Introduction to Programming in Java

An Interdisciplinary Approach

Robert Sedgewick
and
Kevin Wayne

Princeton University

Boston San Francisco New York
London Toronto Sydney Tokyo Singapore Madrid
Mexico City Munich Paris Cape Town Hong Kong Montreal

Publisher	Greg Tobin
Executive Editor	Michael Hirsch
Associate Editor	Lindsey Triebel
Associate Managing Editor	Jeffrey Holcomb
Senior Designer	Joyce Cosentino Wells
Digital Assets Manager	Marianne Groth
Senior Media Producer	Bethany Tidd
Senior Marketing Manager	Michelle Brown
Marketing Assistant	Sarah Milmore
Senior Author Support/ Technology Specialist	Joe Vetere
Senior Manufacturing Buyer	Carol Melville
Copyeditor	Genevieve d'Entremont
Composition and Illustrations	Robert Sedgewick and Kevin Wayne

Cover Image: © Robert Sedgewick and Kevin Wayne

Page 353 © 2006 C. Herscovici, Brussels / Artists Rights Society (ARS), New York Banque d' Images, ADAGP / Art Resource, NY

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

The interior of this book was composed in Adobe InDesign.

Library of Congress Cataloging-in-Publication Data

Sedgewick, Robert, 1946-
Introduction to programming in Java : an interdisciplinary approach / by Robert Sedgewick and Kevin Wayne.
p. cm.
Includes index.
ISBN 978-0-321-49805-2 (alk. paper)
1. Java (Computer program language) 2. Computer programming. I. Wayne, Kevin Daniel, 1971- II. Title.
QA76.73.J38S413 2007
005.13'3--dc22

2007020235

ISBN-13: 978-0-321-49805-2
ISBN-10: 0-321-49805-4

1 2 3 4 5 6 7 8 9 10—CRW—11 10 09 08 07

Preface

THE BASIS FOR EDUCATION IN THE last millennium was "reading, writing, and arithmetic;" now it is reading, writing, and *computing*. Learning to program is an essential part of the education of every student in the sciences and engineering. Beyond direct applications, it is the first step in understanding the nature of computer science's undeniable impact on the modern world. This book aims to teach programming to those who need or want to learn it, in a scientific context.

Our primary goal is to *empower* students by supplying the experience and basic tools necessary to use computation effectively. Our approach is to teach students that writing a program is a natural, satisfying, and creative experience (not an onerous task reserved for experts). We progressively introduce essential concepts, embrace classic applications from applied mathematics and the sciences to illustrate the concepts, and provide opportunities for students to write programs to solve engaging problems.

We use the Java programming language for all of the programs in this book—we refer to Java after programming in the title to emphasize the idea that the book is about *fundamental concepts in programming*, not Java per se. This book teaches basic skills for computational problem-solving that are applicable in many modern computing environments, and is a self-contained treatment intended for people with no previous experience in programming.

This book is an *interdisciplinary* approach to the traditional CS1 curriculum, where we highlight the role of computing in other disciplines, from materials science to genomics to astrophysics to network systems. This approach emphasizes for students the essential idea that mathematics, science, engineering, and computing are intertwined in the modern world. While it is a CS1 textbook designed for any first-year college student interested in mathematics, science, or engineering (including computer science), the book also can be used for self-study or as a supplement in a course that integrates programming with another field.

Coverage The book is organized around four stages of learning to program: basic elements, functions, object-oriented programming, and algorithms (with data structures). We provide the basic information readers need to build confidence in writing programs at each level before moving to the next level. An essential feature of our approach is to use example programs that solve intriguing problems, supported with exercises ranging from self-study drills to challenging problems that call for creative solutions.

Basic elements include variables, assignment statements, built-in types of data, flow of control (conditionals and loops), arrays, and input/output, including graphics and sound.

Functions and modules are the student's first exposure to modular programming. We build upon familiarity with mathematical functions to introduce Java static methods, and then consider the implications of programming with functions, including libraries of functions and recursion. We stress the fundamental idea of dividing a program into components that can be independently debugged, maintained, and reused.

Object-oriented programming is our introduction to data abstraction. We emphasize the concepts of a data type (a set of values and a set of operations on them) and an object (an entity that holds a data-type value) and their implementation using Java's class mechanism. We teach students how to *use*, *create*, and *design* data types. Modularity, encapsulation, and other modern programming paradigms are the central concepts of this stage.

