follows:

```
<strong>Here is text that will appear in a bold font.</strong>
```

To run RainbowApplet we need to create an HTML file that includes the applet's class file inside the <applet> tag.

Example 15.16: RainbowApplet HTML

We want to make RainbowApplet executable via a browser. In addition, we would like the source code of the applet to be linked from the same webpage. The following HTML code embeds the RainbowApplet and links to its source.

```
<html>
  <head>
    <title>Rainbow Applet</title>
  </head>
  <body>
    <applet code="RainbowApplet.class" width="200" height="100"></applet>
    <hr /><a href="RainbowApplet.java">The source.</a>
  </body>
</html>
```

An <html> tag encloses an entire HTML document. Inside is the <head> tag, followed by the <body> tag. The <head> tag contains information about the page such as its title, which is given in the <title> tag. The <body> tag contains the viewable content of the web page. In this case, there is an <applet> tag, a <hr/> tag, and an <a> tag in the body.

The <applet> tag is used to embed our applet. The name of the applet class must be given as the argument, in double quotes, to the code attribute. The width and height attributes specify the size of the applet on the screen. The <hr/> tag puts the horizontal line between the applet and the link. It is the only tag in our example that does not have both an opening and closing half. Finally, the <a> tag is used to make a clickable link. On a web page, it will display "The source," but it will navigate to the file RainbowApplet.java when clicked.

Put this code in an HTML file named RainbowApplet.html. (Note that the name of your HTML fil

should be descriptive, but it does not have to match the name of the applet class.) If your browser is Java compatible and has the appropriate plug-in installed, you should be able to run the applet by double-clicking on the HTML file. Of course, if you are able to upload the HTML file and the class file to a web server, you can view the applet and share it with anyone with an Internet connection. ■

Exercise 15.22

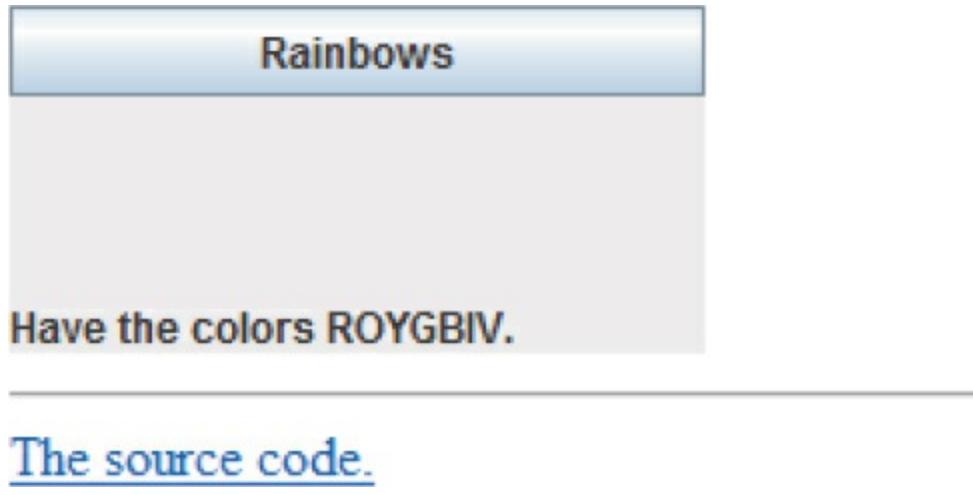


Figure 15.15: The RainbowApplet as seen through a web browser. Clicking on the link displays the Java source code of the applet in [Program 15.16](#).

Applets can be much more complex than RainbowApplet. We now solve the problem posed at the beginning of the chapter with an applet with a fully functional GUI designed to be an arithmetic tutor.

15.4 Solution: Math applet

The applet that meets the specification given at the beginning of the chapter is similar to [Program 15.15](#). Modifications must be made to convert that program into an applet and add full functionality to the event handlers. We now go through this rather long Java program step by step. GUI programs in Java tend to be longer than their command line versions because of the code needed to set up all the widgets.



Figure 15.16: The Type (a) and Operations (b) menus in the MathTutorApplet. Note that the “Multiply” and “Divide” operations are disabled because the “Advanced” option is not checked.

The MathTutorApplet.java file begins with the usual Swing imports and the class declaration

followed by a list of its fields.

```
1 import javax.swing.*;
2 import java.awt.*;
3 import java.awt.event.*;
4
5 public class MathTutorApplet extends JApplet
6     implements ActionListener, ItemListener {
7     private JMenuItem add = new JMenuItem(" Addition ");
8     private JMenuItem subtract = new JMenuItem(" Subtraction ");
9     private JMenuItem multiply = new JMenuItem(" Multiply ");
10    private JMenuItem divide = new JMenuItem(" Divide ");
11    private JLabel score =
12        new JLabel(" Score : 0 Correct 0 Incorrect ");
13    private JLabel label = new JLabel();
14    private JTextField field = new JTextField(10);
15    private JButton submit = new JButton(" Submit ");
16    private int correct = 0;
17    private int incorrect = 0;
18    private int answer = -1;
```

Note that the `int` variables `correct`, `incorrect`, and `answer` have been added to the fields to keep track of the correct and incorrect answers and the current answer that the user is trying to find.

Following the field declarations is the definition of the `init()` method. As you can see, this method is almost identical to the constructor in the `MathTutor` stand-alone application. Indeed, the `init()` method is very much like the constructor for an applet.

```
20    public void init () {
21        JMenuBar menuBar = new JMenuBar();
22        JMenu type = new JMenu(" Type ");
23        JMenu operations = new JMenu(" Operations ");
24        JCheckBoxMenuItem advanced =
25            new JCheckBoxMenuItem(" Advanced ");
26        // Add ActionListener to menu items and buttons
27        add.addActionListener( this );
28        subtract.addActionListener( this );
29        multiply.addActionListener( this );
30        divide.addActionListener( this );
31        submit.addActionListener( this );
32        // Add ItemListener to checkbox menu item
33        advanced.addItemListener( this );
34        // Fill menu for problem type
35        type.add( advanced );
36        // Fill menu for operations
```

```

37     operations.add( add );
38     operations.add( subtract );
39     operations.add( multiply );
40     operations.add( divide );
41     // Disable advanced operations and submit
42     multiply.setEnabled( false );
43     divide.setEnabled( false );
44     submit.setEnabled( false );
45     // Fill menu bar and set on applet
46     menuBar.add( type );
47     menuBar.add( operations );
48     setJMenuBar( menuBar );
49     // Add widgets to applet content
50     add( score, BorderLayout.NORTH );
51     add( label, BorderLayout.WEST );
52     add( field, BorderLayout.EAST );
53     add( submit, BorderLayout.SOUTH );
54 }

```

Just like the MathTutor constructor, this method creates the menu bar, the menus, and the menu items. Then, it adds action listeners to the menu items and the button and an item listener to the check box menu item. Next it adds advanced to the “Type” menu and the four operation menu items to the “Operations” menu. It disables the advanced menu items (since the applet starts in basic mode) and the button (since it is impossible to submit an answer before a problem has been given). Finally, it puts the menus on the menu bar, sets the menu bar on the applet, and adds the two text fields, the label, and the button to the applet content area using appropriate BorderLayout constants.

One big difference between the MathTutor constructor and the MathTutorApplet init() method is that the MathTutor constructor creates a JFrame object. In MathTutorApplet no JFrame is necessary because the class itself is a child of JApplet and thus is a GUI container.

```

56     public void itemStateChanged ( ItemEvent e ) {
57         if(e.getStateChange () == ItemEvent.SELECTED ) {
58             add.setEnabled ( false );
59             subtract.setEnabled ( false );
60             multiply.setEnabled ( true );
61             divide.setEnabled ( true );
62         }
63         else {
64             add.setEnabled ( true );
65             subtract.setEnabled ( true );
66             multiply.setEnabled ( false );
67             divide.setEnabled ( false );
68         }
69     }

```

The `itemStateChanged()` method is the same as its counterpart in `MathTutor`. If the state change is `ItemEvent.SELECTED`, we enable the advanced menus and disable the basic ones. Otherwise, we do the reverse. Note that this method would be more complicated if we were listening to more than one object. Since we are only listening to the advanced check box menu item, we know that it is what is being selected or deselected.

```
71     public void actionPerformed ( ActionEvent e) {  
72         Object object = e.getSource ();  
73         if( object == submit ) {  
74             int response = Integer.parseInt ( field.getText () );  
75             if( response == answer )  
76                 correct ++;  
77             else  
78                 incorrect ++;  
79             label.setText ("");  
80             score.setText (" Score : " + correct + " Correct " +  
81                     incorrect + " Incorrect ");  
82             submit.setEnabled ( false );  
83         }  
84     else {  
85         if( object == add )  
86             answer = ProblemGenerator.addPractice ( label );  
87         else if( object == subtract )  
88             answer = ProblemGenerator.subtractPractice ( label );  
89         else if( object == multiply )  
90             answer = ProblemGenerator.multiplyPractice ( label );  
91         else if( object == divide )  
92             answer = ProblemGenerator.dividePractice ( label );  
93         submit.setEnabled ( true );  
94     }  
95     field.setText ("");  
96 }  
97 }
```

Finally, the `actionPerformed()` method deals with the menu item and button clicks. If `submit` was clicked, we parse the text in the text field to get an `int` value and compare it to the answer. Depending on its correctness, we update the correct and incorrect counts, clear the problem label, and update the score label. Finally, we disable the submit button so that the user can't submit an answer until another problem has been given.

The next part of the `actionPerformed()` method has the same functionality as the `actionPerformed()` method from `MathTutor`. It updates `label` to contain a randomly generated problem with an operation that matches whichever menu item was picked. It also saves the answer into the `answer` field so that we can check the user's response later. Then, it enables the submit button since there is a problem to answer. No matter what object triggered the action, the last line of the method clears the text field.

We need some HTML to test the applet. Here is a suitable HTML file similar to the RainbowApplet.html file presented before.

```
<html>
  <head>
    <title>Math Tutor Applet</title>
  </head>
  <body>
    <applet code="MathTutorApplet.class" width="250" height="125"></applet>
    <hr /><a href="MathTutorApplet.java">The source code.</a>
  </body>
</html>
```

In the stand-alone MathTutor application, we created a JFrame and set its size to 250×125 pixels. The width and height attributes in the HTML allow us to accomplish something similar. Note that the widgets in the applet have more room than in the JFrame version because the size of the JFrame includes the title bar and window borders.

The functionality of this applet is limited, but it still shows off menus, buttons, labels, text fields, applet creation, and two different kinds of event listeners.

15.5 Concurrency: GUIs

Stand-alone Java programs have at least one thread, the main thread. Applets have a similar thread that calls its init() and start() methods. Applications with GUIs (including applets) create additional threads to manage the GUI behind the scenes.

Although a GUI will create several threads, the most important for the programmer to worry about is called the *event dispatch thread* (EDT). This thread handles events like button clicks. When you write your actionPerformed() method, remember that the EDT is the one that will actually execute the code inside.

If you are writing a complex program, the EDT may interact with many other threads, and the synchronization issues discussed in [Chapter 14](#) will become important. However, only the EDT is allowed to change the state of widgets in a GUI. Using other threads to do so will work some of the time, but it is not thread-safe and violates the design of Swing.

15.5.1 Worker threads

Thread safety is not the most common multi-threaded GUI problem, however. Unresponsive GUIs can be found on almost every platform, as you have no doubt experienced. In Java, unresponsive GUIs usually happen because the programmer is using the event dispatch thread to do some task that takes too long. Because the EDT is responsible for updating the GUI, the GUI freezes, and the user has to wait.

This is quite a conundrum. On the one hand, the EDT is the only thread allowed to update widgets. On the other, it has to do its work quickly so that the GUI is responsive. The solution is to spawn *worker threads* to do the job. When they are done, they inform the EDT so that it is able to update the GUI.

Let's look at a GUI with two JButton widgets and two JLabel widgets. When one button is pressed the EDT goes to sleep for 5 seconds before displaying an answer on the first label (in this case, approximately $\sqrt{2}$). When the other button is pressed, it increments a counter and displays the value in the second label.

Program 15.17: A GUI that becomes unresponsive when the “Compute” button is pressed.
(UnresponsiveGUI.java)

```
1 import javax.swing.*;
2 import java.awt.*;
3 import java.awt.event.*;
4
5 public class UnresponsiveGUI extends JFrame
6     implements ActionListener {
7     JLabel label = new JLabel(" Answer :");
8     JButton compute = new JButton(" Compute ");
9     JLabel counter = new JLabel("0");
10    JButton increment = new JButton(" Increment ");
11    int count = 0;
12
13    public static void main( String args [] ) {
14        UnresponsiveGUI frame = new UnresponsiveGUI ();
15        frame.setDefaultCloseOperation ( JFrame.DISPOSE_ON_CLOSE );
16        frame.setSize ( 250 ,150 );
17        frame.setVisible ( true );
18    }
19
20    public UnresponsiveGUI () {
21        setLayout ( new GridLayout (4 ,1));
22        setTitle (" Unresponsive GUI ");
23        add ( label );
24        compute.addActionListener ( this );
25        add ( compute );
26        add ( counter );
27        increment.addActionListener ( this );
28        add ( increment );
29    }
30
31    public void actionPerformed ( ActionEvent e ) {
```

```

32     if( e.getSource () == compute ) {
33         label.setText (" Computing... ");
34         try {
35             Thread.sleep (5000) ;
36         } catch ( Exception ignore ) { }
37         label.setText (" Answer : " + Math.sqrt (2.0) );
38     }
39     else {
40         count ++;
41         counter.setText (" " + count );
42     }
43 }
44 }
```

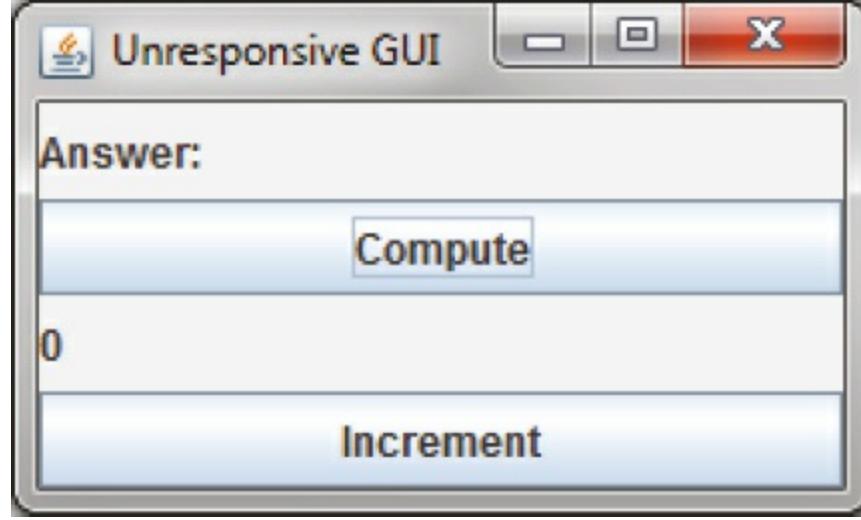


Figure 15.17: GUI generated by [Program 15.17](#).

If you click the “Compute” button, the GUI becomes unresponsive. Specifically, you cannot click on the “Increment” button, but you should still be able to move the frame around the desktop on most systems. Furthermore, some thread in the GUI is registering the clicks you do on the “Increment” button, but events triggered by those clicks are not handled until after the EDT wakes up. At that point, the counter will shoot up in value unpredictably.

[Exercise 15.23](#)

[Exercise 15.24](#)

One solution is to create an anonymous inner class that extends `SwingWorker`. The `SwingWorker` class is abstract, but it is also *generic*, meaning that it has type parameters (given in angle brackets) which specify what type of objects it interacts with. Generic classes are often containers like `LinkedList` where the type parameter says what kind of objects will be kept in the list. [Chapter 18](#) covers generics in some depth. The reason we need generics for `SwingWorker` is so that it can specify what kind of object it will return when it finishes its work. The first type parameter specifies the type that the worker will return when it completes its work. The second specifies the type that the worker will return periodically in the process of doing work (which can be useful for updating

progress bars). Examine the following program which has added a SwingWorker to its actionPerformed() method but is otherwise the same as [Program 15.17](#).

Program 15.18: A GUI that uses SwingWorker to avoid becoming unresponsive. (WorkerGUI.java)

```
1 import javax.swing.*;
2 import java.awt.*;
3 import java.awt.event.*;
4
5 public class WorkerGUI extends JFrame implements ActionListener {
6     JLabel label = new JLabel("Answer :");
7     JButton compute = new JButton("Compute");
8     JLabel counter = new JLabel("0");
9     JButton increment = new JButton("Increment");
10    int count = 0;
11
12    public static void main ( String args [] ) {
13        WorkerGUI frame = new WorkerGUI ();
14        frame.setDefaultCloseOperation ( JFrame.DISPOSE_ON_CLOSE );
15        frame.setSize ( 250 ,150) ;
16        frame.setVisible ( true );
17    }
18
19    public WorkerGUI () {
20        setLayout ( new GridLayout (4 ,1));
21        setTitle (" Worker GUI ");
22        add ( label );
23        compute.addActionListener ( this );
24        add ( compute );
25        add ( counter );
26        increment.addActionListener ( this );
27        add ( increment );
28    }
29
30    public void actionPerformed ( ActionEvent e ) {
31        if( e.getSource () == compute ) {
32            SwingWorker worker = new SwingWorker < String, Void > () {
33                public String doInBackground () {
34                    try {
35                        Thread.sleep (5000) ;
36                    } catch ( Exception ignore ) {}
37                    return " Answer :" + Math.sqrt (2.0) ;
38                }
39            }
```

```

40     public void done () {
41         try {
42             label.setText ( get ());
43         } catch ( Exception ignore ) {}
44     }
45 }
46
47     worker.execute ();
48     label.setText (" Computing... ");
49 }
50 else {
51     count++;
52     counter.setText (" " + count );
53 }
54 }
55 }
```

In this program, the first type parameter for the SwingWorker is String because we are going to set the text in a JLabel with its result. The second parameter is Void, meaning that we do not intend to return any values periodically. Most child classes of SwingWorker should override the doInBackground() and done() methods.

The doInBackground() method performs the time-consuming work that we want done on another thread. In our example, the “work” is going to sleep, but it will generally be some CPU or I/C intensive process. Afterwards, it returns the answer it found. The done() method is called automatically by the EDT after doInBackground() finishes. After the SwingWorker object has been created, the execute() method starts it working. The GUI will look identical to the unresponsive version (except for the title), but it will remain responsive.

Exercise 15.25

This syntax is not particularly elegant, but it accomplishes a complex task. It spawns a thread transparently, and then the EDT is given work when the thread gives back its answer. Using a SwingWorker is not always required, but it is a useful tool to have in your arsenal if you plan on writing industrial-strength GUIs.

15.6 Summary

In this chapter we introduce the basics of constructing a GUI to allow users to interact with an application. We show how to add widgets such as buttons and text boxes and use layout managers to organize their appearance. We show how you can add action listeners and other event handlers so that user actions can perform useful tasks. Finally, we show how these techniques can be applied to GUIs on stand-alone applications as well as applets that run inside of a web browser.

While Java offers a large variety of widgets and listeners, this introduction is limited to a few of the most commonly used. Once you understand the basics of GUI construction as described in this

chapter, it should be easy to understand the extensive Java tutorial at <http://download.oracle.com/javase/tutorial/uiswing/components/> or other reference sources.

Exercises

Conceptual Problems

- 15.1 In [Program 15.3](#), why have we declared the buttons and the text box to be **final**?
- 15.2 Note that both ActionListener and MouseListener interfaces can be used to process button clicks. Under which circumstances is ActionListener better? Under which is MouseListener better?
- 15.3 Why is the MouseAdapter abstract class useful?
- 15.4 What do you expect will happen if you used setJMenuBar() to set two different menu bars on a single JFrame object?
- 15.5 Describe the situations that the following event listeners are useful for: ActionListener, MouseListener, ItemListener, and KeyListener

Programming Practice

- 15.6 Remove the two instances of the keyword **final** from [Program 15.3](#) and try to compile it. Why does the compiler complain? What do you conclude regarding the use of **final** with respect to local variables used in the actionPerformed() method of an anonymous inner class? Why is this not a concern for top-level, named classes used as an ActionListener?
- 15.7 Write a program that creates a GUI containing two buttons labeled “Start” and “Done.” The GUI frame should be labeled “Start and Done.”
- 15.8 Modify [Program 15.5](#) by implementing the mouseExited(), mousePressed(), and mouseReleased() methods. Each method must display a suitable message in the text box when the corresponding event occurs. For example, when the mouse exits button one, the text box should display “Mouse exits One.”
- 15.9 Modify [Program 15.8](#) such that clicking the icon-decorated button generates a roaring sound. Note that this will require you to add an ActionListener to the button, create an audio clip for the desired sound, and then play this click when the button is clicked. Consider visiting <http://www.freesound.org/> for free sound files.
- 15.10 Write a Java program that creates a GUI containing a label with a picture of yourself.
- 15.11 Extend the SoundGame class developed in [Example 15.7](#) to include sounds for various animals. You may find a variety of publicly available sounds files for use in your program. The Freesound link above is only one source.

Note that some sites explicitly ask you not to embed the sound file URL into your program. In those cases, download the sound file into your local directory and load it from there into your application.

15.12 In [Example 15.7](#) we stopped both the chirp and the bark sounds because the action listener corresponding to the “Stop Playing” button does not know which sound is playing. Modify [Program 15.7](#) so that only the sound that is playing is stopped. You may need to declare another variable to keep track of which is playing.

15.13 Modify [Program 15.11](#) so that it plays a sound associated with an animal when the mouse is clicked over its label. Note that this is an example of a situation where a MouseListener can be used to listen for mouse click events though an ActionListener cannot. As in Exercises 15.11 and 15.9, you may need to download sounds from the Internet.

15.14 Modify the applet from [Section 15.4](#) to display the problem number the user is working on. The first problem is numbered “Problem 1” with subsequent problems 2, 3, and so on. The number should increase each time the user hits the Submit button. Find a suitable place on the GUI to display this information. You may need to add a panel to reorganize the GUI.

15.15 Remove the actionPerformed() and itemStateChanged() methods from the MathTutorApple class given in [Section 15.4](#). Move the code from these methods into individual anonymous inner classes added to the add, subtract, multiply, divide, submit, and advanced objects.

Note that you are now able to remove ActionListener and ItemListener from the list of implemented interfaces for the MathTutorApplet class. Reorganizing the code this way should have no impact on the functionality of the applet. Is doing so a good idea or not? Why?

15.16 In [Program 15.3](#) exchange the first two add() method calls on the soundPanel so that bark is added to the panel before the chirp. Explain how the appearance of the GUI is changed.

15.17 Modify [Program 15.2](#) by adding a second panel named secondPanel. Create a new button named train with the label “Train.” Add train to secondPanel. Now add secondPanel to soundCheck and look at the GUI generated. Can you explain why only one panel is visible?

15.18 Remove [line 13](#) from [Program 15.10](#). Run the modified program and resize the frame to various sizes. How does the placement of the buttons change?

15.19 Modify [Program 15.10](#) by deleting the lines that set the frame size. What does the resulting GUI look like? Now add the following code just before [line 13](#):

```
demo.pack();
```

When you run the modified program, what is the impact of using the pack() method?

15.20 Create a GUI with a frame that uses GridLayout and has a suitable size. Use a 3×2 layout with a horizontal and vertical spacing of 5 pixels each. Use a loop to add eight buttons to your frame, labeled 1 through 8. Observe how the frame expands to include all the buttons even though the initially specified GridLayout had only 6 cells. Depending on the frame size, you might have to resize the window to see all cells and buttons. What happens if you add fewer than 6?

15.21 Write a Java program that creates a GUI with a frame and a menu bar containing a single menu. Add a menu item to this menu. Use the add() method, not the setJMenuBar() method, to add the menu bar to the frame. What is the difference between using the add() method and the

setJMenuBar() method to add a menu bar to a JFrame?

15.22 Use the example from the book to create a suitable HTML file that embeds the RainbowApplet class from [Example 15.15](#) and links to the Java source code. Now upload the class file, the source code, and the HTML file to a web server so that you can run the applet over the Internet. Have a friend test out your code on a computer in another location to make sure that it works.

Experiments

Concurrency

15.23 Recall that [Program 15.17](#) is unresponsive because the event dispatch thread goes to sleep for 5 seconds (5,000 milliseconds) on [line 35](#). Experiment with this value to determine what is a reasonable amount of time for the EDT to be blocked before the GUI feels unresponsive.

Concurrency

15.24 [Program 15.17](#) is unrealistic because the EDT simply goes to sleep. Normally, a GUI becomes unresponsive because the EDT is performing extensive calculations or doing slow I/O operations. Replace [line 35](#) with a short loop that performs significant calculations. One simple way to spend a lot of computational time is by summing the sines of random numbers, similar to the work done in [Example 13.10](#). How many sines do you need to compute to make the GUI unresponsive for 5 seconds?

Concurrency

15.25 Take the computationally expensive loop from Exercise 15.24 and use it to replace [line 35](#) in [Program 15.18](#), the SwingWorker version of the program. Does the program become unresponsive if you run it? If possible, run the program on Windows, Mac, and Linux environments. If it is unresponsive in some environments but not others, why do you think that might be?

Chapter 16

Testing and Debugging

I never make stupid mistakes. Only very, very clever ones.

—John Peel

16.1 Fixing bugs

This chapter is about finding, fixing, and, more importantly, preventing bugs in software. This chapter is unique in that it is not based on clearly stated concrete problems with straightforward solutions. As it stands now, finding and fixing bugs in software, especially concurrent software, is a problem which no one has solved completely. The first half of this chapter will focus on common bugs and how to fix them. The second half will focus on design techniques for preventing bugs and testing to see if any hidden bugs remain in your code.

16.1.1 Common sequential bugs

We will begin with bugs commonly found in sequential programs, some of which have already been mentioned as common mistakes in previous chapters. We will not discuss syntax errors or any other errors which can be caught at compile time. Instead, all bugs discussed in this chapter are run-time bugs.

There are an infinite number of possible bugs and so they cannot all be listed. Below are a few of the most common categories of bugs to affect beginner (and occasionally veteran) programmers.

Precision Errors: Floating point numbers have limited precision inside of a computer. Programs that assume infinite precision may break when real results are slightly larger or smaller than expected.

Overflow and Underflow: In some sense the cousin of precision in floating point types is the limited range of values that integer types can take on. If the value goes too high, it wraps back around and becomes a negative number. If the values becomes too low, the opposite can happen.

Casting Errors: Casting can have many subtle effects in a program. Sometimes programmers divide two integers and forget that the result is an integer. Incorrectly casting objects can lead to ClassCastException.

Loop Errors: Loops are a favorite place for bugs to hide. Three of the most common loop mistakes are:

- Off-by-one errors: The loop executes one more or one fewer time than expected.
- Infinite loop: A classic. The loop continues executing until the program runs out of memory

or a user stops the program externally.

- Zero loop: The loop does not execute even once, even though the programmer expected it to.

Equivalence Testing Errors: If a programmer uses the `==` operator to compare two references, it will be true only if the two references point at the same object. Although doing so is occasionally necessary, the `equals()` method should usually be used to compare the attributes of the objects pointed at by the references.

Array Errors: Arrays are often involved with loop errors. Two problems specific to arrays are:

- Out of bounds: A math mistake or an off-by-one error could lead to trying to access an element of an array that comes before element 0 or after the last element.
- Uninitialized object arrays: Arrays of primitive data types can be created and used instantly; however, arrays of object types are filled with nulls until each element is assigned an object, often a new object.

Scope Errors: Some errors can be caused by a programmer's misunderstanding about the visibility of a variable.

- Shadowing variables: If a local variable is declared with the same name as a member or class variable, changes within a method will be made to that local variable. This principle is straightforward, but, if a programmer does not notice the shadow declaration, the behavior of the code can be very confusing.
- Reference vs. value: When primitive data is passed as an argument into a method, its value is copied into the method, and the original data is unchanged. When an object is passed as an argument into a method, the reference is unchanged, but methods called on the object can still affect it.

Null Pointer Errors: By this point in your programming experience, you have almost certainly experienced a `NullPointerException`. This is, in many ways, a catch-all category, because a null reference is generally not due to a simple typographical error. In the simplest case, a reference simply has not been initialized with some default constructor, but more often there is a logical error in the design of the code.

16.2 Concepts: Approaches to debugging

When you have discovered the existence of a bug, pinning down its cause can be difficult. There are a number of different techniques that are useful for narrowing down the possible problems.

16.2.1 Assertions

A common cause of program errors is an incorrect assumption by a programmer. A programmer can

assume that the user will only enter positive numbers, that a library call will not throw any exceptions, or that a linked list is not empty. Some assumptions are reasonable, but it is important to make sure that they are correct.

Using *assertions* is a way to check some of your assumptions, often those surrounding a method call. In most languages, an assertion tests a condition. If that condition is true, nothing happens, but, if it is false, the program shuts down or an error or exception is thrown. Using assertion statements is particularly important with methods because you want to be sure that both your input and output are in the ranges you expect them to be.

Java supports assertions natively, as we will discuss in the next section. However, virtually every language allows you to create your own assertions should they not be present as a language construct.

16.2.2 Print statements

Computer programs execute quickly. Even if they executed a million times more slowly no human is sensitive enough to decipher the flow of electrons inside the processor as a program executes. The simple truth is that we have no idea what is happening when our programs execute. We believe that we understand our programs well, and the output usually confirms that our program is doing what we imagine that it should be doing.

If the output does not match what is expected, we may not have enough data to understand the problem. Since there have been programmers, they have been printing out additional debug information to find their errors. This technique can go a step beyond simple assertion statements because you can print out the values of the variables rather than just test to see if they are in a range.

Once you have found the error in your code, it can be tedious to remove all of your debug statements. Some programmers use a special print command that can be turned off using a global variable or compiler option. Others send their output to stderr so that it doesn't interfere with the legitimate output of the program. Much depends upon the system, the language, and the individual tastes of the programmer.

16.2.3 Step-through execution

With the rise of modern debugging environments, using print statements has lost some of its appeal. Most languages allow the programmer to run his or her program in a special debug mode in which it is possible to execute a single line of code at a time. These tools usually give the option of stepping over method calls or stepping into them, on a case by case basis.

As the program executes, the programmer can inspect the values of the variables in the code. This method of debugging is excellent because it allows the programmer to watch the execution of the code at whatever pace he or she desires. Pinpointing problems becomes trivial if you know which variables you need to watch.

Despite the power of this technique, it has critics. Some older programmers look down on these tools because they make beginning programmers lazier and, in some cases, less careful about writing code correctly in the first place. It should be noted that step-through execution modes are not available for every language and for every system. Some embedded software or operating system programming cannot be debugged on the real hardware in this way. Of course, most of these systems

can be run in virtual environments that do allow step-through debugging.

Even when step through debugging is available, there are difficulties that can limit its effectiveness. If the bug occurs sporadically, perhaps due to race conditions, a programmer may not know where to start looking. Certain data structures such as the list template in C++ may not be easily traversable using the inspection facilities of the debugger. Likewise, the bug or the source of the unexplained behavior could be buried in library code. The debugger does not always have access to library code for stepping through.

16.2.4 Breakpoints

Breakpoints are a feature of step-through debuggers designed to make them easier to use. A user can specify a particular line of code (with some restrictions) as being a place where the debugger should pause execution. Debuggers typically rely on at least one breakpoint in order to skip all the preliminary parts of the code and skip straight to the perceived trouble spot.

Sometimes an error will predictably crop up after many thousands of iterations of a loop or unpredictably in the case of bugs dependent on race conditions or user input. For either of these cases, conditional breakpoints can be used to save the debugger a great deal of time. Rather than always pausing execution on a given line, a conditional breakpoint will only pause if a certain condition is met.

16.3 Syntax: Java debugging tools

16.3.1 Assertions

As we mentioned before, many languages have assertions as a built-in language construct. In Java, there are two forms this feature takes. The simpler can be done by typing the following.

```
assert condition ;
```

In this case, condition is a boolean value that is expected to be true for the program to function properly. The more complicated form of the feature can be used as follows.

```
assert condition : value ;
```

This form adds a value that can be attached to the assertion to give the user more information about the problem. This value can be any primitive data type, any object type, or a statement that evaluates to one of the two.

If you have never used an assert statement before, you might want to test it out by forcing an assertion to fail. You might try

```
int x = 5;  
assert (x < 4) : "x is too large !";
```

Then, if you compile your program and run it through the JVM, you will be shocked when **absolutely nothing happens**. Actually, some of you with older Java compilers may have heard complaints when you tried to compile. If you have a Java 1.3 compiler or earlier, it will treat assert

like an identifier. Some old Java 1.4 compilers may also give warnings or require special flags to be set to compile. However, if you have an up to date compiler, the problem is that the JVM must have assertions enabled at runtime. Assertions are intended to be a special debugging tool and ignored otherwise. To turn run program AssertionTest with assertions enabled, type

```
java -ea AssertionTest
```

With this option, an exception should be thrown at runtime.

```
Exception in thread "main" java.lang.AssertionError: x is too large!
```

There are other options allowing you to enable or disable assertions for specific packages or classes.

Now that you know how to use assertions, you need to know when they are a good idea. The Java Tutorials on the Oracle website suggest five situations where assertions are useful: internal invariants, control-flow invariants, preconditions for methods, postconditions for methods, and class invariants. *Internal invariants* are those situations when you assume that reaching a certain place in your code, like the `else` branch of an `if` statement, will force a variable to have a certain value. For internal invariants, you assert that the variable has the expected value. A *control-flow invariant* means that you assume that your code will always execute along a certain path. For control-flow invariants, you assert `false` if the JVM reaches a point in the code you expected it never would. *Method preconditions* are those conditions you expect to be true about the state of objects or the input to a method before the method is called.

The philosophy of Java is that `public` methods should *not* have assertions used to test their preconditions. Instead, illegal input values for a `public` method should cause exceptions to be thrown, so that improper usage can always be dealt with. In contrast, *method postconditions* are the states that various variables and objects should have at the end of a method call. Using assertions to check these values is fine, since they reflect an error on the part of whoever wrote the method. *Class invariants* are conditions about the state of every instance of a class that should be true as long as the class is in a consistent state. Perhaps a method call rearranges the innards of an object, but, by the end of the method call, the object should be consistent again. You should use assertions to check class invariants at the end of every method that could make the object violate the invariants.

Wonderful as assertions are, there are times when they should not be used. The key danger of assertions is that they are usually turned off. Thus, any statement that is part of an assertion should not have side-effects that are necessary for the normal operation of the program. For example, imagine that you have a object called `bacteria` that mutates periodically. The mutation returns `true` if successful and `false` if there was an unexpected error. You should **not** test for that failure inside an assert, as follows.

```
assert bacteria.mutate () : "Mutation failed !";
```

With assertions disabled, the `bacteria` object will no longer mutate. Instead, your assertion should test only the result of the computation.

```
boolean success = bacteria.mutate();  
assert success : "Mutation failed !";
```

As stated above, checking for bad input coming into `public` methods should not be done with assertions because turning off assertions will remove your error checking.

16.3.2 Print statements

Print statements are one of the most time-honored methods of debugging and remain a quick, dirty, yet effective means of finding errors. Java does not provide any special tools to make print statements easier to use for debugging. Some purists might argue that all of this kind of debugging which focuses on progressively narrowing own the location of a problem until the bad assumption, logical error, or typographical error can be found should be done only with assertions.

Nevertheless, there are a few tips to make print statements a better debugging tool in Java. The first is the use of `System.err`. By now, you have used `System.out.print()` and `System.out.println()` so many times, you are probably tired of them. Any output method that can be used with `System.out` can also be used with `System.err`. For example, there is a `System.err.print()` and a `System.err.println()` method. If you simply run a program from the command line and watch the output, you should see no difference between using `System.out` and `System.err`. However, if you redirect the output of your program to a file using the `>` operator, only the `System.out` code will be sent to the file. Anything printed with `System.err` will be sent to the screen. Alternatively, you can redirect `System.err` to a file by using the `2>` operator. Using `System.err` makes it easier to separate legitimate output from error messages, but it also makes it easier to comment out your debug code by doing a find and replace on your code.

A more extensive method for using print statements to debug is by defining your own class for printing. Every method in it can call a corresponding method in `System.out` or `System.err`. You can define a `boolean` value at the class level that determines whether or not methods in your debug printing class print or stay silent. When you want to change from debugging to your submission or retail version of the code, you can simply switch this value to `false`.

A “modernized” method of using print statements is creating a simple GUI instead. In preparing materials for this textbook, we were occasionally frustrated by the fact that multiple threads can interfere with each other while printing on the screen: You can’t always tell which thread is printing which characters. By displaying the output of each thread in separate `JTextArea` or `JLabel` widgets or a simple GUI, you can disentangle the output of each thread.

16.3.3 Step-through debugging in Java

Since DrJava is a great educational tool and Eclipse is so widely used, we are going to review the step-through debugging features of each program. Similar tools are available with other IDE’s and for most languages.

The debugger in DrJava is very simple. To enable the debugger, check the Debug Mode checkbox in the Debugger menu. Doing so should bring up a debugging pane in the DrJava window. In or out of debug mode, it should be possible to set breakpoints on any executable line of code, either by choosing the Toggle Breakpoint on Current Line command from the Debugger menu or by typing `Ctrl+B`. Once you have set at least one breakpoint, you can run the program. If a line with a

breakpoint on it is reached, then the execution of the program will pause.

In the debugging pane, there should now be a list of threads, with the one that hit the breakpoint highlighted. Having reached this point, the debugging pane has four buttons you can use to move through code: **Resume**, **Step Into**, **Step Over**, and **Step Out**. The **Resume** button allows the program to continue execution, until it hits another breakpoint. The **Step Into** button advances the execution of the program by one statement, moving into a method if there is a method call. The **Step Over** button also advances the execution of the program by one statement, but it skips over method calls. The **Step Out** button advances the execution of the program to the end of the current method and returns, popping the current method off the stack.

In the debugging pane, there will be a Watches tab that allows you to find out the value of a given variable in local scope. All you need to do is type the variable's name, and its value and type should be displayed. Next to the Threads tab, there should also be a Stack tab, showing the current method call stack.

The Eclipse tools are similar but much more advanced. You can set breakpoints in Eclipse either by right clicking on the shaded bar immediately to the left of the line you are interested in or by selecting Toggle Breakpoint from the Run menu. To debug a program in Eclipse, right click on the file you wish to run in the Package Explorer and select Debug As Java Application. If your program is already set up to run, you can simply click the Debug button in the toolbar. Whenever you hit a breakpoint, Eclipse will switch to the Debug perspective if it is not already there. Once execution is suspended on a breakpoint, you can use commands nearly identical to the ones in DrJava. The commands Resume, Step Into, and Step Over from the Run menu are the same as DrJava versions, and Step Return is the equivalent of Step Out. Eclipse adds the useful command Run to Line, which will execute code until it reaches the specified line.

By right-clicking on a breakpoint in Eclipse, you can access its properties. Though properties, you can specify that a breakpoint only halts execution when a specific condition is true or only for a specific thread. The more advanced debugging in Eclipse also provides more comprehensive variable watch and inspection options. Simply by hovering over a variable, its type and value are displayed. You can also inspect an object and traverse its fields. As with DrJava, you can explicitly watch variables, but local variables are also displayed by default.

The difference between a good programmer and a bad programmer is often just experience. Having seen a bug before means you know to expect it in the future. There is no substitute for pulling your hair out over a bug for hours before finally squashing it, but we will give a few examples corresponding to the common bugs listed in [Section 16.1](#).

Example 16.1: Precision errors

Precision can cause some subtle errors, especially with **float** types. Here is an example of a program attributed to Cleve Moler that gives some estimation of the threshold for floating point precision. Note that $a \approx 4/3$, making $b \approx 1/3$, and $d \approx 0$. Nevertheless, the comparison ($d == 0.0$) in the **if** statement in this code will evaluate to **false**.

```
double a, b, c, d;  
a = 4.0 / 3.0;
```

```
b = a - 1;  
c = b + b + b;  
d = c - 1;  
System.out.println (d);  
if( d == 0.0 )  
    System.out.println (" Success !");
```

The output for this fragment is -2.220446049250313E-16. Computer scientists who specialize in numerical analysis have tricks for minimizing the amount of floating point error introduced, but awareness is an easy solution to these kinds of bugs. When testing for specific values of a floating point number, it is wise to test for a range rather than a single value. For example, the condition ($d == 0.0$) could be replaced by ($\text{Math.abs}(d) < 0.000001$). ■

Example 16.2: Overflow and underflow

As you well know, the **int** and **long** types have limited bits for storage. If an arithmetic operation pushes the value of an **int** variable larger than **Integer.MAX_VALUE**, the variable will come full circle and become a negative number, usually with a large magnitude. The converse happens when a variable is pushed lower than the smallest value it can hold. These situations are called overflow and underflow, respectively, and Java throws no exceptions when they occur. Programmers who deal with large magnitude values in **int** or **long** types get used to underflow and overflow, and, when unexpected values are output by their programs, they are usually quick to pin down the problem variable.

Overflow and underflow can cause much more subtle bugs when programmers forget the limited range of values for **byte** and **char** types. For example, a curious beginner programmer might want to print out a table of all of the possible values for **char**. Perhaps the programmer has forgotten the range of values a **char** can take. Perhaps surprisingly, the following loop does not terminate.

```
for ( char letter = '\0'; letter < 100000; letter ++ )  
    System.out.print ( letter );
```

Each time letter reaches **Character.MAX_VALUE** which is '\uFFFF' or 65535 as a numerical value, the next increment pushes its value back to 0. These kinds of errors with **byte** and **char** values are most common when variables of those types are being used as numbers. Some examples are cryptography, low level file operations, and manipulation of multimedia data. The best solution is care and attention. It can help to store the values in variables with more bits such as **int** or **long** values, but care must still be taken to ensure that these values are within the appropriate range before storing them back into variables with a smaller number of bits.

For example, color values in many image formats are stored as red, green blue values with a **byte** used for each of the three colors. In this system, the darkest color, black, is represented as (0,0,0), i.e. zero values for each of the three **bytes**. At the same time, the lightest color, white, is represented conceptually as (255,255,255). In principle, we can perform a very simple filter to increase contrast and lightness by simply doubling all the pixel values. Given red, green, and blue color values stored in three **byte** variables called red, green, and blue, a naive implementation of this filter might be as follows.

```
red *= 2;  
green *= 2;  
blue *= 2;
```

In Java, this code would not work. The first problem is that, even though image standards are written with color values between 0 and 255, Java `byte` values are **signed**. The web standard for the color purple has red, green, and blue values of (128,0,128). Since Java `byte` values are signed, printing the `byte` values for each component of purple directly will actually print (-128,0,-128). Multiplying the green value by 2 is clearly still 0. However, multiplying -128 by 2 as a `byte` value is -256 which underflows back to 0. Thus, “brightening” purple actually turns it into (0,0,0), black. Properly applying the filter to a `byte` requires a conversion to the `int` type, masking out the sign bit, scaling by 2, capping the values at 255, and then casting back into a `byte`. Despite the complicated description, the code is not too unwieldy.

```
red = ( byte ) Math.min ( 255, 2*( red & 0xFF ) ); // bitwise AND automatically upcasts  
to int green = ( byte ) Math.min ( 255, 2*( green & 0xFF ) );  
blue = ( byte ) Math.min ( 255, 2*( blue & 0xFF ) );
```



Example 16.3: Casting errors

The previous example about scaling color component values is an excellent example of the dangers of casting. Someone can easily forget that the implicit cast to convert a `byte` to an `int` always uses a signed conversion. Likewise, the explicit cast needed to store an `int` into a `byte` will cheerfully convert any arbitrarily large `int` into a `byte`, even though the final value might not be expected by the programmer.

Many other casting errors crop up commonly. The most classic example might be muddling floating point and integer types.

```
int x = 5;  
int y = 3;  
double value = 2.0*( x/y );
```

Above, it is easy for a programmer to forget that the division of `x` and `y` is integer division. After all, the 2.0 is right there, causing an implicit cast to `double`. Of course, this cast happens after the division, and the answer stored into `value` is 2.0 and not the 3.3333333333333335 that the programmer might have expected.

Newer programmers sometimes forget that an explicit cast from a floating point type to an integer type always uses truncation, never rounding.

```
int three = ( int ) 2.99999;
```

This assignment will always store 2 into `three`. The `Math.round()` method or some other additional step is needed to perform rounding.

Casting errors are not limited to primitive data types. Object casting will be discussed at length in

[Chapter 17](#). The biggest danger there is an incorrect explicit upcast.

```
Fruit snack = new ChiliPepper();  
Apple apple = ( Apple ) snack ;
```

In a botanical sense, a chili pepper is indeed a fruit and its parallel Java class is apparently a child of the Fruit class. For some reason, the programmer thought that the only Fruit that would be pointed at by a snack reference would be of type Apple. Instead of a mouth on fire, the programmer gets a ClassCastException. This two line example is so simple that it should never come up in serious programming. A much more common example is an array or linked-list whose type is some superclass of the item you generally expect to be in there. If a large team is working on a body of code which such a list in it, half of the team might expect the list to contain only Apple objects while the other expected only ChiliPepper objects. The use of generics, discussed in [Chapter 18](#), can reduce the number of casting errors of this kind, but some applications require a list to hold many different types with a common superclass. In those cases, some amount of explicit (and therefore dangerous) casting will usually be necessary when retrieving the objects from the list. ■

Loops give Java much of its expressive power and unsurprisingly give it much of its power to express incorrect as well as correct code. We are just going to mention a few of the most common loop errors.

Example 16.4: Off-by-one errors

Computer scientists often use zero-based counting. This departure from “normal” practices is just one source of loops that iterate one time more or less than they should. A good rule of thumb is, if you want to iterate n times, start at 0 and go up to but not including n . Alternatively, if you have a reason not to be zero-based, you can start at 1 and go up to and including n .

```
for ( int i = 1; i < 50; i++ )  
    System.out.println (" Question " + i + ".");
```

Perhaps you want to make a template for an exam. Instead of being zero-based, you start at 1 because most exams do not have a Question 0. Unfortunately, you have gotten so used to use a strictly less than for your ending condition, you forget to change it. You only get 49 questions printed out. If your only purpose is making an exam, you can catch your mistake and move on. If you are writing a program that dispenses a quantity of heart medication into a patient’s IV in a hospital, one iteration too few or too many could cause the patient to get too little of the drug to make a difference or too much of the drug to be safe.

Input is another tricky area when it comes to being off by one.

```
int i = 0;  
double sum = 0;  
int count = 0;  
Scanner scanner = new Scanner ( System.in );  
while ( i >= 0 ) {  
    sum += i;
```

```

System.out.print (" Enter an integer ( negative to quit ): ");
i = scanner.nextInt ();
count++;
}
System.out.println (" Average : " + (sum / count));

```

This fragment of code appears to be a perfectly innocent loop that finds the average of the numbers entered by a user. The loop uses a sentinel value so that the user simply enters a negative number when all the numbers have been entered. The value of sum is updated before the user enters a value; thus, the harmless 0 from the declaration of i is included but the final negative number entered to leave the loop is not. Unfortunately, the value of count is incremented for every turn of the loop, even the extra one for the negative number. To combat this problem, an if statement could be used inside of the loop or count could simply be initialized to -1. The mistake is a simple one, but it doesn't jump out at you unless you trace a few executions. What is most insidious is that, especially for large sets of input numbers, the error is going to be small. Catching this kind of bug will be discussed more thoroughly in the second half of this chapter, dealing with testing. ■

Example 16.5: Infinite loops

Infinite loops come in many different flavors, from the `char` overflow example earlier to traversing a linked-list which has a cycle in it. Many infinite loops are caused by simple typographical errors. Perhaps the most classic is:

```

int i = 1;
while ( i <= 100 );
{
    System.out.println (i);
    i++;
}

```

It's usually a beginning programmer who leaves a semicolon at the end of the `while` header, but even veterans can get a little enthusiastic about semicolons. Often a programmer confronted with such a bug (which causes no output, in this case) will scour the body of the loop for some clue as to why it isn't advancing yet never carefully scrutinizing the condition. An extra semicolon at the end of a `for` loop header will usually cause an error but will usually not cause an infinite loop.

```

public double average ( int [] array ) {
    double sum = 0;
    int count = 0;
    for ( int i = 0; i < 100; i++ ) {
        sum += array [i];
        count++;
        if( i == 0 )
            i--;
    }
    return sum / count ;
}

```

}

This example is the kind that might be too confusing to appear in a textbook, but nearly everyone has written worse code while learning to program. We could suppose that this method is meant to average the values in an array, but, for some reason, zero valued entries are not to be counted. The student probably meant to have the following **if** statement:

```
if( array [i] == 0 )
    count --;
```

Those two small changes turn the method into a working but slightly inelegant solution. When debugging remember that index variables in **for** loops can get changed in the body of the loop and change the expected behavior. Generally it is a bad idea to change the value of an index variable anywhere other than the header of a **for** loop, but there are times when doing so gives the cleanest solution.