Algorithms and data structures combine these modern programming paradigms with classic methods of organizing and processing data that remain effective for modern applications. We provide an introduction to classical algorithms for sorting and searching as well as fundamental data structures (including stacks, queues, and symbol tables) and their application, emphasizing the use of the scientific method to understand performance characteristics of implementations.

Applications in science and engineering are a key feature of the text. We motivate each programming concept that we address by examining its impact on specific applications. We draw examples from applied mathematics, the physical and biological sciences, and computer science itself, and include simulation of physical systems, numerical methods, data visualization, sound synthesis, image processing, financial simulation, and information technology. Specific examples include a treatment in the first chapter of Markov chains for web page ranks and case studies that address the percolation problem, N-body simulation, and the small-world

phenomenon. These applications are an integral part of the text. They engage students in the material, illustrate the importance of the programming concepts, and provide persuasive evidence of the critical role played by computation in modern science and engineering.

Our primary goal is to teach the specific mechanisms and skills that are needed to develop effective solutions to any programming problem. We work with complete Java programs and encourage readers to use them. We focus on programming by individuals, not library programming or programming in the large (which we treat briefly in an appendix).

Use in the Curriculum This book is intended for a first-year college course aimed at teaching novices to program in the context of scientific applications. Taught from this book, prospective majors in any area of science and engineering will learn to program in a familiar context. Students completing a course based on this book will be well-prepared to apply their skills in later courses in science and engineering and to recognize when further education in computer science might be beneficial.

Prospective computer science majors, in particular, can benefit from learning to program in the context of scientific applications. A computer scientist needs the same basic background in the scientific method and the same exposure to the role of computation in science as does a biologist, an engineer, or a physicist.

Indeed, our interdisciplinary approach enables colleges and universities to teach prospective computer science majors and prospective majors in other fields of science and engineering in the *same* course. We cover the material prescribed by CS1, but our focus on applications brings life to the concepts and motivates students to learn them. Our interdisciplinary approach exposes students to problems in many different disciplines, helping them to more wisely choose a major.

Whatever the specific mechanism, the use of this book is best positioned early in the curriculum. First, this positioning allows us to leverage familiar material in high school mathematics and science. Second, students who learn to program early in their college curriculum will then be able to use computers more effectively when moving on to courses in their specialty. Like reading and writing, programming is certain to be an essential skill for any scientist or engineer. Students who have grasped the concepts in this book will continually develop that skill through a lifetime, reaping the benefits of exploiting computation to solve or to better understand the problems and projects that arise in their chosen field.

Prerequisites This book is meant to be suitable for typical science and engineering students in their first year of college. That is, we do not expect preparation beyond what is typically required for other entry-level science and mathematics courses.

Mathematical maturity is important. While we do not dwell on mathematical material, we do refer to the mathematics curriculum that students have taken in high school, including algebra, geometry, and trigonometry. Most students in our target audience (those intending to major in the sciences and engineering) automatically meet these requirements. Indeed, we take advantage of their familiarity with the basic curriculum to introduce basic programming concepts.

Scientific curiosity is also an essential ingredient. Science and engineering students bring with them a sense of fascination in the ability of scientific inquiry to help explain what goes on in nature. We leverage this predilection with examples of simple programs that speak volumes about the natural world. We do not assume any specific knowledge beyond that provided by typical high school courses in mathematics, physics, biology, or chemistry.

Programming experience is not necessary, but also is not harmful. Teaching programming is our primary goal, so we assume no prior programming experience. But writing a program to solve a new problem is a challenging intellectual task, so students who have written numerous programs in high school can benefit from taking an introductory programming course based on this book (just as students who have written numerous essays in high school can benefit from an introductory writing course in college). The book can support teaching students with varying backgrounds because the applications appeal to both novices and experts alike.

Experience using a computer is also not necessary, but also is not at all a problem. College students use computers regularly, to communicate with friends and relatives, listen to music, process photos, and many other activities. The realization that they can harness the power of their own computer in interesting and important ways is an exciting and lasting lesson.

In summary, virtually all students in science and engineering are prepared to take a course based on this book as a part of their first-semester curriculum.

Goals What can *instructors* of upper-level courses in science and engineering expect of students who have completed a course based on this book?

We cover the CS1 curriculum, but anyone who has taught an introductory programming course knows that expectations of instructors in later courses are typically high: each instructor expects all students to be familiar with the computing environment and approach that he or she wants to use. A physics professor might expect some students to design a program over the weekend to run a simulation; an engineering professor might expect other students to be using a particular package to numerically solve differential equations; or a computer science professor might expect knowledge of the details of a particular programming environment. Is it realistic to meet such diverse expectations? Should there be a different introductory course for each set of students? Colleges and universities have been wrestling with such questions since computers came into widespread use in the latter part of the 20th century. Our answer to them is found in this common introductory treatment of programming, which is analogous to commonly accepted introductory courses in mathematics, physics, biology, and chemistry. *An Introduction to Programming* strives to provide the basic preparation needed by all students in science and engineering, while sending the clear message that there is much more to understand about computer science than programming. Instructors teaching students who have studied from this book can expect that they have the knowledge and experience necessary to enable them to adapt to new computational environments and to effectively exploit computers in diverse applications.

What can *students* who have completed a course based on this book expect to accomplish in later courses?

Our message is that programming is not difficult to learn and that harnessing the power of the computer is rewarding. Students who master the material in this book are prepared to address computational challenges wherever they might appear later in their careers. They learn that modern programming environments, such as the one provided by Java, help open the door to any computational problem they might encounter later, and they gain the confidence to learn, evaluate, and use other computational tools. Students interested in computer science will be well-prepared to pursue that interest; students in science and engineering will be ready to integrate computation into their studies.

Booksite An extensive amount of information that supplements this text may be found on the web at

```
http://www.cs.princeton.edu/IntroProgramming
```

For economy, we refer to this site as the *booksite* throughout. It contains material for instructors, students, and casual readers of the book. We briefly describe this material here, though, as all web users know, it is best surveyed by browsing. With a few exceptions to support testing, the material is all publicly available.

One of the most important implications of the booksite is that it empowers instructors and students to use their own computers to teach and learn the material. Anyone with a computer and a browser can begin learning to program by following a few instructions on the booksite. The process is no more difficult than downloading a media player or a song. As with any website, our booksite is continually evolving. It is an essential resource for everyone who owns this book. In particular, the supplemental materials are critical to our goal of making computer science an integral component of the education of all scientists and engineers.

For *instructors*, the booksite contains information about teaching. This information is primarily organized around a teaching style that we have developed over the past decade, where we offer two lectures per week to a large audience, supplemented by two class sessions per week where students meet in small groups with instructors or teaching assistants. The booksite has presentation slides for the lectures, which set the tone.

For *teaching assistants*, the booksite contains detailed problem sets and programming projects, which are based on exercises from the book but contain much more detail. Each programming assignment is intended to teach a relevant concept in the context of an interesting application while presenting an inviting and engaging challenge to each student. The progression of assignments embodies our approach to teaching programming. The booksite fully specifies all the assignments and provides detailed, structured information to help students complete them in the allotted time, including descriptions of suggested approaches and outlines for what should be taught in class sessions.

For *students*, the booksite contains quick access to much of the material in the book, including source code, plus extra material to encourage self-learning. Solutions are provided for many of the book's exercises, including complete program code and test data. There is a wealth of information associated with programming assignments, including suggested approaches, checklists, FAQs, and test data.

For *casual readers* (including instructors, teaching assistants, and students!), the booksite is a resource for accessing all manner of extra information associated with the book's content. All of the booksite content provides web links and other routes to pursue more information about the topic under consideration. There is far more information accessible than any individual could fully digest, but our goal is to provide enough to whet any reader's appetite for more information about the book's content.

Acknowledgements This project has been under development since 1992, so far too many people have contributed to its success for us to acknowledge them all here. Special thanks are due to Anne Rogers for helping to start the ball rolling; to Dave Hanson, Andrew Appel, and Chris van Wyk, for their patience in explaining data abstraction; and to Lisa Worthington, for being the first to truly relish the challenge of teaching this material to first-year students. We also gratefully acknowledge the efforts of `/dev/126` (the summer students who have contributed so much of the content); the faculty, graduate students, and teaching staff who have dedicated themselves to teaching this material over the past 15 years here at Princeton; and the thousands of undergraduates who have dedicated themselves to learning it.