Many loop errors are caused by a bad header. Getting the an inequality backwards or switching increment and decrement will usually make a loop that runs a very long time or not at all. We'll see the second possibility just a little later.

```
for (i = 10; i > 0; i++) {
    System.out.println (i + "!");
}
System.out.println ("Blast - off!");
```

In this case, the programmer clearly wanted to count **down** from 10 to 1, but after so much incrementing, he or she forgot to make *i* decrement. As a result, the value of *i* increases for a very long (but not infinite) time, until it overflows. ■

Example 16.6: Zero loops

On the other end of the spectrum, a bad condition can make a loop execute zero times on **for** and **while** loops. For some input, doing so might be intended behavior. In other cases, no input will ever cause the loop to execute.

```
int i = 0;
double sum = 0;
int count = -1;
Scanner scanner = new Scanner ( System.in );
while ( i > 0 ) {
    sum += i;
    System.out.print (" Enter an integer ( negative to quit ): ");
    i = scanner.nextInt ();
    count++;
}
System.out.println (" Average : " + (sum / count));
```

We have just returned to our earlier example of averaging a set of numbers input by the user. This time we have initialized count to be -1 to avoid the off-by-one error, but we have also changed the inequality of the **while** loop from greater than or equal to strictly greater. As a consequence, the loop is never entered because the zero, the initial value of i, is too small.

```
public static boolean isPrime ( int n ) {  
    for ( int i = 1; i < n; i++ ) {  
        if( n % i == 0 )  
            return false ;  
    }  
}
```

Here is a simple method intended to test the number n for primality. Unfortunately, the programmer started the index i at 1 instead of 2. As a consequence, this loop will only run once before finding that every number is divisible by 1. True, this is not a loop that executes zero times, but only once is still just as wrong.

```
public static boolean isPrime ( int n ) {  
    for ( int i = 2; i < n; i++ ) {  
        if( n % i == 0 )  
            return false ;  
        else  
            return true ;  
    }  
}
```

This example is very similar code, trying to solve the same problem. Again, the loop only runs once because the programmer forgot that finding a single case when a number is not evenly divisible by another number does not make it prime. Many, many beginning programmers make this mistake when asked to solve this problem. Perhaps some insight about the nature of bugs can be gained from this example. By the time a student writes a program of this kind, he or she should have a fair idea of how **for** loops and **if** statements work. Likewise, the student will have a fair understanding of the notion of primality. Yet, in the process of combining the ideas together, it is easy to get sloppy and write code that gives some semblance of being correct without being. ■

Example 16.7: Equivalence testing errors

Equivalence is tricky in Java. Very inexperienced programmers confuse the = operator with the == operator, but using the == operator to test for equivalence between two references causes more (and subtler) problems. Comparing two references with the == operator will evaluate to **true** if and only if the two references point at the exact same object.

```
String string1 = new String (" Test ");  
String string2 = new String (" Test ");  
if( string1 == string2 )  
    System.out.println (" Identical ");
```

```
else
    System.out.println (" Different ");
```

Because these two String references point to two different String objects, which happen to have identical contents, the == returns **false** and the output is Different. With String objects this matter is further confused by a Java optimization called String pooling.

```
String string1 = " Test ";
String string2 = " Test ";
if( string1 == string2 )
    System.out.println (" Identical ");
else
    System.out.println (" Different ");
```

Because Java keeps a pool of existing String values, only one copy of “Test” is in the pool, and both string1 and string2 point to it. Thus, this second fragment of code prints Identical. Because of String pooling, programmers can write code which can work in some situations and fail in other, if it is dependent on the == operator.

For String objects as well as almost every reference type, it is almost always the case that the equals() method should be used to test for comparison instead of the == operator. There are a few instances when it is necessary to know if two references really and truly do refer to the same location in memory, but these instances should be a tiny minority.

That said, the equals() method is not bullet-proof. With String objects and most of the rest of the Java API, you can expect very good behavior from the equals() method. However, if you create your own class, you are expected to implement the equals() method. By default, the equals() method inherited from Object only does an equality test using ==.

Properly implementing the equals() method takes care and thought. If your class contains references to other custom classes, you must be certain that they also properly implement their own equals() methods. Likewise, to conform to Java standards, a custom equals() method should also imply that you implement a custom hashCode() method so that objects that are equivalent with equals() give the same hash value. It seems nit-picky to mention this, but many real-world applications depend on the efficient and correct operation of hash tables. ■

16.3.4 Array errors

Any time you have a large collection of data, there are always opportunities for bugs. With catastrophic array bugs, Java usually gives very good exceptions that will point you to the line number. Once you have gotten to this point, the bug should be obvious. The biggest difficulties arise when some unusual course of events is responsible for the bug cropping up and you have to reconstruct what it is.

Example 16.8: Out of bounds

We have all experienced an `ArrayIndexOutOfBoundsException`. Either a little carelessness with our indexes or a mistake about the size of the array can lead us to try to access an element that isn't in the

array. In the C language, a negative index is sometimes a legal location but never in Java. It is very common to go just slightly beyond the bounds of the array, particularly with a loop.

```
int [] array = new int [100];
for ( int i = 0; i <= 100; i++ )
    array [i] = i;
```

In this example, the last iteration of the loop will access index 100 when array only goes up to index 99.

The causes for going out of bounds can be more subtle. We can imagine an array of linked lists used as a hashtable, perhaps for storing words in a dictionary. If we want to hash based on the first letter of the word, we could have an array of length 26. Consider the following helper method used to add a new String to the hashtable.

```
public void add ( String word ) {
    int index = word.toLowerCase ().charAt (0) - 'a';
    list [ index ].add ( word );
}
```

We can assume that the add() method for a given linked list works properly, but we may already have caused other problems. For one thing, we assumed that word began with either an upper or lower case letter. We are depending on other code to check the input and throw out words like “\$1” or “-isms”. Incidentally, we are also assuming that word has at least one character in it. Even if we expect the input to the method to be error free, some error checking is always safe. ■

Example 16.9: Uninitialized object arrays

Another simple mistake that can occur with arrays is failing to initialize an object array. With a primitive data type like int, creating an array with 1,000 elements automatically allocates enough space to hold those elements and even initializes each one to a default value, zero in the case of an int. With an object data type, however, each element of the array is just a reference to null until it is initialized.

```
Hippopotamus [] hippos = new Hippopotamus [15];
hippos [3].feed ();
```

This example causes a NullPointerException. New programmers are often confused by this error because, if they are expecting an error, they are expecting the exception to say something about the array. For more experienced programmers, this kind of mistake is usually more of a forehead-slapping, how-silly-of-me-to-forget error than a mind-numbing puzzler that will take hours to debug. It is probably just a matter of instantiating each element in the array before you try to feed those hungry, hungry hippos. ■

```
Hippopotamus [] hippos = new Hippopotamus [15];
for ( int i = 0; i < hippos.length ; i++ )
    hippos [i] = new Hippopotamus ();
```

16.3.5 Scope errors

We don't have variables in real life, and, as a consequence, our intuition about them is sometimes wrong. Which variable you are accessing at any given time can appear obvious, even if it really isn't.

Example 16.10: Shadowing variables

Java allows variables in different scopes to be declared with the same identifier. If the scopes are two separate methods, then they will never interfere with each other. However, if one scope encloses another, the inner variable will *shadow* or hide the outer variable.

```

1 public class Shadow {
2     int darkness = 10;
3
4     public void deepen ( int darkness ) {
5         darkness += darkness ;
6         if( darkness > 100 )
7             darkness = 100;
8     }
9
10    public int getDarkness () { return darkness ; }
11 }
```

In this example, the field variable `darkness` is being shadowed by the local variable `darkness` in the `deepen()` method. It appears that the programmer wanted to increase the field `darkness` by the amount passed into the parameter `darkness` and failed to notice that both variables had the same name. As a consequence, the parameter `darkness` will double itself and then never be used again while the field `darkness` will never increase. This kind of bug could go uncaught for a long while until a programmer notices that the `Shadow` object is not increasing in darkness no matter how many times it is told to.

This kind of mistake is also common in constructors, since it is reasonable to give a certain parameter a name similar to the field it is about to initialize. Some programmers explicitly prefix all fields with `this` even though it is often redundant. Three additions of `this` will fix the problem in the preceding example.

```

public void deepen ( int darkness ) {
    this.darkness += darkness ;
    if( this.darkness > 100 )
        this.darkness = 100;
}
```

In Java, scope is also defined in terms of classes and their parent classes. A parent class variable can be shadowed by a child class variable of the same name.

```
1 public class Bodybuilder {
```

```

2     public int strength = 8;
3
4     public boolean isStrongEnough ( int strengthNeeded ) {
5         return strength >= strengthNeeded ;
6     }
7
8     public void setStrength ( int value ) { strength = value ; }
9 }
```

```

1 public class BraggingBodybuilder extends Bodybuilder {
2     public int strength = 10;
3
4     public void brag () {
5         System.out.println ("My strength is " + strength + "!");
6     }
7 }
```

This example looks like a simple case of inheritance, but whoever wrote the BraggingBodybuilder class seems to have mistakenly included the field strength again. As a consequence, any BraggingBodybuilder will always brag that his or her strength is 10, even when code sets his or her strength to other values. When strength is tested, it will use the strength field from the superclass Bodybuilder which is set by the setStrength() method. Sometimes similar behavior is desired, but it seems to be accidental here. When classes have large numbers of fields, making such a mistake becomes easier.

```

1 Bodybuilder builder = new BraggingBodybuilder ();
2 builder.strength = 15;
3 BraggingBodybuilder bragger = ( BraggingBodybuilder ) builder ;
4 bragger.brag ();
5 bragger.strength = 20;
6 bragger.brag ();
```

Dynamic and static binding complicate this scope problem further. This fragment of code using the class definitions above highlights these complications. Because fields are statically bound to the class of the object, the strength field for Bodybuilder will be set to 15 on [line 2](#), and the strength field for BraggingBodybuilder will be set to 20 on [line 5](#). Thus, the first call to brag() will print out My strength is 10!, but the second call will print out My strength is 20!. ■

Example 16.11: Reference vs. value

The final category of scope error we will talk about occurs because of confusion between passing by reference and passing by value when using methods. Everything variable in Java is passed by value. However, when that value is itself a reference, it is possible to change the values that it references.

```

public void increaseMagnitude ( int number ) {
    number *= 10;
```

A novice Java programmer might write a method like the above, expecting the value of number to increase by 10 in the calling code. Some languages like Perl use call by reference as default. Other languages like C++ and C# allow the user to mark certain parameters as call by reference. Programmers comfortable with such languages may be confused about the workings of Java.

On the other hand, becoming used to the pass by reference style of Java can cause other errors.

```
public void increaseMagnitude ( int [] numbers ) {  
    numbers [0] *= 10;  
}
```

In this similar example, the 0 index element of numbers is increased by a factor of 10. Unlike the previous code, the increase in the value of that element will affect the array passed in by the calling code. The values in the array are shared by the increaseMagnitude() method and the calling code. The same phenomenon can be observed with the fields of objects whose references are passed into a method. ■

16.3.6 Null pointer errors

Null pointer errors usually raise a NullPointerException in Java. This category of errors is something of a catch-all that could happen for many different reasons, some of which have already been mentioned. Because of loop errors, some variables might not be initialized. A NullPointerException could be raised because the elements of an object array have not be initialized. Scope problems could cause a reference to be null if the programmer was mistakenly updating another reference, leaving the reference in question uninitialized.

Though common, it is very difficult to give a blanket explanation for why most null pointer errors happen. Usually there is some fundamental error in program logic. Linked lists and tree structures that rely on null references to mark the end of a list or an empty child node are especially susceptible to these errors.

One significant source of errors is careless usage of method parameters. A programmer may pass in objects that do not conform to the expectations of the method or even null references instead of objects. Well written methods, particularly library calls, should be designed to throw an appropriate exception when this happens. Poorly designed code may blindly use a null reference without checking it first, causing a NullPointerException.

16.4 Concurrency: Parallel bugs

We will only briefly discuss parallel bugs here because we have already gone into depth about the dangers of parallel programming in [Chapter 14](#). Except in the case of deadlocks and livelocks, the real trouble with parallel bugs is that they make the appearance of ordinary sequential bugs become nondeterministic.

16.4.1 Race conditions

A race condition describes the situation when the output of a program is dependent on the timing of the execution of two or more threads. Because of the complexity of the JVM and the OS and the fact that many other processes may be running and interacting, it is usually impossible to determine how two threads will be scheduled. As a consequence, if the output of the program depends on unpredictable timing, the output will also be unpredictable.

In Java, the way that race conditions usually impact the program is through some variable shared between multiple threads. When the schedule of threads becomes unpredictable, the changes made to this variable can come out of sequence, and its value becomes unpredictable. Incorrect output means that your program has a bug, but the most frustrating aspect of race conditions is that they are nondeterministic. Your program could sometimes have the right answer and sometimes not. Your program could always have the wrong answer, but not always the same one. The truly insidious issue with race conditions is that they will usually cause errors only a tiny percentage of the time. Thus, rigorous testing such as we will discuss in the second half of this chapter is necessary to even be aware that a race condition is occurring.

16.4.2 Deadlocks and livelocks

Both deadlocks and livelocks describe situations in which some part of your program will stop making progress because of thread interaction. In the case of deadlock, there will be some circular wait in which thread A is waiting for thread B which is waiting, directly or indirectly, on thread A. In the case of livelock, some repetitive pattern of waiting for a condition that will never be satisfied is still going on, but the threads continue to use CPU time and are not simply waiting.

If your program reaches a deadlock state, it will not terminate. If threads updating a GUI become deadlocked, your windows may freeze. Typically, deadlocks are nondeterministic and occur only some of the time. Like all race conditions, they can be difficult to detect and duplicate. In fact, `Thread.stop()`, `Thread.suspend()`, and `Thread.resume()`, three seemingly useful and fundamental methods that were originally part of the Java Thread class, have been deprecated because they are deadlock prone.

16.4.3 Sequential execution

One bug which is impossible to achieve in non-parallel code is sequential execution. This situation arises when, usually due to overuse of synchronization tools, parallel code runs sequentially. Each segment of code, instead of running in parallel, is forced to wait for another to complete. A certain amount of serial execution is necessary to maintain program correctness and avoid race conditions, but Amdahl's Law gives a rigid, mathematical characterization of how easily speedup can be lost if the serial execution takes up large portions of the code. Because setting up threads and using other concurrency tools does have some overhead, a parallel program executing sequentially can even run more slowly than a completely sequential version.

Because programs are usually parallelized for the sake of speedup, it is possible to time sections of programs to see how well you have parallelized them. Sequential execution due to synchronization tools is only one of the many problems that can cause slow execution. The threads may be competing for a limited resource such as an I/O device or may be fighting over a small section of memory, causing cache misses. Tuning applications for maximum performance requires an expert

understanding of the concurrency issues within software as well as the underlying OS and hardware characteristics. For now, it's enough to be aware of the risk of sequential execution and be as careful as possible when applying locks and other synchronization tools.

16.5 Finding and avoiding bugs

What would you do if you wanted to design a system for administering a dose of radiation to a specific location on a patient to help treat them for cancer? Depending on the specification of the problem, you might need to control various voltage sources, read data from various sensors, and create a terminal interface or a GUI. With a well designed specification, you could probably apply your knowledge of loops and control structures to some API and provide a software solution that met requirements.

But, how would you know that it worked? Sure, you could run a series of tests, but how many tests would it take for you to be convinced that it worked perfectly? What if your grade was dependent on it working without a single error? Or your job? Or your life?

You have probably already faced the stress of trying to get a program to work as well as possible for the sake of your grade. It is not such a far cry to imagine your job being on the line if you make a mistake as a professional programmer. But, what about your life? Perhaps you will never put your life in the hands of code you write, but odds are that you have already put your life in the hands of someone else's code. Software controls airplanes, automobiles, medical equipment, and countless other applications where a bug in the code could actually result in loss of human life.

Sadly, there have been cases when such bugs have surfaced with deadly consequences. One of the most famous examples of the dangers of badly written software is the Therac-25. The Therac-25 was a machine designed to deliver therapeutic radiation for medical purposes. Between 1985 and 1987, use of the Therac-25 caused at least six incidents of massive radiation overdoses, leading to at least three deaths.

Like most failures of this scope, there was more than a single cause behind the Therac-25 tragedies. For one thing, the machines did give an error code. However, the user manual did not explain the error code, and the technicians were not trained to deal with the errors. Even when patients complained about pain caused by the machine, the technicians and even the manufacturers of the Therac-25 were confident that the machine was operating correctly because neither of the previous models, the Therac-6 and the Therac-20, had suffered any problems. Overconfidence has played a significant role in many of the worst systems failures, including the devastating Chernobyl disaster.

Ignoring the human errors, a number of software errors were also responsible for the Therac-25 overdoses. The overdoses occurred when technicians made incorrect keystrokes giving confusing instructions to the Therac-25 about which mode of operation it should be in. In this situation, the machine would operate with a high-power beam but without the beam spreader that was necessary for its safe operation. The designers ignored the possibility that this series of keystrokes would happen. Also, a race condition was involved in this bug since it depended on one task that set up the equipment and another that received input from the technician. This race condition was never caught

because only technicians with long practice could work fast enough to cause the bug. Finally, a counter was incremented for use as a flag variable, but arithmetic overflow occasionally caused this flag to have the wrong value.

In the remaining half of this chapter, we will discuss a number of testing methodologies and design strategies to minimize errors in software.

16.6 Concepts: Design, implementation, and testing

Unfortunately, there is no foolproof way to design software. There are many researchers who work to design new languages and new development tools that limit certain kinds of mistakes, but it is impossible to design a language as powerful as C or Java which will also prevent all software bugs. A consequence of the halting problem, a fundamental concept in the theory of computation, is that there is no way to design a test that will detect all potential infinite loops (or infinite recursion) for all programs.

With careful design, implementation, and testing, most errors can be reduced almost to nothingness. In the following subsections, we will discuss these three aspects of programming and how you can apply them to writing better programs.

16.6.1 Design

We have remarked in the past that good design pays off ten-fold in implementation, and that payoff continues to increase by factors of ten as you move on to testing and eventually deployment.

One of the first design decisions you may have to make is choice of language. Some languages are better designed for certain tasks than others. For example, languages like Ada have been carefully designed to minimize programming mistakes such as mis-matched else blocks. Many functional languages like ML are designed so that memory errors such as a NullPointerException are impossible. Even Java has taken clear steps to avoid some of the errors possible in C and other languages that allow pointer arithmetic, such as bus errors. However, many other factors such as portability, compatibility, and speed will affect your language decision.

If you are working in industry, you may be given a specification from your client or your supervisors. As you design the software needed to meet the specification, you may use UML diagrams to map out the classes and interactions you plan to implement in your program.

There are many questions you may ask yourself as you design your solution. Will your solution be compatible with the system and future changes made in the system? Is it easy to add features to your solution? Does your solution deal gracefully with mistakes in user input or external hardware and software failures? Is your code easy to maintain, particularly by future programmers who were not involved in its initial development. Are the components of the system modular? Can they be worked on, tested, and upgraded independently? Are the components of your system designed well enough to be reused for other applications? Are the elements of your system secure from malicious attacks? Finally, is it easy for the user to work with your software?

Each one of these questions is related to a separate sub-field in software engineering. It may be impossible to address them all completely, but different applications will have different priorities.

One method for OO software engineering uses *design patterns*. The idea behind design patterns is that most classes share some common design principles with a large category of classes. By naming and recognizing each category, you can apply the same rules to designing new classes from a category you are already familiar with. Each category is called a design pattern. Java uses design patterns extensively in its API. Describing design patterns in greater depth is beyond the scope of this book, but you may want to consult the Gang of Four's excellent book *Design Patterns*.

Another important idea in design is *design by contract*. Although this is also a rich, complex area of software engineering, the idea can be applied to methods in a straightforward way. For each method, you have a formal explanation of what its input should be, what its output should be, and what else can be changed in the process. For some languages and some segments of code, it is possible to prove that a given method does exactly what it is supposed to do. Nevertheless, Donald Knuth, a giant in computer science, is famous for having said, “Beware of bugs in the above code; I have only proved it correct, not tried it.”

16.6.2 Implementation

When the time comes to actually implement your design, there are a number of other techniques you can use to minimize errors in this phase. One interesting technique is *pair programming*, in which two programmers sit at a single computer and work together. Ideally, one programmer is thinking about the immediate problems posed by the next few lines of code while the other is thinking about the larger context of the program. Two sets of eyes is always beneficial when looking at something as detailed and confusing as a computer program.

In keeping with the theme of having more than one set of eyes looking at a program, it is generally recognized that it is useful to have the individuals who test the software be independent from those who develop it. By keeping the testers separate, they are not infected by the assumptions and biases that the developers have made while writing the software. Some communication between the two groups is necessary, but there is a lot of value in black box testing, which we will explain in the next subsection.

Another piece of general advice is to rely on standard libraries as much as possible. Reinventing your own libraries is partly a waste of time and partly dangerous because your own libraries have not undergone as much testing as the standard ones. Likewise, it makes your code less portable. Some expert developers may need to write special libraries for speed or memory efficiency, but they are the exception, not the rule.

There are a number of Java specific implementation guidelines. People have written entire books about good software engineering in Java, and so we will only give a few obvious pointers.

Although it is tempting to do so when working under time pressure, never write empty exception handlers. Doing so swallows exceptions blindly, giving the user no idea what the errors in his or her program are. By the same token, always make your exception handlers as narrow as possible. Simply putting a `catch(Exception e)` at the end of any `try`-block has one of two possible outcomes: In one case your handler is vague and the user is informed that a general error of some kind has occurred. In the other your handler is more precise than it has a right to be. You might have assumed that a file I/O error was most likely to occur and always report that failure. Instead, an `ArrayOutOfBoundsException` could happen and be mistakenly reported as a file I/O problem.

You should test the input to any public methods you write and throw a pre-determined exception if the input is invalid. Never use assertions to test input to public methods. In fact, you should never depend on assertions to catch errors since they must be turned on in the JVM to have effect. Assertions are great for debugging code before it is released but have little or no value in the field.

16.6.3 Testing

Once you have designed and implemented your program (or perhaps even during the process of implementation), you should test it to see if it behaves as expected and required. The most common form of software testing done by students is a form of a *smoke test*. A smoke test is a basic test of functionality. Such a test should simply run through the major features of a program and verify that they seem to work under ordinary circumstances. Often a student will barely finish the program before the deadline and be unable to perform anything but the most basic tests.

Smoke tests are useful because it is pointless to test the finer details of a system that is clearly broken, but the software engineering industry uses many other kinds of testing to ensure that a given piece of software meets its specification. We will briefly cover three broad areas of testing: black box testing, white box testing, and regression testing.

Black-box testing

Black box testing assumes that the tester knows nothing about the internal mechanisms of the software he or she is testing. The software is viewed as a “black box” that only has inputs and outputs. The tester chooses some subset of the possible inputs and tests to see if the output matches the specification.

For simple programs with very little input, it may be possible to test **all** possible input values, but doing so is impractical for most programs. A short list of techniques for determining the appropriate set of input values for black box testing follows.

Equivalence Partitioning: The idea behind equivalence partitioning is that large ranges of data may be functionally equivalent from the point of view of causing errors. If a tester can run a test for one element from a range of data, then the entire range can be tested quickly. To perform this kind of testing, the tester must partition data into ranges that function differently. The partition created is usually not really a partition in a mathematical sense as the sub-domains are overlapping. This is one reason why equivalence partitioning is also referred to as *subdomain* testing.

For example, a program controlling the temperature of the water in an aquarium may have legal input ranges between 32F and 212F. However, if the program warms the water when it is below 75F and cools it when it is above 90F, then values below 0, values from 0 to 74, values from 75 to 90, values from 91 to 212, and values above 212 all constitute different partitions.

Boundary Value Analysis: Once inputs have been partitioned into equivalent ranges, testers can focus on those values which are near the boundary of those ranges. For example, an input containing a person’s age may be allowed to range between 0 and 150. The values -1, 0, 1, 149, 150, and 151 are good candidates for input from the perspective of boundary value analysis. As

with equivalence partitioning, boundary value analysis is useful not only for the boundaries between valid and invalid data but also for the boundaries between any input ranges with different program behavior such as the boundaries separating the five ranges of values for the aquarium thermostat program described above.

All-Pairs Testing: Most software bugs are triggered by a single piece of input. Some harder to discover bug require two separate piece of input to have specific values at the same time before they manifest. With each increase in the number of different inputs that must each have specific values at the same time to cause a bug, the bug becomes increasingly difficult to track down but also increasingly unlikely to exist. It may be possible to test all possible values for a given input but impossible to test all possible values for all inputs at the same time. All-pairs testing is a compromise between these two extremes that tests all possible pairs of inputs.

Fuzz Testing: The concept behind fuzz testing is to use large amounts of invalid, unlikely, or random data as input to a program. Although this kind of testing is used only to test the reliability and robustness of a program receiving unexpected input, it has a number of advantages. One significant advantage of fuzz testing is that it is quick and easy to design test cases. Another is that it makes no assumptions about the program behavior, catching errors that might never occur to a human being.

White-box testing

The philosophy of white box is the opposite of black box testing. When using white box testing techniques, the tester has access to the program internals. The tester should employ techniques to test every possible path that execution can take through the code. Traversing a particular path of execution through a program is called *exercising* that path.

In order to exercise every possible path, it is necessary to force each conditional statement to be true and false on some path. Some combinations of true and false may be impossible, but, ignoring this fact, a program with n independent conditionals would require 2^n runs to test them all. Because of the large number of possible execution paths, white box testing generally tries to maximize coverage over metrics that are not quite so demanding.

Method coverage is the percentage of methods that are called by test cases at least once. Ideally, this number is 100%. Statement coverage is the percentage of statements that are executed by test cases. Again, this number should be as close to 100% as possible. Branch coverage is the percentage of conditionals that have been executed on both their true and false branches. Getting total coverage here is difficult, but good testing can come close.

As with black box testing, equivalence partitioning and boundary value methods can be used to reduce the total number of test cases. Also, it is important to test those parts of your programs reached only in error conditions in addition to normal operation.

Regression testing

Regression testing is a form of testing that is not often necessary for student code because they are small projects. The motivating idea behind this kind of testing is that, in the act of fixing a bug or

adding a feature, existing code can be broken. Thus, even after a system has been thoroughly tested, small changes or additions require the entire system to be retested. As the size of a program grows, the chance of unintended consequences increases, along with the value of performing regression testing.

Regression testing can incorporate both black and white box testing. Doing regression testing could simply mean running all the existing tests over again. At the very minimum, each time a test uncovers a bug, that test should be added to the test suite used after each build of the program. The use of regression testing also implies that regular testing is being done on your code. Regular testing gives developers the opportunity to track changes in other aspects of their program such as memory usage, run time, and other non-functional issues.

16.7 Syntax: Java testing tools

There is an open-source tool for testing Java called JUnit testing. There are other testing tools for Java, and there are a wide array of tools for testing software in virtually any language. We cover JUnit here because it is widely accepted as a standard Java testing tool and because it is open-source. First, we'll explain how to use JUnit, and then we'll discuss some of the tools available to help test concurrent software.

16.7.1 JUnit testing

JUnit testing is used for unit testing Java. Unit testing is the process testing of separate software components that will eventually work together. By testing them individually, debugging can be done before interactions between different components make it more difficult to find the underlying bug. After unit testing comes integration testing to test how the components work together. Finally, system testing is the testing of the complete, integrated system against its specifications.

Annotations

Our coverage of JUnit testing is based on JUnit 4. This version of JUnit simplifies the syntax of creating JUnit tests, but it also relies on *annotations*. Annotations are additional information written into Java code that affects how the compiler or run-time system treats the code. They are like comments, but they can affect code execution or compilation, though usually indirectly. Applying an annotation to a method is called *decorating*. A class, a method, a variable, a package, or even an individual method parameter can be decorated.

Three annotations are built into the language: `@Deprecated`, `@Override`, and `@SuppressWarnings`. If a method is decorated with `@Deprecated`, it is deprecated and included only for backwards compatibility. The compiler will give a warning if you call deprecated code such as the following:

```
@Deprecated  
public void oldMethod () {  
    ...  
}
```

Many methods in the extensive Java API are deprecated, like `Thread.suspend()` due to its inherent deadlock risk. As of Java 5 when annotations were introduced, these methods were all decorated with `@Deprecated`. The `@Override` annotation marks a method that is overriding superclass method, causing a compiler error if the method is not correctly overriding some superclass method. The `@SuppressWarnings` annotation allows certain warning messages to be suppressed, like using deprecated code if you really have to.

Basic JUnit syntax

First of all, JUnit is not a part of the standard Java API. To use it, you should download the latest jar file from <http://www.junit.org> and add the path to that jar file to your class path. To access the JUnit facilities in your code, you need the following import.

```
import org.junit.*;
```

Then, you need to set up a testing class just like you would any other class. The key difference is that each method in the testing class is designed to test some functionality of a code component. For example, let's imagine that we want to test certain functionality of the Java Math library such as the `ceil()`, `pow()`, and `sin()` methods.

To do so, we create a class called `MathTest` with three methods inside of it called `ceil()`, `pow()`, and `sin()`. We will use each method to test the functionality of the three methods that, respectively, have the same names. There is no requirement to name the methods any particular way. Tests in JUnit do not have to test single method calls. They could test any functional aspect of an object or class. Nevertheless, for documentation reasons it is wise to give the test methods names that reflect what is being tested.

So, where do annotations come in? The header for the `ceil()` method would be as follows.

```
@Test  
public void ceil ()
```

The only thing necessary to use a method in a JUnit test is to annotate it with `@Test`. It is also necessary to make any function used for testing `public` with a `void` return type and no parameters. Otherwise, the JUnit framework will crash when you try to run the tests. Each method with a `@Test` annotation is run once by JUnit, but JUnit cannot supply any arguments to them. They should be self-contained tests without any outside input.

The exception to this rule is that you can perform some set up for the tests and then some clean up afterwards. Any method decorated with `@Before` will be run before **every** test, and any method decorated with `@After` will be run after **every** test. If you have some set up or clean up that is expensive to run, you can use the annotations `@BeforeClass` or `@AfterClass` to decorate a static method that is run once before or after all the tests.

So far we have talked about the major aspects of writing a JUnit test class except for the actual test. How does the JUnit test report a success or a failure to the tester? As you would expect in Java, we use the exception handling mechanism to indicate failures. If the test method returns normally, the test is considered a success. If an unhandled exception or error is thrown by the method, the test is

considered a failure. One of the most common ways of implementing this is by using a form of assertions.

Of course, you could simply add an assert into the test code, then enable assertions while running the test, but this approach means that your tests could all incorrectly pass if you forget to enable assertions. Instead, use the following import.

```
import static org.junit.Assert.*;
```

With this static import, you will have access to many static methods that provide useful assertion functionality. The simplest of these is `assertTrue()`, which is essentially equivalent to an assert without requiring assertions to be enabled. For example, we could code the body of the `ceil()` test method as follows.

```
@Test
public void ceil() {
    assertTrue ( 4 == Math.ceil (3.1) );
}
```

Another useful method is `assertEquals()` (and its close cousin `assertArrayEquals()`) which takes two parameters and throws an `AssertionError` if the two are not equal. There are overloaded versions of this method for `long` and `Object` types. Note that the preferred `assertEquals()` method for the `double` type takes three parameters, including an epsilon threshold in case the values don't match exactly.

Example 16.12: JUnit math testing

Using these methods, we can finally write a complete (though very simple) implementation of `MathTest.java`.

Program 16.1: A simple testing suite. (`MathTest.java`)

```
1 import org.junit.*;
2 import static org.junit.Assert.*;
3
4 public class MathTest {
5     private static double sqrt2 ;
6
7     @BeforeClass
8     public static void setUp () { sqrt2 = Math.sqrt (2) ;}
9
10    @Test
11    public void ceil () {
12        assertTrue ( 4 == Math.ceil (3.1) );
13    }
14
15    @Test
16    public void pow () {
```

```

17     assertEquals ( 2, Math.pow ( sqrt2, 2 ), 0.000001 );
18 }
19
20 @Test
21 public void sin () {
22     assertEquals ( sqrt2 /2.0, Math.sin ( Math.PI / 4.0 ), 0.000001 );
23 }
24 }
```

Note that the `setUp()` method is extremely trivial here, and no clean up is needed. JUnit has many other powerful features that allow you to run suites of tests or repeated tests with specific parameterized values, but we are only going to introduce one more feature here. In an ideal world, you are actually developing tests as you develop code. Sometimes, in fact, you might have completed a test for a specific feature before you have finished implementing it. Or, perhaps a feature in your program is broken at the moment, but you want to continue running tests on the rest of the features. ■

In these cases and others, it is useful to turn off a particular test temporarily. To do this, you add the annotation `@Ignore` before the `@Test` annotation. In parentheses after the `@Ignore` annotation, you should ideally put in parentheses a String giving the reason why the test is being ignored.

Running JUnit

Once you have created your JUnit test classes, you will want to run them. There are tools built into IDE's like Eclipse to make this easier, but the command line is always an option. As we said before, you need to include the JUnit jar file in your classpath. You can either do this permanently, by adding it to a CLASSPATH environment variable in a way dependent on your OS, or for a particular run of a Java tool. Assuming that you haven't added the jar file to your classpath permanently, let's say that you are using JUnit 4.5 from a jar file called `junit-4.5.jar` that can be found in `C:\Utilities\Java\JUnit`. To compile `MathTest`, you would type:

```
javac -classpath .;C:\Utilities\Java\JUnit\junit-4.5.jar MathTest.java}
```

To actually run the code, you still need to include `junit-4.5.jar` in your classpath, but you also need to invoke the test runner called `org.junit.runner.JUnitCore` as follows:

```
java -classpath .;C:\Utilities\Java\JUnit\junit-4.5.jar
    org.junit.runner.JUnitCore MathTest
```

If you had multiple test classes, you could just list them all after `org.junit.runner.JUnitCore` and all the methods marked `@Test` in them would be run. Note that methods without an `@Test` decoration will not be run as tests, but there is no rule against having them. In this way, you can use un-decorated methods as helper methods for your test methods.

When you run JUnit tests, you should see the version number of JUnit used, a dot printed out for each test, the amount of time taken, and then something like OK (3 tests) if all of your tests ran without an error. If there is an error, it will list each error, probably with a long stack trace showing the

propagation of the exception.

16.8 Concurrency: Testing tools

In this section, we describe some tools that exist specifically to help you catch those bugs that are present as a direct result of concurrency. You have probably noticed that this section is quite short, and that shortness reflects the shortage of good tools available. The design of concurrent debugging and testing tools is still an open research topic. As always, the nondeterminism of concurrency makes bugs difficult to pin down. You could run a JUnit test 1,000,000 times and never see a peculiar race condition bug. From a brute force perspective, we could try to test all possible interleavings of thread execution, but this approach is not practical for large programs because the number of interleavings grows exponentially. Nevertheless, some research has focused on attacking the problem from this direction.

16.8.1 ConTest

One tool that uses this idea is ConTest from IBM. The way the JVM normally works makes some interleavings more likely than others. If the correct output is very likely and the incorrect is very unlikely, it is easy for you to believe that your program works correctly. ConTest is a tool that *instruments* class files after they have been compiled by Java. When it instruments these files, it adds extra method calls into concurrent code designed to introduce some randomness into the system. By introducing `sleep()` and `yield()` methods in random places, the JVM can be forced into producing interleavings that would otherwise be unusual. The designers of ConTest have used heuristics so that ConTest adds this randomness in “smart” locations designed to maximize unusual interleavings and catch bugs.

ConTest is not a panacea. Although it can reveal bugs that are very rare, it still must be combined with strong testing methodologies so that those bugs can be caught when they appear. Another difficulty with using ConTest is that it cannot tell you where the problem happened or when it is likely to happen under normal circumstances. You are still dependent on your test design to reveal the source of the problem. Finally, ConTest cannot guarantee every possible ordering. Very rare bugs may not manifest even after thousands of runs with ConTest instrumented code. For more information about ConTest, visit <http://www.ibm.com/developerworks/java/library/j-contest.html>

16.8.2 Concutest

We hope we have convinced you of the value of using JUnit testing to unit test your programs. Of course, JUnit has several limitations when it comes to concurrent programs. JUnit uses exceptions to report failed test cases. Unfortunately, JUnit only reports exceptions from the main thread, not from any child threads that may be spawned. ConcJUnit allows exceptions thrown by child threads to be reported and also forces all child threads to join with the main thread.

In this way, it will be clear if any errors happened while a child thread was being executed, either causing an exception to be thrown or causing a child thread to fail to rejoin the main thread.

ConcJUnit is part of a larger suite of tools called **Concutest** maintained at

<http://www.cs.rice.edu/~mgricken/research/concuteest/>. Concuteest includes Thread Checker, a tool that allows programmers to use annotations to test thread invariants. The Concuteest project is also developing a Schedule-Based Execution environment to allow programmers to test programs using specific sets of concurrent interleavings.

16.8.3 Intel tools

There are many other industry tools for debugging and optimizing threaded programs. Intel produces software such as the Intel Thread Checker to find concurrent errors as well as the Intel Vtune Performance Analyzer and Intel Thread Profiler to help tune threaded programs. These products from Intel, like many concurrency tools, are focused on C/C++ and Fortran platforms. Historically concurrency has been centered in the high performance and scientific computing markets. Java, in contrast, has been perceived as a slow language, more suited for desktop applications. As the role of concurrency continues to evolve, so will the tools to help programmers.

16.9 Examples: Testing a class

The larger the system, the more critical testing becomes. We do not have the space to give or explain a complex testing example, but we can give you another example of JUnit testing.

Example 16.13: A broken class

We are going to rely on an example from physics and create a PointCharge class that has a certain charge and a specific location in 3D space. We are also going to introduce some errors into the class. Because the class is so simple, the errors should be obvious. Nevertheless, we have picked errors that are reasonable to make.

Program 16.2: Example physics class with errors. (PointCharge.java)

```
1 public class PointCharge {  
2     private double charge ; // C  
3     private double x; // m  
4     private double y; // m  
5     private double z; // m  
6     public static final double K = 8.9875517873681764 e9; //N m^2 C^-2  
7  
8     public PointCharge ( double charge, double x, double y, double z) {  
9         this.charge = charge ;  
10        this.x = x;  
11        this.y = y;  
12        this.z = y;  
13    }  
14  
15    public double getCharge () { return charge ;}  
16}
```

```

17 public double distance ( PointCharge p ) {
18     return distance ( p.x, p.y, p.z );
19 }
20
21 private double distance ( double x, double y, double z ) {
22     double deltaX = this.x - x;
23     double deltaY = this.y - y;
24     double deltaZ = this.z - z;
25     return Math.sqrt ( deltaX * deltaX + deltaY * deltaY + deltaZ * deltaZ );
26 }
27
28 public double scalarForce ( PointCharge p ) {
29     double r = distance ( p );
30     return K* charge *p. charge /r*r;
31 }
32
33 public double fieldMagnitude ( double x, double y, double z ) {
34     double r = distance ( x, y, z );
35     return charge /(r*r);
36 }
37 }

```

The PointCharge class has the expected constructor and then a method to determine distance to another PointCharge. This method in turn relies on a private helper method that can compute distance to an arbitrary x , y , and z location. Finally, the “important” work done by the class are in determining the scalar force between two charges and the magnitude of the electric field due to the charge at some location. Recall from physics that the force F between two charges q_1 and q_2 is $k_e \frac{q_1 q_2}{r^2}$ where k_e is the proportionality constant $8.9875517873681764 \times 10^9 \text{ N}\cdot\text{m}^2 \text{ C}^{-2}$ and r is the distance between the charges. Likewise the electric field E at a given location due to a charge q is $k_e \frac{q}{r^2}$. ■

Example 16.14: Testing the distance methods

Let’s come up with a test for the distance() methods first. We’re going to need some other PointCharges. Let’s make 4 altogether: one at the origin and three one meter along each positive axis. We can create these charges in a set up method. While we’re at it, we’ll give them a variety of positive and negative charges.

```

@Before
public void setUp () {
    charge1 = new PointCharge ( 1, 0, 0, 0 );
    charge2 = new PointCharge ( 2, 1, 0, 0 );
    charge3 = new PointCharge ( -1, 0, 1, 0 );
    charge4 = new PointCharge ( 0, 0, 0, 1 );
}

```

To test the distance() method thoroughly, we will check the distance from charge1 to all the other charges as well as charge2 to charge3.

```
@Test  
public void distance () {  
    assertEquals ( 1.0, charge1.distance ( charge2 ), 0.001 );  
    assertEquals ( 1.0, charge1.distance ( charge3 ), 0.001 );  
    assertEquals ( 1.0, charge1.distance ( charge4 ), 0.001 );  
    assertEquals ( Math.sqrt ( 2.0 ), charge2.distance ( charge3 ), 0.001 );  
}
```

The distances between charge1 and the other three should be 1, and the distance between charge2 and charge3 should be about $\sqrt{2}$. Yet, when we run this test with JUnit, the test fails. We get:

java.lang.AssertionError: expected:<1.0> but was:<1.4142135623730951>

for the second assertion in the method. But why? If we comb through the distance() methods in PointCharge, they all look correct. The problem must be deeper. PointCharge does not have accessor methods for its location, and so we can't test those. Checking the constructor, we find the culprit: `this.z = y;`, a simple cut and paste error.

With the distance() methods working, we can test other things. We are going to run a similar test for scalarForce() generated by plugging in appropriate values into the equation for F .

```
@Test  
public void scalarForce () {  
    assertEquals ( 2 * PointCharge.K, charge1.scalarForce ( charge2 ),  
        0.001 );  
    assertEquals ( - PointCharge.K, charge1.scalarForce ( charge3 ),  
        0.001 );  
    assertEquals ( 0.0, charge1.scalarForce ( charge4 ), 0.000001 );  
    assertEquals ( - PointCharge.K, charge2.scalarForce ( charge3 ),  
        0.001 );  
}
```

When we run this test with JUnit, the last assertion fails. We get the following output.

```
java.lang.AssertionError: expected:<-8.987551787368176E9>  
but was:<-1.797510357473635E10>
```

A close inspection reveals that the actual value is about twice the expected value. Where does this extra factor of 2 come from? Scanning the code for scalarForce(), we find `return K*charge*p.charge/r*r;`

We forgot parentheses and messed up our equation. What we really wanted was `return K*charge*p.charge/`

The most striking thing about this example is that three test cases passed! Perhaps that means that

we were choosing values that were too simple, but it also illustrates the importance of serious testing. ■

Example 16.15: Testing the field magnitude method

Finally, let's test the value of the fieldMagnitude() method. For simplicity, we'll test the field at the locations of charge1, charge3, and charge4 with respect to charge2.

This time the first assertion fails. We get the following output.

```
java.lang.AssertionError: expected:<1.797510357473635E10> but was:<2.0>
```

2.0 seems like a very strange result when we were expecting a value with an order of magnitude 10 times larger. Perhaps the constant was omitted? Yes, our version of fieldMagnitude() left off a factor of K. Once we fix that, our code finally produces the OK (3 tests) we have been waiting to see from JUnit. Why didn't we fail the assertions after the first one? Because of the exception handling mechanism, each JUnit test method stops once a failure has happened. ■

Here is the fully corrected version of PointCharge renamed FixedPointCharge.

Program 16.3: Corrected version of PointCharge. (FixedPointCharge.java)

```
1 public class FixedPointCharge {  
2     private double charge; // C  
3     private double x; // m  
4     private double y; // m  
5     private double z; // m  
6     public static final double K = 8.9875517873681764e9; // N m^2 C^-2  
7  
8     public FixedPointCharge ( double charge, double x, double y, double z ) {  
9         this.charge = charge;  
10        this.x = x;  
11        this.y = y;  
12        this.z = z;  
13    }  
14  
15    public double getCharge () { return charge; }  
16  
17    public double distance ( FixedPointCharge p ) {  
18        return distance ( p.x, p.y, p.z );  
19    }  
20  
21    private double distance ( double x, double y, double z ) {  
22        double deltaX = this.x - x;  
23        double deltaY = this.y - y;  
24        double deltaZ = this.z - z;  
25        return Math.sqrt ( deltaX * deltaX + deltaY * deltaY + deltaZ * deltaZ );  
}
```

```

26     }
27
28     public double scalarForce ( PointCharge p ) {
29         double r = distance ( p );
30         return K* charge *p. charge /(r*r);
31     }
32
33     public double fieldMagnitude ( double x, double y, double z ) {
34         double r = distance ( x, y, z );
35         return K* charge /(r*r);
36     }
37 }
```

And, for easy readability, here is the full JUnit test class TestPointCharge. Note that you will have to change the name PointCharge to FixedPointCharge if you want to test the corrected class.

Program 16.4: Class for testing PointCharge. (TestPointCharge.java)

```

1 import org.junit.*;
2 import static org.junit.Assert.*;
3
4 public class TestPointCharge {
5     private PointCharge charge1 ;
6     private PointCharge charge2 ;
7     private PointCharge charge3 ;
8     private PointCharge charge4 ;
9
10    @Before
11    public void setUp () {
12        charge1 = new PointCharge ( 1, 0, 0, 0 );
13        charge2 = new PointCharge ( 2, 1, 0, 0 );
14        charge3 = new PointCharge ( -1, 0, 1, 0 );
15        charge4 = new PointCharge ( 0, 0, 0, 1 );
16    }
17
18    @Test
19    public void distance () {
20        assertEquals ( 1.0, charge1.distance ( charge2 ), 0.001 );
21        assertEquals ( 1.0, charge1.distance ( charge3 ), 0.001 );
22        assertEquals ( 1.0, charge1.distance ( charge4 ), 0.001 );
23        assertEquals ( Math.sqrt ( 2.0 ), charge2.distance ( charge3 ), 0.001 );
24    }
25
26    @Test
```

```

27 public void scalarForce () {
28     assertEquals ( 2* PointCharge.K, charge1.scalarForce ( charge2 ), 0.001 );
29     assertEquals ( - PointCharge.K, charge1.scalarForce ( charge3 ), 0.001 );
30     assertEquals ( 0.0, charge1.scalarForce ( charge4 ), 0.001 );
31     assertEquals ( ( double )- PointCharge.K, ( double ) charge2.scalarForce ( charge3 ), 0.001 );
32 }
33
34 @Test
35 public void fieldMagnitude () {
36     assertEquals ( 2* PointCharge.K, charge2.fieldMagnitude ( 0, 0, 0 ), 0.001 );
37     assertEquals ( PointCharge.K, charge2.fieldMagnitude ( 0, 1, 0 ), 0.001 );
38     assertEquals ( PointCharge.K, charge2.fieldMagnitude ( 0, 0, 1 ), 0.001 );
39 }
40 }

```

Exercises

Conceptual Problems

16.1 What is the purpose of the assert keyword in Java? What steps must be taken for it to be active?

16.2 What is the value of j after the following statements are executed?

```

int j = 1;
int i;
for ( i = 0; i < 10; i++ );
    j += i;

```

Assuming the programmer made an error, what category of programming error does it fall under?

16.3 The following loop is intended to print out all possible byte values. What is the conceptual error made in the following loop? How many times will it execute?

```

for ( byte value = 0; value < 256; ++value )
    System.out.println ("Byte :" + value);

```

16.4 What are all the possible run-time errors that could occur in this method that reverses a section of an array?

```

public void reverse ( Object [] array, int start, int end ) {
    Object temp ;
    end --; // up to but not including end
    while ( start < end ) {
        temp = array [ start ]
        array [ start ] = array [ end ];
        array [ end ] = temp;
        start++;
        end--;
    }
}

```

```

        array [ end ] = temp ;
        start++;
        end--;
    }
}

```

What checks could be added to catch these errors?

16.5 Consider the following definition of a stack that holds int values.

```

public class IntegerStack {
    private static Node {
        public int data;
        public Node next;
    }

    private Node head = null;

    public void push( int value ) {
        Node temp = new Node();
        temp.data = value;
        temp.next = head;
        head = temp;
    }

    public void pop() { head = head.next; }
    public int top() { return head.data; }
    public boolean empty() { return head == null; }
}

```

What exceptions could be thrown when using this class? Where could they be thrown?

16.6 This question sometimes comes up in job interviews. Imagine that you have a simple singly linked list such as the one described in [Chapter 18](#). What if there is a loop in the list such that the last element in the list points to an earlier element in the list? For this reason, a simple traversal of the list will go on forever. How could you detect such a problem during program execution?

16.7 What is the difference between black box testing and white box testing? What kinds of bugs are more likely to be caught by black box testing? By white box testing?

16.8 The Microsoft Zune is a portable media player in competition with the Apple iPod. The first generation Zune 30 received negative publicity because many of them froze on December 31, 2008 due to a leap year bug. It is possible to find segments of the source code that caused this problem on the Internet. Essentially, the clock code for the Zune behaved correctly on any day of the year numbered 365 or lower. Likewise, when the day was greater than 366, it would

correctly move to the next year and reset the day counter. When day was exactly 366, however, the Zune became stuck in an infinite loop. What kind of testing should Microsoft have done to prevent this bug?

Programming Practice

16.9 Apply JUnit testing to the last major assignment you did in class. What bugs did you uncover?

Experiments

16.10 James Gosling's original specification for Java contained assertions, but they were not included until Java 1.4. One of the concerns about an assertion mechanism is the additional time required to process the assertions. Time a program of at least moderate length before adding assert statements to its methods. If you use assert statements to check method input and output thoroughly, you should see a slight decrease in performance when assertions are enabled. When disabled, you should see almost none. How great is the performance hit?

16.11 Take another look at your last programming assignment. Calculate the number of branches based on `if` and `switch` statements and compute 2 raised to that power. Time your program executing once under normal circumstances. Multiply that time by the number of different possibilities you would need to exercise every possible combination of branches in your program. How long would it take?

16.12 Take a concurrent program you have written that relies on explicit synchronization mechanisms for correctness. Remove all synchronization tools and run the code many times, testing for race conditions. Then, instrument the code with ConTest and run it many more times. Do you see increased variety in output with ConTest? Was it easier to find race conditions?

Chapter 17

Polymorphism

... how strange it is to be anything at all.