Robert Sedgewick
Madeira, Portugal

Kevin Wayne
San Francisco, California

July, 2007

Contents

Preface . . . . . *v*

Elements of Programming *3*

1.1 Your First Program *4*
1.2 Built-in Types of Data *14*
1.3 Conditionals and Loops *46*
1.4 Arrays *86*
1.5 Input and Output *120*
1.6 Case Study: Random Web Surfer *162*

Functions and Modules *183*

2.1 Static Methods *184*
2.2 Libraries and Clients *218*
2.3 Recursion *254*
2.4 Case Study: Percolation *286*

Object-Oriented Programming *315*

3.1 Data Types *316*
3.2 Creating Data Types *370*
3.3 Designing Data Types *416*
3.4 Case Study: ***N***-body Simulation *456*

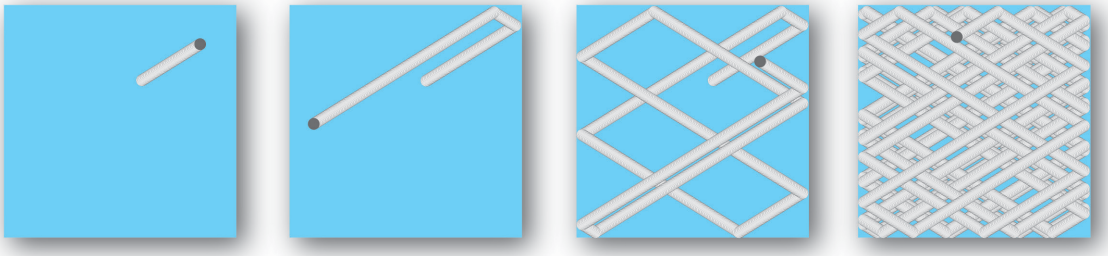
Algorithms and Data Structures *471*

4.1 Performance *472*
4.2 Sorting and Searching *510*
4.3 Stacks and Queues *550*
4.4 Symbol Tables *608*
4.5 Case Study: Small World *650*

Context *695*

Index *699*

Chapter One

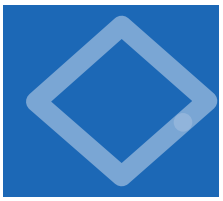


Elements of Programming

1.1 Your First Program 4
1.2 Built-in Types of Data 14
1.3 Conditionals and Loops 46
1.4 Arrays 86
1.5 Input and Output 120
1.6 Case Study: Random Web Surfer . . . 162

OUR GOAL IN THIS CHAPTER IS to convince you that writing a program is easier than writing a piece of text, such as a paragraph or essay. Writing prose is difficult: we spend many years in school to learn how to do it. By contrast, just a few building blocks suffice to enable us to write programs that can help solve all sorts of fascinating, but otherwise unapproachable, problems. In this chapter, we take you through these building blocks, get you started on programming in Java, and study a variety of interesting programs. You will be able to express yourself (by writing programs) within just a few weeks. Like the ability to write prose, the ability to program is a lifetime skill that you can continually refine well into the future.

In this book, you will learn the Java programming language. This task will be much easier for you than, for example, learning a foreign language. Indeed, programming languages are characterized by no more than a few dozen vocabulary words and rules of grammar. Much of the material that we cover in this book could be expressed in the C or C++ languages, or any of several other modern programming languages. But we describe everything specifically in Java so that you can get started creating and running programs right away. On the one hand, we will focus on learning to program, as opposed to learning details about Java. On the other hand, part of the challenge of programming is knowing which details are relevant in a given situation. Java is widely used, so learning to program in this language will enable you to write programs on many computers (your own, for example). Also, learning to program in Java will make it easy for you learn other languages, including lower-level languages such as C and specialized languages such as MATLAB.



1.1 Your First Program

IN THIS SECTION, OUR PLAN IS to lead you into the world of Java programming by taking you through the basic steps required to get a simple program running. The Java system is a collection of applications, not unlike many of the other applications that you are accustomed to using (such as your word processor, email program, and internet browser). As with any application, you need to be sure that Java is properly installed on your computer. It comes preloaded on many computers, or you can download it easily. You also need a text editor and a terminal application. Your first task is to find the instructions for installing such a Java programming environment on *your* computer by visiting

1.1.1 Hello, World 6
1.1.2 Using a command-line argument . . 8
Programs in this section

```
http://www.cs.princeton.edu/IntroProgramming
```

We refer to this site as the *booksite*. It contains an extensive amount of supplementary information about the material in this book for your reference and use. You will find it useful to have your browser open to this site while programming.

Programming in Java To introduce you to developing Java programs, we break the process down into three steps. To program in Java, you need to:

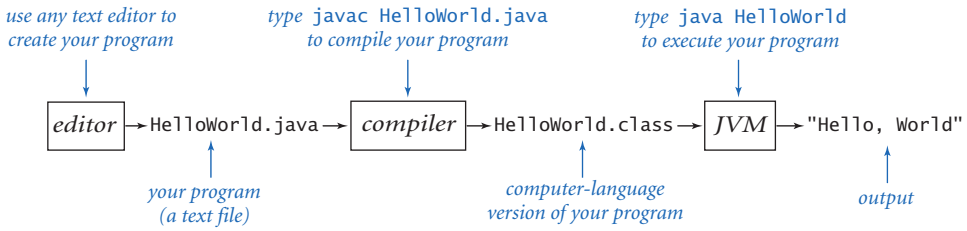
- *Create* a program by typing it into a file named, say, `MyCode.java`.
- *Compile* it by typing `javac MyCode.java` in a terminal window.
- *Run* (or *execute*) it by typing `java MyCode` in the terminal window.

In the first step, you start with a blank screen and end with a sequence of typed characters on the screen, just as when you write an email message or a paper. Programmers use the term *code* to refer to program text and the term *coding* to refer to the act of creating and editing the code. In the second step, you use a system application that *compiles* your program (translates it into a form more suitable for the computer) and puts the result in a file named `MyCode.class`. In the third step, you transfer control of the computer from the system to your program (which returns control back to the system when finished). Many systems have several different ways to create, compile, and execute programs. We choose the sequence described here because it is the simplest to describe and use for simple programs.

Creating a program. A Java program is nothing more than a sequence of characters, like a paragraph or a poem, stored in a file with a `.java` extension. To create one, therefore, you need only define that sequence of characters, in the same way as you do for email or any other computer application. You can use any *text editor* for this task, or you can use one of the more sophisticated program development environments described on the booksite. Such environments are overkill for the sorts of programs we consider in this book, but they are not difficult to use, have many useful features, and are widely used by professionals.

Compiling a program. At first, it might seem that Java is designed to be best understood by the computer. To the contrary, the language is designed to be best understood by the programmer (that's you). The computer's language is far more primitive than Java. A *compiler* is an application that translates a program from the Java language to a language more suitable for executing on the computer. The compiler takes a file with a `.java` extension as input (your program) and produces a file with the same name but with a `.class` extension (the computer-language version). To use your Java compiler, type in a terminal window the `javac` command followed by the file name of the program you want to compile.

Executing a program. Once you compile the program, you can run it. This is the exciting part, where your program takes control of your computer (within the constraints of what the Java system allows). It is perhaps more accurate to say that your computer follows your instructions. It is even more accurate to say that a part of the Java system known as the *Java Virtual Machine* (the *JVM*, for short) directs your computer to follow your instructions. To use the JVM to execute your program, type the `java` command followed by the program name in a terminal window.



Developing a Java program

Program 1.1.1 Hello, World

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

This code is a Java program that accomplishes a simple task. It is traditionally a beginner's first program. The box below shows what happens when you compile and execute the program. The terminal application gives a command prompt (% in this book) and executes the commands that you type (`javac` *and then* `java` *in the example below). The result in this case is that the program prints a message in the terminal window (the third line).*

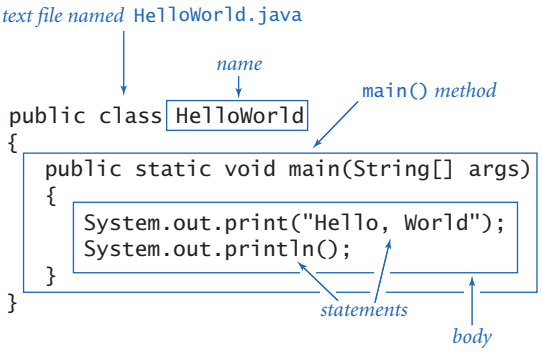
```
% javac HelloWorld.java
% java HelloWorld
Hello, World
```