—Neutral Milk Hotel

17.1 Problem: Banking account with a vengeance

In Chapter 14, we introduced the SynchronizedAccount class that guarantees that checking the balance, making deposits, and making withdrawals will all be safe even in a multi-threaded environment. Unfortunately, SynchronizedAccount gives very few of the options a full bank account should have. The problem we present to you now is to create an entire line of bank accounts which all inherit from SynchronizedAccount. Because of inheritance, all accounts will at least have getBalance(), deposit(), and withdraw() methods.

You must create three new account classes. The constructor for each class must take a String which gives the name of the person opening the account and **double** which gives the starting balance of the account. The first of these classes is CheckingAccount. The rules for the checking account implemented by this class are simply that the customer is charged \$10 every month that the account is open. The second class is DirectDepositAccount.

This account is very similar to the basic checking account except that an additional method directDeposit() has been added. On its own, directDeposit() appears to operate like deposit(); however, if a direct deposit has been made in the last month, no service fee will be charged to the account.

The SavingsAccount class operates somewhat differently. In addition to a name and a starting balance, the constructor for a SavingsAccount takes a **double** which gives the annual interest rate the account earns. Each month the balance is checked. If the balance of the account is greater than \$0, the account earns interest corresponding to $\frac{1}{12}$ of the annual rate. However, if the balance is below \$1000, a \$25 service fee is charged each month, regardless of how low the balance becomes.

In Chapter 11 you were exposed to concepts in inheritance. We are now returning to these concepts and exploring them further. In the first place, concurrency is on the table now, and you must be careful to keep your derived classes thread-safe. In the second, we will discuss the full breadth of inheritance. The tools we discuss are intended to allow you to solve this extended bank account problem and indeed many other problems with as little code as possible.

17.2 Concepts: Polymorphism

Perhaps the most important use for inheritance is code reuse. When you can successfully reuse existing code, you are not just saving the time of writing new code: You are also leveraging the quality and correctness of the existing code. For example, when you create your own class which

extends Thread, you are confident that all the thread mechanisms work properly.

You can reuse code by taking a class that does something you like, say the Racecar class, and enhance it in some way, perhaps so that it becomes the TurboRacecar class. If you use the TurboRacecar class on its own, your code reuse is through simple inheritance. If you use TurboRacecar objects with a RaceTrack class, which was written to take Racecar objects as input, you have entered the realm of *polymorphism*. Polymorphism means that the same method can be used on different types of objects without being rewritten. In Java, polymorphism works by allowing the programmer to use a derived class in any place where a base class could have been used.

17.2.1 The is-a relationship

Consider the two following class definitions:

```
1 public class Racecar {  
2     public double getTopSpeed () { return 200.0; }  
3     public int getHorsepower () { return 700; }  
4     public double speed = 0;  
5 }
```

```
1 public class TurboRacecar extends Racecar {  
2     public int getHorsepower () { return 1100; }  
3 }
```

Now, imagine that a RaceTrack has an addCar(Racecar car) method which adds a Racecar to the list of cars on the track. When the cars begin racing, the RaceTrack object will query the cars to see how much horsepower they have. A Racecar object will return 700 when getHorsepower() is called, but a TurboRacecar will return 1100.

Even though the TurboRacecar does not have an explicit getTopSpeed() method, it inherits one from Racecar. Like all derived classes in Java, TurboRacecar has all the methods and fields that Racecar does. This relationship is called an *is-a* relationship because every TurboRacecar is a Racecar in the sense that you can use a TurboRacecar whenever a Racecar is required.

17.2.2 Dynamic binding

There is a little bit of magic that makes polymorphism work. When you compile your code, the RaceTrack doesn't know which getHorsePower() method will eventually get called. Only at run time does it query the object in question and, if it is a TurboRacecar, use the overridden method that returns 1100. This feature of Java is called *dynamic binding*. Not every object oriented programming language supports dynamic binding. C++ actually allows the programmer to specify whether or not a method is dynamically bound.

Only methods are dynamically bound in Java. Fields are *statically bound*. Consider the following re-definitions of Racecar and TurboRacecar.

```
1 public class StaticRacecar {  
2     public static final double TOP_SPEED = 200.0;
```

```
3     public static final int HORSEPOWER = 700;  
4     public double speed = 0;  
5 }
```

```
1 public class StaticTurboRacecar extends StaticRacecar {  
2     public static final int HORSEPOWER = 1100;  
3 }
```

Assume that RaceTrack contains a method which prints out the horsepower of a StaticRacecar like so:

```
public void printHorsepower ( StaticRacecar car ) {  
    System.out.print ( car.HORSEPOWER );  
}
```

Even if you pass an object of type StaticTurboRacecar, the value 700 will be printed out every time. In Java, all fields, whether normal, **static**, or **final** are statically bound.

17.2.3 General vs. specific

Another way to look at inheritance is as a statement of a specialization and generalization. A TurboRacecar is a specific kind of Racecar while a Racecar is a general category that TurboRacecar belongs to.

The rules of Java say that you can always use a more specific version of a class than you need but never a more general one. You can use a TurboRacecar any time you need a Racecar but not the reverse. A square will do the job of a rectangle, but a rectangle will not always be suitable when a square is needed.

Consider the following two classes:

```
1 public class Vehicle {  
2     public void travel ( String destination ) {  
3         System.out.println (" Traveling to " + destination + "!" );  
4     }  
5 }
```

```
1 public class RocketShip extends Vehicle {  
2     public void blastOff () {  
3         System.out.println (" FOOOOM !");  
4     }  
5 }
```

Here is a method that requires a RocketShip but only uses its travel() method.

```
public void takeVacation ( RocketShip ship, String destination ) {  
    ship.travel ( destination );  
}
```

It seems as though we should be able to pass any Vehicle to the takeVacation() method because the only method in ship used by takeVacation() is the travel() method. However, the programmer specified that the parameter should be a RocketShip, and Java plays it safe. Just because it looks like there won't be a problem, Java isn't going to take any chances on passing an overly general Vehicle when a RocketShip is required. If Java took chances, a problem could arise if the takeVacation() method was overridden by a method that did call ship.blastOff().

In summary, you can pass a RocketShip to a method which takes a Vehicle or store a RocketShip in an array of Vehicles, but not the reverse. Java usually gives a compile time error if you try to put something too general into a location that is too specific. There are, however, some situations which are so tricky that Java doesn't catch the error until runtime. Arrays, specifically, can cause problems. Examine the following code snippet:

```
Vehicle [] transportation = new RocketShip [100];
transportation [0] = new Vehicle ();
```

On the first line, we are using a Vehicle array reference to store an array of 100 RocketShips. But in the second line, we try to store a Vehicle into an array that is really a RocketShip array, even though it looks to the compiler like a Vehicle array. Doing so will compile but throw an ArrayStoreException at runtime.

17.3 Syntax: Inheritance tools in Java

So far, we have described polymorphism in Java with a conceptual focus. In our previous examples, the only language tool needed to use polymorphism was the **extends** keyword which you are well familiar with by now. There are a number of other tools designed to help you structure class hierarchies and enforce design decisions.

17.3.1 Abstract classes and methods

One such tool is abstract classes. An abstract class is one which can never be instantiated. In order to use an abstract class, it is necessary to derive a class from it. To create an abstract class, you just need to add the **abstract** keyword to its definition, as in the following example.

```
1 public abstract class Useless {
2     protected int variable ;
3
4     public Useless ( int input ) {
5         variable = input ;
6     }
7
8     public void print () {
9         System.out.println ( variable );
10    }
11 }
```

This class is useless for a number of reasons. For one thing, there's no way to find out the value of variable except by printing it out. Furthermore, there's no way to change the value of variable after the object has been created. Finally, since an abstract class cannot be instantiated, the following code snippet will not compile.

```
Useless thing = new Useless ( 14 );
```

Instead, we must create a new class that extends Useless.

```
1 public class Useful extends Useless {  
2     public int getVariable () { return variable ; }  
3  
4     public void setVariable ( int value ) {  
5         variable = value ;  
6     }  
7 }
```

Then, we can instantiate an object of type Useful and use it for something.

```
Useless item = new Useful ( 14 );  
item.print ();
```

Note that, in accordance with the rules of Java, we can store a more specific object of type Useful into more general reference of type Useless. Even though Java knows that the object it points to will never actually be a Useless object, it is perfectly legal to have a Useless reference. You can use abstract classes in this way to provide a base class with some fundamental fields and methods that all other classes in a particular hierarchy need. By using the keyword **abstract**, you are marking the class as template for other classes instead of a class that will be used directly.

Methods can be abstract as well. If you have an abstract class, you can create a method header which describes a method that all non-abstract children classes must implement, as show below.

```
1 public abstract class Sequence {  
2     protected int number ;  
3     protected final int CONSTANT ;  
4  
5     public Sequence ( int number, int constant ) {  
6         this.number = number ;  
7         CONSTANT = constant ;  
8     }  
9  
10    public abstract int getNextValue ();  
11 }
```

This abstract class is supposed to be a template for classes which can produce some sequence of numbers. Note that there is no body for the `getNextValue()` method. It simply ends with a semicolon. Every non-abstract derived class must implement a `getNextValue()` method to produce the next

number in the sequence. For example, we could implement an arithmetic or a geometric sequence as follows.

```
1 public class ArithmeticSequence extends Sequence {  
2     public abstract int getNextValue () {  
3         number += CONSTANT ;  
4         return number ;  
5     }  
6 }
```

```
1 public class GeometricSequence extends Sequence {  
2     public abstract int getNextValue () {  
3         number *= CONSTANT ;  
4         return number ;  
5     }  
6 }
```

The Sequence class does not specify **how** the sequence of numbers should be generated, but any derived class must implement the `getNextValue()` method in order to compile. By using an abstract class, we don't have to create a base class which generates a meaningless sequence of numbers just for the sake of establishing the `getNextValue()` method.

Here's a more involved example of an abstract class that gives a first step toward solving the bank account with a vengeance problem posed at the beginning of the chapter.

```
1 import java.util.Calendar;
```

The first step is to import the `Calendar` class for some date stuff we are going to use later.

```
3 public abstract class BankAccount extends SynchronizedAccount {  
4     private String name ;  
5     private Calendar lastAccess ;  
6     private int monthsPassed = 0;
```

We extend `SynchronizedAccount` and declare the new class to be abstract. In this example, we do not use any abstract methods, but, since each bank account has unique characteristics, we don't want people to be able to create a generic `BankAccount`.

```
8     public BankAccount ( String name, double balance )  
9         throws InterruptedException {  
10        this.name = name;  
11        changeBalance ( balance );  
12        lastAccess = Calendar.getInstance ();  
13    }  
14  
15    public String getName () {return name ;}  
16
```

```

17     protected Calendar getLastAccess () {return lastAccess ;}
18
19     protected int getMonthsPassed () { return monthsPassed ; }

```

The constructor and the accessors should be what you expect to see. Note that calling the static method `Calendar.getInstance()` is the correct way to get a `Calendar` object with the current date and time.

```

21     public final double getBalance () throws InterruptedException {
22         update ();
23         return super. getBalance ();
24     }
25
26     public final void deposit ( double amount )
27         throws InterruptedException {
28         update ();
29         super.deposit ( amount );
30     }
31
32     public final boolean withdraw ( double amount )
33         throws InterruptedException {
34         update ();
35         return super.withdraw ( amount );
36     }

```

Then come the balance checking and changing methods. Each simply calls the parent methods after calling an `update()` method we discuss below.

```

38     protected synchronized void update () throws
39         InterruptedException {
40         Calendar current = Calendar.getInstance ();
41         int months = 12*( current.get ( Calendar.YEAR ) -
42             getLastAccess ().get ( Calendar.YEAR ) ) +
43             ( current.get ( Calendar.MONTH ) -
44             getLastAccess ().get ( Calendar.MONTH ) );
45         if( months > 0 ) {
46             lastAccess = current ;
47             monthsPassed = months ;
48         }
49     }

```

Other than adding `String` for a name associated with the account, the `update()` method is the other major addition made in `BankAccount`. Each time `update()` is called, the number of months since the last access is stored in the field `monthsPast` and the timestamp of the last access is stored in

`lastAccess`. We didn't need these time features before, but issues like earning interest or paying monthly service charges will make them necessary. This method is synchronized so that the two fields associated with the last access are updated atomically.

17.3.2 Final classes and methods

If you look at the previous example carefully, you will notice that the methods `getBalance()`, `deposit()`, and `withdraw()` were each declared with the keyword `final`. You have seen this keyword used to declare constants before. When applied to methods, `final` is philosophically similar (and almost the opposite of `abstract`). A method which is declared `final` cannot be overridden by child classes. If you are designing a class hierarchy and you want to lock a method into doing a specific thing and never changing, this is the way to do it.

Like `abstract`, the keyword `final` can be applied to a class as well. If you want to prevent a class from being extended further, apply the `final` keyword to its definition. You may not find yourself using this feature of Java very often. It is primarily useful in situations where a large body of code has been designed to make use of a specific class. The designers of that specific class want to keep it exactly the way it is and prevent anything unexpected from happening.

The most common example of a `final` class is the `String` class. Consider the following.

```
public class SuperString extends String {}
```

This code will give a compiler error. `String` is perfect the way it is (or so the Java designers have decided). There is no reason to restrict your code arbitrarily, but use of the `final` keyword for classes, methods, and especially to specify constants allows the compiler to do some performance optimizations that would otherwise be impossible.

17.3.3 Casting

If we shift the focus back to polymorphism, we have to admit that we have avoided one of the messier issues. It's true that polymorphism gives us lots of power. For example, we can make a `Vehicle` array and store subclasses of `Vehicle` inside, like so:

```
Vehicle [] vehicles = new Vehicle [5];
vehicles [0] = new Skateboard ();
vehicles [1] = new RocketShip ();
vehicles [2] = new SteamBoat ();
vehicles [3] = new Car ();
vehicles [4] = new Skateboard ;
```

This process could be infinitely more complex. We could be reading data out of a file and dynamically creating different kinds of `Vehicle` objects. But, the final product of an array of `Vehicle` objects is the important thing. Now, we can run through the array with a loop and have the code magically work for each kind of `Vehicle`.

```
for ( int i = 0; i < vehicles. length; i++ )
    vehicles [i]. travel ( "Prague" );
```

Each Vehicle will travel to Prague as it should. The only trouble is that we have hidden some information. We know that vehicles[1] is a RocketShip, but we can't treat it like one.

```
vehicles [1].blastOff();
```

This code will not compile.

```
RocketShip ship = vehicles [1];
```

This code will not compile either. In both cases, we must use an explicit cast to tell the compiler that the object really is a RocketShip.

```
RocketShip ship = (RocketShip) vehicles [1];  
((RocketShip) vehicles [1]).blastOff();
```

Both lines of code will work. The compiler is always conservative. It never makes guesses about the type of something. For example:

```
Vehicle ship = new RocketShip();  
ship.blastOff();
```

Even though ship must be a RocketShip, Java does not assume that. The compiler uses the reference type Vehicle to do the check and will refuse to compile. Casting allows us to use our human intellect to overcome the shortsightedness of the compiler. Human intellect is unfortunately flawed. What happens if you cast improperly?

```
Vehicle vehicle = new Skateboard();  
RocketShip ship = (RocketShip) vehicle;  
ship.blastOff();
```

In this example, we are trying to cast a Skateboard into a RocketShip. At compile time, no errors will be found. Because we use explicit casting, the compiler assumes that we, powerful human beings that we are, know what we are doing. The error will happen at runtime while executing the second line. Java will try to cast vehicle into a RocketShip, fail, and throw a ClassCastException.

Java provides some additional tools to make casting easier. One of these is the `instanceof` keyword which can be used to test if an object is an instance of a particular class (or one of its derived classes). For example, we can make an object execute a special command if we know that the object is capable of it.

```
public void visitDenver ( Vehicle vehicle ) {  
    if ( vehicle instanceof RocketShip )  
        ((RocketShip) vehicle).blastOff();  
    vehicle.travel ( "Denver" );  
}
```

Even inside the if statement where it must be the case that vehicle is a RocketShip, we still must perform an explicit cast. Sometimes `instanceof` is not precise enough. If you must be sure that the

object in question is a specific class and not just one of its subclasses, you can use the `getClass()` method on any object and compare it to the static class object. Using this tool, we can rewrite the former example to be more specific.

```
public void visitDenver ( Vehicle vehicle ) {  
    if( vehicle.getClass () == RocketShip.class )  
        (( RocketShip ) vehicle ).blastOff ();  
    vehicle.travel ( "Denver" );  
}
```

This version of the code will only call `blastOff()` for objects of class `RocketShip` and not for objects of a subclass like `FusionPoweredRocketShip`.

17.3.4 Inheritance and exceptions

Beyond `ClassCastException`, there are a few other issues that come up when combining exceptions with inheritance. As you already know, an exception handler for a parent class will work for a child class. As such, when using multiple exception handlers, it is necessary to order them from most specific to most general in terms of class hierarchy.

However, there is another subtle rule that is necessary to keep polymorphism functioning smoothly. Let's consider a `Fruit` class with an `eat()` method that throws an `UnripeFruitException`.

```
public class Fruit {  
    public void eat () throws UnripeFruitException {  
        ...  
    }  
}
```

Almost any fruit can be unripe, and it is unpleasant to try to eat an unripe fruit. But, there are other things that can go wrong when eating fruit. Consider the `Plum` class derived from `Fruit`.

```
public class Plum extends Fruit {  
    public void eat () throws  
        UnripeFruitException, ChokingOnPitException {  
        ...  
    }  
}
```

In the `Plum` class, the `eat()` method has been overridden to tackle the special ways that eating a plum is different from eating fruit in general. When eating a plum, you can make a mistake and try to swallow the pit, throwing, it seems, a `ChokingOnPitException`. This scenario seems natural, but it is not allowed in Java.

The principle behind polymorphism is that a more specialized version of something can be used in place of a more general version. Indeed, if you use a `Plum` in place of a `Fruit`, calling the `eat()` method is no problem. The problem happens if a `ChokingOnPitException` is thrown. Now, code which was designed for `Fruit` objects knows nothing about a `ChokingOnPitException`, so there is no way for such

code to catch the exception and deal with the situation.

There is nothing wrong with throwing exceptions on overridden methods. The rule is that the overriding method must throw a subset (not a proper subset, so it could be all of the original exceptions) of the exceptions that the overridden method throws. This rule is actually a concept called Hoare's rule of consequence that pops up several places in programming languages. Essentially, if you start with something that works, tighten the requirements on the input (use a Plum instead of any Fruit), loosen the requirements on the output (throw fewer exceptions than were originally thrown), it will still work.

Example 17.1: More human than human

Now we have a few additional examples in a somewhat larger class hierarchy.

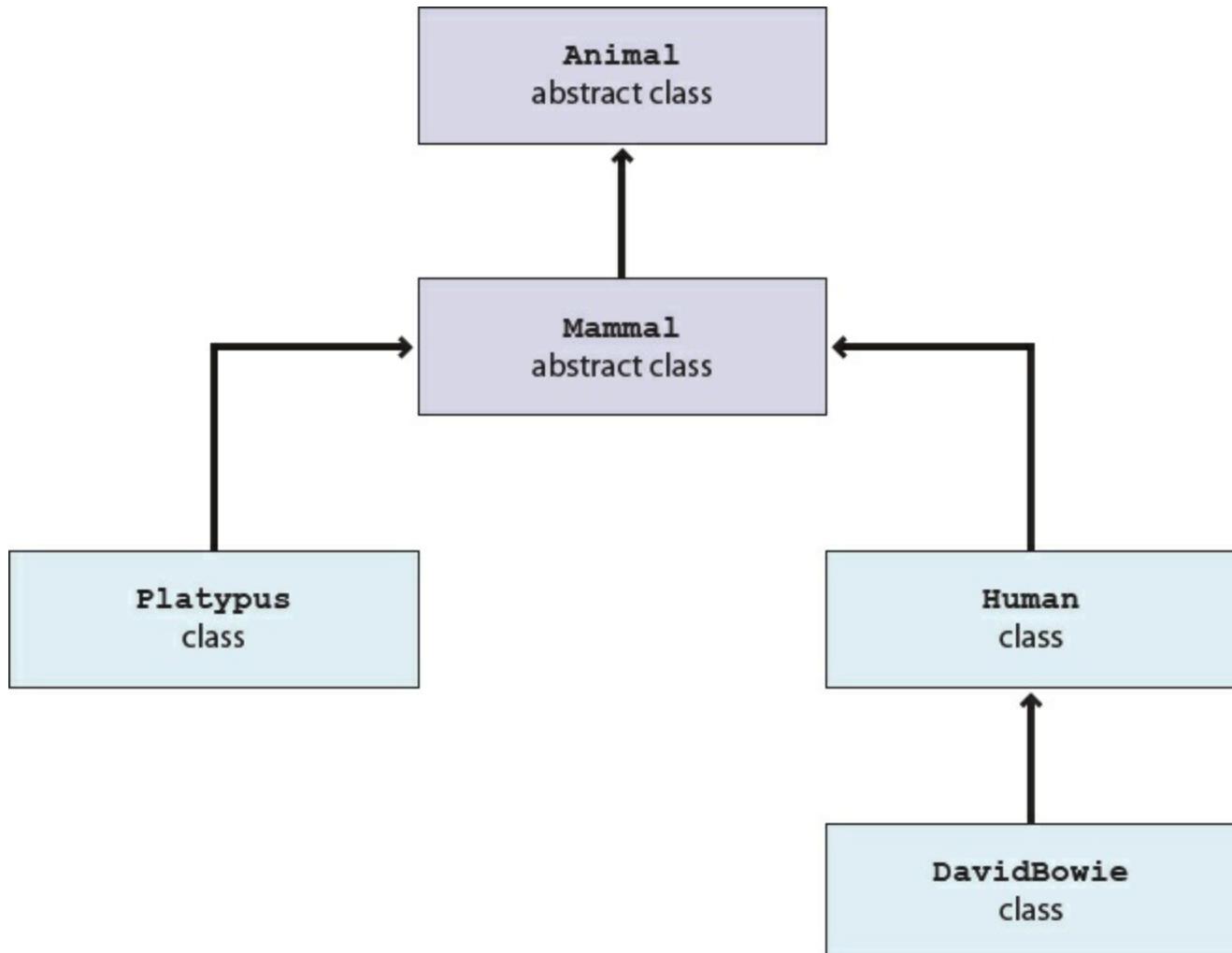


Figure 17.1: Animal class hierarchy.

```
1 public abstract class Animal {  
2     private boolean alive = true ;  
3     private boolean happy = true ;  
4     private final boolean warmblooded ;  
5  
6     public Animal( boolean warmblooded ) {  
7         this.warmblooded = warmblooded ;
```

```

8      }
9
10     public boolean isHappy () { return happy ; }
11
12     public void setHappy ( boolean value ) { happy = value ; }
13
14     public boolean isAlive () { return alive ; }
15
16     public void die () { alive = false ; }
17 }
```

We begin with the abstract Animal class. This class gives a base definition for animals which includes whether the animal is alive, whether the animal is happy, and whether it is warmblooded (declared `final` because an animal can't switch between warmblooded and coldblooded).

```

1 public abstract class Mammal extends Animal {
2     public static final boolean MALE = false ;
3     public static final boolean FEMALE = true ;
4     private final boolean gender ;
5
6     public Mammal ( boolean gender ) {
7         super ( true );
8         this.gender = gender ;
9     }
10
11    public getGender () { return gender ; }
12
13    public abstract String makeSound ();
14 }
```

We then extend Animal into Mammal. All mammals are warmblooded, which is reflected in the constructor call to the base class. In addition, it is assumed that all mammals make some sound. Mammals also have well-defined genders, declared `final` because it cannot change once it has been set. Like Animal, Mammal is an abstract class, and any non-abstract subclass of Mammal must implement `makeSound()`.

```

1 public class Platypus extends Mammal {
2     public Platypus ( boolean gender ) {
3         super ( gender );
4     }
5
6     public String makeSound () {
7         return " Quack !"
8     }
9 }
```

```

10 public Egg layEgg () {
11     if( getGender () == FEMALE )
12         return new Egg ();
13     else
14         return null ;
15 }
16
17 public void poison ( Animal victim ) {
18     if( getGender () == MALE )
19         victim.setHappy ( false );
20 }
21 }
```

The Platypus class extends Mammal and adds the unusual things that a platypus can do: laying eggs (if female) and poisoning other animals (if male).

```

1 public class Human extends Mammal {
2     public Human ( boolean gender, boolean happy ) {
3         super ( gender );
4         setHappy ( happy );
5     }
6
7     public String makeSound () {
8         return "Hello, world."
9     }
10 }
```

The Human class also extends Mammal. Depending on the problem being solved, this class might warrant a great deal more specialization. Right now the main addition is taking happiness as an argument to the constructor. Unfortunately, the default human state is not necessarily happiness.

```

1 public final class DavidBowie extends Human {
2     public DavidBowie () {
3         super ( MALE, true );
4     }
5
6     public String makeSound () {
7         return "I always had a repulsive need to be " +
8             " something more than human."
9     }
10 }
```

Finally, the DavidBowie class extends Human and is declared a final class because it really is impossible to add anything to David Bowie. ■

You will notice that our examples have stretched fairly long in this chapter. It is difficult to give

strong motivation for some aspects of inheritance and polymorphism without a large class hierarchy. These tools are designed to help organize large bodies of code and should become more useful as the size of the problem you are working on grows. One of the best examples of the success of inheritance is the Java API itself. The standard Java library is very large and depends on inheritance a great deal.

17.4 Solution: Banking account with a vengeance

Now we return to the specific problem given at the beginning of the chapter and give its solution. We have already given you the BankAccount abstract class which provides a lot of structure.

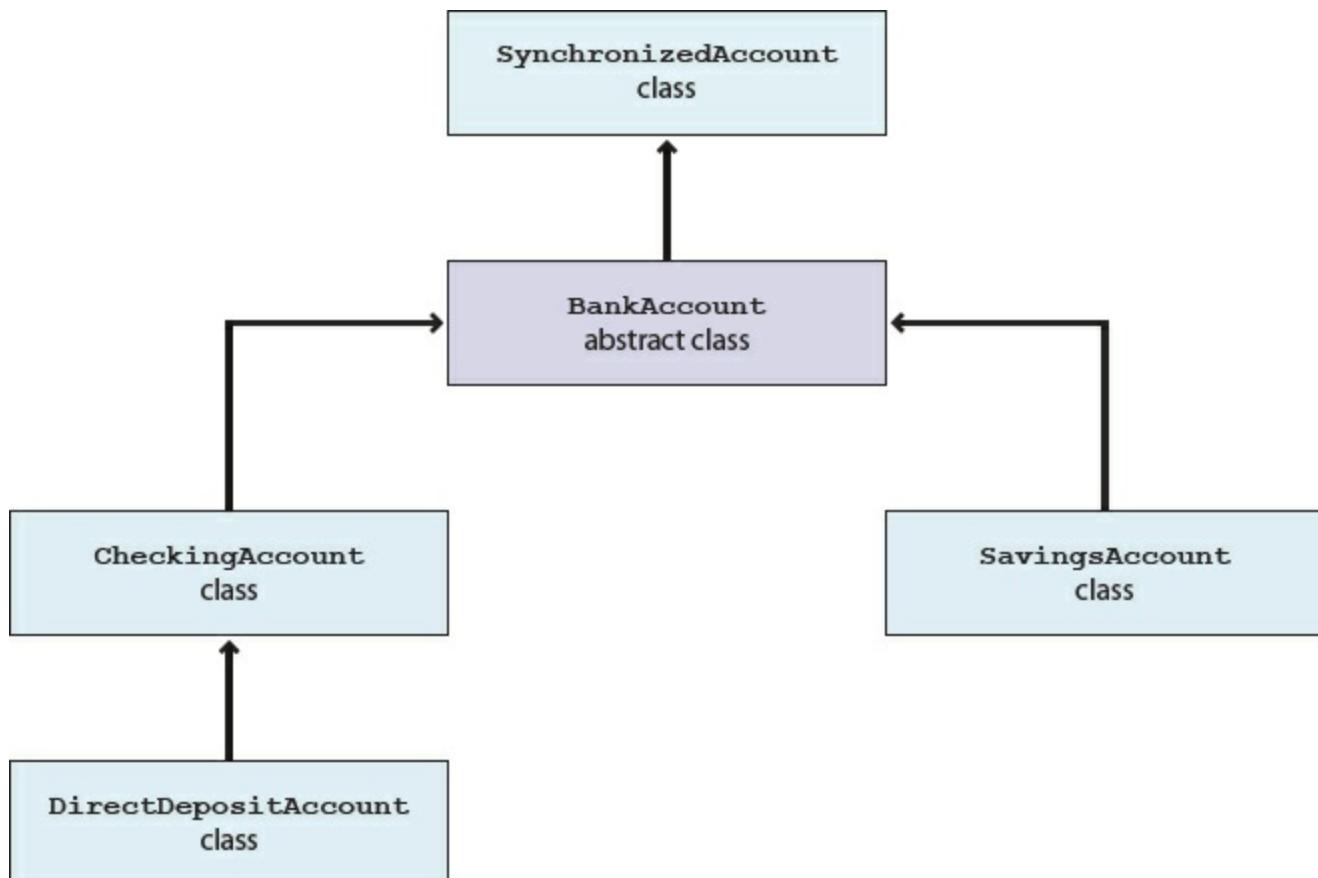


Figure 17.2: Bank account class hierarchy.

Program 17.1: Subclass of BankAccount that models a normal checking account.
(*CheckingAccount.java*)

```
1 public class CheckingAccount extends BankAccount {  
2     public static final double FEE = 10;  
3  
4     public CheckingAccount ( String name, double balance )  
5         throws InterruptedException {  
6         super ( name, balance );  
7     }  
8  
9     protected synchronized void update ()
```

```

10     throws InterruptedException {
11         super.update ();
12         changeBalance ( -getFee ()* getMonthsPassed () );
13     }
14
15     protected double getFee () { return FEE ; }
16 }

```

The most basic account is the `CheckingAccount`. As you recall from the `BankingAccount` class, the `getBalance()`, `deposit()`, and `withdraw()` methods are all declared final. At first it seems as if there is no way to change these methods to add the \$10 service charge. However, each of those methods calls the `update()` method first to take care of any bookkeeping. By overriding the `update()` method, we can easily add the service charge. The new `update()` method calls the parent `update()` to calculate the passage of time, then it changes the balance based on the number of months that have passed.

The system we have adopted may seem unusual at first. Any time the balance is checked, deposited to, or withdrawn from, we call `update()`. By updating the account to reflect any months which may have passed before continuing on, we do not have to write code which periodically updates each bank account. Each bank account is only updated if needed.

We were careful to mark `update()` as `synchronized`. Although the chance of an error happening is small, we make the update of the internal `Calendar` and the application of any fee atomic, just to be safe.

Note that we do not use the constant `FEE` directly in `update()`. Instead, we call the `getFee()` method. The reason for this decision is due to the next class.

Program 17.2: Subclass of `CheckingAccount` that models the behavior of accounts with direct deposits. (`DirectDepositAccount.java`)

```

1 import java.util.Calendar ;
2 public class DirectDepositAccount extends CheckingAccount {
3     protected Calendar lastDirectDeposit ;
4
5     public DirectDepositAccount ( String name, double balance )
6         throws InterruptedException {
7         super ( name, balance );
8         lastDirectDeposit = Calendar.getInstance ();
9     }
10
11    public double getFee () {
12        Calendar current = Calendar.getInstance ();
13        int months = 12*( current.get ( Calendar.YEAR ) -
14            lastDirectDeposit.get ( Calendar.YEAR ) ) +
15            ( current.get ( Calendar.MONTH ) -
16            lastDirectDeposit.get ( Calendar.MONTH ) );

```

```

17     if( months <= 1 )
18         return 0;
19     else
20         return super.getFee ();
21 }
22
23 public void directDeposit ( double amount )
24     throws InterruptedException {
25     deposit ( amount );
26     lastDirectDeposit = Calendar.getInstance ();
27 }
28 }
```

The DirectDepositAccount class extends the CheckingAccount class. Note that the update() method hasn't been overridden. We have added another Calendar object to keep track of the last time a direct deposit was made. Then, we do override the getFee() method. If there has been a recent direct deposit, the fee is nothing, otherwise, it returns the fee from the CheckingAccount. Because of dynamic binding, the update() method defined in CheckingAccount will call this overridden getFee() method for DirectDepositAccount objects.

Program 17.3: Subclass of BankAccount that models the behavior of a savings account.
(SavingsAccount.java)

```

1 public class SavingsAccount extends BankAccount {
2     public static final double MINIMUM = 1000;
3     public static final double FEE = 25;
4     protected final double RATE ;
5
6     public SavingsAccount ( String name, double balance, double rate
7                         )
8         throws InterruptedException {
9         super ( name, balance );
10        RATE = rate ;
11    }
12
13    protected double getFee () { return FEE ; }
14
15    protected double getMinimum () { return MINIMUM ; }
16
17    protected synchronized void update ()
18        throws InterruptedException {
19        super.update ();
20        int months = getMonthsPassed ();
21        for ( int i = 0; i < months ; i++ ) {
```

```

21     if( getBalance () > 0 )
22         changeBalance ( getBalance () * (1 + RATE /12) );
23     if( getBalance () < getMinimum () )
24         changeBalance ( -getFee () );
25 }
26 }
27 }
```

There should be few surprises in the last class, `SavingsAccount`. The biggest difference is that, in the `update()` method, we use a loop to update the balance because the account could be gaining interest and also incurring fees. The interaction of the two operations may give a different result than applying each in a block for the backlog of months.

This set of classes may not resemble the way a real, commercial-grade banking application works. Nevertheless, with inheritance and polymorphism we were able to create bank accounts which do some complicated tasks with a relatively small amount of code. At the same time, we preserved thread safety so that these accounts can be used in concurrent environments.

17.5 Concurrency: Atomic libraries

This chapter has discussed using polymorphism to reuse code. To solve the Bank Account with a Vengeance problem from the beginning of the chapter, we explored the process of extending several bank account classes to add additional features while working hard to maintain thread safety.

Code can be reused by extending classes with child classes or by using instances of existing classes as fields. There is no single solution that is best for every case. As in the bank account examples, it can be difficult to know when to apply the `synchronized` keyword to methods.

To lessen the load on the programmer, the Java API provides a library of atomic primitives in the `java.util.concurrent.atomic` package. These are classes with certain operations that are guaranteed to execute atomically. For example, the `AtomicInteger` class encapsulates the functionality of an `int` variable with atomic accesses. One of its methods is `incrementAndGet()`, which will atomically increment its internal value by 1 and return the result. Recall from [Program 14.1](#) that even an operation as simple as `++` is not atomic. If many different threads try to increment a single variable, some of those increments can get lost, causing the final value to be less than it should be.

Example 17.2: AtomicInteger

We can use the `AtomicInteger` class to rewrite [Program 14.1](#) so that no race condition occurs.

Program 17.4: Program to demonstrate the use of `AtomicInteger`. (`NoRaceCondition.java`)

```

1 import java.util.concurrent.atomic.*;
2
3 public class NoRaceCondition extends Thread {
4     private static AtomicInteger counter = new AtomicInteger ();
5     public static final int THREADS = 4;
```

```

6  public static final int COUNT = 1000000;
7
8  public static void main ( String [] args ) {
9      NoRaceCondition [] threads = new NoRaceCondition [ THREADS ];
10     for ( int i = 0; i < THREADS ; i++ ) {
11         threads [i] = new NoRaceCondition ();
12         threads [i]. start ();
13     }
14     try {
15         for ( int i = 0; i < THREADS ; i++ )
16             threads [i]. join ();
17     }
18     catch ( InterruptedException e ) {
19         e. printStackTrace ();
20     }
21     System.out.println (" Counter \t" + counter.get () );
22 }
23
24 public void run () {
25     for ( int i = 0; i < COUNT / THREADS ; i++ )
26         counter.incrementAndGet ();
27 }
28 }
```

This program is identical to [Program 14.1](#), except that the type of counter has been changed from `int` to `AtomicInteger` (and an appropriate `import` has been added). Consequently, the `++` operation was changed to an `incrementAndGet()` method call, and a `get()` method call was needed to get the final value. If you run this program, the final answer should always be 1000000, no matter what. ■

Exercise 17.9

The `java.util.concurrent.atomic` package includes `AtomicBoolean` and `AtomicLong` as well as `AtomicInteger`. Likewise, the `AtomicIntegerArray` and `AtomicLongArray` classes are included to perform atomic array accesses. For general purposes, the `AtomicReference<V>` class provides an atomic way to store a reference to any object. (The `<V>` is a generic type parameter, which will be discussed in [Chapter 18](#).)

Although you could use the `synchronized` keyword to create each one of these classes yourself, the result would not be as efficient. The atomic classes use a special *lock-free* mechanism. Unlike using the `synchronized` keyword which forces a thread to acquire a lock on a specific object, lock-free mechanisms are built on a *compare-and-swap* (CAS) hardware instruction. Thus, incrementing and the handful of other ways to update an atomic variable execute in one step because of special instructions on the CPU. Since there is no waiting to acquire a lock or fighting over which thread has the lock, the operation is very fast. Many high performance concurrent applications depends on CAS implementations.

Exercise 17.11

Exercises

Conceptual Problems

17.1 Consider the following two classes:

```
1 public class Sale {  
2     public double discount = 0.25;  
3  
4     public double getDiscount () {  
5         return discount ;  
6     }  
7  
8     public void setDiscount ( double value ) {  
9         discount = value ;  
10    }  
11 }
```

```
1 public class Blowout extends Sale {  
2     public double discount = 0.5;  
3  
4     public double getDiscount () {  
5         return discount ;  
6     }  
7  
8     public void setDiscount ( double value ) {  
9         discount = value ;  
10    }  
11 }
```

Given the following snippet of code, what is the output?

```
Sale sale = new Blowout ();  
System.out.println ( sale.discount );  
System.out.println ( sale.getDiscount );  
Blowout blowout = ( Blowout ) sale ;  
System.out.println ( blowout.discount );  
sale.setDiscount ( 0.75 );  
System.out.println ( sale.discount );
```

17.2 What are the differences and similarities between abstract classes and interfaces?

17.3 Assume that the Corn, Carrot, and Potato classes are all derived from Vegetable. Both Carrot and Potato classes have a peel() method, but Corn does not. Examine the following code and identify which line will cause an error and why.

```
Vegetable [] vegetables = new Vegetable [30];
```

```

for ( int i = 0; i < vegetables.length ; i += 3 ) {
    vegetables [i] = new Corn ();
    vegetables [i + 1] = new Carrot ();
    vegetables [i + 2] = new Potato ();
}
int index = vegetables.length - 1;
Potato potato ;
while ( index >= 0 ) {
    potato = ( Potato ) vegetable [ index ];
    potato.peel ();
}

```

17.4 How many different meanings of the keyword **final** are there in Java, and what does each mean?

17.5 Assume that Quicksand is a subclass of Danger.

What is the output of the following code?

```

Quicksand quicksand = new Quicksand ();
if( quicksand instanceof Danger ) {
    System.out.printf ( " Run for your lives !" );
if( quicksand.getClass () == Danger.class )
    System.out.printf ( " Run even faster !" );
if( quicksand instanceof Quicksand ) {
    System.out.printf ( " The more you struggle ,"+
        " the faster you'll sink !" );
if( quicksand.getClass () == Quicksand.class )
    System.out.printf ( " You'll need to find a vine to escape !"
        );
}

```

Programming Practice

17.6 Implement a program to assess income tax on normal employees, students, and international students using a class hierarchy. Normal employees pay a 6.2% social security tax and a 1.45% Medicare tax every year, but neither kind of student pays these taxes. All three groups pay normal income tax according to the following table.

Marginal Tax Rate	Income Bracket
10%	\$0 - \$7,825
15%	\$7,826 - \$31,850
25%	\$31,851 - \$77,100
28%	\$77,101 - \$160,850
33%	\$160,851 - \$349,700
35%	\$349,701+

Tax is assessed at a given rate for every dollar in the range. For example, if someone makes \$10,000, she pays 10% tax on the first \$7,825 of her income and 15% on the remaining \$2,175. The exception is international students whose country has a treaty with the U.S. so that they do not have to pay tax on the first \$50,000 of income.

17.7 Refer to the sort given in [Section 10.5](#) as the solution to the Sort It Out problem. Add another **boolean** to the parameters of the sort which specifies whether the sort is ascending or descending. Make the needed changes throughout the code to add this functionality.

Concurrency

17.8 Again refer to the sort from [Section 10.5](#). The goal now is to parallelize the sort. Change the sort to work with **int** values then write some code which will generate an array of random **int** values. Design your code so that you can spawn n threads. Partition the single array into n arrays and map one partition to each thread. Use your bubble sort implementation to sort each partition. Finally, merge the arrays back together, in sorted order, into one final array. For now, just use one thread (ideally the main thread) to do the merge.

The merge operation is a simple idea, but it is easy to make mistakes in implementation. The idea is to have three indexes, one for each of the two arrays you are merging and one for the result array. Always take the smaller (or larger, if sorting in descending order) index value from the two arrays and put it in the result. Then increment the index from the array you took the data from as well as the index of the result array. Make sure that you are careful not to go beyond the end of the arrays which are being merged.

17.9 Re-implement the original `SynchronizedAccount` class from [Example 14.2](#) using atomic classes. For simplicity, you can change the balance type from double to `AtomicInteger` since there is no `AtomicDouble` class. How much has this simplified the implementation? Is the `readers` field still necessary? Why or why not?

Concurrency

Experiments

17.10 Once you have implemented the sort in parallel from Exercise 17.8, time it against the sequential version. Try 2, 4, and 8 different threads, particularly if you have 8 cores. Be sure to

create one random array and use the same array for both the parallel and sequential versions. Try array sizes of 1000, 100000, and 1000000. Did the performance increase as much as you expect?

Concurrency

17.11 Take [Program 17.4](#) and increase COUNT to 100000000. Run it several times and time how long it takes to run to completion.

Concurrency

Then, take [Program 14.1](#) and increase its COUNT to 100000000 as well. Change the body of the `for` loop inside the `run()` method so that `count++;` is inside of a `synchronized` block that uses `RaceCondition.class` as the lock. (The choice of `RaceCondition.class` is arbitrary, but it is an object that all the threads can see.) In this way, the increment will occur atomically, since only the thread that has acquired the `RaceCondition.class` lock will be able to do the operation. Now, run this modified program several times and time it.

How different are the running times? They may be similar, depending on the implementation of locks and CAS on your OS and hardware platform.

Chapter 18

Dynamic Data Structures

Algorithms + Data Structures = Programs

—Niklaus Wirth

18.1 Problem: Infix conversion

If a math teacher writes the expression $(1 + 7 \times 8 - 6 \times (4 + 5) \div 3)$ on the blackboard and asks a group of 10 year old children to solve it, different children may give different answers. The difficulty is that the children may not understand the order of operations. Modern graphing calculators and many computer programs can, of course, evaluate such expressions correctly, but how do they do it? You intuitively grasp order of operations (left to right, multiplication and division take higher precedence than addition and subtraction), but encoding that intuition into a computer program is more difficult.

One way a computer scientist might approach this problem is to turn the mathematical expression from one that is difficult to evaluate to one that is easy. The normal style of writing mathematical expressions is called *infix notation*, because the operators are written in between the operands they are used on. A much easier notation for automatic evaluation is called *postfix notation*, because an operator is placed after the operands it works on. [Table 18.1](#) gives a few examples of expressions written in both infix and postfix notation.

Infix	Postfix
$3 + 7$	$3\ 7\ +$
$4 * 2$	$4\ 2\ *$
$1 + 9 * 2$	$1\ 9\ 2\ *\ +$
$(1 + 9) * 2$	$1\ 9\ +\ 2\ *$

Table 18.1: Examples of expressions in infix and equivalent postfix forms.

Although infix notation is probably more familiar to you, postfix notation has the benefit of exactly specifying the order of operations without using any precedence rules and without needing parentheses to clarify. To understand how to compute an expression in postfix notation, we rely on the idea of a stack, which we first introduce in [Chapter 9](#) and examine in much greater depth here.

Recall that a stack has three operations: *push*, *pop*, and *top*. It works like a stack of physical objects. The push operation places an object on the top, the pop operation removes an item from the top, and the top operation tells you what is at the top of the stack.

Using a stack, the postfix evaluation rules are easy. Scan the expression from left to right, if you see

an operand (a number, in our case), put it on the stack. If you see an operator, pop the last two operands off the stack and use the operator on them. Then, push the result back on the stack. When you run out of input, the value at the top of the stack is your answer.

For example, with $1\ 9\ 2\ *\ +$, all three operands are pushed onto the stack. Then, the $*$ is read, and so 2 and 9 is popped off the stack and multiplied. The result 18 is pushed back on the stack. Then, the $+$ is read, and 18 and 1 is popped off the stack and summed, resulting in 19, which is pushed back on the stack and is the final answer.

Our problem, however, is not to evaluate an expression in postfix notation but to convert an expression in infix notation to postfix notation. Again, the concept of a stack is useful. To do the conversion, we initialize a stack and scan through the input in infix notation. As we scan through the input, we do one of four things depending on which of the four possible inputs we see:

Operand: Simply copy the operand to the postfix output.

Operator: If the stack is empty, push the operator onto the stack. If the stack is not empty and the operator at the top of the stack has the same or greater precedence than our new operator, put the top operator into our postfix output and pop the stack. Continue this process as long as the top operator has the same or greater precedence compared to our new operator and the stack is not empty. Finally, push the new operator onto the stack.

Left Parenthesis: Push the left parenthesis onto the stack.

Right Parenthesis: Pop everything off the stack and add it to the output until you find a left parenthesis in the stack. Then pop the left parenthesis.

Precedence comes from order of operations: $*$ and $/$ have high precedence and $+$ and $-$ have low precedence. When you encounter it on the stack, treat $($ as if it has even lower precedence than $+$ and $-$. A right parenthesis should never appear on the stack.

Following this algorithm, we are able to write a program that converts infix notation to postfix notation. We further restrict our problem to the case when the only operands are positive integers in the range $[0,9]$. Since each is a single character, parsing the input is much easier. The same ideas for postfix conversion holds no matter how the input is formatted, but parsing arbitrarily formatted numbers is a difficult problem in its own right. This restriction also makes spaces unnecessary.

To solve the infix conversion problem, we need to create a stack data structure whose elements are terms from an infix expression, where a term is an operator, operand, or a parenthesis. We created a stack to solve the nesting expression problem in [Section 9.6](#), but we explore stacks in this chapter as one of many different kinds of dynamic data structure.

18.2 Concepts: Dynamic data structures

By now you have seen several ways to organize data in your programs. For example, you have used arrays to store a sequence of values and class definitions to store (and operate on) collections of related values. These data structures have the property that they are *static* in size: Once allocated,

they do not grow as the program runs. If you allocate an array to store 100 integers, you'll get an error if you try to store 101 integers into it.

In this chapter, we examine *dynamic* data structures: As more data is read or processed, these data structures grow and shrink in memory to store what is needed. The stack used to solve the nesting expressions problem from [Chapter 9](#) is not actually a dynamic data structure since it is defined with a fixed maximum size. In this chapter, we implement a true stack as well as many other kinds of dynamic data structures.

18.2.1 Dynamic arrays

There are two broad classes of dynamic data structures we examine here. The first kind are based on arrays that grow and shrink. Dynamic arrays allow for fast access to individual elements in the data structure. One drawback of dynamic arrays is that the array that stores that data has a fixed amount of space. When too many elements are added, a new array has to be reallocated and all the original elements copied over.

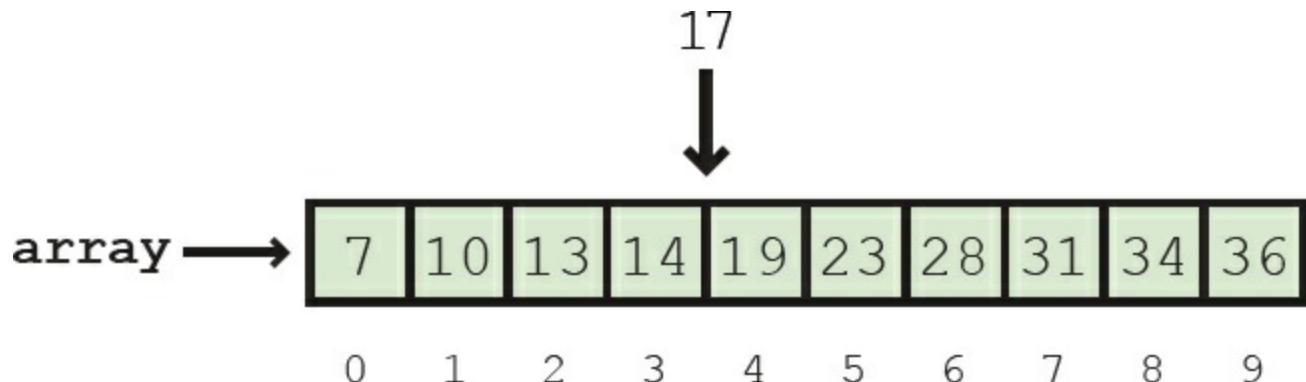


Figure 18.1: Insertion into full dynamic array requiring reallocation. Even if the array wasn't full, all values after 14 would need to be moved back to insert 17 in order.

Another drawback of dynamic arrays is that they are poorly suited for insertion or deletion of elements in the middle of the array. When an element is inserted, each element after it must be moved back one position. Likewise, when an element is deleted, all of the elements after it must be moved forward one position. Thus, insertions and deletions at the end of a dynamic array are usually efficient, but insertions and deletions in the middle are very time consuming.

18.2.2 Linked lists

The second kind of dynamic data structure is based on objects that link to (or reference) other objects. The simplest form of such a data type is a *linked list*. A linked list is a data structure made up of a sequence of objects. Each object contains some data value (such as a String) and a *link* to the next object in the sequence.

Linked lists are flexible because they have no preset size. Whenever a new element is needed, it can be created and linked into the list. Unfortunately, they can be slow if you need to access arbitrary elements in the list. The only way to reach an element in the list is to walk from element to element until you find what you're looking for. If the element is at the beginning (or the end) of the list, this process can be quick. If the element is in the middle, there is no fast way to get there.

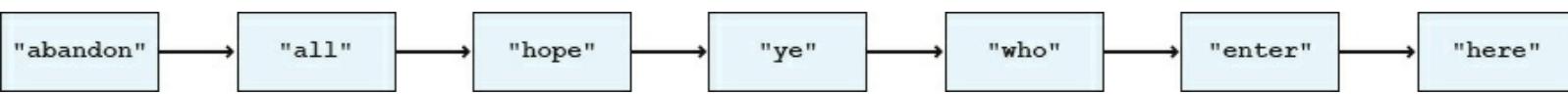


Figure 18.2: Visualization of a linked list.

Linked lists work well when inserting new elements at arbitrary locations in the list. Unlike arrays, they are not implemented as a contiguous block of memory. Linking a new element into the middle of the list automatically creates the correct relationship among elements, and there is no need to move all the elements after an insertion.

Among their downsides is the memory overhead of linked lists. Because a new object must be allocated for each element in the list, which must include a reference to the next element. Consequently, using a linked list to solve a problem usually take more memory than an equivalent dynamic array solution.

It turns out that either dynamic arrays or linked lists can be used to create an efficient solution to the infix conversion problem defined at the beginning of the chapter.

18.2.3 Abstract data types

The fact that dynamic arrays and linked lists can be used to solve similar problems points out that we may often be more interested in the **capabilities** of a data structure rather than its implementation.

An *abstract data type* (ADT) is a set of operations that can be applied to a set of data values with well-defined results that are independent of any particular implementation. In other words, it is a list of things that a data type can do (or have done to it).

A stack is a great example of an ADT. A stack needs to be able to push a value, pop a value, and tell us what value is on top. The internal workings of the stack are irrelevant (as long as they are efficient). It is possible to use either a dynamic array or a linked list to implement a stack ADT. A queue is another ADT we discuss in [Section 18.4](#), but there are many other useful ADTs.

18.3 Syntax: Dynamic arrays and linked lists

18.3.1 Dynamic arrays

Suppose you are faced with the problem of reading a list of names from a file, sorting them into alphabetical order, and printing them out. You have already looked at simple sorting algorithms to handle the sorting part, or you could use the Java Arrays.sort() method. In previous problems when you needed to use an array for storing items, you knew in advance how many (or a maximum of how many) items you would need to store. In this new problem, the number of names in the input file is unspecified, so you must allow an arbitrary number to be handled.

One approach is to make a guess at how many names are in the input file and allocate an array of that size. If your guess is too small, and you don't check array accesses, you'll cause an exception once you have filled the array and try to store the next name into the index one past the last. If your guess is too large, you could be wasting a significant amount of storage space.

Our first solution to the problem of dealing with dynamic or unknown amounts of data is to watch

our array accesses and expand the array as necessary during processing. (It is also possible to contract an array once you determine that the array has more space than needed.)

A simple solution

Program 18.1 allocates an array of 10 strings and reads a list of names from standard input until it reaches the end of the file, storing each name in successive array locations. If the number of names in the input is larger than the size of the array, it generates an exception.

Exercise 18.3

Since programs that generate uncaught exceptions are, in general, a bad idea, our first change to this program should be either to catch the exception or check the index before storing the name in the array. In either case, we would then take some action that is more user friendly than generating an exception, perhaps simply printing an explanatory message before exiting.

Program 18.1: Program to read names into an array, sort, and print. If there are more than 10 lines in the input, an exception is generated. (ReadIntoFixedArray.java)

```
1 import java.util.Arrays ;  
2 import java.util.Scanner ;  
3  
4 public class ReadIntoFixedArray {  
5     public static void main ( String [] args ) {  
6         Scanner in = new Scanner ( System.in );  
7         String [] names = new String [10];  
8         int n = 0;  
9  
10        while ( in.hasNextLine () )  
11            names [n ++] = in.nextLine ();  
12  
13        Arrays.sort (names, 0, n);  
14  
15        for ( int i = 0; i < n; i ++ )  
16            System.out.println ( names [i]);  
17    }  
18 }
```

Our second change is to take a recovery action that allows the program to proceed. What went wrong? We made a guess of the input size, allocated an array of that size, but our guess was too small. We could start over again (modify the code to initially allocate a larger array, recompile, and re-run the program), but that option may not be available to us if the program has been distributed to users around the world. Instead, we fix the problem on the fly by allocating a larger array, copying the old array into the new array, and continuing.

Program 18.2 begins like the previous program by allocating a fixed array. However, it now catches the `ArrayOutOfBoundsException` at line 16 if it tries to store too many names into the array.

The catch clause allocates a new array, twice the size of the original (current) array, copies the existing array into it, and replaces the reference to the current array with a reference to the new array.

Exercise 18.4

Program 18.2: Read names into an array, enlarging the array as necessary.

(ReadAndGrowArray.java)

```
1 import java.util.Arrays ;
2 import java.util.Scanner ;
3
4 public class ReadAndGrowArray {
5     public static void main ( String [] args ) {
6         String [] names = new String [10];
7         Scanner in = new Scanner ( System.in );
8         int n = 0;
9         String name = null ;
10
11         while (in. hasNextLine ()) {
12             name = in. nextLine ();
13             try {
14                 names [n] = name ;
15             }
16             catch ( ArrayIndexOutOfBoundsException e ) {
17                 names = Arrays.copyOfRange (names, 0,
18                     names.length *2) ;
19                 names [n] = name ;
20             }
21             n++;
22         }
23
24         Arrays.sort (names, 0, n);
25
26         for ( int i = 0; i < n; i++)
27             System.out.println ( names [i]);
28     }
29 }
```

Note that it was necessary to refactor the code in [Program 18.1](#) slightly: Add the name variable to hold the temporary result of reading the input line, and move the counter increment to outside the **try-catch** block.

Can this new, improved program still fail? Yes, but only for **very large** input, in the case when the Java virtual machine runs out of memory when doubling the size of the array.

A potentially more serious problem is the way we set names to point at a new array.

```
names = Arrays.copyOfRange (names, 0, names.length *2) ;
```

This line works because we know the only variable that references the array is `names`. If other variables referenced that array, they would continue to reference the old, smaller, and now out-of-date version of the `names` array. [Figure 18.3](#) gives an example of this problem.

A more complete solution

The problem of updating variables that reference the dynamic array is a serious issue in large programs. It may not be enough to allocate a larger array and assign the new reference to only one variable. There may be hundreds of variables (or objects) that reference the original array.

A solution to this problem is to create a new class whose objects contain the array as a private field. References to the array are then mediated, as usual, via accessor methods, which always refers to the same version of the array. [Program 18.3](#) is a simple implementation of a dynamic array class. This class maintains an internal array of `String` objects, which it extends whenever a call to `set()` tries to write a new element just past the end of the array.

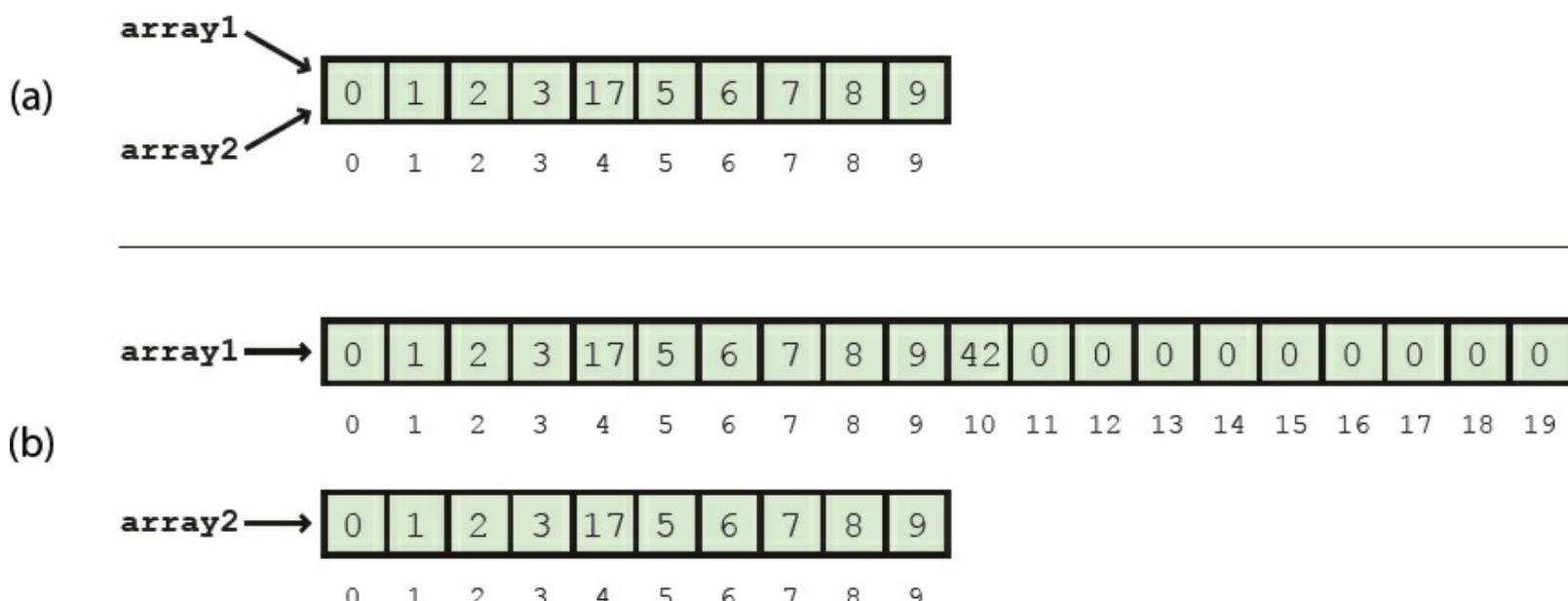


Figure 18.3: A poorly designed dynamic array implementation. (a) Both `array1` and `array2` begin pointing at the same array. (b) A new array has been allocated, and 42 has been added to it. `array1` has been updated to point at the new array, but `array2` still points at the original.

Program 18.3: A class to manage a dynamic array. This array grows by doubling when more space is needed. (`DynamicArray.java`)

```
1 import java.util.Arrays ;  
2  
3 public class DynamicArray {  
4     private String [] strings = new String [10];  
5  
6     public synchronized void set ( int i, String string ) {  
7         if (i == strings.length )
```

```

8     strings = Arrays.copyOfRange ( strings, 0,
9                               strings.length *2 );
10    strings [i] = string ;
11 }
12
13 public String get ( int i ) {
14     return strings [i];
15 }
16
17 public synchronized void sort ( int first, int last ) {
18     Arrays.sort ( strings, first, last );
19 }
20 }
```

Note that the set() and sort() methods are both **synchronized** in case this class is used by multiple threads simultaneously. Exercise 18.12 explores the need to synchronize these methods in the presence of multiple threads.

Exercise 18.12

Exercise 18.9

[Program 18.4](#) illustrates how to modify and extend [Program 18.1](#) to use this new class. Since the array grows automatically, there is no need for the original program to check for out-of-bounds exceptions. Of course, the array expansion only works if the reference occurs exactly at the index corresponding to one beyond the end of the array. Other out-of-bound references generate an exception.

Program 18.4: A program that uses the DynamicArray class to store input read from a file.
(UseDynamicArray.java)

```

1 import java.util.Scanner ;
2
3 public class UseDynamicArray {
4     public static void main ( String [] args ) {
5         DynamicArray names = new DynamicArray ();
6
7         Scanner in = new Scanner (System.in);
8         int n = 0;
9         String name = null ;
10        while (in. hasNextLine ()) {
11            name = in. nextLine ();
12            names.set (n, name );
13            n++;
14        }
15 }
```

```
16     names.sort(0, n);
17
18     for ( int i = 0; i < n; i++ )
19         System.out.println( names.get(i));
20     }
21 }
```

Since `names` is no longer an array, but rather an object of class `DynamicArray`, we can no longer use braces (`[]`) to access elements, but must use accessor methods `set()` and `get()`. Also, `Arrays.sort()` cannot sort this object, so we need to provide a `sort()` method in the class itself to sort the private array on demand.

This implementation, like most implementations of dynamic arrays, has potentially serious performance penalties. If the initial array is too small, compared to the final size, then it will have been doubled and the elements copied multiple times, resulting in slower execution. After a resize, the array is only half full, resulting in wasted space. Even on average, the array will only be three-quarters full

Exercise 18.5

18.3.2 Linked lists

As we've seen, while dynamic arrays can grow to accommodate a large number of items, the performance penalties of repeated copying and the space wasted by unoccupied array elements can negatively affect program behavior. In this section, we introduce the *linked list*, an alternative data structure that can efficiently grow to accommodate a large number of objects. As we shall see, this efficient growth comes at the expense of limitations on how the structure can be accessed.

Consider again the problem of reading an arbitrary number of names from an input file and storing them. Since we don't know in advance how many names there are, it may not be efficient to pre-allocate or dynamically grow an array to store them. Imagine, instead, that we could write each name on a small index card, and then link the index cards together to keep track of them, much like the cars of a railroad train are linked by the coupling from one to the next.

Constructing a linked list

In Java, a linked list is usually implemented as a class that provides methods to interact with a sequence of objects. The objects in the list are implemented as a private static nested class. A private static nested class behaves like a normal class but can only be created and accessed by the class surrounding it. In this way, the internal representation of the list is hidden and protected from outside modification. The nested class has two fields, one containing the data to be stored and the other containing a link or reference to the next object, or *node*, in the list. Since they are only accessed by the outer class, it is reasonable to make these fields public. If you need a refresher on static nested classes, refer to [Section 9.5](#).

```
public class LinkedList {
    private static class Node {
        public String value;
```

```
    public Node next ;  
}  
// methods for interacting with the list  
}
```

Note that the type next is the same as the class it's inside of! This apparent circular reference works because the variable only **references** an object, but the object is not actually contained within the variable. In fact, the value of the link may be **null**, indicating that there are no additional nodes in the list.

In the railroad metaphor, the node is a train car (with its freight as the value), and the link to the next node is the coupling to the next car.

The definition of LinkedList given above is a good start, but it needs a head reference that keeps track of the first node in the list. Initially, this value is **null**. We also need an add() method so that we can add nodes to the list. Without checking through the entire list, it is useful to know how many nodes are in it. We can create a size field that we increment whenever we add a node, as well as an accessor to read its value. Finally, we can create a fillArray() method that fills an array with the values in the list.

Program 18.5: A basic implementation of a linked list class to hold String objects. (LinkedList.java)

```
1  public class LinkedList {  
2      private static class Node {  
3          public String value ;  
4          public Node next ;  
5      }  
6  
7      private Node head = null ;  
8      private int size = 0;  
9  
10     public void add ( String value ) {  
11         Node temp = new Node ();  
12         temp.value = value ;  
13         temp.next = head ;  
14         head = temp ;  
15         size ++;  
16     }  
17  
18     public int size () {  
19         return size ;  
20     }  
21  
22     public void fillArray ( String [] array ) {  
23         Node temp = head ;  
24         int position = 0;
```

```

25     while ( temp != null ) {
26         array [ position ++] = temp.value ;
27         temp = temp.next ;
28     }
29 }
30 }
```

[Program 18.6](#) is a re-implementation of the name-reading program using class `LinkedList`. Note that no array needs to be pre-allocated. Instead, we capture all the lines of input into a linked list called `list`.

Program 18.6: A program that uses the `LinkedList` class to store input read from a file.
(`UseLinkedList.java`)

```

1 import java.util.Arrays ;
2 import java.util.Scanner ;
3
4 public class UseLinkedList {
5     public static void main ( String [] args ) {
6         Scanner in = new Scanner ( System.in );
7         LinkedList list = new LinkedList ();
8
9         while ( in.hasNextLine () )
10            list.add ( in.nextLine () );
11
12        String [] names = new String [ list.size () ];
13        list.fillArray ( names );
14
15        Arrays.sort ( names );
16
17        for ( int i = 0; i < names.length ; i ++ )
18            System.out.println ( names [i] );
19    }
20 }
```

Each time we read a new line from the file, the `LinkedList` class internally creates a new `Node` with the input line as its value. It also sets its next reference to the **current** head so that the rest of the list (which could be empty if head is `null`) comes after the new Node. We then update the `head` field to reference the new Node. Thus, each new line read from the file is stored at the **beginning** of the linked list. The last node in the list, which contains the first String read in, has a next value of `null`. [Figure 18.4](#) shows a visualization of the contents of this implementation of a linked list. An “X” is used in place of an arrow that point to `null`.

LinkedList
object

Node objects

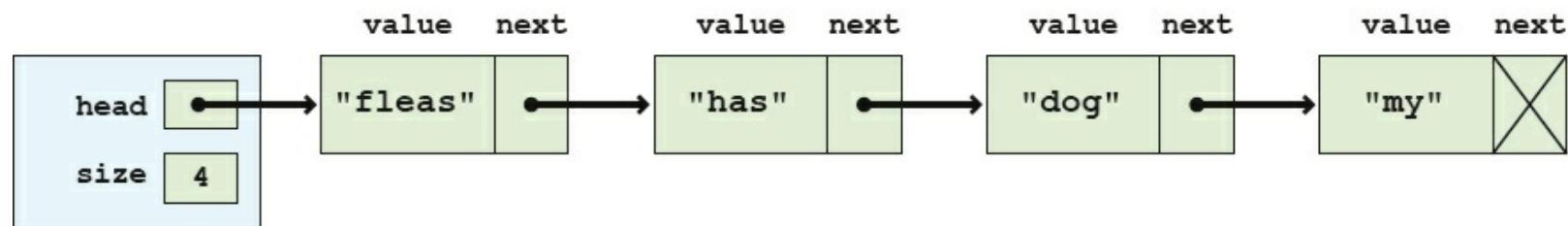


Figure 18.4: Visualization of linked list implementation with classes.

Since we also increment the `size` field inside of `LinkedList` on each add, we know how many `String` objects it contains. Thus, the `toString()` method knows how large of an array to allocate. It then visits every node in the linked list, storing its value into the array. In `UseLinkedList`, we sort the returned array as before and then print it.

Appending to the end of a linked list

The `LinkedList` class maintains a field named `head` that references the first node in the linked list. As we saw, that element was actually the **last** or **most recent** `String` read from input. This `head` element was followed by the next most recent `String`, followed by the next most recent `String`, and so on. The last node contained the first `String` read from input and had a **null** `next` field.

If we want the linked list to be ordered in the natural way, with `head` pointing to the first element read from the file and the last element on the list (the one with `next` pointing to **null**) containing the `String` most recently read, we can maintain a second field that references the *tail* of the list.

Program 18.7 adds a *tail pointer* called `tail` to the `LinkedList` class. Note that we have changed the `add()` method to the `addFirst()` method, and we have also added an `addLast()` method to make it easy to append elements to the end of a linked list. Note that the `addFirst()` method has been updated to change the `tail` pointer, but only if the list is empty (`head` is **null**). After all, adding to the front of a list only changes `tail` if the front is **also** the back. In the `addLast()` method, adding a value to an empty list also sets both the `head` and `tail` to point at a node containing that value. Once the list has a node in it, subsequent calls to `addLast()` creates a new `Node`, points the `next` field of the old `tail` at it, and changes the `tail` field so that it also points at it.

Program 18.7: We can append to the end of a linked list by using an additional variable, `tail`, to reference the last element (`tail`) of the list. (`LinkedListWithTail.java`)

```
1 public class LinkedListWithTail {  
2     private static class Node {  
3         public String value;  
4         public Node next;  
5     }  
6     private Node head = null;
```

```

8     private Node tail = null ;
9     private int size = 0;
10
11    public void addFirst ( String value ) {
12        Node temp = new Node ();
13        temp.value = value ;
14        temp.next = head ;
15        head = temp ;
16        if( tail == null )
17            tail = head ;
18        size++;
19    }
20
21    public void addLast ( String value ) {
22        Node temp = new Node ();
23        temp.value = value ;
24        temp.next = null ;
25        if( tail == null )
26            head = tail = temp ;
27        else
28            tail.next = temp ;
29        size++;
30    }
31
32    public int getSize () {
33        return size ;
34    }
35
36    public void fillArray ( String [] array ) {
37        Node temp = head ;
38        int position = 0;
39        while ( temp != null ) {
40            array [ position ++] = temp.value ;
41            temp = temp.next ;
42        }
43    }
44 }
```

Inserting into a linked list

In the running example for this chapter, we are interested in printing a sorted list of String objects read from input. Thus far we have captured the lines into a linked list of elements, dumped these elements into an array of the right size, and then sorted the array. An alternative solution is to insert the elements into the linked list at the right point in the first place.

Program 18.8 is a version of a linked list that inserts elements into the linked list in sorted order. The only significant difference between it and the previous implementations of a linked list is its add() method. This method walks down the linked list, starting at head, until it either walks off the end of the list or finds an element before which the new String should go. There are special cases that must be handled to make this process work correctly.

Empty list: The first time an item is inserted into a linked list, the head and tail fields must be set to reference this new node. The next field of the new node is **null**.

Insert at beginning: If a node is inserted at the beginning of the list, the head must be updated to point to this new node. The next field of the new node is set to the old value of head.

Insert in middle: To insert a node in the middle of a linked list, it is typically necessary to maintain two variables to reference the current and previous nodes while walking down the list. Once the proper insertion point is found (between the previous and current nodes), the next field for the previous node is adjusted to reference the new node, and next field for the new node is set to current.

Insert at end: If the insertion is taking place at the end of the list, current is **null**, and the new node has a next field of **null**. However, the tail field must be updated to reference the new node.

Program 18.8: A linked list class in which calling the add() method inserts each value in sorted order. (SortedLinkedList.java)

```
1 public class SortedLinkedList {
2     private static class Node {
3         public String value;
4         public Node next;
5     }
6
7     private Node head = null;
8     private Node tail = null;
9     private int size = 0;
10
11    public void add ( String value ) {
12        Node temp = new Node ();
13        temp.value = value;
14        temp.next = null;
15
16        if( head == null )
17            // empty list
18            head = tail = temp;
19        else if( value.compareTo ( head.value ) < 0 ) {
20            // insert at beginning
```

```

21         temp.next = head ;
22         head = temp ;
23     }
24     else {
25         // insert at middle or end
26         Node previous = head ;
27         Node current = head.next ;
28
29         while ( current != null &&
30                 value.compareTo ( current.value ) >= 0 ) {
31             previous = current ;
32             current = current.next ;
33         }
34
35         previous.next = temp ;
36         temp.next = current ;
37
38         if( current == null ) // insert at end of list
39             tail = temp ;
40     }
41     size++;
42 }
43
44 public int size () {
45     return size ;
46 }
47
48 public void fillArray ( String [] array ) {
49     Node temp = head ;
50     int position = 0;
51     while ( temp != null ) {
52         array [ position ++] = temp.value ;
53         temp = temp.next ;
54     }
55 }
56 }
```

18.4 Syntax: Abstract data types (ADT)

We've seen two examples so far of dynamic data structures: dynamic arrays and linked lists. A great deal of complexity can go on inside these data structures, but code that uses these data structures does not need to be aware of the details of the internal implementation. Ideally, user programs could use any data structure that provided the needed set of operations.

Our dynamic array and linked list classes were simple examples of abstract data types (ADT). We continue to design data structures that hide the details of their implementation inside a class. The user of each class is aware of the operations (public methods) that can be performed on objects of the class, but not on the techniques used to implement those operations. Defining an ADT without regard to an implementation keeps users of the ADT from becoming dependent on details of any particular implementation. It gives maximum freedom to the programmer to choose (and change) the implementation as appropriate for the overall system design.

We generalize a data structure by observing which operations are applied to it. Then, we create an abstraction that formalizes these observations. The idea is to cleanly separate the use and behavior of the data structure from the way in which it is implemented.

Interfaces are the obvious tool for defining the behavior of a class in Java without specifying its implementation. When defining an ADT in Java, the set of operations becomes the set of methods given by the interface. Then, any class that implements the ADT must implement the interface that defines that ADT.

In subsequent sections we look at two fundamental abstract data types, *stacks* and *queues*, and sample classes that implement them.

18.4.1 Stacks

We have already used stacks to solve problems in [Chapter 9](#). Recall that a stack data structure behaves like a stack of books on your desk. When you place a book on the stack it covers the books that are already there. When you take a book off the stack, you remove the book most recently placed there, exposing the one beneath it.

You can find a simple implementation of a stack in the solution to the infix conversion problem in [Section 18.6](#), but we now examine the stack more deeply as an archetypal ADT. A stack's restricted set of operations (pushing and popping) is adequate for many tasks and can be implemented in a number of different ways, some more efficient than others.

The acronym FILO (first in, last out) is sometimes used to describe a stack. The last item that has been pushed onto the stack is the first item to be popped off the stack. In the next section, we'll study the *queue*, which is a FIFO (first in, first out) data structure.

18.4.2 Abstract Data Type: Operations on a stack

There are two essential operations on a stack abstract data type (corresponding to placing a book on the pile and removing it): `push()` and `pop()`. We also define two additional operations, `top()` and `isEmpty()`.

- `push(x)`: Push value `x` onto the stack.
- `pop()`: Pop the value on the top of the stack, and return its value.
- `top()`: Return the value on the top of the stack, but do not pop it off.
- `isEmpty()`: Return true if the stack is empty, false otherwise.

Because a stack is an abstract data type, we are not specifically concerned with how these operations are implemented, merely that they are. Thus, we can specify an interface called Stack that requires these four methods.

Program 18.9: An interface specifying the stack ADT. (Stack.java)

```
1 public interface Stack {  
2     void push ( String value );  
3     String pop ();  
4     String top ();  
5     boolean isEmpty ();  
6 }
```

Linked list implementation

All the operations defined by the stack ADT (and interface) are implemented as methods in the class LinkedListStack, shown in [Program 18.10](#).

Program 18.10: A class to implement a stack ADT using a linked list. (LinkedListStack.java)

```
1 public class LinkedListStack implements Stack {  
2     private static class Node {  
3         public String value ;  
4         public Node next ;  
5     }  
6  
7     private Node head = null ;  
8  
9     public void push ( String value ) {  
10        Node temp = new Node ();  
11        temp.value = value ;  
12        temp.next = head ;  
13        head = temp ;  
14    }  
15  
16    public String pop () {  
17        String value = null ;  
18        if ( isEmpty () )  
19            System.out.println (" Can 't pop empty stack !");  
20        else {  
21            value = head.value ;  
22            head = head.next ;  
23        }  
24        return value ;  
25    }
```

```

26
27     public String top () {
28         String value = null ;
29         if ( isEmpty () )
30             System.out.println ("No top on an empty stack !");
31         else
32             value = head.value ;
33         return value ;
34     }
35
36     public boolean isEmpty () {
37         return head == null ;
38     }
39 }
```

The head field is used to maintain a reference to the linked list that defines the stack. It is initialized to `null`.

The method `push()` must create a new node for the linked list and push it onto the front of the list. It does so by creating a new `Node`, setting its `value` field to the incoming value, and pointing its `next` pointer to the beginning of the list, stored by `head`. Since `temp` is now the new top of the stack, `head` is made to point at it.

The `pop()` method needs to return the value of the `head` node and remove that node from the linked list. It does this by replacing the `head` node with the node pointed at by the `next` link in `head`. The `pop()` method from the simpler stack used in the solution to the nested expressions problem in [Section 9.6](#) merely removed the top and did not return the value. Most real-world stack implementations of `pop()` **do** return this value, giving programmers more flexibility.

Note that both `pop()` and `top()` print an error message if the stack is empty. Other more elaborate error handling is possible, for example, by throwing an exception.

Dynamic array implementation

Like the dynamic array example of [Program 18.4](#), [Program 18.11](#) implements a stack of `String` values using a dynamic array data structure.

Program 18.11: Program illustrating a stack ADT partially implemented using a dynamic array.
(`DynamicArrayStack.java`)

```

1 import java.util.Arrays ;
2
3 public class DynamicArrayStack implements Stack {
4     private String [] strings = new String [10];
5     private int size = 0;
6
7     public void push ( String string ) {
```

```

8     if( size == strings.length )
9         doubleArray ();
10    strings [ size ++] = string ;
11 }
12
13 public String pop () {
14     String value = null ;
15     if( size == 0 )
16         System.out.println (" Can 't pop empty stack !");
17     else
18         value = strings [-- size ];
19     return value ;
20 }
21
22 private void doubleArray () {
23     strings = Arrays.copyOfRange ( strings, 0, strings.length *2 ) ;
24 }
25 }
```

This stack implementation using a dynamic array omits the top() and isEmpty() methods (causing a compiler error in [Program 18.11](#) until the Stack interface is properly implemented). Exercise 18.7 has you provide implementations of these methods.

Exercise 18.7

Example 18.1: Postfix computation

At the beginning of the chapter, we introduced the problem of converting an expression from infix to postfix notation. In [Section 18.6](#), we give the solution to this problem, but without a program that can evaluate a postfix expression, the conversion tool is not very useful.

Here we give a simple postfix evaluator. Recall the algorithm: Scan the input expression from left to right, if you see a number, put it on the stack. If you see an operator, pop the last two operands off the stack and use the operator on them. Then, push the result back on the stack. When you run out of input, the value at the top of the stack is your answer.

Like the infix to postfix converter, we restrict our input to positive integers of a single digit. To make this program simpler, we introduce two new classes that are also useful in our infix to postfix converter. The first is Term.

```

public class Term {
    private int value ;
    public Term ( int value ) { this.value = value ; }
    public int getValue () { return value ; }
}
```

This class is a *wrapper* for an `int` value. Although its structure is simple, we update the definition

of Term later in the solution to the infix to postfix conversion problem. By doing so, we can keep exactly the same definition for TermStack given next.

Program 18.12: Class to manage a stack of Term objects. (TermStack.java)

```
1 public class TermStack {  
2     private static class Node {  
3         public Term value ;  
4         public Node next ;  
5     }  
6  
7     private Node head = null ;  
8  
9     public void push ( Term value ) {  
10        Node temp = new Node ();  
11        temp.value = value ;  
12        temp.next = head ;  
13        head = temp ;  
14    }  
15  
16    public Term pop () {  
17        Term value = null ;  
18        if ( isEmpty () )  
19            System.out.println (" Can 't pop empty stack !");  
20        else {  
21            value = head.value ;  
22            head = head.next ;  
23        }  
24        return value ;  
25    }  
26  
27    public Term top () {  
28        Term value = null ;  
29        if ( isEmpty () )  
30            System.out.println ("No top on an empty stack !");  
31        else  
32            value = head.value ;  
33        return value ;  
34    }  
35  
36    public boolean isEmpty () {  
37        return head == null ;  
38    }  
39 }
```

This class gives a linked list implementation of a stack. In fact, it is virtually identical to [Program 18.10](#) with the substitution of Term for String.

Program 18.13: Program to evaluate a postfix expression. (PostfixCalculator.java)

```
1 import java.util.*;
2
3 public class PostfixCalculator {
4     public static void main ( String [] args ) {
5         Scanner in = new Scanner ( System.in );
6         String expression = in.nextLine ();
7         TermStack stack = new TermStack ();
8         for ( int i = 0; i < expression.length (); i++ ) {
9             char term = expression.charAt ( i );
10            if( term >= '0' && term <= '9' )
11                stack.push ( new Term ( (int)( term - '0' ) ) );
12            else {
13                int b = stack.pop ().getValue ();
14                int a = stack.pop ().getValue ();
15                switch ( term ) {
16                    case '+': stack.push ( new Term ( a + b ) );
17                        break ;
18                    case '-': stack.push ( new Term ( a - b ) );
19                        break ;
20                    case '*': stack.push ( new Term ( a * b ) );
21                        break ;
22                    case '/': stack.push ( new Term ( a / b ) );
23                        break ;
24                }
25            }
26        }
27        System.out.println ("The answer is: " +
28                            stack.top ().getValue ());
29    }
30 }
```

With our utility classes in place, the code for the postfix evaluator is short. Our main() method reads in the expression from the user and creates a TermStack called stack. Then, it iterates through the expression with a for loop. For each number we find, we supply it as an argument to the constructor of a new Term object, which we push onto stack.

For each operator, we pop two items off stack and apply the operator to them. We create a new Term from the result and push this value onto stack. Finally, after all input is exhausted, we print the value on the top of stack. To test it properly, you have to supply expressions in postfix form. Also, remember that these operations are all integer operations without fractional parts. Be careful to avoid

division by zero! ■

18.4.3 Queues

A *queue* data structure is similar to a stack data structure, except that when getting an item from a queue, the item that has been in the queue longest is the one retrieved. A queue data structure models an ordinary queue or line of people. The first person into the queue or line at a bank, for example, is the first one to receive service. Late comers are served in the order in which they arrive.

A queue is sometimes called a FIFO (first in, first out) data structure due to this property. To distinguish the operations on a queue from those on a stack, we use the terms enqueue and dequeue instead of push and pop.

18.4.4 Abstract Data Type: Operations on a queue

Four typical operations on a queue data structure are:

- enqueue(x): Put value x onto the end of the queue.
- dequeue(): Remove and return the value at the front of the queue, that is, the value that has been on the queue the longest.
- front(): Return (but do not remove) the value at the front of the queue.
- isEmpty(): Return true if the queue is empty, false otherwise.

As with stacks, we can specify an interface called Queue that requires these four methods.

Program 18.14: An interface specifying the queue ADT. (Queue.java)

```
1 public interface Queue {  
2     void enqueue ( String value );  
3     String dequeue ();  
4     String front ();  
5     boolean isEmpty ();  
6 }
```

Linked list implementation

Program 18.15 shows an implementation of the queue ADT operations using a linked list. Because we need to keep track of nodes at both ends of the linked list, we maintain head and tail variables to reference these nodes. The enqueue() and dequeue() methods manipulate these variables to manage the queue as values are put onto it and removed from it.

Program 18.15: Program illustrating a queue ADT implemented using a linked list.
(LinkedListQueue.java)

```
1 public class LinkedListQueue implements Queue {
```

```
2  private static class Node {
3      public String value ;
4      public Node next ;
5  }
6
7  private Node head = null ;
8  private Node tail = null ;
9
10 public void enqueue ( String value ) {
11     Node temp = new Node ();
12     temp.value = value ;
13     temp.next = null ;
14
15     if( isEmpty () )
16         head = tail = temp ;
17     else {
18         tail.next = temp ;
19         tail = temp ;
20     }
21 }
22
23 public String dequeue () {
24     String value = null ;
25     if( isEmpty () )
26         System.out.println (" Can't dequeue an empty queue !");
27     else {
28         value = head.value ;
29         head = head.next ;
30         if( head == null )
31             tail = null ;
32     }
33     return value ;
34 }
35
36 public String front () {
37     String value = null ;
38     if( isEmpty () )
39         System.out.println ("No front on an empty queue !");
40     else
41         value = head.value ;
42     return value ;
43 }
44
```

```
45     public boolean isEmpty () {
46         return head == null ;
47     }
48 }
```

Note that the implementation of the LinkedListQueue class is very similar to the implementation of the LinkedListWithTail class. The enqueue() method in the former is almost identical to the addLast() method in the latter.

18.5 Advanced Syntax: Generic data structures

Most of the dynamic data structures we have seen in this chapter store values of type String. We explore dynamic arrays of String values, linked lists of String objects, queues of String objects, and stacks of String objects. In [Example 18.1](#), we create the stack class TermStack to hold Term objects, but TermStack is identical to the existing LinkedListStack class with the substitution of Term for String.

What if you wanted to store values of some other type in these data structures? What if you wanted a stack of int values or a queue of Thread objects? You might think that you need to create a distinct but similar implementation of each ADT for each type, as we do in [Example 18.1](#).

One possible solution is to take advantage of the fact that a variable of type Object can hold a reference to a value of any reference type (since all classes are subtypes of Object). If we create data structures using Object as the underlying type, we can store values of any type in the data structure. For example, [Program 18.16](#) is an implementation of a stack ADT with an underlying data type of Object.

Program 18.16: A class that implements a stack of Object references. (ObjectStack.java)

```
1  public class ObjectStack {
2      private static class Node {
3          public Object value ;
4          public Node next ;
5      }
6
7      private Node head = null ;
8
9      public void push ( Object value ) {
10         Node temp = new Node ();
11         temp.value = value ;
12         temp.next = head ;
13         head = temp ;
14     }
15
16     public Object pop () {
```

```

17     Object value = null ;
18     if( isEmpty () )
19         System.out.println (" Can 't pop empty stack !");
20     else {
21         value = head.value ;
22         head = head.next ;
23     }
24     return value ;
25 }
26
27 public Object top () {
28     Object value = null ;
29     if( isEmpty () )
30         System.out.println (" Can 't get top of empty stack !");
31     else
32         value = head.value ;
33     return value ;
34 }
35
36 public boolean isEmpty () {
37     return head == null ;
38 }
39 }
```

Note that a stack of Object references is an example of a *heterogeneous data structure*. It is possible to put objects of different types onto the same stack. While there are situations in which this technique is useful, in most cases a *homogeneous data structure* (where all values are of the same type) is all that is needed. Homogeneous data structures allow type checking to occur at compile time, thus helping to avoid run-time errors.

Using a stack of Object references is generally more cumbersome, since you must cast values returned from `pop()` or `top()` to the appropriate data type.

```

ObjectStack stack = new ObjectStack ();
stack.push (" hello ");
String s = ( String ) stack.pop ();
```

Without the cast to `(String)`, the compiler gives an error: Type mismatch: cannot convert from `Object` to `String`.

Casting the returning value from a heterogeneous data structure essentially forces type checking to move from compile-time to run-time. Instead of having the Java compiler verify the type correctness of operations, we force the Java virtual machine to do the check.

18.5.1 Generics in Java

Java provides a general facility to create classes that implement the same basic ADT but with a

different underlying data type. This mechanism preserves the advantages of compile-time type checking and eliminates the need for run-time casting. A *generic class* is a class that gives a template for creating classes in which a placeholder for the underlying data type can be filled in when a specific instance of that class is created. In the case of [Example 18.1](#), we need a stack that can hold Term objects instead of String objects, and a generic class allows us to create a stack of any reference type.

The generics facility in Java only supports underlying data types that are reference types (such as String and user-defined types), not primitive types (such as `int` or `boolean`). Thus, a generic stack of `int` values needs to be implemented as a stack of Integer objects. Fortunately, Java automatically converts between `int` and Integer in most cases.

Defining a simple generic class in Java is done by appending a *type parameter* within angle brackets (`<>`) to the end of the class name being defined.

```
public class GenericClass <T> {  
    ...  
    T transform (T item ) {  
        ...  
    }  
    ...  
}
```

This code defines a new generic class (think class template) `GenericClass` with underlying type `T`. It includes a method `transform()` that takes a value of type `T` and transforms it (in some unspecified way) to another value of type `T`.

To use a generic class properly, you must create instances of it specifying the underlying type. In actual fact, the compiler fills in the appropriate type at compile time. The compiler must make sure that all the operations are valid with the supplied type substituted for the type parameter (`T` in this example).

For example, to create and use an instance of `GenericClass` with underlying type `String`, you would type:

```
GenericClass < String > genericString = new GenericClass <String >();  
String s = genericString.transform (" hello ");
```

Because this use of the `GenericClass` class is defined for underlying type `String`, no casting is necessary to assign the result of the `transform()` method to the `String` variable `s`.

To create and use an instance of `GenericClass` with underlying type `Integer`, you would type:

```
GenericClass <Integer> genericInteger = new GenericClass < Integer >();  
int i = genericInteger.transform (27);
```

The same definition of `GenericClass` is used in both instances with different underlying data types and the compiler is able to verify at compile time that the uses are type safe.

If you omit the underlying type when declaring a generic variable or creating an instance of a

generic type, the compiler uses Object as the underlying type. This use, called a *raw type*, is essentially like not using generics at all. There is no compile-time type checking, and references must be cast as needed.

```
GenericClass genericRaw = new GenericClass () ; // raw type  
int i = ( Integer ) genericRaw.transform (27) ; // cast needed
```

The next two examples illustrate defining generic classes in Java.

Example 18.2: Defining a generic linked list

Program 18.17 defines a generic version of the LinkedList class shown earlier. Note that it is necessary to include the type parameter T on the outer class as well as the nested class Node.

Program 18.17: A class that implements a generic linked list. (GenericLinkedList.java)

```
1 public class GenericLinkedList <T> {  
2     private static class Node <T> {  
3         public T value ;  
4         public Node <T> next ;  
5     }  
6  
7     private Node <T> head = null ;  
8     private int size = 0;  
9  
10    public void add (T value ) {  
11        Node <T> temp = new Node <T >();  
12        temp.value = value ;  
13        temp.next = head ;  
14        head = temp ;  
15        size ++;  
16    }  
17  
18    public int size () {  
19        return size ;  
20    }  
21  
22    public void fillArray (T[] array ) {  
23        Node <T> temp = head ;  
24        int position = 0;  
25        while ( temp != null ) {  
26            array [ position ++] = temp.value ;  
27            temp = temp.next ;  
28        }  
29    }
```

Using generics can be very easy, but there are some oddities. In particular, there are problems instantiating arrays with generic types. The `fillArray()` method works because it never creates the array, only fills it. ■

18.5.2 Using a Generic Class

Creating an instance of a generic class is similar to creating an instance of a regular class, except that (to avoid warnings) you must specify the missing type (or types) used to parameterize the generic class. For example, if you want to create an instance of the `GenericClass<T>` class, you must specify the type `T`, for example `new GenericClass<String>()`.

[Program 18.18](#) uses the generic class `GenericLinkedList` parameterized by `String` to re-implement Program 18.6.

Program 18.18: Program that uses the generic class `GenericLinkedList` to create and use a linked list of Strings. (`UseGenericLinkedList.java`)

```

1 import java.util.Arrays ;
2 import java.util.Scanner ;
3
4 public class UseGenericLinkedList {
5     public static void main ( String [] args ) {
6         Scanner in = new Scanner ( System.in );
7         GenericLinkedList < String > list =
8             new GenericLinkedList <String> () ;
9
10        while ( in.hasNextLine () )
11            list.add( in.nextLine () );
12
13        String [] names = new String [ list.size () ];
14        list.fillArray ( names );
15
16        Arrays.sort ( names );
17
18        for ( int i = 0; i < names.length ; i ++ )
19            System.out.println ( names [i]);
20    }
21 }
```

18.5.3 Using Java Libraries

Many of the Java library classes use generics to make them more general purpose. The `java.util` package includes many classes to implement stacks, queues, dynamic arrays, sets, and other useful data structures. These classes are parameterized so that they can be created with different underlying

types. We illustrate three examples here: Vector, ArrayList, and HashMap. Note that there is also a LinkedList class, which is a great deal more powerful than the LinkedList class defined in this chapter. Any class that implements the Iterable interface can be used in the for-each loops described in [Section 6.9.1](#). The ArrayDeque, ArrayList, HashSet, TreeSet, and Vector, classes all implement Iterable. In our examples, a Vector object and a Set (returned by the entrySet() method of a HashMap) are used as targets of for-each loops.

Example 18.3: Vectors

A Vector (java.util.Vector) implements an array of objects that can grow at run time. The array is automatically extended whenever an attempt is made to store an item exactly one location beyond the last element. Unlike a linked list, Vector elements can be efficiently accessed in any order (by specifying the index, just like an ordinary array). Elements can be inserted into the middle of the Vector, causing following elements to be pushed back to later indexes. Arbitrary elements can also be deleted from the Vector using the remove() method.

[Program 18.19](#) illustrates a use of the Vector class. The program creates an empty Vector and generates random integers between 1 and 10, appending them to the end of the vector, until their sum is at least 100. Then, it prints the integers and their sum (including how many were generated).

Program 18.19: A simple program to illustrate the use of the Vector class. (VectorExample.java)

```
1 import java.util.Random ;
2 import java.util.Vector ;
3
4 public class VectorExample {
5     public static void main ( String [] args ) {
6         Random random = new Random () ;
7
8         Vector < Integer > vector = new Vector < Integer > () ;
9
10        int sum = 0 ;
11        while ( sum < 100 ) {
12            int n = random.nextInt ( 10 ) + 1 ;
13            vector.add ( n ); // append n to end of vector
14            sum += n ;
15        }
16
17        for ( int n : vector )
18            System.out.format ("%3d\n", n );
19        System.out.println (" --- ");
20        System.out.format ("%3d (%d values )\n", sum, vector.size ());
21    }
22 }
```

Output from a typical run of [Program 18.19](#) is shown below:

```
9  
9  
8  
7  
7  
4  
7  
6  
8  
7  
9  
4  
9  
10
```

104 (14 values)

Example 18.4: Maps

The `HashMap`(`java.util.HashMap`) is a very useful, general-purpose data structure that maintains a dictionary of entries. A dictionary associates unique keys with values. You can think of it as *mapping* a *key* to a *value*. In the Java `HashMap` class, keys and values can be arbitrary Java classes.

[Program 18.20](#) reads a sequence of lines containing names and ages (for simplicity, the name is one word and the age is a simple integer). It stores these (name, age) pairs in a `HashMap<String, Integer>` data structure. Once all the input is read (`in.hasNext()` returns `false`), the program prints all the keys (names), then all the values (ages), and finally it prints the names and ages of each person in the input file.

Program 18.20: A program that illustrates using a `HashMap` dictionary to store a set of names and ages. (`HashMapExample.java`)

```
1 import java.util.HashMap ;  
2 import java.util.Map ;  
3 import java.util.Scanner ;  
4  
5 public class HashMapExample {  
6     public static void main ( String [] args ) {  
7         HashMap < String, Integer > map =  
8             new HashMap < String, Integer > () ;
```

```

9
10     Scanner in = new Scanner ( System.in);
11     while ( in.hasNext() ) {
12         String name = in.next();
13         int age = in.nextInt();
14         map.put (name, age );
15     }
16
17     System.out.format (" Keys \n");
18     for ( String name : map.keySet () )
19         System.out.println ("\t" + name );
20
21     System.out.println (" Values ");
22     for ( int age : map.values () )
23         System.out.println ("\t" + age );
24
25     for ( Map.Entry <String, Integer > entry : map.entrySet () ) {
26         System.out.format ( entry.getKey () + " -> " +
27                             entry.getValue ());
28     }
29 }
30 }
```

Shown below is the output for a simple input file.

Keys

```

kathy
martha
fred
henway
michael
henry
john
margarette
edward
tim
hamcost
```

Values

```

60
22
15
```

```
1  
21  
31  
23  
57  
12  
57  
2  
  
kathy -> 60  
martha -> 22  
fred -> 15  
henway -> 1  
michael -> 21  
henry -> 31  
john -> 23  
margarette -> 57  
edward -> 12  
tim -> 57  
hamcost -> 2
```

■

18.6 Solution: Infix conversion

Here we give our solution to the infix conversion problem from the beginning of the chapter. As in [Example 18.1](#), we use a stack of Term objects to solve the problem. However, we expand the Term class to hold both operands and operators. We only add methods and fields to the earlier definition, taking nothing away. In this way, we should be able to use the Term class for both infix to postfix conversion and postfix calculation.

```
1 public class Term {  
2     private int value ;  
3     private char operator ;  
4     private boolean isOperator ;
```

Here we have augmented the earlier Term class by adding two more fields, a char called operator to hold an operator and a **boolean** called isOperator to keep track of whether or not our Term object holds an operator or an operand.

```
6     public Term ( int value ) {
```

```

7     this.value = value ;
8     isOperator = false ;
9 }
10
11 public Term ( char operator ) {
12     this.operator = operator ;
13     isOperator = true ;
14 }
```

We now have two constructors. The first one takes an `int` value and stores it into `value`, setting `isOperator` to `false` to indicate that the `Term` object must be an operand. The second constructor takes a `char` value and stores it into `operator`, setting `isOperator` to `true` to indicate that the `Term` object must be an operator (such as `+`, `-`, `*`, or `/`).

```

16     public int getValue () { return value ; }
17     public char getOperator () { return operator ; }
18     public boolean isOperator () { return isOperator ; }
```

These three accessors give back the operand value, the operator character, and whether or not the object is an operator, respectively. This solution is not necessarily the most elegant from an OOP perspective. The code that uses a `Term` object needs to chose the `getValue()` method or the `getOperator()` method depending on whether the `Term` is an operator or not. This design opens up the possibility that some code will call the wrong accessor method and get a useless default value.

```

20     public boolean greaterOrEqual ( Term term ) {
21         if( isOperator () )
22             switch ( operator ) {
23                 case '*':
24                 case '/': return true ;
25                 case '+':
26                 case '-':
27                     return ( term.operator != '*' &&
28                             term.operator != '/' );
29                 default : return false ;
30             }
31         else
32             return false ;
33     }
34 }
```

The most complicated addition to the `Term` class is the `greaterOrEqual()` method, which takes in another `Term` object. This method compares the operator of the `Term` object being called with the one that is being passed in as a parameter. Because this method is in the `Term` class, it can access the `private` variables of the `term` parameter. This method returns `true` if the operator of the called object has a greater or equal precedence compared to the operator of the parameter object. The meat of the

method is the **switch** statement that establishes the high precedence of * and /, the medium precedence of + and -, and the low precedence of anything else, namely the left parenthesis (.