PROGRAM 1.1.1 IS AN EXAMPLE OF a complete Java program. Its name is `HelloWorld`, which means that its code resides in a file named `HelloWorld.java` (by convention in Java). The program's sole action is to print a message back to the terminal window. For continuity, we will use some standard Java terms to describe the program, but we will not define them until later in the book: PROGRAM 1.1.1 consists of a single *class* named `HelloWorld` that has a single *method* named `main()`. This method uses two other methods named `System.out.print()` and `System.out.println()` to do the job. (When referring to a method in the text, we use `()` after the name to distinguish it from other kinds of names.) Until SECTION 2.1, where we learn about classes that define multiple methods, all of our classes will have this same structure. For the time being, you can think of "class" as meaning "program."

The first line of a method specifies its name and other information; the rest is a sequence of *statements* enclosed in braces and each followed by a semicolon. For the time being, you can think of "programming" as meaning "specifying a class

name and a sequence of statements for its `main()` method." In the next two sections, you will learn many different kinds of statements that you can use to make programs. For the moment, we will just use statements for printing to the terminal like the ones in `HelloWorld`.

When you type `java` followed by a class name in your terminal application, the system calls the `main()` method that you defined in that class, and executes its statements in order, one by one. Thus, typing `java HelloWorld` causes the system to call on the `main()` method in PROGRAM 1.1.1 and execute its two statements. The first statement calls on `System.out.print()` to print in the terminal window the message between the quotation marks, and the second statement calls on `System.out.println()` to terminate the line.



Anatomy of a program

Since the 1970s, it has been a tradition that a beginning programmer's first program should print `"Hello, World"`. So, you should type the code in PROGRAM 1.1.1 into a file, compile it, and execute it. By doing so, you will be following in the footsteps of countless others who have learned how to program. Also, you will be checking that you have a usable editor and terminal application. At first, accomplishing the task of printing something out in a terminal window might not seem very interesting; upon reflection, however, you will see that one of the most basic functions that we need from a program is its ability to tell us what it is doing.

For the time being, all our program code will be just like PROGRAM 1.1.1, except with a different sequence of statements in `main()`. Thus, you do not need to start with a blank page to write a program. Instead, you can

- Copy `HelloWorld.java` into a new file having a new program name of your choice, followed by `.java`.
- Replace `HelloWorld` on the first line with the new program name.
- Replace the `System.out.print()` and `System.out.println()` statements with a different sequence of statements (each ending with a semicolon).

Your program is characterized by its sequence of statements and its name. Each Java program must reside in a file whose name matches the one after the word `class` on the first line, and it also must have a `.java` extension.

Program 1.1.2 Using a command-line argument

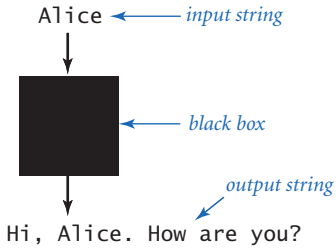
```
public class UseArgument
{
   public static void main(String[] args)
   {
      System.out.print("Hi, ");
      System.out.print(args[0]);
      System.out.println(". How are you?");
   }
}
```

This program shows the way in which we can control the actions of our programs: by providing an argument on the command line. Doing so allows us to tailor the behavior of our programs.

```
% javac UseArgument.java
% java UseArgument Alice
Hi, Alice. How are you?
% java UseArgument Bob
Hi, Bob. How are you?
```

Errors. It is easy to blur the distinction among editing, compiling, and executing programs. You should keep them separate in your mind when you are learning to program, to better understand the effects of the errors that inevitably arise. You can find several examples of errors in the Q&A at the end of this section. You can fix or avoid most errors by carefully examining the program as you create it, the same way you fix spelling and grammatical errors when you compose an email message. Some errors, known as *compile-time* errors, are caught when you compile the program, because they prevent the compiler from doing the translation. Other errors, known as *run-time* errors, do not show up until you execute the program. In general, errors in programs, also commonly known as *bugs*, are the bane of a programmer's existence: the error messages can be confusing or misleading, and the source of the error can be very hard to find. One of the first skills that you will learn is to identify errors; you will also learn to be sufficiently careful when coding, to avoid making many of them in the first place.