With this updated Term class, we can create Term objects that hold either an operator or an operand and allow the precedence of operators to be compared. We use exactly the same TermStack class from [Example 18.1](#) for our stack. All that remains is the client code that parses the input.

```
1 import java.util.*;
2
3 public class InfixToPostfix {
4     public static void main ( String [] args ) {
5         Scanner in = new Scanner ( System.in );
6         String expression = in.nextLine ();
7         TermStack stack = new TermStack ( expression.length () );
8         String postfix = "";
9         char term ;
```

The main() method of this class reads in the input expression and creates a TermStack called stack with a maximum size of the length of the expression. We also declare a String called postfix to hold the output.

```
10    for ( int i = 0; i < expression.length (); i++ ) {
11        term = expression.charAt ( i );
12        if( term >= '0' && term <= '9' )
13            postfix += term ;
14        else if( term == '(' )
15            stack.push ( new Term ( term ) );
16        else if( term == ')' ) {
17            while ( stack.top ().getOperator () != '(' ) {
18                postfix += stack.top ().getOperator ();
19                stack.pop ();
20            }
21            stack.pop (); // pop off the '('
22        }
23        else if( term == '*' || term == '/' ||
24                  term == '+' || term == '-' ) {
25            Term operator = new Term ( term );
26            while ( stack.size () > 0 &&
27                    stack.top ().greaterOrEqual ( operator ) ) {
28                postfix += stack.top ().getOperator ();
29                stack.pop ();
30            }
31            stack.push ( operator );
32        }
33    }
```

This `for` loop runs through each `char` in the input expression and applies the four rules given in the description of the infix conversion problem. If a term is an operand, it is added to the output. If a term is a left parenthesis, it is pushed onto the stack. If a term is a right parenthesis, all the terms on the stack are popped off and added to the output until a left parenthesis is reached. If a term is a normal operator, the top of the stack is repeatedly popped and added to output as long as it has a precedence greater than or equal to the new operator. The complexity of doing this precedence comparison is now tucked away inside of the `Term` class.

```
35     while ( stack.size () > 0 ) {
36         postfix += stack.top ().getOperator ();
37         stack.pop ();
38     }
39     System.out.println ( postfix );
40 }
41 }
```

After the input has all been consumed, we pop all the elements off the stack and add them to the output. Finally, we print the output. The output to this program could be used as the input to the postfix evaluator program from [Example 18.1](#). A more complex program that did both the conversion and the calculation might want to store everything in `Term` objects instead of outputting a `String` and then recreating `Term` objects.

18.7 Concurrency: Linked lists and thread safety

The implementations of stacks and queues in the previous sections are **not** thread-safe. If multiple threads use a stack or queue object simultaneously, the head or tail pointers can become inconsistent or be updated incorrectly, potentially causing the stack or queue to lose elements. As you have seen, multiple threads operating on the same data can produce unexpected results.

[Program 18.21](#) is a simple multi-threaded program to test (and break!) the thread safety of the queue implementation in [Program 18.15](#). This program (18.21) creates a queue and stores a reference to it in a static (class) variable `queue`. It then creates and starts 10 threads. During the adding phase (indicated by `adding` being `true`), each thread adds its thread ID number to the queue and prints it to standard output. Then, the program joins the threads until each has finished. The program then ends the adding phase (by setting the boolean variable `adding` to `false`) and starts 10 more threads. These threads each read one value from the queue and print it to standard output.

Program 18.21: Program to test the queue implementation, including its thread safety.
(`UseLinkedListQueue.java`)

```
1 public class UseLinkedListQueue extends Thread {
2     private static final int THREADS = 10;
3     private static LinkedListQueue queue = new LinkedListQueue ();
4     private static boolean adding = true ;
5 }
```

```

6  public static void main ( String [] args ) {
7      Thread [] threads = new Thread [ THREADS ];
8      for ( int i = 0; i < THREADS ; i ++ ) {
9          threads [i] = new UseLinkedListQueue ();
10         threads [i]. start ();
11     }
12
13     for ( int i = 0; i < THREADS ; i ++ )
14         try { threads [i]. join (); }
15         catch ( InterruptedException e) {}
16
17     adding = false ;
18
19     for ( int i = 0; i < THREADS ; i ++ ) {
20         threads [i] = new UseLinkedListQueue ();
21         threads [i]. start ();
22     }
23
24     for ( int i = 0; i < THREADS ; i ++ )
25         try { threads [i]. join (); }
26         catch ( InterruptedException e) {}
27
28     while ( ! queue.isEmpty () )
29         System.out.println (" Left in queue : ID = " +
30                             queue.dequeue ());
31     }
32
33     public void run () {
34         if( adding ) {
35             long ID = Thread.currentThread (). getId ();
36             System.out.println (" Thread ID added to queue : " +
37                                 Thread.currentThread (). getId ());
38             queue.enqueue ( String.valueOf (ID));
39         }
40         else {
41             String ID = queue.dequeue ();
42             System.out.println (" Thread ID removed from queue : "
43                                 + ID);
44         }
45     }
46 }
```

Without appropriate synchronization, the program may not correctly link all values into the queue nor remove them at the end. A typical error-prone output run is shown here:

```
Thread ID added to queue: 9
Thread ID added to queue: 14
Thread ID added to queue: 13
Thread ID added to queue: 12
Thread ID added to queue: 11
Thread ID added to queue: 10
Thread ID added to queue: 18
Thread ID added to queue: 17
Thread ID added to queue: 16
Thread ID added to queue: 15
Thread ID removed from queue: 14
Thread ID removed from queue: 11
Thread ID removed from queue: 12
Thread ID removed from queue: 16
Thread ID removed from queue: 17
Thread ID removed from queue: 18
Thread ID removed from queue: 10
Can't dequeue an empty queue!
Can't dequeue an empty queue!
Thread ID removed from queue: 15
Thread ID removed from queue: null
Thread ID removed from queue: null
```

How does this implementation fail? Consider the situation in which two threads are attempting to put a value in the queue simultaneously (see [line 10](#) in [Program 18.15](#)). Suppose the first thread tests the queue and finds it empty (`isEmpty()` returns `true`) but is then interrupted. If a second thread gets control it will also see that the queue is empty then sets the head and tail variables to the new `Node` object `temp` at [line 16](#) and return. The first thread will eventually wake up, still thinking that the queue is empty, and also set the head and tail variables to its own new `Node` `temp`. But these assignments overwrite the assignments just done by the previous thread! The initial node that was in the queue is now lost.

This problem can be fixed by ensuring that once one thread starts examining and modifying queue variables, no other thread can access the same variables until the first one is finished. As shown in [Chapter 14](#), this mutual exclusion can be achieved by using the `synchronized` keyword on methods that need to have exclusive access to object variables. In this queue implementation, we need to synchronize access by threads that are using either the `enqueue()` or `dequeue()` methods, since both methods access and manipulate variables in the object. Although it is not called in this program, the `front()` method should also be synchronized so that a `null` head is not accessed accidentally. The

`isEmpty()` method does not need to be synchronized since the only methods that call it that can do any harm are already synchronized. Outside code that calls `isEmpty()` might get the wrong value if another thread modifies the contents of the queue, but there is no guarantee that other threads will not modify the state of the queue at any point after the `isEmpty()` method is called anyway.

Program 18.22: A synchronized version of the queue class that allows thread-safe use.
(`LinkedListQueueTS.java`)

```
1  public class LinkedListQueueTS implements Queue {  
2      private static class Node {  
3          public String value ;  
4          public Node next ;  
5      }  
6  
7      private Node head = null ;  
8      private Node tail = null ;  
9  
10     public synchronized void enqueue ( String value ) {  
11         Node temp = new Node ();  
12         temp.value = value ;  
13         temp.next = null ;  
14  
15         if( isEmpty () )  
16             head = tail = temp ;  
17         else {  
18             tail.next = temp ;  
19             tail = temp ;  
20         }  
21     }  
22  
23     public synchronized String dequeue () {  
24         String value = null ;  
25         if( isEmpty () )  
26             System.out.println (" Can't dequeue an empty queue !");  
27         else {  
28             value = head.value ;  
29             head = head.next ;  
30             if( head == null )  
31                 tail = null ;  
32         }  
33         return value ;  
34     }  
35  
36     public synchronized String front () {
```

```

37     String value = null ;
38     if( isEmpty () )
39         System.out.println ("No front on an empty queue !");
40     else
41         value = head.value ;
42     return value ;
43 }
44
45     public boolean isEmpty () {
46         return head == null ;
47     }
48 }
```

With both enqueue() and dequeue() methods synchronized as in [Program 18.22](#), a typical output generated by the program is shown below.

```

Thread ID added to queue: 9
Thread ID added to queue: 14
Thread ID added to queue: 12
Thread ID added to queue: 13
Thread ID added to queue: 10
Thread ID added to queue: 11
Thread ID added to queue: 18
Thread ID added to queue: 17
Thread ID added to queue: 16
Thread ID added to queue: 15
Thread ID removed from queue: 9
Thread ID removed from queue: 18
Thread ID removed from queue: 13
Thread ID removed from queue: 17
Thread ID removed from queue: 15
Thread ID removed from queue: 16
Thread ID removed from queue: 14
Thread ID removed from queue: 12
Thread ID removed from queue: 10
Thread ID removed from queue: 11
```

18.8 Concurrency: Thread-safe libraries

As we mentioned in [Section 9.7](#), some libraries are thread-safe and some are not. The Java Collections Framework (JCF) is a very useful library, but it is also a library that requires thread safety to be at the forefront of your mind.

The JCF defines the Collection interface and the Map interface. The Collection interface, which any collection of objects should implement, has subinterfaces Set, List, and Queue which define the basic operations in Java that are needed to implement a set, list, or queue of items. The Map interface gives the basic operations for a dictionary, a collection of key-value pairs, one implementation of which is the HashMap from [Example 18.4](#).

As we mentioned in [Chapter 10](#), an interface cannot mark a method with the `synchronized` keyword. Consequently, the JCF can make no guarantee about the thread safety of a container based on which interface it implements. The programmer must read the documentation carefully in order to know if a container is thread-safe and react accordingly.

Example 18.5: ArrayList

An ArrayList is like a Vector, with essentially the same interface but lacks synchronization. That is, if two threads attempt to insert or remove an element from the same ArrayList at the same time, the ArrayList internal state may become corrupt or the results may be incorrect.

[Program 18.23](#) is an example of synchronizing updates to the ArrayList class with multiple threads. The program creates an ArrayList and places a reference to it in the static class variable list. It then creates and starts two threads. Each thread repeats a loop 10 times, appending a String to the ArrayList on each iteration. To increase the likelihood of concurrent update attempts, the thread sleeps for a millisecond on each iteration. To prevent concurrent updates from actually happening, each thread synchronizes on the common (shared) class variable list at [line 26](#).

Program 18.23: Example of thread-safe use of an ArrayList. (ArrayListExample.java)

```
1 import java.util.ArrayList ;  
2  
3 public class ArrayListExample extends Thread {  
4     private static ArrayList< String > list ;  
5  
6     public static void main ( String [] args ) {  
7         Thread t1 = new ArrayListExample ();  
8         Thread t2 = new ArrayListExample ();  
9  
10        list = new ArrayList< String >();  
11  
12        t1. start ();  
13        t2. start ();  
14  
15        try {  
16            t1. join ();  
17            t2. join ();  
18        } catch ( InterruptedException e ) {  
19            e. printStackTrace ();  
20        }  
21    }  
22}
```

```

18     } catch ( InterruptedException e) { e.printStackTrace (); }
19
20     for ( String s : list )
21         System.out.println (s);
22 }
23
24 public void run () {
25     for ( int i = 0; i < 10; i ++ ) {
26         synchronized ( list ) {
27             list.add ( this.getName () + ":" + i);
28         }
29         try { Thread.sleep (1); }
30         catch ( InterruptedException e) {
31             e.printStackTrace ();
32         }
33     }
34 }
35 }
```

Without the **synchronized** keyword, a typical run, shown below, includes a **null** reference in the output, indicating that the internal ArrayList data structure was not updated correctly.

```

Thread-1: 0
Thread-0: 0
Thread-1: 1
Thread-0: 1
Thread-1: 2
Thread-0: 2
Thread-0: 3
Thread-1: 3
Thread-1: 4
Thread-0: 4
null
Thread-0: 5
Thread-1: 6
Thread-0: 6
Thread-1: 7
Thread-0: 7
Thread-1: 8
```

```
Thread-0: 8  
Thread-0: 9  
Thread-1: 9
```

With the **synchronized** keyword, each run includes exactly the same number of entries from each thread, although the threads do not always alternate in strict lock-step.

```
Thread-0: 0  
Thread-1: 0  
Thread-0: 1  
Thread-1: 1  
Thread-1: 2  
Thread-0: 2  
Thread-1: 3  
Thread-0: 3  
Thread-1: 4  
Thread-0: 4  
Thread-0: 5  
Thread-1: 5  
Thread-1: 6  
Thread-0: 6  
Thread-1: 7  
Thread-0: 7  
Thread-1: 8  
Thread-0: 8  
Thread-1: 9  
Thread-0: 9
```



Exercises

Conceptual Problems

- 18.1 Explain the difference between static data structures and dynamic data structures.
- 18.2 In which situations is it better to use a dynamic array? In which situations is it better to use a linked list? Explain why in each case.
- 18.3 On which line in [Program 18.1](#) is an exception generated?
- 18.4 In [Program 18.2](#), is it possible to post-increment n inside the **try** clause rather than at the

bottom of the **while** loop?

18.5 Explain why the names array in [Program 18.4](#) is, on average, only three-quarters full.

18.6 Based on the stack implementation in [Program 18.10](#), draw a picture of the linked list structure after each of the following statements.

```
LinkedListStack stack = new LinkedListStack();  
stack.push ("hello");  
stack.push ("goodbye");  
stack.pop ();  
stack.push ("there");  
stack.push ("cruel");  
stack.pop ();  
stack.push ("world");
```

18.7 Implement the methods `top()` and `isEmpty()` for the dynamic array implementation of the stack in [Program 18.11](#).

18.8 Based on queue implementation in [Program 18.15](#), draw a picture of the linked list structure after each of the following statements.

```
LinkedListStack queue = new LinkedListStack();  
queue.enqueue ("hello");  
queue.enqueue ("there");  
queue.enqueue ("world");  
queue.dequeue ();  
queue.enqueue ("cruel");  
queue.dequeue ();  
queue.enqueue ("goodbye");
```

Programming Practice

18.9 Implement a version of `DynamicArray` that shrinks the size of its internal storage array to half its size when only one quarter of its capacity is being used. This design can save significant amounts of space if a large number of items are added to the dynamic array at once and then removed.

18.10 Consider Program 18.7 which defines the `LinkedListWithTail` class for storing a linked list of `String` values. Add a public `reverse()` method to the class which reverses the order of the nodes in the linked list. The key idea is make a new linked list that holds the head of the list. Then, remove the head from the original linked list. Put the next node **in front** of the head in the new linked list and remove it from the old. Continue the process until there is nothing left in the original list. Be sure to reset the head and tail references correctly after the reversal.

18.11 In [Section 18.2.2](#), we use two kinds of linked lists to store data, but copy all of that data back into an array before sorting it. We use third linked list class (`SortedLinkedList`) to insert data and maintain a sorted order. However, it is possible to add data in non-sorted order to a linked

list and then sort it afterwards. Add a `sort()` method to the `LinkedListWithTail` class that performs a bubble sort on the nodes inside.

The algorithm for a bubble sort is described in [Section 10.1](#). The idea is to make repeated passes through a list, swapping two adjacent items if they are out of order. You keep making passes over the list until no adjacent items are out of order. For this `sort()` method, you will need to use the `compareTo()` method to compare the `String` values in the linked list nodes. Also, it may be necessary to have special cases that update the head and tail pointers if those nodes are swapped with other nodes. Note that bubble sort is not the fastest way to sort a linked list. We introduce a faster approach in [Chapter 19](#).

Concurrency 18.12 Create JUnit test cases to verify that the synchronized keywords are needed or the `set()` and `sort()` methods of the `DynamicArray` example (Program 18.3). To test the `set()` method, you can create one thread that repeatedly sets, gets, and tests a changing value at a fixed location (e.g., 0) and another thread that continuously appends to the array (causing it to grow by copy and replace, thus occasionally overwriting the value at the fixed location). To test the `sort()` method, create two threads that sort the same large random array at the same time. Check to see if the array is, in fact, actually sorted after the threads have exited. For both tests, you may need to repeat the operations a number of times to trigger the race condition.

18.13 To make an infix calculator that can handle floating point values or even just integers with more than one digit, you need to make a pass over the input, parsing the sequence of characters into terms. When an expression is in infix notation, the order of terms is an operand followed by an operator, repeated over and over, and finishing on an operand. There are two exceptions: Whenever you are expecting an operand, you might get a left parenthesis, but, after the parenthesis, you continue to look for an operand. Whenever you are expecting an operator, you might get a right parenthesis, but, after that parenthesis, you continue to look for an operator.

Using this first pass over input to separate terms as well as the `parseDouble()` method from [Example 8.3](#) to compute the equivalent double values of operands, rewrite the solution from [Section 18.6](#) to convert your terms into postfix ordering and then calculate the answer.

18.14 Re-implement the solution to the infix conversion problem given in [Section 18.6](#) so that it uses `GenericStack` with a type parameter of `Term` instead of `TermStack`.

18.15 Interfaces can also be generic. Consider the following generic version of `Queue`.

```
public interface Queue <T> {  
    void enqueue (T value );  
    T dequeue ();  
    T front ();  
    boolean isEmpty ();  
}
```

Re-implement `LinkedListQueue` so that it is generic with type parameter `T` and implements interface `Queue<T>`.

Chapter 19

Recursion

In order to understand recursion, you must first understand recursion.

—Anonymous

19.1 Problem: Maze of doom

The evil mastermind from [Chapter 13](#) has returned with a new attempt at world domination. Since he now knows that you can use concurrency to crack his security code, this time he has hidden his deadly virus in a secret location protected by a complex maze of walls and passageways. Fortunately, you've been able to get a copy of the maze floor plan, but now you must write a program to find a path through the maze so you can steal the deadly virus before the evil mastermind releases it on the world.

Finding a path through a maze involves systematically exploring twists and turns, keeping track of where you've been, backtracking out of dead ends, and producing a resulting path once you make it through. You already have the basic tools necessary to solve this problem. You can, for example, represent the maze with a two-dimensional array of characters, where a plus ('+') represents a wall and a space (' ') represents a passageway. You could create a path through the maze by replacing a contiguous (vertical and horizontal but not diagonal) sequence of ' ' characters by '*' characters, that lead from the starting square to the final square where the deadly virus is located.

The difficulty is dead ends or, worse, loops. How do you keep track of which paths you've tried that didn't work? While you could use additional data structures to store this additional information, *recursion* is a solution technique that makes solving problems like this one surprisingly straightforward.

19.2 Concepts: Recursive problem solving

The idea that we use to solve this maze problem is called recursion. Imagine that you are in a maze and have the choice to go right, left, or straight. No matter which of the three paths you take, you will probably be confronted by more choices of going right, left, or straight. You need to explore them all systematically. The process of systematically exploring the right path is similar to the process of systematically exploring the left path. The choice at this moment between left, right, and straight is in fact part of the same systematic process that you want to follow when you are in the left, right, or straight branches of the maze.



Figure 19.1: Example of a maze to solve.

Solutions that can be described in terms of themselves are recursive. But what is recursion? How can describing something in terms of itself be useful? Since recursion sounds circular, how can it be applied to problem solving in Java? How does the computer keep track of these self-references? The following subsections address precisely these questions.

19.2.1 What is recursion?

In the context of computer science and mathematics, recursion means describing some repetitive process in terms of itself. Many complex things can be described elegantly using recursion.

Consider the question, “How old are you now?” If you were 27, you could answer, “I am one year older than I was last year.” If then asked, “Well, how old were you last year?” Again, you could answer, “I was one year older than I was the year before.” Assuming that the person who wanted to know your age was very patient, you could repeat this answer over and over, explaining that each year you have been one year older than the previous year. However, after asking you 27

times about your age on the previous year, you would run out of years of life and be forced to answer, “Zero years old.”

This absurd dialog shows an important feature of useful recursive definitions: They have at least one *base case* and at least one *recursive case*. The base case is the part of the definition that is not described in terms of itself. It is a concrete answer. Without the base case, the process would never end, and the definition would be meaningless. The recursive case is the part of the definition that is defined in terms of itself. Without the recursive case, the definition could only describe a finite set of things.

In the example above, the base case is the age of zero years old. You have no age before that. The recursive case is any age greater than zero years old. We can use mathematical notation to describe your age in a given year. Here $\text{age}(\text{year})$ is a function that gives the age you were during year year .

$$\text{age}(\text{year}) = \begin{cases} 0 & \text{(Base Case) if you were born in } \text{year} \\ 1 + \text{age}(\text{year} - 1) & \text{(Recursive Case) if you were born before } \text{year} \end{cases}$$

To be meaningful, recursive cases should be defined in terms of simpler or smaller values. In English, it is equally correct to say that you are now a year younger than you’ll be next year. Unfortunately, the age that you’ll be next year is not any closer to a base case, making that recursion useless.

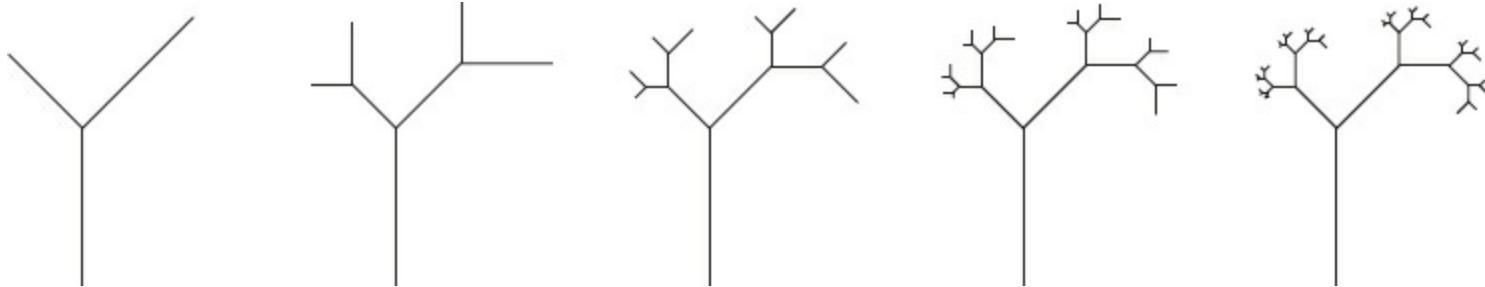


Figure 19.2: Recursive refinement generating a tree-like image.

A recursive definition of your age suggests that recursion is all around us. It takes some work to write it down in formal notation, but self-similarity is a constant theme in art and nature. The branching of a trunk of a tree is similar to the branching of its limbs, which is similar to the branching of its branches, which in turn is similar to the branching of its twigs. In fact, we borrow the idea of the branching of a tree to define a recursive data structure in [Section 19.4](#). [Figure 19.2](#) starts with a simple Y-shaped branching. By successively replacing the branches with the previous shape, a tree is generated recursively. *Fractals* are images generated by similar recursive techniques. Although many real trees exhibit recursive tendencies, they do not follow rules consistently or rigidly.

19.2.2 Recursive definitions

Although recursion has many fascinating applications, certain forms of recursion are more useful for solving problems. As with many aspects of programming, there is a strong connection to mathematical principles.

In mathematics, a *recursive definition* is one that is defined in terms of itself. It is common to define functions and sequences (and also sets) recursively. Functions (such as $f(n)$) are usually defined in relation to the same function with a smaller input (such as $f(n - 1)$). Sequences (such as s_n) are usually defined in relation to earlier elements in the sequence (such as s_{n-1}).

Example 19.1: Multiplication defined recursively

Even very common functions can be defined recursively. Consider the multiplication $x \times y$. This multiplication means repeatedly adding the x a total of y times. If y is a positive integer, we can describe this multiplication with the following recursive definition. Note the base case and recursive case.

$$x \times y = \begin{cases} x & \text{(Base Case) if } y = 1 \\ x + (x \times (y - 1)) & \text{(Recursive Case) if } y > 1 \end{cases}$$

Multiplication seems like such a basic operation that there would be no need to have such a definition. Yet mathematicians often use multiple equivalent definitions to prove results. Furthermore, this elementary definition provides intuition for creating more complex definitions. ■

Exercise 19.1

Example 19.2: Factorial defined recursively

Another mathematical function with a natural recursive definition is the factorial function, often written $n!$. The factorial function is used heavily in probability and counting. The value of $n! = n \cdot (n - 1) \cdot (n - 2) \dots 3 \cdot 2 \cdot 1$. Mathematicians like recursive definitions because they are able to describe functions and sequences precisely without using ellipses (...).

$$n! = \begin{cases} 1 & \text{(Base Case) if } n = 0 \\ n \cdot (n - 1)! & \text{(Recursive Case) if } n > 0 \end{cases}$$

Note that the base case gives 1 as the answer when $n = 0$. By convention $0! = 1$. Thus, this definition correctly gives $0! = 1$, $1! = 1 \cdot 0! = 1$, $2! = 2 \cdot 1! = 2$, $3! = 3 \cdot 2! = 6$, and so on. ■

Example 19.3: Fibonacci defined recursively

The Fibonacci sequence is an infinite sequence of integers starting with 1, 1, 2, 3, 5, 8, 13, 21, Each term after the two initial 1s is the sum of the previous two terms in the sequence. Fibonacci has many interesting properties and crops up in surprisingly diverse areas of mathematics. It was originally used to model the growth of rabbit populations.

The Fibonacci sequence also has a natural recursive mathematical definition. Indeed, you may have noticed that we described each term as the sum of the two previous terms. We can formally define the n^{th} Fibonacci number F_n as follows, starting with $n = 0$.

$$F_n = \begin{cases} 1 & \text{(Base Cases) if } n = 0 \text{ or } n = 1 \\ F_{n-1} + F_{n-2} & \text{(Recursive Case) if } n > 1 \end{cases}$$

Since the F_n depends on the two previous terms, it is necessary to have two base cases. The Fibonacci sequence is a special kind of Lucas sequence. Other Lucas sequences specify different values for the two base cases (and sometimes coefficients to multiply the previous terms by). ■

19.2.3 Iteration vs. recursion

These mathematical definitions are interesting, but what is their relationship to Java code? So far, we have considered algorithms that are iterative in nature: processing is performed as a sequence of operations on elements of a sequential data structure. We sum the elements of an array by iterating through them from first to last. We multiply two matrices by using nested for loops to sequence through the matrix contents in the proper order. Similarly, one method may make a sequence of one or more calls to other methods. We're confident that such computations terminate because we start at the beginning and work to the end of a finite structure. But what if the structure is not a simple linear or multidimensional array? The path we're trying to find through the maze is of unknown length and may not take a "straight line" through the array.

A method may call other methods to complete its operation. For example, a method that sorts a list of String values calls another method to do pairwise comparison of the values in the list. A method that calls **itself**, either directly or indirectly, is called a *recursive* method.

A recursive method may seem like a circular argument that never ends. In fact, a recursive method only calls itself under certain circumstances. Other times, it does not. A recursive method has the same two parts that a mathematical recursive definition has.

Base case: The operation being computed is done without any recursive calls.

Recursive case: The operation is broken down into smaller pieces, one or more of which results in a recursive call to the method itself.

Each time a method calls itself recursively, it does so on a smaller problem. Eventually it reaches a base case, and the recursion terminates.

A recursive method is useful when a problem can be broken down into smaller subproblems where each subproblem has the same structure as the original, complete problem. These subproblems can be solved by recursive calls and the results of those calls assembled to create a larger solution.

Recursive methods are often surprisingly small given their complexity. Each recursive call only makes a single step forward in the process of solving the problem. In fact, it can appear that the problem is never solved. The code has something like a "leap of faith" inside of it. Assuming that you can solve a smaller subproblem, how do you put the solutions together to solve the full problem? This assumption is the leap of faith, but it works out as long as the subproblems get broken down into smaller and smaller pieces that eventually reach a base case.

From a theoretical standpoint, any problem that can be solved iteratively can be solved

recursively, and vice versa. Iteration and recursion are equivalent in computational power. Sometimes it is more efficient or more elegant to use one approach or the other, and some languages are designed to work better with a given approach.

19.2.4 Call stack

Many programmers who are new to recursion feel uncomfortable about the syntax. How can a method call itself? What does that even mean?

Recursion in Java is grounded in the idea of a call stack. We discuss the stack abstract data type in [Chapter 18](#). A similar structure is used to control the flow of control of a program as it calls methods.

Recall that a stack is a first in, last out (FILO) data structure. Each time a method is called, its local variables are put on the call stack. As the method executes, a pointer to the current operation it is executing is kept on the call stack as well. This collection of local variables and execution details for a method call is called the *stack frame* or *activation record*. When another method is called, it pushes its own stack frame onto the call stack as well, and its caller remembers what it was executing before the call. When a method returns, it pops its stack frame (the variables and state associated with its execution) off the call stack.

A recursive method is called in exactly the same way. It puts another copy of its stack frame on the call stack. Each call of the method has its own stack frame and operates independently. There is no way to access the variables from one call to the next, other than by passing in parameters or returning values.

[Figure 19.3](#) shows the stack frames being pushed onto the call stack as the `main()` method calls the `factorial()` method, starting with the argument 4. The `factorial()` method recursively calls itself with successively smaller values.

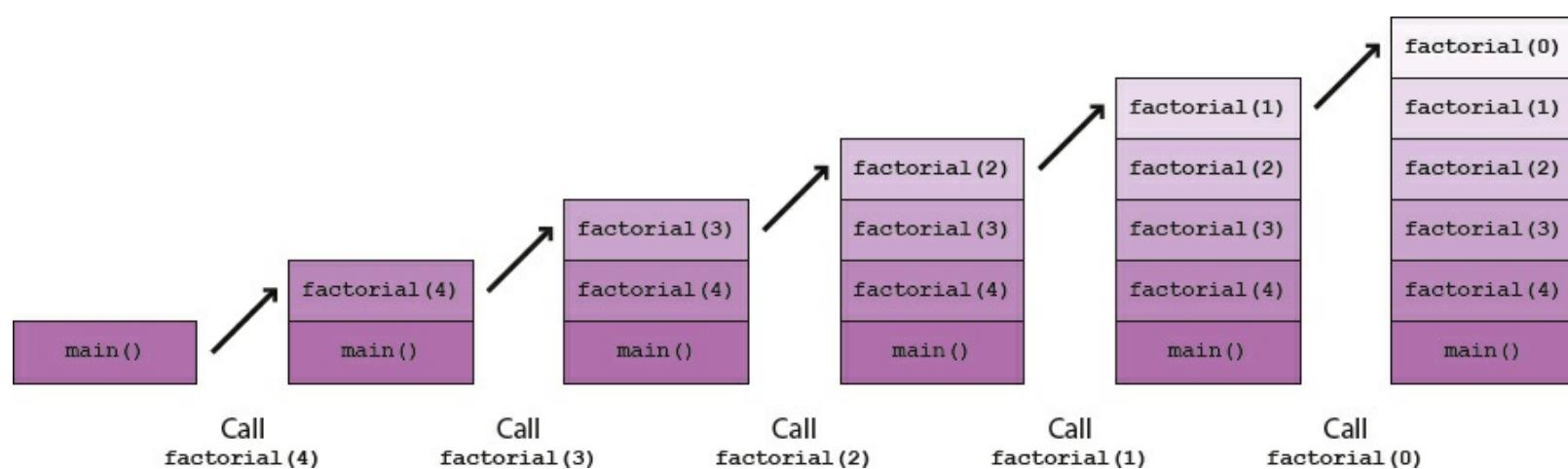


Figure 19.3: Successive recursive method calls getting added to the call stack.

[Figure 19.4](#) shows the stack frames popping off the call stack as each call to `factorial()` returns. As the answers are returned, they are incorporated into the answer that is generated and returned to the next caller in the sequence until the final answer 24 ($4!$) is returned to `main()`.

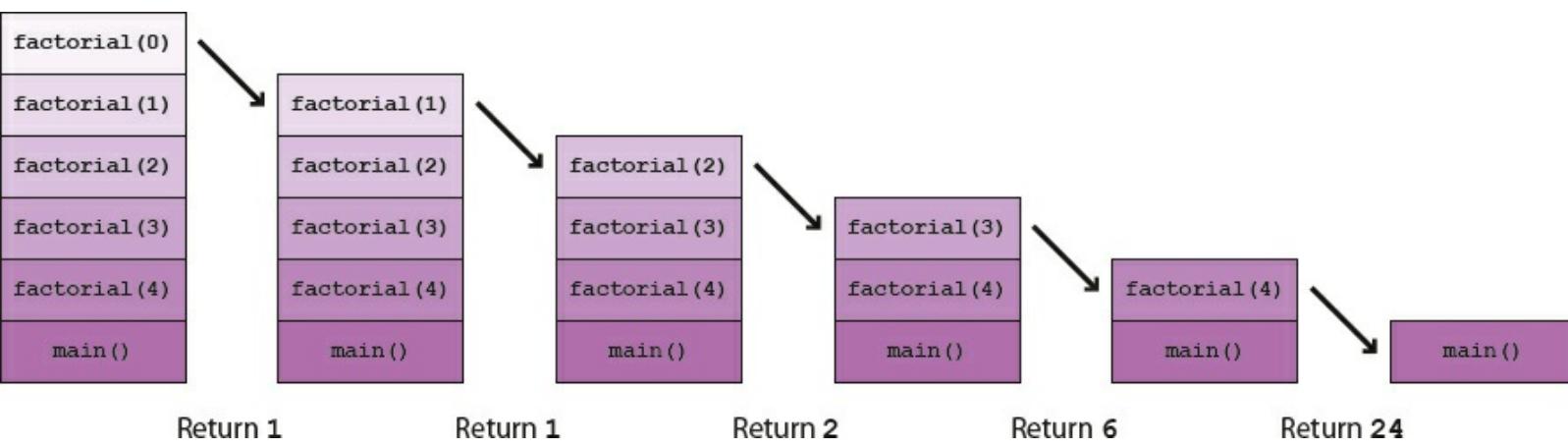


Figure 19.4: Recursive methods returning results to their callers.

19.3 Syntax: Recursive methods

Unlike many **Syntax** sections in other chapters, there is no new Java syntax to introduce here. Any method that calls itself, directly or indirectly, is a recursive method. Recursive methods are simply methods like any others, called in the normal way.

The real difficulty in learning to program recursively lies in breaking out of the way you are used to thinking about program control flow. All that you have learned about solving problems with iteration in previous chapters may make it harder for you to embrace recursion.

Iteration views the whole problem at once and tries to sequence all the pieces of the solution in some organized way. Recursion is only concerned with the current step in the solution. If the current step is one in which the answer is clear, you're in a base case. Otherwise, the solution takes one step toward the answer and then makes the leap of faith, allowing the recursion to take care of the rest. Programmers who are new to recursion are often tempted to do too much in each recursive call. Don't rush it!

The use of recursion in languages like Java owes much to the development of *functional programming*. In many functional languages (such as Scheme), there are no loops, and **only** recursion is allowed. In a number of these languages, there is no assignment either. Each variable has one value for its entire lifetime, and that value comes as a parameter from whatever method called the current method.

It may seem odd to you, but this approach is a good one to follow when writing recursive methods. Try **not** to assign variables inside your methods. See if the work done in each method can be passed on as an argument to the next method rather than changing the state inside the current method. In that way, each recursive method is a frozen snapshot of some part of the process of solving the problem. Of course, this guideline is only a suggestion. Many practical recursive methods need to assign variables internally, but a surprisingly large number do not.

Because the data inside these methods is tied so closely to the input parameters and the return values given back to the caller, these methods are often made **static**. Ideally, recursive methods do not change the state of member variables or class variables. Again, sometimes changing external state is necessary, but recursive methods are meant to take in only their input parameters and give back only return values. Recursive code that reads and writes variables inside of objects or classes can be

difficult to understand and debug, since it depends on outside data.

With this information as background, we focus on examples for the rest of this section. Because recursion is a new way of thinking, approach these examples with an open mind. Many students have the experience that recursion makes no sense until they see the right example. Then, the way it works suddenly “clicks.” Do not be discouraged if recursion seems difficult at first.

In this section, we work through examples of factorial computation, Fibonacci numbers, the classic Tower of Hanoi problem, exponentiation. These problems are mathematical in nature because the mathematical recursion is easy to model in code. The next section applies recursion to processing data structures.

Example 19.4: Factorial implemented recursively

In our first example of a recursive implementation, we return to the factorial function. Recall the recursive definition that describes the function.

$$n! = \begin{cases} 1 & \text{(Base Case) if } n = 0 \\ n \cdot (n-1)! & \text{(Recursive Case) if } n > 0 \end{cases}$$

By translating this mathematical definition almost directly into Java, we can generate a method that computes the factorial function.

```
public static long factorial( int n ) {  
    if (n == 0)      // base case  
        return 1;  
    else            // recursive case  
        return n * factorial( n -1 );  
}
```

Note the base case and recursive case are exactly the same as in the recursive definition. The return type of the method is **long** because factorial grows so quickly that only the first few values are small enough to fit inside of an **int**. ■

Example 19.5: Fibonacci implemented recursively

Let us return to the recursive definition of Fibonacci.

$$F_n = \begin{cases} 1 & \text{(Base Cases) if } n = 0 \text{ or } n = 1 \\ F_{n-1} + F_{n-2} & \text{(Recursive Case) if } n > 1 \end{cases}$$

Like factorial, this definition translates naturally into a recursive method in Java.

```
public static int fibonacci( int n ) {  
    if( n == 0 || n == 1 ) // base cases  
        return 1;
```

```

else      // recursive case
    return fibonacci(n -1) + fibonacci(n -2);
}

```

One significant problem with this example is performance. In this case, the double recursion performs a great deal of redundant computation.

Exercise 19.3

One technique for eliminating redundant computation in recursion is called *memoization*. Whenever the value for a subproblem is computed, we note down the result (like a memo). When we go to compute a value, we first check to see if we have already found it.

To perform memoization for Fibonacci, we can pass an array of `int` values of length $n + 1$. The values in this array all begin with a value of 0. When computing the Fibonacci value for a particular n , we first check to see if its value is in the array. If not, we perform the recursion and store the result in the array.

```

public static int fibonacci( int n, int [] results ) {
    if( results [n] == 0 ) {
        if( n == 0 || n == 1 )
            results [n] = 1;
        else
            results [n] = fibonacci (n -1) + fibonacci (n -2);
    }
    return results [n];
}

```

This change makes the computation of the n^{th} Fibonacci number much more efficient; however, even more efficient approaches are described in the exercises. ■

Exercise 19.6

Exercise 19.8

Example 19.6: Tower of Hanoi

The famous Tower of Hanoi puzzle is another example commonly used to illustrate recursion. In this puzzle, there are three poles containing a number of different sized disks. The puzzle begins with all disks arranged in a tower on one pole in decreasing size, with the smallest diameter disk on top and the largest on the bottom. [Figure 19.5](#) shows an example of the puzzle. The goal is to move all the disks from the initial pole to the final pole, with two restrictions:

1. only one disk can be moved at a time
2. a larger disk can never be placed on top of a smaller disk

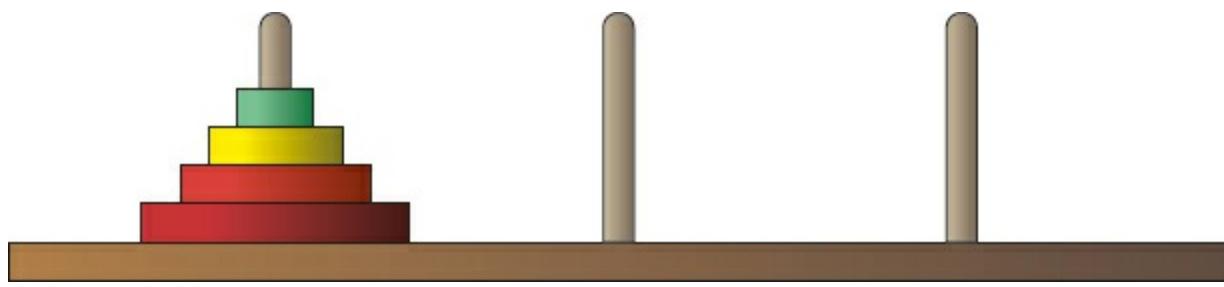


Figure 19.5: Tower of Hanoi puzzle with 4 disks on the initial pole.

The extra pole is used as a holder for intermediate moves. The idea behind the recursive solution follows.

Base Case: Moving one disk is easy. Just move it from the pole it's on to the destination pole.

Recursive Case: In order to move $n > 1$ disks from one pole to another, we can move $n - 1$ disks to an intermediate pole, move the n^{th} disk to the destination pole, then move the $n - 1$ disks from the intermediate pole to the destination pole.

The Tower of Hanoi solution in Java translates this outline into code.

Program 19.1: A recursive solution to the Tower of Hanoi with four disks and poles named '**A**', '**B**', and '**C**'. (TowerOfHanoi.java)

```

1 public class TowerOfHanoi {
2     public static void main ( String [] args ) {
3         move (4, 'A', 'C', 'B');
4     }
5
6     public static void move (int n, char fromPole, char toPole,
7             char viaPole ) {
8         if (n == 1)
9             System.out.format (
10                 " Move disk from pole %c to pole %c.\n",
11                 fromPole, toPole );
12         else {
13             move (n - 1, fromPole, viaPole, toPole );
14             move (1, fromPole, toPole, viaPole );
15             move (n - 1, viaPole, toPole, fromPole );
16         }
17     }
18 }
```

A legend tells of monks that are solving the Tower of Hanoi puzzle with 64 disks. The legend predicts that the world will end when they finish. Run the implementation above with different numbers of disks to see how long the sequence of moves is. Try small numbers of disks, since large

numbers of disks takes a very long time. ■

Example 19.7: Exponentiation

Both Fibonacci and the Tower of Hanoi have natural recursive structures. In the case of Fibonacci one way to implement its natural recursive definition results in very wasteful computation. In the case of the Tower of Hanoi, the **only** way to solve the problem takes an excruciatingly long amount of time.

However, we can apply recursion to many practical problems and get efficient solutions. Consider the problem of exponentiation, which looks trivial: Given a rational number a and a positive integer n , find the value of a^n .

It's tempting to call `Math.pow(a, n)` or to use a short `for` loop to compute this value, but what if neither tool existed in Java? A simple recursive formulation can describe exponentiation.

$$a^n = \begin{cases} a & \text{(Base case) if } n = 1 \\ a \cdot a^{n-1} & \text{(Recursive case) if } n > 1 \end{cases}$$

As with factorial and Fibonacci, directly converting the recursive definition into Java syntax yields a method that computes the correct value.

```
public static double power ( double a, int n ) {  
    if( n == 1 ) // base case  
        return a;  
    else // recursive case  
        return a * power (a, n - 1);  
}
```

Admittedly, this method only works for positive integer values of n . Ignoring that limitation, what can we say about its efficiency? For any legal value of n , the method is called n times. If n has a small value, like 2 or 3, the process is quick. But if n is 1,000,000 or so, the method might take a while to finish. Another problem is that stack size is limited. On most systems, the JVM crashes with a `StackOverflowError` if a method tries to call itself recursively a 1,000,000 times.

If we limit n to a power of 2, we can do something clever that makes the method much more efficient with many fewer recursive calls. Consider this alternative recursion definition of exponentiation.

$$a^n = \begin{cases} a & \text{(Base Case) if } n = 1 \\ \left(a^{\frac{n}{2}}\right)^2 & \text{(Base Case) if } n > 1 \end{cases}$$

Recalling basic rules of exponents, $a^n = \left(a^{\frac{n}{2}}\right)^2$, but what does that buy us? If we structure our method correctly, we cut the size of n in half at each recursive step instead of only reducing n by 1.

```
public static double power ( double a, int n ) {
```

```

if( n == 1 ) // base case
    return a;
else { // recursive case
    double temp = power (a, n /2) ;
    return temp * temp ;
}

```

Note that we only make the recursive call once and save it in temp. If we made two recursive calls, we would no longer be more efficient than the previous method. That version took n recursive calls. How efficient is this version? The answer is the number of times you have to cut n in half before you get 1. Let's call that value x . Recall that n is a power of 2, meaning that $n = 2^k$ for some integer $k \geq 0$.

$$\begin{aligned}
\left(\frac{1}{2}\right)^x \cdot n &= 1 \\
2^x \left(\frac{1}{2}\right)^x \cdot n &= 2^x \\
n &= 2^x \\
2^k &= 2^x \\
k &= x \\
\log_2 n &= x
\end{aligned}$$

In other words, the number of times you have to divide n in half to get 1 is the logarithm base 2 of n , written $\log_2 n$. The logarithm function is the inverse of exponentiation. It cuts any number down to size very quickly (just as exponentiation blows up the value of a number very quickly). For example, $2^{20} = 1,048,576$. Thus, $\log_2 1,048,576 = 20$. The original version of power() would have to make 1,048,576 calls to raise a number to that power. This second version would only have to make 20 calls.

It is critical that n is a power of 2 (1, 2, 4, 8, ...), otherwise the process of repeatedly cutting n in half loses some data due to integer division. The problem is that, at some point in the recursion, the value of n is odd unless you start with a power of 2. There is a way to extend this clever approach to all values of n , even and odd, but we leave it as an exercise. ■

Exercise 19.7

Recursion offers elegant ways to compute mathematical functions like those we have explored in this section. Recursion also offers powerful ways to manipulate data structures. As we show in the next section, recursive methods are especially well suited to use with recursive data structures.

19.4 Syntax: Recursive data structures

Because recursion can be used to do anything that iteration can do, it is clear that data structures can

be processed recursively. For example, the following recursive method reverses the contents of an array. It keeps track of the position in the array it is swapping with the position parameter. This method is initially called with a value of 0 passed as an argument for position.

```
public static void reverse ( int [] array, int position ) {  
    if( position < array.length / 2 ) {  
        int temp = array [ position ];  
        array [ position ] = array [ array.length - position - 1];  
        array [ array.length - position - 1] = temp ;  
        reverse ( array, position + 1 );  
    }  
}
```

Note that nothing is done in the base case for this recursive method. The recursion swaps the first element of the array (at index 0) with the last (at index `array.length - 1`). Recursion continues until position has reached half the length of array. If execution continued past the halfway point, it would begin to swap elements that had already been swapped.

Exercise 19.5

Although it is possible to reverse an array recursively, there is usually no advantage in doing so. We introduce bubble sort and selection sort in previous chapters, but neither of these algorithms is very fast. Many of the best sorting algorithms are recursive, as in the following example of merge sort.

Example 19.8: Merge sort

Merge sort is an efficient sorting algorithm that is usually implemented recursively. The idea of the sort is to break a list of items in half and recursively merge sort each half. Then, these two sorted halves are merged back together into the final sorted list. The base case of the recursion is when there's only a single item in the list, since a list with only one thing in it is, by definition, sorted.

Here is a method that recursively sorts an `int` array using the merge sort algorithm.

```
public static void mergeSort( int [] array ) {  
    if( array.length > 1 ) {  
        int[] a = new int [array.length / 2];  
        int[] b = new int [array.length - a.length];  
        for( int i = 0; i < a.length; ++i )  
            a [i] = array [i];  
        for( int i = 0; i < b.length; ++i )  
            b [i] = array [a.length + i];  
        mergeSort( a );  
        mergeSort( b );  
        merge( array, a, b );  
    }  
}
```

The mergeSort() method is quite short and appears to do very little. It starts by creating arrays a and b and copying roughly half of the elements in array into each. We make a half the size of array, but we can't do the same thing for b because an odd length for array would leave us without enough space in a and b to hold everything from array. Instead, we let b hold however much is leftover after the elements for a have been accounted for.

Then, arrays a and b are recursively sorted. Finally, these two sorted arrays are merged back into array in sorted order using a helper method called merge(). This method is non-recursive and does much of the real work in the algorithm.

```
public static void merge( int [] array, int [] a, int [] b ) {  
    int aIndex = 0;  
    int bIndex = 0;  
    for( int i = 0; i < array.length; ++i ) {  
        if( bIndex >= b.length )  
            array [i] = a [aIndex++];  
        else if( aIndex >= a.length )  
            array [i] = b [bIndex++];  
        else if( a [aIndex] <= b [bIndex] )  
            array [i] = a [aIndex++];  
        else  
            array [i] = b [bIndex++];  
    }  
}
```

The merge() method loops through all the elements in array, filling them in. We keep two indexes, aIndex and bIndex, that keep track of our current positions in the a and b arrays, respectively. This method assumes that a and b are sorted and that the sum of their lengths is the length of array. We want to compare each element in a and b, always taking the smaller and putting it into the next available location in array. Since the next smallest item could be in either a or b, we never know when we'll run out of elements in either array. That's why the first two **if** statements in the merge() method check to see if the bIndex or the aIndex is already past the last element in its respective array. If so, the next element from the other array is automatically used. By the time the third **if** statement is reached, we are certain that both indexes are valid and can compare the elements at those locations to see which is smaller.

Sorting lists using the merge sort algorithm seems more complicated than using bubble sort or selection sort, but this additional complication pays dividends. For large lists, merge sort performs much faster than either of those sorts. In fact, it is comparable in speed to the best general sorting algorithms that are possible. ■

Exercise 19.15

Although recursive sorting algorithms are useful for arrays, recursion really shines when manipulating *recursive data structures*. A recursive data structure is one that is defined in terms of itself. For example, class X is recursive if there is a field inside X with type X.

```
public class X {  
    private int a, b;  
    private X x;  
}
```

The linked list examples from [Chapter 18](#) are recursive data structures, since a linked list node is defined in terms of itself. You may not have thought of the linked list Node class as being recursive since it simply has a reference to another Node inside it. However, this self-reference is the essence of a recursive data structure.

Data structures are often defined recursively. We typically need to represent an unbounded collection of data, but we always write bounded programs to describe the data. A recursive data structure allows us to bridge the gap between a compile-time, fixed length definition and a run-time, unbounded collection of objects.

Recursive data structures have a base case to end the recursion. Typically, the end of the recursion is indicated by a link with a `null` value. For example, in the last node of a linked list, the link field is `null`. Unsurprisingly, recursive methods are frequently used to manipulate recursive data structures.

Example 19.9: Recursive linked list size

How would you get the size of a linked list? The implementation in [Program 18.5](#) keeps track of its size as it grows, but what if it didn't? A standard way to count the elements in the list would be to start with a reference to the head of the list and a counter with value zero. As long as the reference is not `null`, add one to the counter and set the reference to the next element on the list. [Program 19.2](#) counts the elements in this way.

Program 19.2: Linked list implementation whose `size()` method counts its elements iteratively.
(IterativeListSize.java)

```
1  public class IterativeListSize {  
2      private static class Node {  
3          public String value ;  
4          public Node next ;  
5      }  
6  
7      private Node head = null ;  
8  
9      public void add( String value ) {  
10         Node temp = new Node ();  
11         temp.value = value ;  
12         temp.next = head ;  
13         head = temp ;  
14     }  
15  
16     public int size () {
```

```

17     Node current = head ;
18     int counter = 0;
19     while ( current != null ) {
20         current = current.next ;
21         counter++;
22     }
23     return counter ;
24 }
25 }
```

An alternative way to count the number of elements in a linked list is to use the natural recursion of the linked list itself. We can say that the length of a linked list is 0 if the list is empty (the current link is `null`), otherwise, it is one more than the size of the rest of the list.

[Program 19.3](#) counts the elements in a linked list using this recursive procedure. Note that there is a non-recursive `size()` method that calls the recursive `size()` method. This non-recursive method is called a *proxy method*. The recursive method requires access to the internals of the data structure. The proxy method calls the recursive method with the appropriate starting point (`head`), while providing a public way to get the list's size without exposing its internals.

[Exercise 19.9](#)

[Exercise 19.10](#)

Program 19.3: Linked list implementation with a recursive `size()` method for counting its elements.
(`RecursiveListSize.java`)

```

1  public class RecursiveListSize {
2      private static class Node {
3          public String value ;
4          public Node next ;
5      }
6
7      private Node head = null ;
8
9      public void add( String value ) {
10         Node temp = new Node ();
11         temp.value = value ;
12         temp.next = head ;
13         head = temp ;
14     }
15
16     // proxy method
17     public int size () {
18         return size ( head );
19     }
```

```

20
21     private static int size ( Node list ) {
22         if( list == null ) // base case
23             return 0;
24         // recursive case
25         return 1 + size ( list.next );
26     }
27 }
```

19.4.1 Trees

A linked list models a linear, one-to-one relationship between its elements. Each item in the list is linked to a maximum of one following element. Another useful relationship to model is a hierarchical, one-to-many relationship: parent to children, boss to employees, directory to files, and so on. These relationships can be modeled using a *tree* structure, which begins with a single *root*, and proceeds through *branches*, to the *leaves*. Typically, the elements of a tree are also called *nodes*, with three special cases:

Root node: The root of the tree has no parents.

Leaf node: A leaf is at the edge of a tree and has no children.

Interior node: An interior node has a parent and at least one child; it is neither the root nor a leaf.

Figure 19.6 shows a visualization of a tree. In nature, a tree has its root at the bottom and branches upward. Since the root is the starting point for a tree data structure, it is almost always drawn at the top.

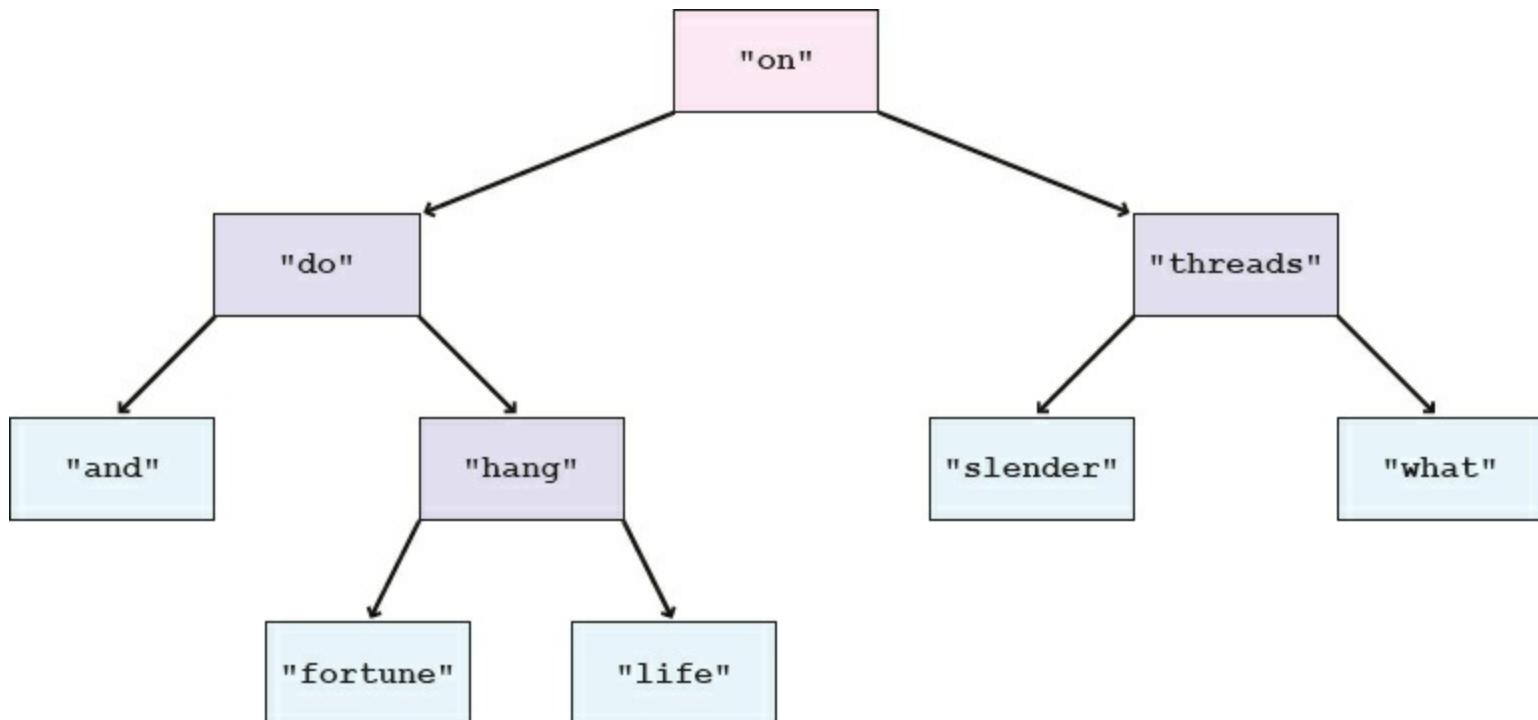


Figure 19.6: Visualization of a tree. The root is shown in light red. The leaves are shown in blue. The

interior nodes are show in light purple.

Abstractly, a tree is either empty (the base case) or contains references to 0 or more other trees (the recursive case). Trees are very useful for storing and retrieving sortable data efficiently. Some applications include dictionaries, catalogs, ordered lists, and any other sorted set of objects. For these purposes, we can define an abstract data type that includes operations such as add() and find().

Exercise 19.13

A special case of a tree that is used frequently is a *binary tree*, in which each node references at most two other trees.

Example 19.10: Binary search tree

A *binary search tree* is a binary tree with the following three properties.

1. The value in the left child of the root is smaller than the value in the root.
2. The value in the right child of the root is larger than the value in the root.
3. Both the left and the right subtrees are binary search trees.

This recursive definition describes a tree that makes items with a natural ordering easy to find. If you are looking for an item, you first look at the root of the tree. If the item you want is in the root, you've found it! If the item you want is smaller than the root, go left. If the item you want is larger than the root, go right. If you ever run out of tree (hit a **null**), the item is not in the tree.

This example is a simple binary tree that can stores a list of strings and print them out in alphabetical order. Program 19.4 shows the Tree class that defines the fields and two public methods, add() and print() that operate on the tree. Each is a proxy method that calls its private recursive version, which takes a reference to a Node object. The Node static nested class contains three fields.

- value: the String value stored at the node
- left: a link to the left subtree
- right: a link to the right subtree

Program 19.4: A class that implements a simple binary search tree ADT for creating a sorted list of strings. (Tree.java)

```
1 public class Tree {  
2     private static class Node {  
3         String value ;  
4         Node left = null ;  
5         Node right = null ;  
6     }  
7 }
```

```

8     private Node root = null ;
9
10    // proxy add
11    public void add ( String value ) {
12        root = add ( value, root );
13    }
14
15    private static Node add ( String value, Node tree ) {
16        if( tree == null ) { // base case
17            tree = new Node ();
18            tree.value = value ;
19        }
20        // left recursive case
21        else if( value.compareTo ( tree.value ) < 0 )
22            tree.left = add ( value, tree.left );
23        // right recursive case
24        else if( value.compareTo ( tree.value ) > 0 )
25            tree.right = add ( value, tree.right );
26        return tree ;
27    }
28
29    // proxy print
30    public void print () {
31        print ( root );
32    }
33
34    private static void print ( Node tree ) {
35        if( tree != null ) {
36            print ( tree.left );
37            System.out.println ( tree.value );
38            print ( tree.right );
39        }
40    }
41 }
```

[Figure 19.7](#) shows a visualization of the contents of this implementation of a binary search tree. As with a linked list, an “X” is used in place of arrows that point to `null`.

Tree object

Node objects

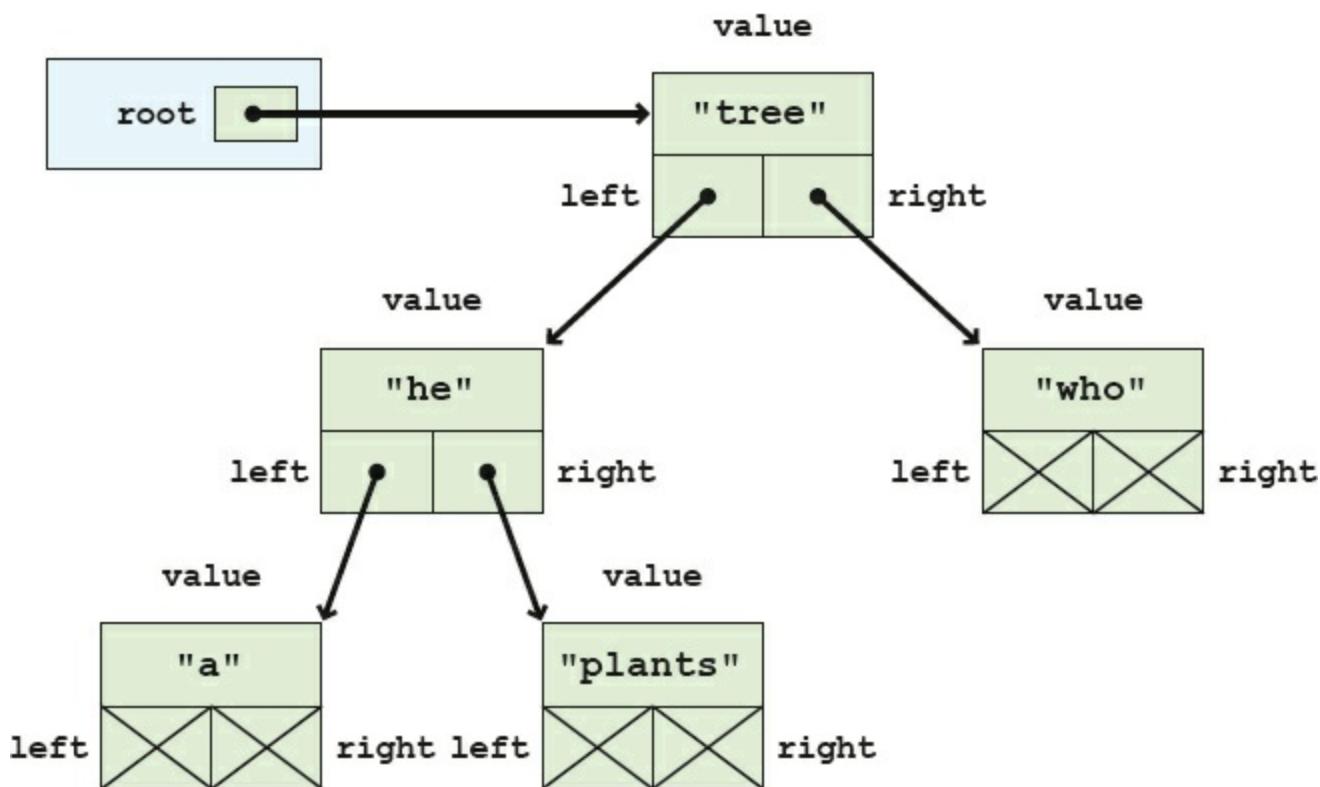


Figure 19.7: Visualization of a tree implementation with classes.

The recursive add() method first checks to see if the current subtree is empty (`null`). If so, it creates a new Node and puts value inside it. If the current subtree is not `null`, it checks to see if value is smaller or larger than the value at the root of the subtree. If it is smaller, it recurses down the left subtree. If it is larger, it recurses down the right subtree. If value is already in the root node, it does nothing.

Remember that all parameters are pass by value in Java. Thus, assigning a new Node to tree does not by itself change anything at higher levels of the tree. What does change the links in the parent of the current subtree is returning the tree pointer. If the recursive call to add() was made with a left or a right subtree, the left or right link, respectively, of the parent Node is assigned the return value. If the call was made with root, the parent of the entire tree, the non-recursive add() method sets its value when the recursive add() returns.

Exercise 19.11

Exercise 19.12

The recursive print() method starts by walking down the left subtree. Those values are all alphabetically less than the value of the current node. When it finishes, it prints the current node value. Finally, it walks the right subtree to print the values that alphabetically follow the value in the current node. This path through the nodes of the tree is called an *inorder traversal*.

With the power of a binary search tree, it takes virtually no code at all to store a list of String values and then print them out in sorted order. [Program 19.5](#) gives an example of this process using a Tree object for storage.

Program 19.5: A program to read String values, store them in a binary search tree, and print the results in sorted order. (ReadAndSortStrings.java)

```
1 import java.util.Scanner ;  
2  
3 public class ReadAndSortStrings {  
4     public static void main ( String [] args ) {  
5         Tree tree = new Tree ();  
6         Scanner in = new Scanner ( System.in );  
7  
8         while ( in.hasNextLine () )  
9             tree.add ( in.nextLine () );  
10  
11     tree.print ();  
12 }  
13 }
```

Binary search trees (and other trees, including heaps, tries, B-trees, and more) are fundamental data structures that have been studied heavily. Designing them to have efficient implementations that balance the size of their left and right subtrees is an important topic that is beyond the scope of this book. ■

19.4.2 Generic dynamic data structures and recursion

Combining dynamic data structures and generics from the previous chapter and recursion from this chapter gives us the full power of generic dynamic data structures and recursive methods to process them.

Example 19.11: Binary search tree to hold integers

Consider [Program 19.6](#), which implements a tree that stores values of type Integer. Although it would be more efficient to store int values, we use the Integer wrapper class to ease our eventual transition into a parameterized generic type.

Program 19.6: A variant of [Program 19.4](#) that stores Integer values instead of String values. (IntegerTree.java)

```
1 public class IntegerTree {  
2     private static class Node {  
3         Integer value ;  
4         Node left = null ;  
5         Node right = null ;  
6     }  
7  
8     private Node root = null ;  
9 }
```

```

10 // proxy add
11 public void add ( Integer value ) {
12     root = add ( value, root );
13 }
14
15 private static Node add ( Integer value, Node tree ) {
16     if( tree == null ) { // base case
17         tree = new Node ();
18         tree.value = value ;
19     }
20     // left recursive case
21     else if( value.compareTo ( tree.value ) < 0 )
22         tree.left = add ( value, tree.left );
23     // right recursive case
24     else if( value.compareTo ( tree.value ) > 0 )
25         tree.right = add ( value, tree.right );
26     return tree ;
27 }
28
29 // proxy print
30 public void print () {
31     print ( root );
32 }
33
34 private static void print ( Node tree ) {
35     if( tree != null ) {
36         print ( tree.left );
37         System.out.println ( tree.value );
38         print ( tree.right );
39     }
40 }
41 }

```



It is a waste to create class IntegerTree, which is identical to Tree except that the type String has been replaced by Integer. As in [Chapter 18](#), we want our data structures, recursive or otherwise, to hold any type. In this way, we can reuse code.

Example 19.12: Defining a generic binary search tree

[Program 19.7](#) defines a generic version of the Tree class. This example is complicated by the fact that we need to be able to compare the value we want to store with the value in each Node object. We can't make a tree with any arbitrary type. Objects of the type **must** have the ability to be compared to each other and ordered. Thus, we use a *bounded type parameter* specifying that the type T stored in each Tree must implement the Comparable interface. This requirement complicates the generic syntax

significantly but guarantees that any type that cannot be compared with itself is rejected at compile-time.

Program 19.7: A class that implements a generic tree. (GenericTree.java)

```
1 public class GenericTree <T extends Comparable < Comparable < T >>> {
2     private static class Node <T extends Comparable < Comparable < T >>>{
3         T value ;
4         Node left = null ;
5         Node right = null ;
6     }
7
8     private Node <T> root = null ;
9
10    // proxy add
11    public void add (T value ) {
12        root = add ( value, root );
13    }
14
15    private Node <T> add(T value, Node tree ) {
16        if( tree == null ) { // base case
17            tree = new Node <T>();
18            tree.value = value ;
19        }
20        // left recursive case
21        else if( value.compareTo ( tree.value ) < 0 )
22            tree.left = add ( value, tree.left );
23        // right recursive case
24        else if( value.compareTo ( tree.value ) > 0 )
25            tree.right = add ( value, tree.right );
26        return tree ;
27    }
28
29    // proxy print
30    public void print () {
31        print ( root );
32    }
33
34    private void print (Node <T> tree ) {
35        if( tree != null ) {
36            print ( tree.left );
37            System.out.println ( tree.value );
38            print ( tree.right );
39        }
50
```

```
40    }
41 }
```

First, note that the recursive methods are no longer **static**. The generic syntax for keeping them **static** is unnecessarily complex. The type specifier T **extends Comparable<T>** guarantees that type T implements the interface Comparable<T>. The generic Comparable interface defined in the Java AP is as follows.

```
public interface Comparable <T> {
    int compareTo (T object );
}
```

The syntax for generics in Java with type bounds is complex, and we only scratch the surface here. The good news is that these subtleties are more important for people designing data structures and libraries and come up infrequently for programmers who are only using the libraries. ■

Example 19.13: Using a generic class

Program 19.8 uses the generic tree class to create two kinds of trees, a tree of String objects and a tree of Integer objects. Java library implementations of binary search trees are available as the TreeSet and TreeMap classes.

Program 19.8: Program to create two trees with different underlying types.
(ReadAndSortGenerics.java)

```
1 import java.util.Random ;
2 import java.util.Scanner ;
3
4 public class ReadAndSortGenerics {
5     public static void main ( String [] args ) {
6         Scanner in = new Scanner ( System.in );
7
8         GenericTree < String > stringTree = new GenericTree < String > () ;
9         GenericTree < Integer > integerTree =
10            new GenericTree < Integer > () ;
11
12        while ( in.hasNextLine () )
13            stringTree.add ( in.nextLine () );
14        stringTree.print ();
15
16        Random random = new Random ();
17        for ( int i = 0; i < 10; i ++ )
18            integerTree.add ( random.nextInt () );
19        integerTree.print ();
20    }
21 }
```

19.5 Solution: Maze of doom

Our algorithm for solving the maze follows the conventional pencil-and-paper method: trial and error! We mark locations in the maze with '*' as we explore them. If we come to a dead end, we unmark the location (by replacing '*' with ' ') and return to our previous location to try a different direction.

Start at the beginning square of the maze, which must be a passageway. Mark that location as part of the path by putting '*' in the cell. Now, what can we do? There are, in general, four possible directions to head: up, down, left, or right. If that direction doesn't take you outside the bounds of the array, then you find either a wall or a passageway. If you have been walking through the maze, you may also find a part of the current path (often the square you were on before the current one).

Suppose from your current point in the maze you could send a scout ahead in each of the four directions. If the direction did not take the scout out of bounds, he would find either a wall, a part of the current path (the path that led into that space), or an open passageway. If the scout doesn't find an open passageway, he reports back that that direction doesn't work.

On the other hand, if the scout finds an open passageway, what does he do? Brace yourself! He does the exact same thing you just did: Sends out scouts of his own in each of the four possible directions.

With careful, consistent coding, you and the scout follow the exact same process. And the scout's scouts. And so on. There is, in fact, only one method and instead of calling a scout method to investigate each of the squares in the four directions, you call yourself recursively.

```
1 import java.util.Scanner ;  
2  
3 public class MazeSolver {  
4     private char [][] maze ;  
5     private int rows, columns ;  
6  
7     public static void main ( String [] args ) {  
8         MazeSolver solver = new MazeSolver ();  
9         if( solver.solve (0, 0) )  
10             System.out.println ("\\ nSolved !");  
11         else  
12             System.out.println ("\\ nNot solvable !");  
13         solver.print ();  
14     }  
15 }
```

The MazeSolver class needs a two-dimensional array of **char** values to store a representation of the maze. Likewise, it is convenient to store the number of rows and columns in member variables.

The `main()` method creates a new `MazeSolver` object and then calls its `solve()` method with a starting location of `(0, 0)`. It prints an appropriate message depending on whether or not the maze was solved. Finally, it prints out the maze, which includes a path marked with '*' symbols if the maze is solvable.

```

16     public MazeSolver () {
17         Scanner in = new Scanner ( System.in );
18         rows = in. nextInt ();
19         columns = in. nextInt ();
20         in. nextLine ();
21         maze = new char [ rows ][ columns ];
22         for ( int row = 0; row < rows ; row ++ ) {
23             String line = in. nextLine ();
24             System.out.println ( line );
25             for ( int column = 0; column < columns ; column ++ ) {
26                 maze [ row ][ column ] = line.charAt ( column );
27             }
28         }
29     }

```

The constructor for MazeSolver creates a Scanner. It assumes that the file describing the maze is redirected from standard input, although it would be easy to modify the constructor to take a file name and read from there instead. Next, it reads two integers and sets rows and columns to those values. It allocates a two-dimensional array of `char` values with `rows` rows and `columns` columns. Finally, it reads through the file, storing each line of `char` values into this array. As it reads, it prints out each line to the screen, showing the initial (unsolved) maze.

```

31     public void print () {
32         for ( int row = 0; row < rows ; row ++ ) {
33             for ( int column = 0; column < columns ; column ++ )
34                 System.out.print ( maze [ row ][ column ] );
35             System.out.println ();
36         }
37     }

```

The `print()` method is a utility method that prints out the maze. It iterates through each row, printing out the values for the columns in that row.

```

39     public boolean solve ( int row, int column ) {
40         if( row < 0 || column < 0 || row >= rows || column >=
41             columns )
42             return false ;
43         else if( maze [ row ][ column ] == 'E' )
44             return true ;
45         else if( maze [ row ][ column ] != ' ' )
46             return false ;
47         else {
48             maze [ row ][ column ] = '*';
49             if( solve ( row - 1, column ) || solve ( row + 1, column ) ||
50                 solve ( row, column - 1 ) || solve ( row, column + 1 ) )
51                 return true ;
52             maze [ row ][ column ] = ' ';
53         }
54     }

```

```

49     solve (row, column - 1) || solve (row, column + 1) )
50     return true ;
51
52     else {
53         maze [ row ][ column ] = '+' ;
54         return false ;
55     }
56 }
57 }
```

The heart of the solution is the recursive method `solve()`. The `solve()` method takes two parameters, `row` and `column`, and tries to find a solution to the maze starting at location `maze[row][column]`. It assumes that the maze is filled with `'+'` for walls, `' '` for passageways, and may include `'*'` characters at locations that are part of the partially completed solution.

If `solve()` is able to find a solution from the current location, it returns `true`, otherwise it returns `false`. There are three base cases for the current location in the maze.

1. The current location is outside the maze. Return `false`.
2. The current location is the final location. We have a winner, return `true!`
3. The current location is not a passage (either a wall or a location in the current path that has already been marked). This call to `solve` is not making progress toward the finish. Return `false`.

If none of the base cases applies, then the current location, which must contain a `' '` character, **might** be on a successful path, so `solve()` gives it a try. The method tentatively marks the current position with `'*'`. Then, it recursively tries to find a path from the current location to each of the four neighboring cells (line 48). If any of those four neighbors returns `true`, then `solve()` knows it has found a completed path and returns `true` to its caller.

If none of the four neighbors was on a path to the destination, then the current location is not on a path. The method unmarks the current location (by storing a `' '`) and returns `false`. Presumably, its caller figures out what to do next, perhaps calling a different one of its neighbors or giving up and returning `false` to its caller.

The very first call to `solve()` from the `main()` method either returns `true` if a complete path through the maze is found or `false` if no path exists. Note that this solver has no guarantee of finding the **shortest** path through the maze, but if there is at least one path to the goal, it finds one.

19.6 Concurrency: Futures

This section does not deal explicitly with recursion, but it does deal with concurrency and methods in an interesting way. When we call a method in Java, a stack frame for the method is put on the stack, and the thread of execution begins executing code inside the method. When it is done, it returns a value (or not), and execution resumes in the caller. But what if calling the method began executing an

independent thread, and the caller continued on without waiting for the method to return?

This second situation should seem familiar, since it is very much what happens when the start() method is called on a Thread object: the start() method returns immediately to its caller, but an independent thread of execution has begun executing the code in the run() method of the Thread.

What if we only care about the value that is computed by the new thread of execution? We can think of spawning the thread as an asynchronous method call, a value that is computed **at some point** rather than one we have to wait for. The name for such an asynchronous method call is a *future*. In some languages, particularly functional languages, all concurrency is expressed as a future. In Java, only a little bit of code is needed to create threads that can behave like futures. However, the idea of futures is pervasive enough that Java API tools were created to make the process of creating them easy.

We introduce three interfaces and a factory method call that can allow you to use futures in Java. This section is not a complete introduction to futures, but these tools are enough to get you started with them.

The first interface is the Future interface, which allows you to store a reference to the asynchronous computation while it is computing (and before you ask for its value). The second interface is the Callable interface, which is similar to the Runnable interface in that it allows you to specify a class whose objects can be started as independent threads. Both the Future interface and the Callable are generic interfaces that require to specify a type. Remember that futures are supposed to give back an answer, and that's the type that you supply as a parameter. For example, when creating a future that returns an int value, you would create a class that implemented the Callable<Integer> interface, requiring it to contain a method with the signature Integer call(). Likewise, you would store a reference to the future you create in a Future<Integer> reference.

And how do you create such a future? Usually, many futures are running at once to leverage the power of multiple cores. What if you want to create 100 futures but only have 8 cores? The process of creating threads is expensive, and it might not be worthwhile to create 100 threads when only 8 are able to run concurrently. To deal with this problem, the Java API provides classes that implement the ExecutorService interface, which can maintain a fixed-size *pool* of threads. When a thread finishes computing one future, it is automatically assigned another. To create an object that can manage threads this way, call the static factory method newFixedThreadPool() on the Executors class with the size of the thread pool you want create. For example, we can create an ExecutorService with a pool of 8 threads as follows.

```
ExecutorService executor = Executors.newFixedThreadPool(8);
```

Once you have an ExecutorService, you can give it a Callable object of a particular type (such as Callable<Integer>) as a parameter to its submit() method, and it returns a Future object of a matching type (such as Future<Integer>). Then, the future is running (or at least scheduled to run). At any later point you can call the get() method on the Future object, which returns the value of its computation. Like calling join(), calling get() is a blocking call that may have to wait for the future to finish executing.

All of this messy syntax becomes clearer in the following example, which uses futures to compute the sum of the square roots of the first 100,000,000 integers concurrently.

Example 19.14: Futures to sum square roots

To use futures to sum the square roots of integers, we first need a worker class that implements Callable. Since the result of the sum of square roots is a double, it must implement Callable<Double>. Recall that primitive types such as double cannot be used as generic type parameters, requiring us to use wrapper classes in those cases.

```
1 import java.util.concurrent.*;
2
3 public class RootSummer implements Callable < Double > {
4     private int min;
5     private int max;
6
7     public RootSummer ( int min, int max ) {
8         this.min = min ;
9         this.max = max ;
10    }
11
12    public Double call () {
13        double sum = 0.0;
14        for ( int i = min; i < max ; ++i )
15            sum += Math.sqrt ( i );
16        return sum ;
17    }
18 }
```

RootSummer is a simple worker class that takes a min and a max value in its constructor. Its call() method sums the square roots of all the int values greater than or equal to min and less than max. It imports java.util.concurrent.* to have access to the Callable interface.

Of course, we need another class to create the ExecutorService, start the futures running, and collect the results.

```
1 import java.util.concurrent.*;
2 import java.util.ArrayList ;
3
4 public class RootFutures {
5     private static final int THREADS = 10;
6     private static final int N = 100000000;
7     private static final int FUTURES = 1000;
8
9     public static void main ( String [] args ) {
10        ArrayList < Future < Double >> futures =
11            new ArrayList < Future < Double >>( FUTURES );
12        ExecutorService executor =
13            Executors.newFixedThreadPool ( THREADS );
```

```
int work = N / FUTURES ;
```

The first part of RootFutures is setup. The imports give us the concurrency tools we need and a list to store our futures in. We have three constants. THREADS specifies the number of threads to create N gives the number we are going up to. FUTURES is the total number of futures we create considerably larger than the number of threads they share.

Inside the main() method, we create an ArrayList to hold the futures. Since we know the number of futures ahead of time, an array would be ideal. Unfortunately, quirks in the way Java handles generics makes it illegal to create an array with a generic type. Instead, we create an ArrayList with the size we'll need pre-allocated. Next, we create an ExecutorService with a thread pool of size THREADS. Finally, we find the amount of work done by each future by dividing N by FUTURES.

```
16     System.out.println (" Creating futures ... ");
17     for ( int i = 0; i < FUTURES ; i++ ) {
18         Callable < Double > summer =
19             new RootSummer ( 1 + i*work, 1 + (i + 1)* work );
20         Future < Double > future = executor.submit ( summer );
21         futures.add ( future );
22     }
```

In this section of code, we create each future and start it running. First, we instantiate a RootSummer object with the appropriate bounds for the work it's going to compute. Then, we supply that object to the submit() method on the ExecutorService, which returns a Future object. We could have saved a line of code by storing this return value directly into the list futures.

```
24     System.out.println (" Getting results from futures ... ");
25     double sum = 0;
26     for ( Future < Double > future : futures ) {
27         try {
28             sum += future.get ();
29         }
30         catch ( InterruptedException e ) {
31             e. printStackTrace ();
32         }
33         catch ( ExecutionException e ) {
34             e. printStackTrace ();
35         }
36     }
37     executor.shutdown ();
38     System.out.println ("The sum of square roots is: " + sum);
39 }
40 }
```

All that remains is to collect the values from each future. We iterate through the list of futures with a for-each loop and add the return value each future's get() method to our running total sum. Because

`get()` is a blocking call, we have to catch an `InterruptedException` in case we are interrupted while waiting for the future to respond. However, we also have to catch an `ExecutionException` in case an exception occurred during the execution of the future. This exception handling mechanism is one of the big advantages of using futures: Exceptions thrown by the future are propagated back to the thread that gets the answer from the future. Normal threads simply die if they have unhandled exceptions.

After all the values have been read and summed, we shut the `ExecutorService` down. If we had wanted, we could have submitted additional `Callable` objects to it to run more futures. Finally, we print out the result. ■

Exercises

Conceptual Problems

- 19.1 Example 19.1 gave a mathematical recursive definition for $x \times y$. Give a similar recursive definition for $x+y$. The structure is similar to the recursion to determine your current age given in Section 19.2.1.
- 19.2 In principle, every problem that can be solved with an iterative solution can be solved with a recursive one (and vice versa). However, the limited size of the call stack can present problems for recursive solutions with very deep recursion. Why? Conversely, are there any recursive solutions that are impossible to turn into iterative ones?
- 19.3 Consider the first (non-memoized) recursive version of the Fibonacci method given in Example 19.5. How many times is `fibonacci()` called with argument 1 to compute `fibonacci(n)`? Instrument your program and count the number of calls for $n = 2,3,4 \dots 20$.
- 19.4 In the recursive `solve()` method in the `MazeSolver` program given in Section 19.5, the current location in the maze array is set to a blank character (' ') after no solution had been found. What value was in that location? How would the program behave if the value was not changed?

Programming Practice

- 19.5 Exercise 8.11 from Chapter 8 challenges you to write a method to determine whether a `String` contains a palindrome. Recall that a palindrome (if punctuation and spaces are ignored) can be described as a `String` in which the first and last characters are equal, and all the characters in between form a palindrome. Write a recursive method to test if a `String` is a palindrome, using the following signature.

```
public static boolean isPalindrome ( String text, int start, int end )
```

In this method, the `start` parameter is the index of the first character you are examining, and the `end` parameter is the index immediately after the last character you are examining. Thus, it would initially be called with a `String`, 0, and the length of the `String`, as follows.

```
boolean result =
    isPalindrome ( "A man, a plan, a canal : Panama ", 0, 30 );
```

19.6 The efficient implementation of Fibonacci from [Example 19.5](#) eliminates redundant computation through memoization, storing values in an array as they are found. It is possible to carry along the computations of the previous two Fibonacci numbers **without** the overhead of storing an array. Consider the following method signature.

```
public static int fibonacci( int previous, int current, int n)
```

The next recursive call to the fibonacci() method passes in $n - 1$ and suitably altered versions of previous and current. When n reaches 0, the current parameter holds the value of the Fibonacci number you were originally looking for.

The method would be called as follows for any value of n .

```
int result = fibonacci( 0, 1, n );
```

Complete the implementation of this recursive method.

19.7 Write an implementation of fast exponentiation that works for even and odd n . This implementation is exactly the same as the one given at the end of [Example 19.7](#) except when n is odd. Use the following recursive definition of exponentiation to guide your implementation.

$$a^n = \begin{cases} a & (\text{Base Case}) \text{ if } n = 1 \\ \left(a^{\frac{n}{2}}\right)^2 & (\text{Base Case}) \text{ if } n > 1 \text{ and even} \\ a \cdot \left(a^{\frac{n-1}{2}}\right)^2 & (\text{Base Case}) \text{ if } n > 1 \text{ and odd} \end{cases}$$

19.8 [Example 19.5](#) shows two implementations that can be used to find the n^{th} Fibonacci number. With a little bit of math, it is possible to show that there is a closed-form equation that gives the n^{th} Fibonacci number F_n where $F_0 = F_1 = 1$.

$$F_n = \frac{\left(\frac{1+\sqrt{5}}{2}\right)^{n+1} - \left(\frac{1-\sqrt{5}}{2}\right)^{n+1}}{\sqrt{5}}$$

Although this math is a bit ugly, you can plug numbers into this equation and discover the value of F_n quickly, provided that you have an efficient way to raise values to the n^{th} power. Use the recursive algorithm for fast exponentiation from Exercise 19.7 to make an implementation that finds the n^{th} Fibonacci number **very** quickly.

Note that this approach uses real numbers (including $\sqrt{5}$) that need to be represented as **double** values. There are exact methods that use fast exponentiation of integer matrices to do this computation without doing any floating-point arithmetic, but we do not go into those details here.

19.9 [Example 19.9](#) shows a way to calculate the size of a linked list recursively. Add a recursive method called print() to the RecursiveListSize class that prints out the values in the linked list.

recursively, on each line.

19.10 Expand the previous exercise to add another method called `reversePrint()` that prints out the values in the linked list the **opposite** order that they appear. It should take only a slight modification of the `print()` method you have already written.

19.11 Create a recursive `find()` method (and a non-recursive proxy method to call it) for the `Tree` class given in [Program 19.4](#). Its operation is similar to the `add()` method. If the subtree it is examining is empty (`null`), it should return `false`. If the value it is looking for is at the root of the current subtree, it should return `true`. These are the two base cases. If the value it is looking for comes earlier in the alphabet than the value at the root of the current subtree, it should look in the left subtree. If the value it is looking for comes later in the alphabet than the value at the root of the current subtree, it should look in the right subtree. These are the two recursive cases.

19.12 The height of a binary tree is defined as the longest path from the root to any leaf node. Thus, the height of a tree with only a root node in it is 0. By convention, the height of an empty tree is -1.

Create a recursive `getHeight()` method (and a non-recursive proxy method to call it) for the `Tree` class given in [Program 19.4](#). The base case is an empty tree (a null pointer), which has a height of -1. For the recursive case of a non-empty tree, its height is one more than the height of the larger of its two subtrees.

19.13 Create an **interface** that describes a tree ADT. Modify the programs in [Example 19.10](#) to use this **interface**.

Experiments

19.14 Write an iterative version of the factorial function and compare its speed to the recursive version given in the text. Use the `System.currentTimeMillis()` or `System.nanoTime()` methods before and after `for` loops that call the factorial methods 1,000,000 times each for random values.

19.15 Write a program that generates four arrays of random int values with lengths 1,000, 10,000, 100,000, and 1,000,000. Make two additional copies of each array. Then, sort each of three copies of the array with the selection sort algorithm given in [Example 6.2](#), the bubble sort algorithm given in [Section 10.1](#), and the merge sort algorithm given in [Example 19.8](#), respectively. Use the `System.currentTimeMillis()` or `System.nanoTime()` methods to time each of the sorts. Note that both selection sort and bubble sort may take quite a while to sort an array of 1,000,000 elements.

Run the program several times and find average values for each algorithm on each array size. Plot those times on a graph. The times needed to run selection sort and bubble sort should increase quadratically, but the time to run merge sort should increase linearithmically. In other words, an array length of n should take time proportional to n^2 (multiplied by some constant) for selection sort and bubble sort, but it should take time proportional to $n \log n$ (multiplied by some constant) for merge sort. For large arrays, the difference in time is significant.

19.16 Investigate the performance of using recursion to compute Fibonacci numbers. Implement the naive recursive solution, the memoization method, and an iterative solution similar to the memoization method. Use the `System.currentTimeMillis()` or `System.nanoTime()` methods to time the computations for large values of n . (**Warning:** It may take a very long time to compute the n^{th} Fibonacci number with the naive recursive solution.)

19.17 Exercise 6.9 from [Chapter 6](#) explains how binary search can be used to search for a value in a sorted array of values. The idea is to play a “high-low” game, first looking at the middle value. If the value is the one you’re looking for, you’re done. If it is too high, look in the lower half of the array. If it is too low, look in the upper half of the array. Implement binary search both iteratively and recursively. Populate an array with 100,000 `int` values between 1 and 10,000,000 and sort it. Then, search for 1,000,000 random values generated between 1 and 10,000,000 using iterative binary search and then recursive binary search. Use the `System.currentTimeMillis()` or `System.nanoTime()` methods to time each process. Was the iterative or recursive approach faster? By how much?

Chapter 20

File I/O

Kira: *What are those funny marks?*

Jen: *This is all writing.*

Kira: *What's writing?*

Jen: *Words that stay, my master said.*

—The Dark Crystal

20.1 Problem: A picture is worth 1,000 bytes

If you are familiar with bitmap (bmp) files, you know that they can get very large. People often use a technique called data compression to reduce the size of large files for storage. There are many different kinds of compression and many which are tailored to work well on images. Your task is to write a program that will do a particular kind of compression called run length encoding (RLE), which we will test on bitmaps. The idea behind RLE is simple: Imagine a file as a stream of bytes. As you look through the stream, replace repeating sequences of a single byte with a count telling how many times it repeats followed by the byte that repeats. Consider the following sequence.

215 7 7 7 7 7 7 7 7 123 94 94 94 71

Its RLE compressed version could be as follows.

1 215 9 7 1 123 3 94 1 71

Since there is no simple way to keep track of which numbers are counts and which ones are actual bytes of data, we have to keep a count for every byte, even unrepeated ones. In this example, we went down from 15 numbers to 10 numbers, a savings of a third. In the worst case, a file that has no repetition at all would actually double in size after being “compressed” with this kind of RLE. Nevertheless, RLE compression is used in practice and performs very well in some situations.

Your job is to write a Java program that takes two arguments from the command line. The first is either -c for compress or -d for decompress. The second argument is the name of the file to be compressed or decompressed. When compressing, append the suffix .compress to the end of the file name. When decompressing, remove that suffix.

Executing `java BitmapCompression -c test.bmp` from the command line should generate an RLE compressed file called `test.bmp.compress`.

Likewise, executing `java BitmapCompression -d test.bmp.compress` should create an uncompressed file called `test.bmp`. Be sure to make a backup of the original file (in this case `test.bmp`) because decompressing will overwrite a file of the same name.

To perform the compression, go byte by byte through the file. For each repeating sequence of a byte (which can be as short as a single byte long), write the length of the sequence as a byte and then the byte itself into the compressed file. If a repeating sequence is more than 127 bytes, you must break the

sequence into more than one piece since the largest value the `byte` data type can hold in Java is 127. (The `byte` type in Java is always signed, giving a range of -128 to 127.) Decompression simply reads the count then the byte value and writes the appropriate number of copies of the byte value into the decompressed file.

20.2 Concepts: File I/O

Before you can begin to tackle the problem of compressing, or even reading from and writing to, files, some background on files might be useful. By now, you have had many experiences with files: editing Java source files, compiling those files, and running them on a virtual machine, at the very least. You have probably done some word processing, looked at photos, listened to music, and watched videos, all on a computer. Each of these activities centers on one or more files. In order to reach these files, you probably had to look through some directories, which are a special kind of file as well. But, what is a file?

20.2.1 Non-volatile storage

A computer program is like a living thing, always moving and restless. The variables in a program are stored in RAM, which is volatile storage. The data in volatile memory will only persist as long as there is electricity powering it. But programs do not run constantly, and neither do most computers. We need a place to store data between runs of a particular program. Likewise, we need a place to store data when our computer isn't turned on. Both scenarios share a common solution: secondary storage such as hard drives, flash drives, and optical media like CD-ROMs and DVD-ROMs.

Files are not always stored in non-volatile memory. It is possible to load entire files into RAM and keep them there for long periods of time. Likewise, all input and output on Unix and Linux systems are viewed as file operations. Nevertheless, the characteristics of non-volatile memory are often associated with file I/O: slow access times and the possibility of errors in the event of hardware problems.

20.2.2 Stream model

While discussing RLE encoding, we described a file as a stream of bytes, and that is a good definition for a file, especially in Java. Since Java is platform independent, and different operating systems and hardware will deal with the nitty gritty details of storing files in different ways, we want to think about files as abstractly as possible. The idea of a stream of bytes should make a connection with all the other input and output you have done so far. For the most part, file I/O will be similar and, in fact, can use some of the same classes.

Although reading and writing from the files will be very much like reading from the keyboard and writing to the screen, there are a few additional complications. For one thing, you must open a file before you can read and write. Sometimes opening the file will fail: You could try to open a file for reading which does not exist or try to open a file for writing which you do not have permissions for. When reading data, you might try to read past the end of the file or try to read an int when the next item is a String. Unlike reading from the keyboard, you cannot ask the user to try again if there is a mistake in input. To deal with all these possible errors, exception handling will accompany many

different file I/O operations in Java.

20.2.3 Text files and binary files

When talking about files, many people divide files into two categories: *text files* and *binary files*. A text file can be read by humans. That is, when you open a text file with a simple file reader, it will not be filled by gibberish and nonsense characters. A Java source file is an excellent example of a text file.

In contrast, a binary file is a file meant only to be read by a computer. Instead of printing out characters meant to be read by a human, the raw bytes of memory for specific pieces of data are written to binary files. To clarify, if we wanted to write the number 2127480645 in a text file, the file would contain the following.

2127480645

However, if we wanted to write the same number in a binary file, the file would contain the following.

~ÎŒ

If you recall, an `int` in Java uses four bytes of storage. There is a system of encoding called the ASCII table which maps each of the 256 (0–255) numerical bytes to a character. The four characters given above are the ASCII representation of the four bytes of the number 2127480645.

In some sense, the idea of a text file is artificial. All files are binary in the sense that they are readable by a computer. You will take different steps and create different objects depending on whether you want to do file I/O in the text or binary paradigms, but the overall process will be similar in either case.

20.3 Syntax: File operations in Java

20.3.1 The File class

The most basic object for interacting with a file in Java are objects of the `File` class. A `File` object allows you to interact with a file at the operating system level. You can create new files, test to see if a file is a directory, find out the size of a file, and so on. A number of file I/O classes require a `File` object as a parameter. To use the `File` class, import `java.io.File` or `java.io.*`.

To create a `File` object, you can call the `File` constructor with a `String` specifying the name of the file.

```
File file = new File ("file.txt");
```

Doing so will create a virtual file object associated with the name `file.txt` (which may not exist yet) in the working directory of the Java program. In this case, the extension `.txt` doesn't have any real meaning. On many systems, the extension (like `.doc` or `.html`) is used by the operating system to guess which application should open the file. To Java, however, the extension is just part of the file name. A file name passed to the constructor of a `File` object can have any number of periods in it.

Creating a file is all well and good, but file systems are useful in part because of their hierarchical

structure. If we want to create a file in a particular location, we specify the *path* in the String before the name of the file.

```
File file = new File ("/homes/owilde/programs/file.txt");
```

In this case, the prefix “`/homes/owilde/programs/”` is the path and “`file .txt”` is still the file name. Each slash (‘/’) separates a parent directory from the files or directories inside of it. This path specifies that we start at the root, go into the homes directory, then the owilde directory, and then the programs directory. Note that we can also use a single period (.) in a path to refer to the current working directory and two periods (..) to refer to a parent directory.

This is one of those sticky places where Java is trying to be platform independent, but the platforms each have different needs. The example we gave above is for a Unix or Linux system. In Windows the way to specify the path is slightly different. Creating a similar File object on Windows system might be done as follows.

```
File file = new File ("C:\\My Documents\\Programs\\file.txt");
```

Then, the path specifies that we start in the C drive, go into the My Documents directory and then the Programs directory. Windows systems use a backslash (\) to separate a parent directory from its children. But, in Java, a backslash is not allowed to be by itself in a string literal, and so each backslash must be escaped with another backslash. To simplify things somewhat, Java allows Windows paths to be separated with regular slashes as well, so we will use this style for the rest of the book.

If we return to objects of class File, there are a number of things we can do directly. A File object has methods that can test if a file with the associated name and path exists, if it is readable, if it is writable, and many other things. Because there are so many classes associated with file I/O and each class has so many methods, now is a good time to remind you of the usefulness of the Java API. If you visit <http://docs.oracle.com/javase/>, you can choose the edition of Java you are using and get detailed documentation for all of the standard library, including file I/O classes.

20.3.2 Reading and writing text files

Once you have a File object, its true usefulness comes from combining it with other classes. You are already familiar with the Scanner class, and once you have a File object, reading from a text file is the same as reading from the keyboard.

```
Scanner in = null ;
try {
    in = new Scanner ( file );
    while ( in.hasNextInt () )
        process ( in.nextInt () );
}
catch ( FileNotFoundException e ) {
    System.out.println (" File " + file.getName () + " not found !");
}
```

```
finally { if( in != null ) in.close(); }
```

Assuming that file is linked to a file which the program has read access to, this block of code will extract `int` values from the file and pass them to the `process()` method. If the file does not exist or is not readable to the program, a `FileNotFoundException` will be thrown and an error message printed. Creating a Scanner from a File object instead of `System.in` can throw a checked exception, and so the `try` and `catch` are needed before the program will compile. Note that you will need to import `java.util.Scanner` or `java.util.*` just like any other time you use the Scanner class.

And that's all there is to it. After opening the file, using the Scanner class will be almost the same as before. One difference is that you should close the Scanner object (and by extension the file) when you are done reading from it, as we do in the example. Closing files is key to writing robust code. You'll notice that we put `in.close()` in a `finally` block. Using `finally` is a good habit for file I/O. File operations could fail for any number of reasons, but you will still need to close the file afterwards. We put in the `null` check in case the file didn't exist and the reference in never pointed to a valid object. (We also begin by setting `in` to `null`. Otherwise, Java complains that it might not have been initialized.)

Writing information to a file is similar to using `System.out`. First, you need to create a `PrintWriter` object. Unlike Scanner, you cannot create a `PrintWriter` object directly from a `File` object. Instead you have to create a `FileOutputStream` object first. If we want to write a list of random numbers to the file we were reading from earlier, we could do it as follows.

```
PrintWriter out = null;  
try {  
    out = new PrintWriter ( new FileOutputStream ( file ) );  
    Random random = new Random ();  
    for ( int i = 0; i < 100; i++ )  
        out.println ( random.nextInt () );  
}  
catch ( FileNotFoundException e ) {  
    System.out.println ( " File " + file.getName () + " not found ! " );  
}  
finally { if( out != null ) out.close (); }
```

Again, once you have a `PrintWriter` object, the methods for outputting data will be just like using `System.out`. In this case, we already had an `File` object lying around. To save time, the `FileOutputStream` constructor can take a path name instead of a `File` object. So, it would be equivalent to create the `PrintWriter` from above by supplying a path like so.

```
PrintWriter out = new PrintWriter (  
    new FileOutputStream ( "/homes/owilde/programs/file.txt" ) );
```

Be sure to import `java.io.*` in order to have access to the `FileOutputStream` and `PrintWriter` classes.

20.3.3 Reading and writing binary files

We covered text files first because the input and output is similar to console I/O. When reading and writing text files, it's easy to verify that what you wanted to write was written and what you read was what was in the file. Binary files, however, are more powerful. Data, as in the example with the integer 2127480645, can often be stored more compactly. Even better, Java provides facilities for easily dumping (and later retrieving) primitive data types, objects, and even complex data structures to binary files.

The first object you'll need to read input from a binary file is a `FileInputStream` object. As before all you need is a `File` object to create one.

```
File file = new File( "file.bin" );
FileInputStream in = new FileInputStream( file );
```

As it happens, `FileInputStream` also allows you to call its constructor with a `String` specifying the file path and name.

```
FileInputStream in = new FileInputStream( "file.bin" );
```

The bad news is that you can't do much with a `FileInputStream` object. Its methods allow you to read single bytes, either one at a time, or a group of them into an array. The basic `read()` method returns the next byte in the file, or a -1 if the end of the file has been reached. Working at the low level of bytes, we can still write useful code like the following method which prints the size of a file.

```
public static void printFileSize( String fileName ) {
    FileInputStream in = null;
    try {
        in = new FileInputStream( fileName );
        int count = 0;
        while ( in.read() != -1 )
            count++;
        System.out.println( "File size :" + count + " bytes" );
    }
    catch ( Exception e ) {
        System.out.println( "File access failed." );
    }
    finally { try { in.close(); } catch ( Exception e ) {} }
}
```

Note the extra `try-catch` block inside of the `finally`. Like the other binary file I/O objects we will discuss in this chapter, `FileInputStream` can throw a `IOException` when closing. Usually, you will not need to deal with this exception, but you still must catch it. By catching any `Exception`, we can save a little bit of code by eliminating the `null` check. If `in` is `null` in this example, a `NullPointerException` will be thrown and immediately caught, causing no damage.

To output a number of bytes, you can create a `FileOutputStream` object. Its `write()` methods are the mirror images of the `read()` methods in `FileInputStream`. For output, what we really want is an object which will chop up primitive data types and objects into their component bytes and send those bytes

to a FileOutputStream. Then, for input, we would want an object which could read a sequence of bytes from a FileInputStream and reassemble them into whatever kind of data they are supposed to be.

These objects exist, and they belong to the ObjectInputStream and ObjectOutputStream classes respectively. To create an ObjectInputStream, you supply a FileInputStream to its constructor.

```
ObjectInputStream in =  
    new ObjectInputStream ( new FileInputStream ( "baseball.bin" ) );
```

Now, let's assume that baseball.bin contains baseball statistics. The first thing in the file is an int indicating the number of records it contains. Then, for each record, it will list home runs, RBI, and batting average, as an int, an int, and a float, respectively. Assuming that we've opened the file correctly above, we can read these statistics into three arrays with the following code.

```
int records = in. readInt ();  
int [] homeRuns = new int [ records ];  
int [] RBI = new int [ records ];  
float [] battingAverage = new float [ records ];  
for ( int i = 0; i < records ; i++ ) {  
    homeRuns [i] = in. readInt ();  
    RBI [i] = in. readInt ();  
    battingAverage [i] = in. readFloat ();  
}
```

Of course, all of the code should be enclosed in a try block with appropriate exception handling and in.close() in a finally block at the end. If an ObjectInputStream object tries to read past the end of a file, an EOFException exception will be thrown. Using an ObjectInputStream object to read from a file also assumes that the file was created with an ObjectOutputStream object. If you substitute write for read, ObjectOutputStream methods are almost the same as ObjectInputStream methods. Below is a companion piece of code which assumes that homeRuns, RBI, and battingAverage are filled with data and writes them to a file.

```
ObjectOutputStream out = null ;  
try {  
    out = new ObjectOutputStream (  
        new FileOutputStream ( "baseball.bin" ) );  
    out.writeInt ( homeRuns.length );  
    for ( int i = 0; i < homeRuns.length ; i++ ) {  
        out.writeInt ( homeRuns [i] );  
        out.writeInt ( RBI [i] );  
        out.writeFloat ( battingAverage [i] );  
    }  
}  
catch ( Exception e ) {  
    System.out.println ( "File writing failed." );  
}
```

```
finally { try { out.close (); } catch ( Exception e ){} }
```

Using `ObjectInputStream` and `ObjectOutputStream` in this way is not too difficult, but it seem cumbersome. The objects provide methods which elegantly allow you to read and write a whole object at a time. To do so, we need to define a new class.

Program 20.1: Serializable BaseballPlayer class. (BaseballPlayer.java)

```
1 import java.io.Serializable ;  
2  
3 public class BaseballPlayer implements Serializable {  
4     private int homeRuns ;  
5     private int RBI;  
6     private float battingAverage ;  
7  
8     public BaseballPlayer ( int homeRuns, int RBI, float  
9         battingAverage ) {  
10        this.homeRuns = homeRuns ;  
11        this.RBI = RBI ;  
12        this.battingAverage = battingAverage ;  
13    }  
14  
15    public int getHomeRuns () { return homeRuns ; }  
16    public int getRBI () { return RBI ; }  
17    public float getBattingAverage () { return battingAverage ; }  
18 }
```

The new class `BaseballPlayer` encapsulates the three pieces of information we want. Note that it also implements the interface `Serializable`, but it doesn't seem to implement any special methods to conform to the interface. We'll discuss this more after we show how using this new class can simplify the file I/O. The input will change to the following.

```
ObjectInputStream in = null ;  
try {  
    in = new ObjectInputStream (  
        new FileInputStream ( "baseball.bin" ));  
    int records = in. readInt ();  
    BaseballPlayer [] players = new BaseballPlayer [ records ];  
    for ( int i = 0; i < players.length ; i++ )  
        players [i] = ( BaseballPlayer )in. readObject ();  
}  
catch ( Exception e ) {  
    System.out.println ( "File reading failed." );  
}  
finally { try { in. close (); } catch ( Exception e ){} }
```

The output will become what follows.

```
ObjectOutputStream out = null ;
try {
    out = new ObjectOutputStream (
        new FileOutputStream ( "baseball.bin" ) );
    out.writeInt ( players.length );
    for ( int i = 0; i < players.length ; i++ )
        out.writeObject ( players [i] );
}
catch ( Exception e ) {
    System.out.println ( "File writing failed." );
}
finally { try { out.close () } catch ( Exception e ){} }
```

This process of outputting an entire object at a time is called *serialization*. The BaseballPlayer class is very simple, but even complex objects can be serialized, and Java takes care of almost everything for you. The only magic needed is for the class which is going to be serialized to implement Serializable. There are no methods in Serializable. It is just a tag for a class which can be packed up and stored. The catch is that, if there are any references to other objects inside of the object being serialized, they must also be serializable. Otherwise, a NotSerializableException will be thrown when the JVM tries to serialize. Most things are serializable, including the vast majority of the Java API.

However, objects which have some kind of special system dependent state, like a Thread or a FileInputStream object cannot be serialized. If you need to serialize a class with references to objects like these, add the transient keyword to the declaration of each unserializable reference. That said, these should be few and far between. For BaseballPlayer, adding implements Serializable was all that was needed, and we can still get more mileage out of it. Consider that an array can be treated like an Object and is also serializable. We can further simplify the input as below.

```
ObjectInputStream in = null ;
try {
    in = new ObjectInputStream (
        new FileInputStream ( "baseball.bin" ) );
    BaseballPlayer [] players = ( BaseballPlayer [] ) in.readObject ();
}
catch ( Exception e ) {
    System.out.println ( "File reading failed." );
}
finally { try { in.close () } catch ( Exception e ){} }
```

And the corresponding output code can be simplified to:

```
ObjectOutputStream out = null ;
try {
```

```

out = new ObjectOutputStream (
    new FileOutputStream ( "baseball.bin" ));
out.writeObject ( players );
}
catch ( Exception e ) {
    System.out.println ( "File writing failed." );
}
finally { try { out.close (); } catch ( Exception e ){} }

```

It is worth noting that `DataInputStream` and `DataOutputStream` objects can be used in place of `ObjectInputStream` and `ObjectOutputStream` objects if you only need to read and write primitive data.

20.4 Examples: File examples

Example 20.1: Directory listing

Let's return to the `File` class and look at another example of how to use it. It is often useful to know the contents of a directory. At the Windows command prompt, this is usually done using the `dir` command; in Linux and Unix, the `ls` command is generally used. In a few lines of code, we can write a directory listing tool which lists all the files in a directory, the dates each was last modified, and whether or not a file is a directory.

Program 20.2: Directory listing tool. (`Directory.java`)

```

1 import java.io.*;
2 import java.text.*;
3 import java.util.*;
4
5 public class Directory {
6     public static void main ( String [] args ) {
7         File directory = new File (".");
8         File [] files = directory.listFiles ();
9         for ( File file : files ) {
10             System.out.print ( DateFormat.getDateInstance ().format (
11                 new Date ( file.lastModified () ) + "\t");
12             if( file.isDirectory () )
13                 System.out.print ( "directory " );
14             else
15                 System.out.print ( "\t" );
16             System.out.println ( "\t" + file.getName () );
17         }
18     }
19 }

```

As you can see, the code first creates a `File` object using `.` to specify the current working

directory. The `listFiles()` method returns an array of `File` objects which we then iterate over. We call `lastModified()` on each file to get its date, `isDirectory()` to see if it is a directory, and finally print the name given by `getName()`. ■

Example 20.2: Radiiuses stored in a file

Now, let's look at a data processing application of files. Let's assume that there is a file called `radiiuses.txt` which holds the radiiuses of a number of circles formatted as text, one on each line of the file. It's our job to read each radius, compute the area of the circle, and write those areas to a file called `areas.txt`, using the formula $A = \pi r^2$.

Program 20.3: Program to read a list of radiiuses from a text file and output their areas to another file. (`AreaFromRadiusText.java`)

```
1 import java.io.*;
2 import java.util.*;
3
4 public class AreaFromRadiusText {
5     public static void main ( String [] args ) {
6         File inFile = new File ( "radiiuses.txt" );
7         File outFile = new File ( "areas.txt" );
8         Scanner in = null ;
9         PrintWriter out = null ;
10        double radius ;
11        try {
12            in = new Scanner ( inFile );
13            out = new PrintWriter ( outFile );
14            while ( in. hasNextDouble () ) {
15                radius = in. nextDouble ();
16                out.println ( Math.PI* radius * radius );
17            }
18        }
19        catch ( Exception e ) {
20            System.out.println ( e. getMessage () );
21        }
22        finally {
23            if( in != null ) in. close ();
24            if( out != null ) out.close ();
25        }
26    }
27 }
```

The previous class did all of its input and output with text files. We will also implement this program to read from a binary file called `radiiuses.bin` and write to a binary file called `areas.bin`.

Program 20.4: Program to read a list of radii from a binary file and output their areas to another file. (AreaFromRadiusBinary.java)

```
1 import java.io.*;
2
3 public class AreaFromRadiusBinary {
4     public static void main ( String [] args ) {
5         File inFile = new File ( "radii.bin" );
6         File outFile = new File ( "areas.bin" );
7         ObjectInputStream in = null ;
8         ObjectOutputStream out = null ;
9         double radius ;
10    try {
11        in = new ObjectInputStream (
12            new FileInputStream ( inFile ) );
13        out = new ObjectOutputStream (
14            new FileOutputStream ( outFile ) );
15        while ( true ) {
16            radius = in. readDouble ();
17            out.writeDouble ( Math.PI* radius * radius );
18        }
19    }
20    catch ( EOFException e ) {}
21    catch ( Exception e ) {
22        System.out.println ( e. getMessage () );
23    }
24    finally {
25        try { in. close (); } catch ( Exception e ) {}
26        try { out.close (); } catch ( Exception e ) {}
27    }
28    }
29 }
```

There should be few surprises in this piece of code as only a few changes have been made so that ObjectInputStream and ObjectOutputStream objects could be used. You may notice that the input **while** loop is an infinite loop. The easiest way to see if there is any more data in a binary file is to keep reading until an EOFException is thrown. As you can see, we do nothing to handle this exception, because, in this case, it is just a signal to stop reading. ■

20.5 Solution: A picture is worth 1,000 bytes

Now we will give the solution to the problem posed at the beginning of the chapter. First, let's look at the class definition and main() method.

```

1 import java.io.*;
2
3 public class BitmapCompression {
4     public static void main ( String [] args ) {
5         ObjectInputStream in = null ;
6         try {
7             in = new ObjectInputStream ( new FileInputStream ( args [1] )
8                             );
9             if( args [0]. equals ("-c"))
10                 compress ( in, args [1] );
11             else if( args [0]. equals ("-d"))
12                 decompress ( in, args [1] );
13         }
14         catch ( Exception e ) {
15             System.out.println (" Bad input : " + e. getMessage () );
16         }
17         finally { try{ in. close (); } catch ( Exception e ){} {} }
18     }

```

Here we open an ObjectInputStream based on the file named passed as the second command line parameter. Then, we either compress or decompress the file depending on which switch is passed as the first command line parameter. The **catch** block will deal with the FileNotFoundException or the IOException which could be thrown by opening the file, as well as ArrayIndexOutOfBoundsException which could be caused if there are not enough command line arguments. In either case, the vague message “**Bad input:**” will be output with the message from the exception. Commercial-grade code should give a more specific error.

After the rest of the method, note the usual **finally** block where the file is closed, including the inner try-catch blocks needed to safely close binary files.

```

19     public static void compress ( ObjectInputStream in,
20                                 String file ) {
21         int temp, current, count = 1;
22         ObjectOutputStream out = null ;
23         try {
24             out = new ObjectOutputStream (
25                             new FileOutputStream ( file + ". compress " ) );
26             current = in. read ();
27             while ( ( temp = in. read () ) != -1 ) {
28                 if( temp == current && count < 127 )
29                     count++;
30                 else {
31                     out.writeByte ( count );
32                     out.writeByte ( current );
33                     count = 1;

```

```

34         current = temp ;
35     }
36     out.write ( count );
37     out.write ( current );
38 }
39 }
40 catch ( Exception e ) {
41     System.out.println (" Compression failed : "
42         + e. getMessage () );
43 }
44 finally { try { out.close (); } catch ( Exception e ){} }
45 }

```

In the compress() method we first open a new ObjectOutputStream for a file named the same as the input file with .compress tacked on the end. Then, we read in bytes of data from the input file. As long as we keep seeing the same byte, we increment a counter. When we run into a new byte (or when we reach the limit of 127 of the same consecutive byte), we write the count and the byte we've been reading and move on. When in.read() returns -1, we know that we've reached the end of the file and output the last count and last byte value. The method finishes with the usual **catch** and **finally** blocks needed to catch errors and safely close the output file.

```

47 public static void decompress ( ObjectInputStream in,
48     String file ) {
49     int count, temp ;
50     ObjectOutputStream out = null ;
51     try {
52         out = new ObjectOutputStream (
53             new FileOutputStream ( file.substring ( 0,
54                 file.lastIndexOf ( ". compress " ) ) );
55         while ( ( count = in. read () ) != -1 ) {
56             temp = in. readByte ();
57             for ( int i = 0; i < count ; i++ )
58                 out.writeByte ( temp );
59         }
60     }
61     catch ( Exception e ) {
62         System.out.println (" Decompression failed : "
63             + e. getMessage () );
64     }
65     finally { try{ out.close (); } catch ( Exception e ){} }
66 }
67 }

```

The decompress() method is even simpler than compress(). It begins by opening a new

`ObjectOutputStream` for a file named the same as the input file with a `.compress` extension stripped off. Then, it reads a count, reads a byte value, and writes the byte value as many times as the count specifies.

20.6 Concurrency: File I/O

By now, you have seen threads behave in unpredictable ways because of the way they are reading and writing to shared variables. Well, isn't a file a shared resource as well? What happens when two threads try to access a file at the same time? If both threads are reading from the file, everything should work fine. If the threads are both writing or doing a combination of reading and writing, there can be problems.

As we mentioned in [Section 20.3](#), file operations are OS dependent. Although Java tries to give a uniform interface, different system calls are happening at a low level. Consequently, the results may be different as well.

Consider the following program that spawns two threads that both print a series of numbers to a file called `concurrent.out`. The first thread prints the even numbers between 0 and 9,999 while the second thread prints the odd ones.

Program 20.5: Program that spawns threads that print odd and even numbers to a file concurrently. (`ConcurrentFileAccess.java`)

```
1 import java.io.*;
2
3 public class ConcurrentFileAccess implements Runnable {
4     private boolean even;
5
6     public static void main ( String args [] ) {
7         Thread writer1 =
8             new Thread ( new ConcurrentFileAccess ( true ) );
9         Thread writer2 =
10            new Thread ( new ConcurrentFileAccess ( false ) );
11         writer1.start ();
12         writer2.start ();
13     }
14
15     public ConcurrentFileAccess ( boolean even ) {
16         this.even = even;
17     }
18
19     public void run () {
20         PrintWriter out = null;
21         int start = 0;
22         if( ! even )
```

```

23     start = 1;
24
25     try {
26         out = new PrintWriter (
27             new FileOutputStream ( "concurrent.out ", true )
28             );
29
30         for ( int i = start ; i < 10000; i += 2 ) {
31             out.println ( i );
32             out.flush ();
33         }
34     } catch ( FileNotFoundException e ) {
35         System.out.println ( "concurrent.out not found !" );
36     } finally { if( out != null ) out. close (); }
37 }
38 }
```

The code in this program should have no surprises. The main() method creates two Thread objects based on ConcurrentFileAccess objects, each with a different value for its even field. Then, the main() method starts the threads running. In each thread's run() method, it opens the file and starts printing out even or odd numbers, depending on which thread it is. Afterwards, each thread closes the file and ends.

What do you expect the file concurrent.out to look like after the program has completed? Run it several times, on Windows, Linux, and Mac computers, if you can. Most likely, the file will contain either all the even numbers from 0 to 9,998 or all the odd numbers from 1 to 9,999. If you run the program enough times, you should be able to see both possibilities.

Why are half the numbers getting lost? When you open a file for writing, by default it erases everything that was already in the file. So, an entire sequence of numbers is getting saved and then lost. We can change this behavior by changing the line below.

```
out = new PrintWriter ( new FileOutputStream ( "concurrent.out " ) );
```

We replace it with the following.

```
out = new PrintWriter ( new FileOutputStream ( "concurrent.out ", true
) );
```

This second **boolean** parameter to the FileOutputStream constructor specifies that output will *append* to the file instead of overwriting it.

After this change, what does the file look like when we run the program? Since we are going to append to any preexisting file, make sure that you delete concurrent.out before running the program again. The file may look different on different systems. The file probably contains long runs of numbers from each thread. In fact, it is quite possible to have the complete output from one thread

followed by the complete output from the other.

For performance reasons, file operations are usually done in batches. Instead of writing each number to the file as the thread produces it, output is usually stored in a buffer which is written as a whole. If we call `out.flush()` after [line 28](#), we can flush the buffer to the file after each number is generated. Doing so will not be as efficient, but it may give us some insight into how concurrent writes on files work.

Using flushes, the output from the two threads should be thoroughly intermixed. On a Windows machine, if you copy the data from the file and sort it, you will probably see some numbers missing. This lost output is similar to situations where updates to variables were lost because they were overwritten by another thread. On the other hand, most Linux systems have better concurrent file writing and will not lose any numbers. (Even on Linux, it is possible for a number to be printed in the middle of another number, but no digits should be lost.)

Under ideal circumstances, no two threads or processes should be writing to the same file. However, this situation is sometimes unavoidable, as with a database program that must support concurrent writes for the sake of speed. If you need to enforce file locking, you can prevent threads within your own program from accessing a file concurrently by using normal Java synchronization tools. If you expect other programs to interact with the same files that your program will use, Java provides a `FileLock` class which allows the user to lock up portions of a file. As with everything file-related, `FileLock` is dependent on the underlying OS and may behave differently on different systems.

Exercises

Conceptual Problems

- 20.1 What is the difference between volatile and non-volatile memory? Which is often associated with files and why?
- 20.2 What is the difference between text and binary files? What are the pros and cons of using each?
- 20.3 Define compression ratio to be the size of the uncompressed data in bytes divided by the size of the compressed data in bytes. What is the theoretical maximum compression ratio you can get out of the RLE encoding we used? What is the theoretical lowest compression ratio you can get out of the RLE encoding we used?
- 20.4 What is serialization in Java? What do you have to do to serialize an object?
- 20.5 What kinds of objects cannot be serialized?

Programming Practice

- 20.6 Implement the RLE bitmap compression program using only `FileInputStream` and `FileOutputStream` for file input and output.
- 20.7 Re-implement the maze solving program from [Section 19.5](#) to ask the user for a file instead of reading from standard input.

20.8 An HTML file contains many tags such as `<p>`, which marks the beginning of a paragraph, and `</p>`, which marks the end of a paragraph. A lesser known feature of HTML is that ampersand (`&`) can mark special HTML entities used to produce symbols on a web page. For example `π` is the entity for the Greek letter π . Because of these features of the language, raw text that is going to be marked up in HTML should not contain less than signs (`<`), greater than signs (`>`), or ampersands (`&`).

Write a program that reads in an input text file specified by the user and writes to an output text file also specified by the user. The output file should be exactly the same as the input file except that every occurrence of a less than sign should be replaced by `<`, every occurrence of a greater than sign should be replaced by `>`, and every occurrence of an ampersand should be replaced by `&`.

20.9 Write a program that prompts the user for an input text file. Open the file and read each word from the file, where a word is defined as any String made up only of upper and lower case letters. You can use the `next()` method in the `Scanner` class to break up text by whitespace, but your code will still need to examine the input character by character, ending a word when any punctuation or other characters are reached. Store each word (with a count of the number of times you find it) in a binary search tree such as those described in [Example 19.10](#). Then, traverse the tree, printing all the words found (and the number of times found) to the screen in alphabetical order.

20.10 Expand the program from Exercise 20.9 so that it also prompts for a second file containing a dictionary in the form of a word list with one word on each line. Store the words from the dictionary in another binary search tree. Then, for each word in the larger document that you cannot find in the dictionary tree, add it to a third binary search tree. Finally, print out the contents of this third binary search tree to the screen, and you will have implemented a rudimentary spell checker. You can test the quality of your implementation by using a novel from Project Gutenberg (<http://www.gutenberg.org/>) and a dictionary file from an open source spell checker or from a Scrabble word list.

20.11 Files can become corrupted when they are transferred over a network. It is common to make a *checksum*, a short code generated using the entire contents of a file. The checksum can be generated before and after file transmission. If both of the checksums match, there's a good chance that there were no transmission errors. Of course, there can be problems sending checksums, but checksums are much smaller and therefore less likely to be corrupted. Modern checksums are often generated using cryptographic hash functions, which are more complex than we want to deal with here. An older checksum algorithm works in the following way. Although we use mathematical notation, the operations specified below are integer modulus and integer division.

1. Add up the values of all the bytes, storing this sum in a long variable
2. Set $sum = sum \bmod 2^{32}$
3. Let $r = (sum \bmod 2^{16}) + (sum \div 2^{16})$

4. Let $s = (r \bmod 2^{16}) + (r \div 2^{16})$

5. The final checksum is s

Remember that finding powers of 2 is easy with bitwise shifts. Write a program that opens a file for binary reading using `FileInputStream` and outputs the checksum described. On Linux systems, you can check the operation of your program with the `sum` utility, using the `-s` option. The following is an example of the command used on a file called `wombat.dat`. The first number in the output below it, 6892, is the checksum.

```
sum -s wombat.dat
```

```
6892 213 wombat.dat
```

Experiments

20.12 Write the RLE bitmap compression program in parallel so that a file is evenly divided into as many pieces as you have threads, compressed, and then each compressed portion is output in the correct order. Compare the speed for 2, 4, and 8 threads to the sequential implementation. Are any of the threaded versions faster? Why or why not? Run some experiments to see how long it takes to read 1,000,000 bytes from a file compared to the time it takes to compress 1,000,000 bytes which are already stored in an array.

Chapter 21

Network Communication

Arguing with anonymous strangers on the Internet is a sucker's game because they almost always turn out to be - or to be indistinguishable from - self-righteous sixteen-year-olds possessing infinite amounts of free time.

—Neal Stephenson

21.1 Problem: Web server

It is no accident that the previous chapter about file I/O is followed by this one about networking. At first glance, the two probably seem unrelated. As it happens, both files and networks are used for input and output, and the designers of Java were careful to create an API with a similar interface for both.

In the next two sections, we discuss how this API works, but first we introduce the problem: You need to create a web server application. The term *server* is used to describe a computer on a network which other computers, called *clients*, connect to in order to access some services or resources. When you surf the Internet, your computer is a client connecting to web servers all over the world. Writing a web server might seem like a daunting task. The web browser you run on your client computer (e.g. Firefox, Internet Explorer, or Safari) is, in modern times, a very complicated program capable of playing sounds and movies, browsing in multiple tabs, automatically encrypting and decrypting secure information, and, at the very least, correctly displaying web pages of every description.

In contrast, a web server application is much simpler. At its heart, a web server application gets requests for files and sends those files over the network. More advanced servers can execute code and dynamically generate pages, and many web servers are multi-threaded to support heavy traffic. The web server you are writing needs only to focus on getting requests for files and sending those files back to the requester.

21.1.1 HTTP requests

To receive requests, a web server uses something called *hypertext transfer protocol (HTTP)*, which is just a way of specifying the format of the requests. The only request we are interested in is the GET request. All GET requests have the following format.

GET path HTTP/version

where **path** is the path of the file being requested and **version** is the HTTP version number. A typical request might be as follows.

GET/images/banner.jpg HTTP/1.1

You should also note that all HTTP commands end with two newline characters ('\n'). The extra blank line makes it easier to separate the commands from other data being sent.

21.1.2 HTTP responses

After your web server receives a GET message, it looks for the file specified by the **path**. If the server finds the file, it sends the following message.

HTTP/1.1 200 OK

Again, note that this message is followed with two newline characters. After this message is sent, the server sends the requested file, byte by byte across the network. If the file cannot be found by the web server, it sends an error message as follows.

HTTP/1.1 404 Not Found

Of course, two newlines will be sent after this as well. After the error message, servers will also typically send some default web page with an explanation in HTML.

Now, we return to the more fundamental problem of how to communicate over a network.

21.2 Concepts: TCP/IP communication

We begin where nearly every discussion of computer networking begins, with the Open Systems Interconnection Basic Reference Model (or OSI model). As we mentioned before, the designers of Java wanted to make a networking API which was very similar to the file system API. This single API must be used for JVM's running on Windows or on Mac OS or on Linux or any other operating system. Even with the same operating system, different computers have different hardware. Some computers have wired connections to a router or gateway. Others are connected wirelessly. And somehow, you have to figure out the address of the computer you want to send messages to and deal with its network, hardware, and software.

There are so many steps in the process that it seems hopelessly complicated. To combat this problem, the OSI seven layer model was developed. Each layer defines a specification for one aspect of the communication path between two computers. As long as a particular layer interacts smoothly with the one above it and below it, that layer could take the form of many different hardware or software choices. Listing them in order from the highest level (closest to the user) to the lowest level (closest to the hardware), the layers are as follows.

- Layer 7: Application Layer
- Layer 6: Presentation Layer
- Layer 5: Session Layer
- Layer 4: Transport Layer

- Layer 3: Network Layer
- Layer 2: Data Link Layer
- Layer 1: Physical Layer

The application layer is where your code is. The Java networking API calls that your code uses to send and receive data comprise the application layer for your purposes. The only thing above this layer is the user. Protocols like HTTP and FTP are the province of this layer. All the other communication problems have been solved, and the key issue is what to do with the data that is communicated.

The presentation layer changes one kind of message encoding to another. This layer is not one people usually spend a lot of time worrying about, but some kinds of encryption and compression can happen here.

The session layer allows for the creation of sessions when communicating between computers. Sessions requiring authentication and permissions can be dealt with here, but, in practice, this layer is not often used. One notable exception is the Secure Sockets Layer (SSL), the technology most commonly used to protect passwords and credit card numbers when you make online purchases.

The transport layer is concerned with the making the lower level communication of data more transparent to higher layers. This layer typically breaks larger messages into smaller packets of data which can be sent across the network. This layer can also provide reliability by checking to see if these packets make it to their destinations and resending them otherwise. The two most important protocols for this layer are Transmission Control Protocol (TCP) and User Datagram Protocol (UDP). TCP is more commonly used and provides communication that is reliable and ensures that packets are delivered in order. TCP is used for file transfers, e-mail, web browsing, and any number of other web applications. UDP does not have guarantees about reliability or ordering; however, UDP is faster. For this reason, UDP is used for streaming media and online games.

The network layer is responsible for packet transmission from source to destination. It is concerned with addressing schemes and routing. The most well-known example of a network layer protocol is the Internet Protocol (IP) used to make the Internet work.

If the network layer worries about sending of packets from source to destination, the data link layer is responsible for the actual transmission of packets between each link in the chain. Here hardware becomes more important because there are so many different kinds of networks. Examples of data link layers include Ethernet, token ring networks, IEEE 802.11 wireless networks, and many more.

Finally, the lowest level is the physical layer. This layer defines the actual physical specifications for sending raw information from one place to another, over a wire or wirelessly. This layer is typically the least interesting to programmers but is a key area for electrical engineers.

21.3 Syntax: Networking in Java

The seven layer model might seem overwhelming, but there are only a few key pieces that we'll need on a regular basis. In fact, the system of layers is designed to help people focus on the one or two

layers specified to their needs and ignore the rest.

21.3.1 Addresses

The first topic we touch on is the network layer. What we need from this layer are addresses. A network address is much like a street address. It gives the location on the network of a computer so that messages can be sent there.

For most systems you use, such an address is be an *IP address*. There are two current versions of IP addresses, IPv4 and IPv6. IPv6 is the way of the future and provides a huge number of possible addresses. Not all systems support IPv6, and the general public is not very aware of it. Although it will one day be the standard, we use the more common IPv4 addresses here. An IPv4 address is typically written as four decimal numbers separated by dots. Each of these four numbers is in the range 0 - 255. For example, 64.233.187.99 and 192.168.1.1 are IPv4 addresses.

21.3.2 Sockets

The second topic we focus on is the transport layer. Here, you need to make a choice between TCP or UDP for communication. In this book, we only cover TCP communication because it is reliable and much more commonly used than UDP. If you need to use UDP communication, the basics are not too different from TCP, and there are many excellent resources online.

To create a TCP connection, you typically need a server program and a client program. The difference between the two is not necessarily big. In fact, both the client and the server could be running on the same computer. What distinguishes the server is that it sets up a *port* and listens to it, waiting for a connection. Once the client makes a connection, the two programs can send and receive data on an equal footing.

We just mentioned the term port. As you know, an address is the location of a computer in a network, but a single computer may be performing many different kinds of network communications. For example, your computer could be running a web browser, an instant message application, an online game, and a number of other things. So that none of these programs become confused and get each others' messages, each program uses a separate port for communication. To the outside world, your computer usually only has a single address but thousands of available ports. Many of these ports are set aside for specific purposes. For example, port 20 is for FTP, port 23 is for Telnet, and port 80 is for HTTP (webpages).

When you write a server program, you will usually create a `ServerSocket` object which is linked to a particular port. For example, if you wanted to write a web server, you might create a `ServerSocket` as follows.

```
ServerSocket serverSocket = new ServerSocket( 80 );
```

Once the `ServerSocket` object has been created, the server will typically listen to the socket and try to accept incoming connections. When a connection is accepted, a new `Socket` object is created for that connection. The purpose of the `ServerSocket` is just to set up this `Socket`. The `ServerSocket` doesn't do any real communication on its own. This system may seem indirect, but it allows for greater flexibility. For example, a server could have a thread just listening for connections. When a

connection is made, it could spawn a new thread to do the communication. Commercial web servers often function in this way. The code for a server to listen for a connection is:

```
Socket socket = serverSocket.accept();
```

The `accept()` method is a blocking method; thus, the server will wait for a connection before doing anything else.

Now, if you want to write the client which connects to such a server, you can create the `Socket` object directly.

```
Socket socket = new Socket( "64.233.187.99 ", 80 );
```

The first parameter is a `String` specifying the address of the server, either as an IP address as shown or as domain like “google.com”. The second parameter is, of course, the port you want to connect on.

21.3.3 Receiving and sending data

From here on out, we no longer have to worry about the differences between the client and server. Both programs have a `Socket` object that can be used for communication.

In order to get input from a `Socket`, you first call its `getInputStream()` method. You can use the `InputStream` returned to create an object used for normal file input like in the first half of the chapter. The considerations are similar. If you only need to receive plain, human readable from the `Socket`, you can create a `Scanner` object.

```
Scanner in = new Scanner( socket.getInputStream() );
```

Over the network, it will be much more common to send files and other binary data. For that purpose you can create an `ObjectInputStream` or `DataInput-Stream` from the `Socket` in much the same way.

```
ObjectInputStream in = new ObjectInputStream( socket.getInputStream()
    () );
```

It should be unsurprising that output is just as easy as input. Text output can be accomplished by creating a `PrintWriter`.

```
PrintWriter out = new PrintWriter( socket.getOutputStream() );
```

Likewise, binary output can be accomplished by creating an `ObjectOutputStream` or a `DataOutputStream`.

```
ObjectOutputStream out = new ObjectOutputStream(
    socket.getOutputStream() );
```

Once you have these input and output objects, you use them in the same way you would for file processing. There are a few minor differences to keep in mind. In the first place, when reading data, you may not know when more is coming. There is no explicit end of file. Also, it is sometimes

necessary to call a flush() method after doing a write. Unlike writing to a disk, a socket may wait for a sizable chunk of data to be accumulated before it gets sent across the network. Without a flush(), the data you write may not be sent immediately.

Example 21.1: Simple client and server

Here's an example of a piece of server code which listens on port 4321 waits for a connection, then reads 100 int values in binary form from the socket, and prints their sum.

```
try {  
    ServerSocket serverSocket = new ServerSocket( 4321 );  
    Socket socket = serverSocket.accept();  
    ObjectInputStream in = new ObjectInputStream( socket.  
        getInputStream());  
    int sum = 0;  
    for ( int i = 0; i < 100; i++ )  
        sum += in. readInt();  
    in. close();  
    System.out.println( " Sum : " + sum );  
}  
catch ( IOException e )  
{}
```

Now, here's a companion piece of client code which connects to port 4321 and sends 100 int values in binary form, specifically, the first 100 perfect squares.

```
try {  
    Socket socket = new Socket( " 127.0.0.1 ", 4321 );  
    ObjectOutputStream out = new ObjectOutputStream(   
        socket.getOutputStream());  
    for ( int i = 1; i <= 100; i++ )  
        out.writeInt( i*i );  
    out.close();  
}  
catch ( IOException e )  
{}
```

Note that this client code connects to the IP address 127.0.0.1. This is a special loopback IP address. When you connect to this IP address, it connects to the machine you are currently working on. In this way, you can test your networking code without needing two separate computers. To test this client and server code together, you will need to run two virtual machines. The simplest way to do this is open two command line prompts and run the client from one and the server from the other. Be sure that you start the server first so that the client has something to connect to. ■

Example 21.2: Chat client and server

Now we look at a more complicated example of network communication which should be familiar: a chat program. If you want to apply the GUI design from [Chapter 15](#), you can make a windowed version of this chat program which looks more like chat programs you are used to. For now, our chat program is be text only. .

Exercise 21.9

The functionality of the program is simple. Once connected to a single other chat program, the user will enter his or her name, then enter lines of text each followed by a newline. The program will insert the user's name at the beginning of each line of text and then send it across the network to the other chat program, which will display it. We encapsulate both client and server functionality in a class called Chat.

The first step is the appropriate import statements and the main() method, which creates a client or a server Chat object, depending on command line parameters.

```
1 import java.io.*;
2 import java.net.*;
3 import java.util.*;
4 public class Chat {
5     private Socket socket ;
6
7     public static void main ( String [] args ) {
8         if( args [0]. equals ("-s") )
9             new Chat ( Integer.parseInt ( args [1] ) );
10        else if( args [0]. equals ("-c") )
11            new Chat ( args [1], Integer.parseInt ( args [2] ) );
12        else
13            System.out.println (" Invalid command line flag.");
14    }
```

The code given here calls the server version of the Chat constructor if the argument “-s” is given and the client version of the Chat constructor if the argument “-c” is given. For the server, only a port is required, but the client also needs an IP address to connect to.

The server Chat constructor takes the port and listens for a connection on it. After a connection, it calls the runChat() method to perform the actual business of sending and receiving chats.

```
16 // Server
17 public Chat ( int port ) {
18     try {
19         ServerSocket serverSocket = new ServerSocket ( port );
20         socket = serverSocket.accept ();
21         runChat ();
22     }
23     catch ( Exception e ) {}
24 }
```

The client constructor is similar but connects directly to the specified IP address on the specified port.

```
26 // Client
27 public Chat ( String address, int port ) {
28     try {
29         socket = new Socket ( address, port );
30         runChat ();
31     }
32     catch ( Exception e )
33     {}
34 }
```

Once the client and server are connected, they both run the runChat() method, which creates a new Sender and a new Receiver to do the sending and receiving. Note that both start() and join() are called on the Sender and Receiver objects. These calls are needed because both classes are subclasses of Thread. Sending messages is an independent task concerned with reading input from the keyboard and then sending it across the network. Receiving messages is also an independent task, but it is concerned with reading input from the network and printing it on the screen. Since both tasks are independent, it is reasonable to allocate a separate thread to each.

```
36     public void runChat () throws InterruptedException {
37         Sender sender = new Sender ();
38         Receiver receiver = new Receiver ();
39         sender.start ();
40         receiver.start ();
41         sender.join ();
42         receiver.join ();
43     }
```

Below is the private inner class Sender. In this case it is convenient but not necessary to make Sender an inner class, especially since it is so short. The only piece of data Sender shares with Chat is the all important socket variable. The Sender begins by creating a PrintWriter object from the socket's output stream. After reading a name from the user, it waits for a line from the user. Each time a line is ready, it is printed and flushed, with the user name inserted at the beginning, through the PrintWriter connected to the Socket's output stream. When the user types quit, the Socket will be closed.

```
45     private class Sender extends Thread {
46         public void run () {
47             try {
48                 PrintWriter netOut = new PrintWriter (
49                     socket.getOutputStream () );
50                 Scanner in = new Scanner ( System.in );
51                 System.out.print (" Enter your name : ");
```

```

52     String name = in.nextLine();
53     String buffer = "";
54     while ( ! socket.isClosed() ) {
55         if( in.hasNextLine() ) {
56             buffer = in.nextLine();
57             if( buffer.equals("quit") )
58                 socket.close();
59             else {
60                 netOut.println( name + ":" +
61                     buffer );
62                 netOut.flush();
63             }
64         }
65     }
66 }
67 catch ( IOException e ) {}
68 }
69 }
```

Below is the private inner class Receiver, the counterpart of Sender, as well as the last thing defined in the Chat class. The Receiver class is even simpler than the Sender class. After creating a Scanner object connected to the input stream of the Socket, it waits for a line of text to arrive from the connection. Each time a line arrives, it prints it to the screen. Here again, you can see that this problem is solved with threads much more easily than without them. Both the in.hasNextLine() method called by Sender and the netIn.hasNextLine() method called by Receiver are blocking functions. Because each might wait for input before continuing, they cannot easily be combined in one thread of execution.

```

71 private class Receiver extends Thread {
72     public void run() {
73         try {
74             Scanner netIn = new Scanner (
75                 socket.getInputStream() );
76             while ( ! socket.isClosed() )
77                 if( netIn.hasNextLine() )
78                     System.out.println( netIn.nextLine() );
79         }
80     catch ( IOException e ) {}
81 }
82 }
```

Although the fundamentals are present in this example, a real chat client should provide a buddy list, the opportunity to talk to more than one other user at a time, real error-handling code in the **catch**-

blocks, and many other features. Some of these features are easier to provide in a GUI. ■

In the next section, we give a solution for the Web Server problem. Since only the server side is provided, some of the networking is simpler, and there are no threads. However, the communication is done in both binary and text mode.

21.4 Solution: Web server

Here is our solution to the Web Server problem. As usual, our solution doesn't provide all the error checking or features that a real web server would, but it is entirely functional. When you compile and run the code, it will start a web server on port 8080 (an alternative port for HTTP) in the directory you run it from. Feel free to change those settings in the main() method. When the server is running, you should be able to open any web browser and go to <http://127.0.0.1>. If you put some sample HTML files in the directory you run the server from, you should be able to browse them.

As for our code, we start with the imports and constructor below. Note that the server has fields for the port communication will take place on, the root directory for the web page, and a ServerSocket. The main() method does nothing but call the constructor using the current directory as an argument and then start the server.

```
1 import java.io.*;
2 import java.net.*;
3 import java.util.*;
4 public class WebServer {
5     private int port ;
6     private String webRoot ;
7     private ServerSocket serverSocket = null ;
8
9     public WebServer ( int port, String webRoot ) {
10         this.port = port ;
11         this.webRoot = webRoot ;
12     }
13
14     public static void main ( String [] args ) {
15         File currentDirectory = new File (".");
16         try {
17             WebServer server = new WebServer (8080,
18                 currentDirectory.getCanonicalPath () );
19             server.start ();
20         }
21         catch ( IOException e )
22         {}
23     }
```

Below is the start() method. This method contains the central loop of the web server that waits for

connections and loops forever. Once a connection has been made, the server creates input and output objects from the socket connection. Then, it tries to serve requests coming from the socket input. Our web server ignores any request other than a GET request and closes the connection after the first GET. When a GET request is made, the server removes the “HTTP” at the beginning and passes off the remaining path to the serve() method. Everything else in the start() method is made up of the necessary exception handling machinery.

Note that the out object is of type ObjectOutputStream, allowing us to send binary data over the socket. However, the in variable is of type Scanner, because HTTP requests are generally only text.

```
25 public void start () {
26     Socket socket = null ;
27     Scanner in = null ;
28     ObjectOutputStream out = null ;
29     String line ;
30     try {
31         serverSocket = new ServerSocket ( port );
32         while ( true ) {
33             socket = serverSocket.accept ();
34             try {
35                 in = new Scanner ( socket.getInputStream () );
36                 out = new ObjectOutputStream (
37                     socket.getOutputStream () );
38                 while ( in.hasNextLine () ) {
39                     line = in.nextLine ();
40                     if( line.startsWith ("GET ") ) {
41                         String path = line.substring (4,
42                                         line.lastIndexOf (" HTTP ")).trim ();
43                         System.out.println (
44                             " Received request for :" + path );
45                         serve ( out, getPath ( path ) );
46                         socket.close ();
47                         break ;
48                     }
49                 }
50             }
51             catch ( IOException e ) {
52                 System.out.println (" Error : " + e.getMessage ());
53             }
54             finally {
55                 if( in != null ) in.close ();
56                 if( out != null ) out.close ();
57             }
58         }
```

```

59     }
60     catch ( IOException e ) {
61         System.out.println (" Error : " + e.getMessage ());
62     }
63 }

```

Next is a short utility method which provides some amount of platform independence. The getPath() method take in a String representation of a path requested by a web browser and format it. This path should always be given in the Unix or Linux style with slashes (/) separating each directory. To function smoothly with other operating systems, getPath() uses the class variable File.separatorChar which gives the **char** used to separate directories on whichever platform the JVM is currently running. In addition, getPath() adds “[index.html](#)” to the end of the path if the path ends with a directory rather than a file name. Real web servers try a list of many different files such as index.html, index.htm, index.php, and so on, until a file is found or the list runs out.

```

65     public String getPath ( String path ) {
66         if ('/' != File.separatorChar )
67             path = path.replace ('/', File.separatorChar );
68         if ( path.endsWith ("" + File.separatorChar ) )
69             return webRoot + path + " index.html ";
70         else
71             return webRoot + path ;
72     }

```

The last method in the WebServer class takes in a path and sends the corresponding file over the network. The serve() method first tries to find the specified file. If it fails, it sends an HTTP 404 message with a short explanatory piece of HTML. Anyone who surfs the Internet should be familiar with 404 messages. On the other hand, if this method finds the file, it sends the HTTP 200 method indicating success and then creates a new ObjectInputStream object to read the file in binary format. In this case, it is necessary to read the file in binary. In general, HTML files are simple text files which are human readable, but the image files that web servers must often send such as GIF and JPEG files are binary files which are filled with unprintable characters. Because we need to send binary data, we were also careful to open an ObjectOutputStream on the socket earlier.

Once the file is open, the serve() method simply reads it in, byte by byte, and sends each byte out over the socket. After the file has been sent, the method closes it and returns.

```

74     public void serve ( ObjectOutputStream out, String path )
75         throws IOException {
76     System.out.println (" Trying to serve " + path );
77     File file = new File ( path );
78     if( ! file.exists () ) {
79         out.writeBytes (" HTTP /1.1 404 Not Found \n\n");
80         out.writeBytes ("<html><head><title>404 Not Found " +
81             "</title ></ head ><body ><h1> Not Found " +
82             "</h1> The requested URL " + path +

```

```

83     " was not found on this server.</ body ></ html >);
84     System.out.println (" File not found.");
85 }
86 else {
87     out.writeBytes (" HTTP /1.1 200 OK\n\n");
88     ObjectInputStream in = null ;
89     try {
90         in = new ObjectInputStream (
91             new FileInputStream ( file ));
92         int data ;
93         while ( ( data = in. readByte () ) != -1 )
94             out.writeByte ( ( byte ) data );
95         System.out.println (" Request succeeded.");
96     }
97     catch ( IOException e ) {
98         System.out.println (" Error sending file : " +
99             e. getMessage ());
100    }
101    finally { if( in != null ) in. close (); }
102 }
103 }
104 }
```

Because a web server is a real world application, we have to give the usual caveat that this implementation is quite bare-bones. There are other HTTP requests and many features, including error handling, that a web server should do better. Feel free to extend the functionality.

Also, you might notice that there is no way to stop the web server. It has an infinite loop which is only broken if there is an IOException thrown. From a Windows, Linux, or Mac command prompt you can usually stop a running program by typing Ctrl-C.

21.5 Concurrency: Networking

Throughout this book, we have used concurrency primarily for the purpose of speedup. For that kind of performance improvement, concurrency is essentially icing on the cake. Unless you are performing massively parallel computations such as code breaking or scientific computing, concurrency will probably make your application run just a little faster or a little smoother.

With network programming, the situation is different. Many networked programs, including chat clients, web servers, and peer-to-peer file sharing software, will simultaneously be connected to tens if not hundreds of other computers at the same time. While there are single-threaded strategies to handle these scenarios, it is natural to handle them in a multi-threaded way.

A web server at Google, for example, may service thousands of requests per second. If each request had to wait for the previous one to come to completion, the server would become hopelessly

bogged down. By using a single thread to listen to requests and then spawn worker threads as needed, the server can run more smoothly.

Exercise 21.10

Exercise 21.6

Even in [Example 21.2](#), it was convenient to create two different threads, Sender and Receiver. We did not create them for speedup but simply because they were doing two different jobs. Since the Sender waits for the user to type a line and the Receiver waits for a line of text to arrive over the network, it would be difficult to write a single thread that could handle both jobs. Both threads call the `hasNextLine()` method, which can block execution. A single thread waiting to see if the user had entered more text could not respond to text arriving over the network until the user hit enter.

We only touch briefly on networking in this book. As the Internet evolves, standards and APIs evolve as well. Some libraries can create and manage threads transparently, without the user worrying about the details. In other cases, your program must explicitly use multiple threads to solve the networking problem effectively.

Exercises

Conceptual Problems

- 21.1 Why are there so many similarities between the network I/O and the file I/O APIs in Java?
- 21.2 Explain the difference between client and server computers in network communication. Is it possible for a single computer to be both a client and a server?
- 21.3 Why is writing a web browser so much more complicated than writing a web server?
- 21.4 Name and briefly describe the seven layers of the OSI model.
- 21.5 Modern computers often have many programs running that are all in communication over a network. Since a computer often has only one IP address that the outside world can send to, how are messages that arrive at the computer connected to the right program?
- 21.6 What are the most popular choices of protocols at the transport layer of the OSI model? What are the advantages and disadvantages of each?
- 21.7 How many possible IP addresses are there in IPv4? IPv6 addresses are often written as eight groups of four hexadecimal digits, totaling 32 hexadecimal digits. How many possible IP addresses are there in IPv6?

Programming Practice

- 21.8 Recall the client and server from [Section 21.3.3](#) that, respectively, send 100 int values and sum them. Rewrite these fragments to send and receive the int values in text rather than binary format.
- 21.9 Add a GUI based on JFrame for the chat program given in [Example 21.2](#). Use a (non-editable) JTextArea to display the log of messages, including user name. Provide a JTextField for

entering messages, a JButton for sending messages, and another JButton for closing the network connections and ending the program.

Concurrency 21.10 Study the web server implementation from [Section 21.4](#). Implement a similar web server which is multithreaded. Instead of serving each request with the same thread that is listening for connections, spawn a new thread to handle the request each time a connection is made.

Concurrency 21.11 One of the weaknesses of the web server from the previous exercise is that a new thread has to be created for each connection. An alternative approach is to create a pool of threads to handle requests. Then, when a new request arrives, an idle thread is selected from the pool. Extend the solution to the previous exercise to use a fixed pool of 10 worker threads.

Experiments

Concurrency 21.12 Consider the multi-threaded implementation of a web server from Exercise 21.10. Can you design an experiment to measure the average amount of time a client waits to receive the requested file? How does this time change from the single threaded to the multi-threaded version? If the file size is larger, is the increase in the waiting time the same in both the single and multi-threaded versions?

Index

! (logical NOT), [85](#), [116](#)
!= (not equal to), [85](#), [118](#)
 π , [94](#)
+ (String concatenate), [87](#)
+ (unary plus), [76](#)
- (negate), [76](#)
/* */ (block comment), [56](#)
/** */ (documentation comment), [56](#)
// (single-line comment), [56](#)
< (less than), [118](#)
<< (signed left shift), [20](#), [79](#)
<= (less than or equal to), [85](#), [118](#)
= (assign), [45](#)
== (equal to), [85](#), [118](#)
> (greater than), [85](#), [118](#)
>= (greater than or equal to), [85](#), [118](#)
>> (signed right shift), [20](#), [79](#)
>>> (unsigned right shift), [20](#), [79](#)
& (bitwise AND), [20](#), [79](#)
| (bitwise OR), [20](#), [79](#)
&& (logical AND), [85](#), [116](#)
|| (logical OR), [85](#), [116](#)
^ (bitwise XOR), [20](#), [79](#)
^ (logical XOR), [85](#), [116](#)
_ (underscore), [92](#)
{ } (braces), [114](#), [141](#), [223](#)
~ (bitwise NOT), [20](#), [79](#)
20 Questions, [110](#), [121](#)

abstract

 class, [442–444](#)
 method, [261](#), [443](#), [444](#)

abstract, 442, 443, 445
abstract data type (ADT), 461, 470, 471, 499
access modifier, 240, 245, 246, 280
accessor, 244, 248, 288
ActionEvent, 367, 370
 getSource(), 370
ActionListener, 367, 369, 370, 403
 actionPerformed(), 367, 370
activation record, 500
actual parameter, 224
ADT, *see* abstract data type (ADT), 477, 478
algorithm, 37, 499
aliasing, 183
all-pairs testing, 426
AMD, 8
Amdahl's Law, 338
AND (^), 111
Android OS, 7
angle brackets (<>), 479
anniversary, 124
annotation, 428
anonymous class, 251, 265, 267, 367
Apache Harmony, 12
API, *see* application programming interface (API), 297
Apple Inc., 7, 9
Applet, 375, 392
applet, 361, 392–394, 404
application programming interface (API), 283
area code, 127
argument, 43, 68, 224
Ariane 5 rocket, 69
ArithmeticException, 297
array, 71, 165, 220
 as parameter, 226
 as return type, 223

default initial values, 169
dynamic, 461, 465, 470
indexing, 168, 170
instantiation, 169
length, 168, 171, 199
merge, 274, 456
multidimensional, 183, 187, 499
of arrays, 184, 199
of references, 187
one-dimensional, 180–182
out of bounds, 186
overhead in memory, 199
partition, 274, 456
pass by reference, 232
performance penalty, 465
ragged, 185, 199
sorting, 461
static, 168
two-dimensional, 180–182, 184, 327, 495
wasted space, 465
zero-based counting, 168

ArrayDeque, 481

ArrayIndexOutOfBoundsException, 186

ArrayIndexOutOfBoundsException, 171, 186, 261, 297, 302, 418, 534

ArrayList, 481, 490

Arrays, 192, 193

 copyOfRange(), 463
 sort(), 461

ArrayStoreException, 442

ascending order, 176

ASCII, 526

assembly language, 69

assert, 409, 429

assertion, 408–410

 turned off, 410

assignment, 45, 90

astrology

 Chinese, 129

 Western, 131

atomic operation, 344

audio clip, 403

audio sample, 339

AWT: Abstract Window Toolkit, 362

background task, 7

backup

 tape, 6

bank account, 348

base

 2, 13

 8, 14

 10, 12, 230

 16, 13

base case, 497, 499

base class, 277, 440

Bateson, Mary Catherine, 342

bias, 24

BigDecimal, 73

binary, 13

 addition, 16

 digit, 144

 file, 526, 528, 538

 negative numbers, 17

 numbers, 80, 143

 operator, 76, 288

 subtraction, 17

 tree, 510

binary search, 200

binary search tree, 510

binary to decimal conversion, 13

binding, 225

dynamic, 440, 441

static, 441

bioinformatics, 138

bit, 4, 13

masking, 413

bitmap file (bmp), 524

bitwise, 79

bitwise AND, 20

bitwise NOT, 20

bitwise operators, 19

bitwise OR, 20

bitwise XOR, 20

black box testing, 427

block, 114, 141

synchronized, 357

block comment /* */, 56

blocking call, 321

blocking I/O, 347

Bonaparte, Napoleon, 361

Boolean, 96

boolean, 146, 157

operations, 116

boolean, 67

boolean

type, 116

variable, 115

Boolean logic, 84, 110, 111, 136

BorderLayout, 384

Borges, Jorge Luis, 312

borrow, 17

bound

exclusive, 95, 326, 329

inclusive, 95, 326, 329

boundary value analysis, 426, 427

bounded type parameter, 513

BoxLayout, 387

braces ({ }), 114, 141, 223

brackets, 182

 in array declaration, 169

break, 127, 142, 146, 158

break, 142

breakpoint, 409, 412

broader type, 71

bubble sort, 260, 261, 270

 parallel, 274, 456

 pass, 260

buffer, 348, 359

bug, 38, 152, 406

busy waiting, 330, 334, 347

button, 203

 Cancel, 208, 217

 No, 217

 OK, 204, 211, 218

 Yes, 217

Byte, 96, 97

 MAX_VALUE, 99

 MIN_VALUE, 99

byte, 4, 71

 stream, 524

byte, 67, 71, 413

bytecode, 11, 32

C, 10, 74, 168, 169, 186, 221, 231, 296, 418, 432

 printf(), 74

C++, 10, 168, 169, 186, 279, 283, 408, 432, 441

 multiple inheritance, 279

C#, 10

cache, 3, 6

calculator, 301, 494

calendar, 120, 129, 188

call

blocking, 321

method, 220, 224

recursive, 515

static method, 224

Callable, 264

camel case, 54, 92

Camus, Albert, 109

cards, 219, 231

CAS, *see* compare-and-swap (CAS)

cast, 69

error, 413, 414

explicit, 446

implicit, 170, 414

upcast, 81

catch, 295

exception, 295

read-error handling, 547

catch, 298, 306, 535

catch

first matching, 298

catch or specify, 297, 299

cellular automata, 165

Celsius, 203

char, 67, 83, 139, 143, 149, 157, 187

Character, 96

MAX_VALUE, 413

chat program, 545

checked exception, 299

checksum, 539

chemistry, 211

chess

game, 181

representation, 181

chess board, 181

child

- class, 277, 278, 280, 281, 445
- constructor, 281
- of Object, 283

Chinese

- astrology, 129
- New Year, 130

choice, 134

circuit, 289, 290

circular reference, 466

circular wait, 350, 353, 360

circumference of a circle, 94

city distance table, 199

Class, 346

class, 42

- abstract, 373, 442, 444
- anonymous, 251, 265, 267, 367
- as template, 240
- base, 277, 440
- child, 277, 278, 280, 281, 445
- constants, 230
- constructor, 247, 281
- derived, 277, 440, 446
- exception, 301
- for printing, 411
- generic, 400, 478, 479
- hierarchy, 292, 445
- inherit, 277
- inner, 251, 546
- local, 251, 265
- nested, 249
- parameterized, 481
- parent, 277, 279, 281, 283
- static nested, 249

subclass, 277

superclass, 277

variable, 222, 226, 227

wrapper, 96, 104, 259, 512

class file, 172

class variable, 241

ClassCastException, 271

classes

ActionEvent, 367, 370

Applet, 375, 392

ArrayDeque, 481

ArrayList, 481, 490

Arrays, 192, 193

BigDecimal, 73

Boolean, 96

BorderLayout, 384

BoxLayout, 387

Byte, 96, 97

Character, 96

Class, 346

Color, 303

DataInputStream, 532

DataOutputStream, 532

DirectDepositAccount, 439

Double, 96, 97

File, 526, 532

FileInputStream, 528, 529, 538

FileOutputStream, 528, 538

Float, 96, 97

FlowLayout, 380

Graphics, 285, 286

GridBagLayout, 387

GridLayout, 381

GroupLayout, 387

HashMap, 481, 482

HashSet, 481
Integer, 96, 97, 259
JApplet, 392, 393
 JButton, 367
JCheckBoxMenuItem, 389
JFrame, 363, 403
JList, 213
JMenu, 388
JMenuBar, 388
JMenuItem, 388, 389
JOptionPane, 204, 205, 207, 216, 229
 JPanel, 365, 380
JTextArea, 374
 JTextField, 366, 374
LinkedList, 481
Long, 96, 97
Math, 92, 104, 120, 221, 224
MouseAdapter, 373, 403
MouseEvent, 371
Object, 278, 281, 283, 477
ObjectInputStream, 530, 532
 ObjectOutputStream, 530, 532, 535, 548
Point, 285, 293
PrintWriter, 528
Random, 104, 351
Rectangle, 243, 244, 246
Scanner, 173, 229, 289, 528
ServerSocket, 543
Short, 96, 97
Socket, 543, 544
SpringLayout, 387
String, 68, 87–89, 96, 145, 149, 162, 169, 200, 239, 287, 417
SwingWorker, 400
SynchronizedAccount, 349
Thread, 290, 319, 342, 422, 440

TreeMap, 514

TreeSet, 481, 514

Vector, 481, 490

CLASSPATH, 430

client, 540

client code, 270

client program, 247, 248

clock rate, 3

cloud computing, 6

Cobol, 10

code

block, 141

client, 270

instrumenting, 431

malicious, 257

refactor, 463

reuse, 278

code reuse, 240, 513

codon, 202

coefficient of restitution, 39

collection, 180

Color, 303

color

scaling values, 413

column, 180

command

time, 163, 201, 259

command line, 178, 202, 203

command line interface, 41

comment, 47

block, 56

documentation, 56

single-line, 56

to avoid confusion, 180

common mistakes, 406

Comparable, 264, 271, 514
compare-and-swap (CAS), 453
compile time, 478
compiler, 8
 Java, 11
 optimization, 278
 warning, 428
compression, 524 compression ratio, 538
computer, 34
computer program, 35
computer scientist, 174, 260, 414, 458
concat(), 88
concatenate, 52, 87
concurrency, 59
 in the presence of dependency, 103
 unsafe, 348
concurrent, 312
 program, 60, 312
 solution, 62
condition
 boolean, 141
conditional execution, 110
Connect Four, 200
connection
 listening to, 544
console, 41, 368
 I/O, 528
constant, 91, 92
 class, 227
 interface, 261
 named, 91
 public, 246
constructor, 86, 243, 247, 258, 259, 281, 289, 484
 child, 281
 default, 281

invocation, 243

more than one, 243

parent, 281

consumer, 348

container, 365

ConTest, 431

testing for race condition, 438

control flow, 221

non-local, 297

conversion

base 10 to base 3, 162

binary to decimal, 13

decimal to binary, 14, 143

decimal to hexadecimal, 15

Conway's Game of Life, 165, 194, 222

coordinated universal time (UTC), 325

core, 29

CPU, 4

dual, 315

counter

incrementation, 344

coverage

branch, 427

method, 427

metrics, 427

statement, 427

CPU, 331, 356

cache, 3, 6

core, 4

cycles, 335

pipelining, 3

single core, 7

time, 356

wasted cycles, 347

crash, 257

critical section, 344, 360

cryptographic hash function, 539

cryptography, 413

 public key, 313

Ctrl-C, 142

Ctrl-C, 550

Dalvik, 12

Dark Crystal, The, 524

data

 compact storage, 528

 compression, 524

 encapsulation, 239

 leak, 359

 parallelism, 159

 sortable, 509

 type, 66

data structure

 array, 167

 base case, 507

 dynamic, 167, 199, 470

 heterogeneous, 167, 478

 homogeneous, 167, 478

 recursive, 507

 size, 167

 stack, 238, 471

 static, 167, 169, 460

 tree, 509

data structures

 dynamic, 460

data type

 primitive, 528

database, 258

DataInputStream, 532

DataOutputStream, 532

deadlock, 350, 359
deadlock prone methods, 422

debugging
in DrJava, 411
method, 408

decimal, 12, 230

decimal point, 230

decimal to binary conversion, 14

decimal to hexadecimal conversion, 15

declaration

shadow, same name, 407

declare, 45, 70

decomposition

domain, 60, 62, 159, 313, 336

task, 60, 313, 336

decompression, 524, 525

decoration, 428

default label, 127

Dell, 8

@Deprecated, 428

deprecated, 321

dequeue, 475

derive, 277

derived class, 277, 440, 446

descending order, 176

design by contract, 424

design pattern, 424

dialog, 207, 212

CANCEL_OPTION, 204

confirm, 204

input, 204

message, 204

modal, 204, 217

multiple options, 208

NO_OPTIO, 204

non-modal, 204, 217

OK_OPTION, 204

option, 204

yes or no, 207

Yes-No, 204

YES_OPTION, 204

dialog box, 48

dictionary, 482, 510

digit

hexadecimal, 552

most significant, 144

dining philosophers, 342

dir command, 532

DirectDepositAccount, 439

discriminant, 121

distance, 228

divide by zero, 301, 302, 475

division (/), 75

DNA, 138, 148, 202

do, 141, 150

do-while loop, 192

do-while, 141

do-while loop, 150, 151, 221

documentation comment ((/** */), 56

DocumentListener, 374

domain, 315, 336

decomposition, 313, 336

decomposition, 60, 62, 159

input, 61, 62

subdomain, 62

dot notation, 242

Double, 96, 97

MAX_VALUE, 99

MIN_VALUE, 99

parseDouble(), 97, 229

POSITIVE_INFINITY, 99

double, 73, 82

double, 67, 72, 118

double precision, 24, 27

driver, 7

DrJava, 8, 411

 debug mode, 411

drop down list, 212

Dumas, Alexandre, père, 258

DVD, 296

dynamic array, 460, 465

dynamic binding, 440, 441

E, 72

E notation, 22

echo, 339

 delay, 339

Eclipse, 8, 411

EDT, *see* event dispatch thread (EDT)

Einstein, Albert, 240

electrical outlet, 62

ellipse, 265

else, 115

empty list, 469

empty String, 87

encapsulation, 239, 240

English, 111

enqueue, 475

enum, 332

EOFException, 534

EOFException, 530

epoch, 325

equals(), 283

equivalence

partitioning, 426

equivalence partitioning, 427

Error, 300, 301

error

array out of bounds, 186

casting, 406, 413

checking, 230

code, 296

divide by zero, 475

equivalence testing, 407, 417

fencepost, 154

hardware, 296

infinite loop, 407, 415

message, 63

misplaced semicolon, 156

named, 301

null pointer, 407, 421

off-by-one, 154, 407, 415

out of bounds, 407

out of memory, 201, 407

overflow, 406

parallel, 421

precision, 26, 406, 412

propagation, 26

read-error, 547

reference vs. value, 407

scope, 407, 419

shadowing variables, 407

silent, 299

skipped loop, 155

typographical, 152

underflow, 406

uninitialized array, 407

zero loop, 407, 416

escape sequence, 83

Euclidean distance, 228

event, 367

event dispatch thread (EDT), [398](#)

event handling, [204](#)

Excel, [178](#)

Exception, [298](#), [300](#), [301](#)

exception, [97](#), [218](#), [295](#), [297](#)

array out of bounds, [464](#)

catch, [310](#)

checked, [299](#)

class, [301](#)

conversion of string to number, [97](#)

custom, [297](#), [303](#)

handling, [301](#), [526](#), [548](#)

mistake in program, [297](#)

none for overflow/underflow, [412](#)

on overridden method, [447](#)

propagation, [297](#)

thousands of, [311](#)

throw, [295](#), [310](#)

uncaught, [462](#)

unchecked, [299](#)

with inheritance, [447](#)

exceptions

ArithmaticException, [297](#)

ArrayIndexOutOfBoundsException, [171](#), [186](#), [261](#), [297](#), [302](#), [418](#), [534](#)

ArrayStoreException, [442](#)

ClassCastException, [271](#)

EOFException, [530](#)

Error, [300](#), [301](#)

Exception, [298](#), [300](#), [301](#)

ExecutionException, [520](#)

FileNotFoundException, [527](#)

IllegalMonitorStateException, [359](#)

InterruptedException, [299](#), [321](#), [520](#)

IOException, [550](#)

NotSerializableException, [531](#)

NullPointerException, 297, 298, 306, 407, 419, 421

NumberFormatException, 310

RuntimeException, 298, 300, 301

StackOverflowError, 504

exclusive, 326, 329

execution

conditional, 110

sequential, 312

ExecutionException, 520

ExecutorService, 518

exercise path, 427

exponent, 21

expression

concurrent evaluation, 103

splitting for concurrent evaluation, 101

extends, 278–280, 283, 442

extends

omit, 283

factorial, 498

factorization, 313

factory method, 205, 211, 518

Fahrenheit, 203

false, 84

fencepost error, 154

Fibonacci, 498, 523

field, 240, 241, 245, 257, 280, 282

 hide, 281, 282

 static, 226, 227

FIFO, *see* first in, first out (FIFO)

File, 526, 532

file

 append, 537

 binary, 526, 538

 I/O, 172, 540

input, 465

open, 526

system

 hierarchical structure, 527

text, 526, 533, 538

FileInputStream, 528, 529, 538

FileNotFoundException, 527

FileOutputStream, 528, 538

files, 525

FILO, *see* first in, last out (FILO)

final, 91, 195, 227, 403, 445, 455

finally, 297, 300, 534, 535

Firefox, 540

first in, first out (FIFO), 471, 475

first in, last out (FILO), 238, 471, 500

flash drive, 6, 9, 10

Float, 96, 97

 MAX_VALUE, 99

 MIN_VALUE, 99

 parseFloat(), 97

float, 73

float, 67, 72

floating point, 494

 64-bit, 69

 accuracy, 73

 arithmetic, 26

 infinity, 26

 not-a-number, 26

 notation, 24

 number, 81, 143, 229

 precision, 412

 representation, 24

 special numbers, 26

FlowLayout, 380

flush(), 544

for, 141, 144

for loop, 175, 192, 196, 221, 254, 324, 357, 417, 486

bounds, 171

condition, 144

initialization, 144

nested, 195

update, 144

for-each loop, 192

formal parameter, 224

format string, 73

Fortran, 10, 432

fractal, 497

fractions

conversion to binary, 22

non-terminating, 23

frame, 205, 362, 363

title, 364

FTP, 542

function, 221

trigonometric, 93

functional programming, 501

Future, 518

get(), 518

future, 518

fuzz testing, 426

game

complex programs, 190

Connect Four, 200

Conway's Game of Life, 165

poker, 219, 220

Tic Tac Toe, 190, 200

Gandhi, Mahatma, 202

generic, 271, 414

class, 400, 478, 479

 instantiate, 479

 use in Java library, 481

Gibbon, Edward, 260

Gibran, Khalil, 275

global variable, 226

Google, 7, 316

 Android, 7, 12

GPA, 247, 248, 285

graphical user interface (GUI), 42, 203, 216

Graphics, 285, 286

gravity, 293

grid

 infinite, 165

GridLayout, 387

GridLayout, 381

GroupLayout, 387

GUI, *see* graphical user interface, 166, *see also* graphical user interface (GUI)

 creation, 203

 for debugging, 411

GUI: Graphical User Interface, 202, 362

Guitton, Jean, 65

Hamlet, 219

hard drive, 6

hard drive crash, 296

hardware, 2

hash code, 283

hash table, 283, 418

HashMap, 481, 482

HashSet, 481

heap, 201

heterogeneous data structure, 478

hexadecimal, 13, 84, 552

hide

field, 281

variable, 419

high definition (HD), 30

high performance computing, 432

higher order digit, 13

Hoare's rule of consequence, 447

hold and wait, 350

homogeneous data structure, 478

HotSpot, 12

HTML, *see* hypertext markup language (HTML), 541

HTTP, 540, 542, 543, 548

request, 540, 548

response, 541

human user, 202

hypertext markup language (HTML), 394

file, 394

tag, 394

I/O

blocking, 347

console, 528

file, 172

in Java, 526

IBM, 431

icon, 205, 209, 210

IDE, *see* integrated development environment, 411

IDE: Integrated Development Environment, 362

identifier, 45, 54

IEEE floating point, 24, 34, 73

if, 113, 124, 221, 319, 417

IllegalMonitorStateException, 359

immutable, 88, 245

implement

algorithm, 38

interface, 261

import, 246, 285, 286, 289, 364

in place, 226

inclusive, 326, 329

indentation, 116

index

negative, 186

negative, illegal in Java, 418

index card, 465

index variable, 145

indexing

array elements, 170

indexing into an array, 170

infinite loop, 142, 153, 156, 407, 550

infinity, 26

infix notation, 458

Inge, W. R., 295

inheritance, 239, 276, 277, 283, 294, 439, 449

as specialization, 287, 441

code reuse, 440

hierarchy, 276

single, 279

with exceptions, 447

initialize, 70

inlining methods, 237

inner class, 251

inorder traversal, 512

input

command line, 202

domain, 61

keyboard, 203

parsing, 289

redirection, 172, 176

standard, 462

instance, 240

instance data, 240
instance method, 239, 240
`instanceof`, 263, 446
instantiate, 68
instrument code, 431
`int`, 223, 230
`int`, 67, 71, 118
`Integer`, 96, 97, 259

- `MAX_VALUE`, 99, 412
- `MIN_VALUE`, 99
- `parseInt()`, 97, 310

integer

- even, 207
- odd, 207

`Integer.MIN_VALUE`, 174
integers

- negative, 17

integrated development environment (IDE), 42
Intel, 8, 29, 432
Core, 29
Core 2, 29
i3 Core, 29
i5 Core, 29
i7 Core, 29
Thread Checker, 432
Thread Profiler, 432
Vtune Performance Analyzer, 432
interest rate, 439
interface, 261, 265

- command line, 203
- declaration, 261
- implement, 261
- multiple, 261, 280
- no static methods, 263
- set of methods, 261

interfaces

ActionListener, 367, 369, 370, 403

Callable, 264

Comparable, 264, 271, 514

DocumentListener, 374

ExecutorService, 518

Future, 518

ItemListener, 374, 403

Iterable, 481

KeyListener, 403

MouseListener, 370, 384, 403

MouseMotionListener, 374

Runnable, 264, 290, 323

Serializable, 530

interior node, 509

Internet Explorer, 540

interpreter, 10

InterruptedException, 299, 321, 520

invariant, 410

control flow, 410

internal, 410

IOException, 550

iOS, 7

IP: Internet Protocol, 542

address, 543, 552

IPv4, 543

IPv6, 543

iPad, 7

iPhone, 7, 9

iPod, 438

is-a relationship, 440

ItemListener, 374, 403

Iterable, 481

iterate, 139

iteration, 139

JApplet, 392, 393

Java, 10, 36, 167–169, 172, 178, 182, 184, 186, 199, 216, 318, 319, 465, 471

API, 323, 418, 428

bytecode, 11

I/O, 526

library, 481

memory model, 334

optimization, 417

platform independence, 330

relational operators, 118

shared memory, 318

statically typed, 67

strongly typed, 67, 69

testing tools, 427

Java 5, 428

Java virtual machine (JVM), 11, 127, 187, 201, 223, 278, 297, 310, 329–331, 344, 409, 431, 463

javac, 8

javadoc, 56

JButton, 365

JButton, 367

JButton

 adding ActionListener, 369

JCheckBoxMenuItem, 389

JFrame, 363, 403

 DISPOSE_ON_CLOSE, 363

 EXIT_ON_CLOSE, 364

 HIDE_ON_CLOSE, 364

 pack(), 404

 setDefaultCloseOperation(), 363

 setJMenuBar(), 388, 403

 setResizable(), 363

 setSize(), 363

 setVisible(), 363

JIT, *see* just-in-time compiler (JIT)

JList, 213

JMenu, 388

JMenuBar, 388

JMenuItem, 388, 389

job interview, 438

JOptionPane, 48, 68

QUESTION_MESSAGE, 63

showInputDialog(), 63

JOptionPane, 204, 205, 207, 216, 229

CANCEL_OPTION, 208

ERROR_MESSAGE, 206

INFORMATION_MESSAGE, 206

PLAIN_MESSAGE, 206

QUESTION_MESSAGE, 206

showConfirmDialog(), 204, 208

showInputDialog(), 204, 211

showMessageDialog(), 204–206, 211, 217

showOptionDialog(), 204

WARNING_MESSAGE, 206

YES_NO_CANCEL_OPTION, 208

YES_NO_OPTION, 208

JPanel, 365, 380

JTextArea, 374

JTextField, 366, 374

JUnit, 428, 431

annotation, 429

assert, 429

test cases, 429

just-in-time compiler (JIT), 11

JVM, *see* Java Virtual Machine (JVM)

for scheduling threads, 342

memory, 283

key, 482

keyboard, 6

reading from, 526

keyboard input, 203

KeyListener, 403

keyword, 43, 63

`new`, 267

keywords, 55

`abstract`, 442, 443, 445

`assert`, 409, 429

`boolean`, 67

`break`, 142

`byte`, 67, 71, 413

`catch`, 298, 306, 535

`char`, 67, 83, 139, 143, 149, 157, 187

`do`, 141, 150

`double`, 67, 72

`else`, 115

`extends`, 278–280, 283, 442

`false`, 84

`final`, 91, 195, 227, 403, 445, 455

`finally`, 297, 300, 534, 535

`float`, 67, 72

`for`, 141, 144

`if`, 113, 124, 221, 319, 417

`import`, 246, 285, 286, 289, 364

`instanceof`, 263, 446

`int`, 67, 71

`long`, 67, 71

`new`, 169, 170, 182, 184, 242, 243, 297

`null`, 297, 468

`private`, 226, 228, 246, 257, 280

`protected`, 246, 280, 286, 292, 294

`public`, 42, 222, 226, 228, 243, 244, 246, 257, 292

`return`, 223

`short`, 67, 71

static, 222, 226, 241, 286
super, 281, 282, 286–288, 292
switch, 221
synchronized, 256, 290, 345–347, 355, 453, 488, 491
this, 244, 245, 257, 281, 345
throw, 297
throws, 300
transient, 531
true, 84, 85
try, 302, 306
void, 223, 245
volatile, 334
while, 141, 144

killer app, 8

King's pawn, 181

L & F, *see* look and feel (L & F)

label

- in switch, 127

language

- high-level, 10
- intermediate, 11
- low-level, 10
- machine, 10

largest value, 174

last in, first out (LIFO), 238

layer

- network, 543
- physical, 542

layout manager, 380

LCG, *see* linear congruential generator

leaf

- of a tree, 509

leaf node, 509

leak data, 359

leap year, 120, 136

left brace, 43

Let's Make a Deal, television show, 109

library, 92

LIFO, *see* last in, first out (LIFO)

linear congruential generator, 233

link, 460, 466

linked list, 199, 460, 465, 467, 470

empty, 469

implementation, 465

insertion at beginning, 469

insertion at end, 469

insertion in middle, 469

insertion in order, 469

link, 460

node, 466

not empty error, 407

picture, 493

recursive data structure, 507

size, 507

tail, 468

walk, 469

with a cycle, 415

LinkedList, 481

Linux, 7, 163, 172, 541, 549

file system, 527

list, 180

drop down, 212

empty, 469

linked, 460, 465

multidimensional, 182

of words, 176

singly linked, 438

traversal, 438

listener, 268, 367

register, 367

literal, 66, 70

livelock, 353

load balancing, 197

local

 class, 251

 variable, 226

local class, 265

lock, 345

lock-free, 453

logic

 Boolean, 84

logic gate, 275

logical AND, 275, 289

logical AND (&&), 85, 116

logical NAND, 289

logical NOR, 289

logical NOT, 288

logical NOT (!), 85

logical NOT(!), 116

logical OR, 289

logical OR (||), 85, 116

logical XOR, 289

logical XOR (^), 85, 116

Long, 96, 97

 MAX_VALUE, 99

 MIN_VALUE, 99

long, 67, 71

look and feel, 363

look and feel (L & F), 205

loop, 139, 221, 319

loops

 almost infinite, 153, 163

 bad header, 416

 condition, 144

 do-while, 141, 150, 192

`for`, 141, 163, 192
`for-each`, 192
header, 156
infinite, 142, 153, 156, 415
initialization, 144
inner, 151
nested, 140, 151, 152
nested for, 158, 159, 163
never executed, 156, 416
outer, 151
skipped, 155
syntax, 141
update, 144
use with arrays, 171
`while`, 141, 150, 192, 493
low-level language, 10
lowest order digit, 13
`ls` command, 532

Mac OS, 7, 541
machine code, 10
machine language, 10
main thread, 318
`main()`, 230
`main()`, 223
maintenance, 38
makespan, 196
malicious code, 257
mantissa, 21, 24, 34
map, 482
map reduce, 316
masking bits, 413
Math, 92, 104, 120, 221, 224
 `ceil()`, 428
 `cos()`, 93

`exp()`, 93

`log()`, 93

`PI`, 227

`pow()`, 93, 428, 504

`random()`, 93, 231

`round()`, 93

`sin()`, 93, 428

`sqrt()`, 93, 221, 222

`tan()`, 93

`matrix`, 189

`matrix multiplication`, 316, 340

`maze`, 495

`mean`, 179, 200, 235

`median`, 180, 235

`member variables`, 241

`memoization`, 502

`memory`

 non-volatile, 525

 volatile, 525

`message`

 broadcast, 318

`message passing`, 318

`method`, 42, 68, 220, 222

 abstract, 261, 443, 444

 accessor, 244, 248

 actual parameter, 224

 argument, 43, 224

 body, 223

 call, 114, 220, 224

 constructor, 243

 decoration, 428

 dynamic binding, 441

 factory, 518

 formal parameter, 224

 helper, 418

instance, 239, 240
main(), 223
mutator, 244, 245, 247
non-static, 239
overloading, 224
overridden, 440
override, 278, 281, 283, 428
parameter, 223
parent, 281, 282
pass into, 224
postcondition, 410
precondition, 410
proxy, 508
recursive, 499, 502, 517
return, 223
return nothing, 223
scope, 225
signature, 224
static, 220–222, 227, 235, 240, 257
synchronized, 348, 349
synchronized static, 346
syntax, 222
void, 223

microprocessor, 3

Microsoft, 7, 8

Excel, 8, 178

Office, 8

PowerPoint, 8

Windows, 7, 9

Word, 8

Xbox 360, 8

Zune, 438

milliseconds (ms), 325

modal dialog, 204, 206, 217

mode, 200

modularity, 220
modulus (%), 75
Moler, Cleve, 412
monitor, 344
 hardware, 6
monolithic code, 220
month to days conversion, 128
Monty Hall
 simulation, 163
Monty Hall problem, 109, 110
Moore's Law, 30
Motorola Xoom, 7, 10
mouse, 6
MouseAdapter, 373, 403
 mouseClicked(), 373
 mouseEntered(), 373
MouseEvent, 371
MouseListener, 370, 384, 403
 mouseClicked(), 370
 mouseEntered(), 370
 mouseExited(), 371
 mousePressed(), 371
 mouseReleased(), 371
MouseMotionListener, 374
MPI, 318
ms, *see* milliseonds (ms)
multi-processor system, 60
multi-threaded, 60
 environment, 439
multi-threaded program
 testing, 486
multicore, 4, 29, 34, 60, 312, 344
 machine, 360
multidimensional, 499
multiple interfaces, 280

mutator, 244, 245, 247, 280, 288

mutual exclusion, 344, 350, 488

named

constants, 91

NaN, *see* not-a-number

nanoseconds (ns), 325

native code, 10

natural language, 10

nested

choice, 113

class, 249

for loop, 195

loop, 140

nested for loops, 163

nested class

access modifiers, 250

nested loops, 151

NetBeans, 362

network

address, 543

API, 542

client, 540

communications, 543

layer, 542, 543

server, 540

storage, 6

Neutral Milk Hotel, 439

new, 169, 170, 182, 184, 242, 243, 297

new, 267

New York, 137

newline('n'), 83

Nintendo, 8

Nintendo Wii, 8

no preemption, 350

node, 466

 interior, 509

 leaf, 509

 root, 509

 tree, 509

non-atomic operation, 344

non-modal dialog, 204, 217

NOT (\neg), 113

not-a-number, 26

notation

 dot, 242

 infix, 458

 postfix, 246, 458

 scientific, 72

notify(), 347

notifyAll(), 347

notifyAll(), 349

NotSerializableException, 531

nucleotide base, 202

null

 reference, 407

null, 212, 297, 468, 507

NullPointerException, 297, 298, 306, 407, 419, 421

NullPointerException, 187

number

 binary, 143

 decimal to binary, 143

 even, 217

 floating point, 143

 odd, 217

 prime, 147, 312

 pseudorandom, 96

 random, 208, 217

 triangular, 151, 152

number system, 12

binary, 13

decimal, 12

hexadecimal, 13

octal, 14

NumberFormatException, 310

numbers

E notation, 22

fractional part, 22

integer part, 22

scientific notation, 21

numeral, 12

Nvidia, 8

Object, 278, 281, 283, 477

clone(), 283

equals(), 283, 418

getClass(), 446

getClass(), 283

hashCode(), 283

notify(), 283

notifyAll(), 283

toString(), 283

wait(), 283, 347

object, 42, 68, 114

collection of data and methods, 240

lock, 346

waiting threads, 346

object oriented design, 261

object oriented programming, 239–241, 246, 248, 258

controversy, 239

performance, 278

ObjectInputStream, 530, 532

ObjectOutputStream, 530, 532, 535, 548

objects, 239

octal, 14
off-by-one, 407
 error, 414
off-by-one error, 154
OK, 204, 211, 218
one's complement, 32
one-dimensional array, 180, 182
ones's complement, 18
OOP, *see* object oriented programming
open source, 7
operating system, 7, 541
operating system (OS), 172, 363
operation
 atomic, 344
 non-atomic, 344
operator, 67
 binary, 288
 bitwise, 79
 logical, 85
 precedence, 75
 relational, 84
 unary, 76, 287
operator precedence, 485, 486
OR (V), 111
Oracle Corporation, 12
order of operations, 75, 458
ordinal numbers, 129
OS, *see* operating system, 172, 201, 259, 329, 330, 344
OSI, 541
 Application layer, 542
 Data link layer, 542
 Network layer, 542
 Physical layer, 542
 Presentation layer, 542
 Session layer, 542

seven layer model, 541

Transport layer, 542

out of memory, 407

OutOfMemoryError, 201

output

command line, 202

redirection, 173

to file, 173

to screen, 173

oval, 265

overflow, 19, 412

error, 413

overload, 224

@Override, 428

override, 428

override method, 278, 281

Pólya, George, 35

package, 45, 246, 280

wildcare, 47

package-private, 246

pair programming, 425

palindrome, 229, 236

panel, 362

parallel

program, 338

parallel program, 355

parallelism, 59

data, 159

parameter, 68, 223

actual, 224

binding, 225

formal, 224

pass by value, 225, 226, 511

type, 479

parameters, 40
parent, 283
 method, 281, 282
parent class, 277, 279, 281, 282
parse, 229
 input, 289
parseDouble(), 494
parsing, 459
pass, 224
 bubble sort, 260
pass by value, 225, 226, 511
path, 527
PCI Express, 6
Peel, John, 406
performance penalty, 465
perimeter, 285
peripherals, 6
PHP, 263
physical layer, 542
physicist, 165
physics, example from, 432
Picasso, Pablo, 1
pipelining, 3
pixel, 30, 413
platform independence, 549
Point, 285, 293
poker, 219
 five card, 236
 three card, 219, 220, 230
polling, 330, 334
polymorphism, 239, 283, 440, 442, 445, 449
pool of threads, 518
pop, 238, 459, 471
port, 543
post-increment, 493

postcondition, 410
postfix notation, 246, 458
precedence of operators, 459
precision, 22
 double, 24, 27
 loss of, 81
 single, 24, 25
precondition, 410
presentation layer, 542
primality test, 417
primality testing, 147, 163
prime factor, 335
prime number, 312
prime numbers, 147
prime, large, 163
primitive type, 67
print
 for debugging, 408
 use in debugging, 410
printer, 6
printf() function, 74
PrintWriter, 528
priority inversion, 356
private, 285, 485
private, 226, 228, 245, 246, 257, 258, 280
proactive password checker, 310
procedural language, 221
process
 message passing, 318
processor
 dual-core, 315
 multicore, 312
 quadcore, 338
processor time, 201
processors, 344

producer, 348

producer/consumer, 347, 359

program, 35

 concurrent, 312, 325, 336, 342

 crash, 168, 257

 debugging, 319

 parallel, 338, 355

 prompt, 202

 sequential, 60, 312

programmer

 veteran, 406

programming, 35

 functional, 501

programming language, 10, 36

programs

 large, 463

Project Gutenberg, 201

promotion, 81

prompt, 202

protected, 246, 280, 286, 292, 294

proxy method, 508

pseudocode, 37

pseudorandom, 96

pseudorandom numbers, 234

public, 245

 methods, 410

public, 42, 222, 226, 228, 243–246, 257, 258, 292

public key cryptography, 313

push, 238, 459, 471

PVM, 318

Python, 10

quadcore, 338

quadratic formula, 120

queue, 471, 475

dequeue, 475

enqueue, 475

linked list, 476

operations, 475

quiet failure, 378

quotient, 75

race condition, 135, 343, 438

ragged array, 185, 199

 to save space, 185

RAM, *see* random access memory (RAM)

Random, 104, 351

 nextBoolean(), 95

 nextDouble(), 95

 nextInt(), 95

random, 133

 boolean value, 96

 number generator, 208

 numbers, 109, 110

random access, 168

random access memory (RAM), 6, 525

random door, 110

random number, 94, 133, 151, 208

rational numbers, 26

re-entrant lock, 345

read-error, 547

readability, 221

real world problems, 174

Rectangle, 243, 244, 246

recursion, 495

recursive

 call, 499

 call, 515

 case, 497, 499

 data structure, 507

definition, 497

method, 499

recursive method

base case, 499

recursive case, 499

redirect

input, 172, 176

output, 173

refactor, 463

reference, 67

comparison, 417

comparison of, 407

reference type, 67, 183

reference variable, 242

register, 367

regression testing, 427

black box, 427

white box, 427

relational operator, 84

remainder, 75

repetition, 139

requirements document, 37

reserved word, 43

resource starvation, 353

restaurant

simulation, 359

return, 49

optional, 223

type, 281

return, 223

return type, 222

reusability, 221

right brace, 43

RLE, *see* run length encoding (RLE)

bitmap compression, 539

encoding, 525, 538

root, 509

row, 180

 instantiation, 184

run length encoding (RLE), 524

run time

 type checking, 478

run(), 320, 352

run(), 320

Runnable, 264, 290, 323

RuntimeException, 298, 300, 301

Safari, 540

safe

 banking operations, 439

sample, 339

Scanner, 157, 544, 547, 548

Scanner, 173, 229, 289, 528

 close(), 528

 hasNext(), 482

 hasNextInt(), 173

Scanner, 175

scheduling

 fairness, 353

 nondeterministic, 343

scientific computing, 432

scientific notation, 21, 72, 229, 230

 normalized, 22

scope, 225

 error, 419

 variable hiding, 243

score

 maximum, 178

 minimum, 178

search

 binary, 200

 fast, 200

secondary storage, 525

selection sort, 175, 200, 201, 248, 260

semantics, 42

semaphore, 353

semicolon, 144, 159

 misplaced, 156

semicolon (;)

 in **for** header, 415

while header, 415

sentinel value, 155, 302

sequential, 59

 program, 60

sequential access, 168

sequential instructions, 239

sequential program, 60

Serializable

 no methods, 531

Serializable, 530

server, 540, 543

ServerSocket, 543

setLength(), 245

shadow, 419

Shakespeare, William, 219

shared

 variable, 339, 343

shared memory, 318

Short, 96, 97

 MAX_VALUE, 99

 MIN_VALUE, 99

short, 67, 71

short circuit logic, 117

shuffle, 220, 231

side effect, 103

signature, 224

signed left shift, 20

signed right shift, 20

significand, 24

significant digits, 24

simulation

 Conway's Game of Life, 194

 mechanics, 310

single inheritance, 279

single precision, 24, 25

single-line comment (//), 56

skipped loops, 155
smallest value, 174
smartphone, 9
smoke test, 425
Socket, 543, 544
socket, 543, 550
software, 2, 7
 development lifecycle, 37
 maintenance, 38
solid state drive, 6
solution
 concurrent, 62
sequential, 62
Sony, 8
Sony PS3, 8
sort
 ascending or descending, 274
 ascending order, 176
 bubble parallel, 456
 data, 175
 descending order, 176
 in parallel, 457
 list, 200
 parallel, 274
 parallel vs. sequential, 274
 selection, 175, 200, 201, 248
sorting, 174, 248, 260
 bubble sort, 260, 270
 selection sort, 175, 260
speedup, 63, 103, 324, 325, 340
matrix multiplication, 329
maximum, 338
SpringLayout, 387
square root, 222
 of variance, 179

SSD, *see* solid state drive

stack, 238, 239, 257, 459, 471, 499

ADT, 477

as FILO structure, 471

empty, 257, 471

operations, 472

pop, 238, 459, 471

push, 238, 459, 471

top, 238, 459, 471

stack frame, 500

StackOverflowError, 504

standard deviation, 179, 200, 235

standard input, 462

starvation, 353, 360

statement

assignment, 90

conditional, 156

executable, 156

static

binding, 441

data structure, 460

field, 226, 227

method, 220, 222, 227, 235, 257

static, 222, 226, 241, 286

static binding, 441

static nested class, 249

statically typed language, 67

statistics, 178

Stephenson, Neal, 540

stop(), 321

String, 68, 87–89, 96, 145, 149, 162, 169, 200, 239, 287, 417

charAt(), 145

indexOf(), 89

length(), 88

String

concatenation, 52, 87, 88

immutable, 88

pooling, 417

string, 44

String conversion, 97

strongly typed language, 67

student roster, 247

subclass, 277

subdomain, 62

 testing, 426

subexpression

 tasks for, 318

subproblem, 499

substring, 89

Sun Microsystems, Inc., 12

super, 281, 282, 286–288, 292

superclass, 277, 298

@SuppressWarnings, 428

swap

 in bubble sort, 260

Swing library, 204, 362

SwingWorker, 400

switch, 126, 127

 use of break, 146

switch, 221

synchronization, 283, 355

 tools, 342

synchronized

 synchronized method, 345

synchronized, 256, 290, 345–347, 355, 453, 488, 491

synchronized

 block, 346, 349, 357

 method, 348, 349

 read and write, 349

 static, 346

SynchronizedAccount, 349

syntax, 42

System

 currentTimeMillis(), 325, 340

 err, 411

 exit(), 364

 nanoTime(), 294, 311, 325

 out, 411, 528

 out.print(), 410

 out.println(), 151, 298, 410

tablet computer, 10

tag, 394

 <applet>, 394

 <body>, 394

 <head>, 394

 <html>, 394

 , 394

 <title>, 394

tail

 linked list, 468

 pointer, 468

tape backup, 6

task, 336

 coordination, 62

 decomposition, 60, 313, 336

 division, 315

tax rate, 456

TCP, 542, 543

 connection, 543

Telnet, 543

temperature, 203

template

 for exam, 414

terminal, 41, 166, 167

ternary operator, 191

testing

- all-pairs, 426
- black box, 427, 438

- boundary value analysis, 426, 427

- branch coverage, 427

- coverage metrics, 427

- equivalence partitioning, 427

- exercise path, 427

- for primality, 147

- functional, 428

- JUnit, 428, 429, 432

- method coverage, 427

- multi-threaded program, 486

- regression, 427

- statement coverage, 427

- subdomain, 426

- tools for concurrency, 431

- white box, 427

text

- box, 203

- editor, 173

- file, 526, 533, 538

text editor, 172

this, 244, 245, 257, 281, 345

Thread, 290, 319, 342, 422, 440

- join(), 102, 321, 324, 329, 336, 339, 518, 546

- NORM_PRIORITY, 332

- resume(), 422

- sleep(), 167, 299, 330–332, 339, 347

- start(), 321, 323, 517, 546

- stop, 422

- suspend(), 422, 428

- wait(), 330

- yield(), 330, 331, 333

Thread

deprecated methods, [422](#)

thread, [60](#), [317](#)

background task, [329](#)

combine results, [101](#)

creation, [320](#)

data sharing, [321](#)

independent execution, [321](#)

main, [318](#), [321](#)

order of execution, [329](#)

overhead, [340](#)

pool, [518](#)

possible orderings, [431](#)

print order, [340](#)

priority, [329](#), [332](#)

running time, [340](#)

scheduling, [329](#)

share memory space, [318](#)

starting, [321](#)

task completion, [103](#)

time sharing, [329](#)

to evaluate a sub-expression, [101](#)

waiting, [321](#)

thread of execution, [233](#)

thread-safe, [256](#), [440](#), [486](#)

Threads tab

in debugging pane, [411](#)

throw

checked exception, [299](#)

throw, [297](#)

throw exception, [295](#)

throws, [300](#)

Tic Tac Toe, [190](#), [200](#)

time

concurrent, [324](#)

current, 325

sequential, 324

timing, 259

OS time command, 163, 201, 259

System.currentTimeMillis(), 325

System.nanoTime(), 294, 325

top, 238, 459, 471

top-down programming, 37

toString(), 248, 283, 290

Tower of Hanoi, 503

transient, 531

tree, 509

binary, 510

binary search, 510

leaf of, 509

node, 509

root, 509

traversal, 512

TreeMap, 514

TreeSet, 481, 514

triangular numbers, 151

Tribe Called Quest, A, 138

trigonometric functions, 93

true, 84, 85

truth table, 111

try, 302, 306

try-catch, 218, 299

try-catch, 321, 324

two's complement, 17, 18, 32, 71

two-dimensional array, 182, 184, 495

type, 45, 66

broader, 71

cast, 69

primitive, 67

promotion, 81

raw, 479

reference, 67, 183

return, 281

safe, 479

upcast, 81

type checking, 478

type parameter, 479, 513

U.S. Marine Corps, 292

U.S. Navy, 292

UDP, 542, 543

 communication, 543

unary operator, 76, 287

unchecked exception, 299

underflow, 19, 412

 error, 413

underscore (_), 92

Unicode, 83, 107, 187

United States, 180

universal resource locator (URL), 375, 403

universe, 165

Unix, 163, 172, 296, 549

 epoch, 325

 file system, 527

unresponsive GUI, 399

unsigned right shift, 20

upcast, 81

URL, *see* universal resource locator (URL)

UTC, *see* coordinated universal time (UTC)

variable, 45, 66

 binding, 225

 class, 222, 226, 227

 global, 226, 296

 hide, 419

index, 145

initialize, 70

local, 226

member, 241

reference, 67

shadow, 419

shadowing, 407

shared, 339

value in debugging pane, 411

variable hiding, 243

variance, 179

Vector, 481, 490

vector, 189

vertex, 293

video game, 95, 314

virtual address, 283

virtual file, 526

virus, 335

Visual Basic, 10

void, 223, 245

volatile, 334

volatile memory, 525

von Neumann architecture, 2

wait(), 347

web browser, 543

web server, 540, 552

West, Mae, 165

while, 141, 144

while loop, 192, 221, 493

white box testing, 427

widget, 203, 365

wildcard

 package, 47

Windows, 7, 541

7, 9

8, 9

command prompt, 532

file system, 527

use of \, 527

Wirth, Niklaus, 458

word size, 3

worker thread, 399

wrapper, 474

Boolean, 96

Byte, 96

Character, 96

Double, 96

Float, 96

Integer, 96

Long, 96

Short, 96

wrapper class, 96, 104, 259, 512

XOR (\oplus), 112

Yes-No dialog, 204

zero loop, 416

zero-based counting, 154, 168, 414

zodiac, 129

Zune bug, 438

START CONCURRENT

2014 EDITION

Multicore microprocessors are now at the heart of nearly all desktop and laptop computers. While these chips offer exciting opportunities for the creation of newer and faster applications, they also challenge students and educators. How can the new generation of computer scientists growing up with multicore chips learn to program applications that exploit this latent processing power? This unique book is an attempt to introduce concurrent programming to first-year computer science students, much earlier than most competing products.

This book assumes no programming background but offers a broad coverage of Java. It includes 167 numbered and numerous inline examples as well as 334 exercises categorized as "conceptual," "programming," and "experiments." The problem-oriented approach presents a problem, explains supporting concepts, outlines necessary syntax, and finally provides its solution. All programs in the book are available for download and experimentation. A substantial index of 3,038 entries makes it easy for readers to locate relevant information.

In a fast-changing field, this book is continually updated and refined. The 2014 version is the seventh "draft edition" of this volume, and features numerous revisions based on student feedback.

ABOUT THE AUTHORS

Barry Wittman is an Assistant Professor of Computer Science at Elizabethtown College. His areas of active research are computer science pedagogy, particularly for parallel computing, and approximation algorithms for graph problems.

Aditya Mathur is a Professor of Computer Science at Purdue University. He has taught courses in computer science at all levels since 1972. He has written several books, the most well-known being an introduction to microprocessors, the first book of its kind published in India. Aditya has published extensively in international journals and conferences in the area of software engineering.

Tim Korb is the Assistant Department Head of the Computer Science Department at Purdue University. He was previously at the University of Arizona, where he was a co-designer of the Icon programming language. His technical interests also include operating systems, networking, databases, and security.

PURDUE UNIVERSITY