Input and Output Typically, we want to provide *input* to our programs: data that they can process to produce a result. The simplest way to provide input data is illustrated in `UseArgument` (PROGRAM 1.1.2). Whenever `UseArgument` is executed, it reads the *command-line argument* that you type after the program name and prints it back out to the terminal as part of the message. The result of executing this program depends on what we type after the program name. After compiling the program once, we can run it for different command-line arguments and get different printed results. We will discuss in more detail the mechanism that we use to pass arguments to our programs later, in SECTION 2.1. In the meantime, you can use `args[0]` within your program's body to represent the string that you type on the command line when it is executed, just as in `UseArgument`.



A bird's-eye view of a Java program

Again, accomplishing the task of getting a program to write back out what we type in to it may not seem interesting at first, but upon reflection you will realize that another basic function of a program is its ability to respond to basic information from the user to control what the program does. The simple model that `UseArgument` represents will suffice to allow us to consider Java's basic programming mechanism and to address all sorts of interesting computational problems.

Stepping back, we can see that `UseArgument` does neither more nor less than implement a function that maps a string of characters (the argument) into another string of characters (the message printed back to the terminal). When using it, we might think of our Java program as a black box that converts our input string to some output string. This model is attractive because it is not only simple but also sufficiently general to allow completion, in principle, of any computational task. For example, the Java compiler itself is nothing more than a program that takes one string of characters as input (a `.java` file) and produces another string of characters as output (the corresponding `.class` file). Later, we will be able to write programs that accomplish a variety of interesting tasks (though we stop short of programs as complicated as a compiler). For the moment, we live with various limitations on the size and type of the input and output to our programs; in SECTION 1.5, we will see how to incorporate more sophisticated mechanisms for program input and output. In particular, we can work with arbitrarily long input and output strings and other types of data such as sound and pictures.

Q&A

Q. Why Java?

A. The programs that we are writing are very similar to their counterparts in several other languages, so our choice of language is not crucial. We use Java because it is widely available, embraces a full set of modern abstractions, and has a variety of automatic checks for mistakes in programs, so it is suitable for learning to program. There is no perfect language, and you certainly will be programming in other languages in the future.

Q. Do I really have to type in the programs in the book to try them out? I believe that you ran them and that they produce the indicated output.

A. Everyone should type in and run `HelloWorld`. Your understanding will be greatly magnified if you also run `UseArgument`, try it on various inputs, and modify it to test different ideas of your own. To save some typing, you can find all of the code in this book (and much more) on the booksite. This site also has information about installing and running Java on your computer, answers to selected exercises, web links, and other extra information that you may find useful or interesting.

Q. What is the meaning of the words `public`, `static` and `void`?

A. These keywords specify certain properties of `main()` that you will learn about later in the book. For the moment, we just include these keywords in the code (because they are required) but do not refer to them in the text.

Q. What is the meaning of the `//`, `/*`, and `*/` character sequences in the code?

A. They denote *comments*, which are ignored by the compiler. A comment is either text in between `/*` and `*/` or at the end of a line after `//`. As with most online code, the code on the booksite is liberally annotated with comments that explain what it does; we use fewer comments in code in this book because the accompanying text and figures provide the explanation.

Q. What are Java's rules regarding tabs, spaces, and newline characters?

A. Such characters are known as *whitespace* characters. Java compilers consider all whitespace in program text to be equivalent. For example, we could write `Hel-`

loWorld as follows:

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

But we do normally adhere to spacing and indenting conventions when we write Java programs, just as we always indent paragraphs and lines consistently when we write prose or poetry.

Q. What are the rules regarding quotation marks?

A. Material inside quotation marks is an exception to the rule defined in the previous question: things within quotes are taken literally so that you can precisely specify what gets printed. If you put any number of successive spaces within the quotes, you get that number of spaces in the output. If you accidentally omit a quotation mark, the compiler may get very confused, because it needs that mark to distinguish between characters in the string and other parts of the program.

Q. What happens when you omit a brace or misspell one of the words, like `public` or `static` or `void` or `main`?

A. It depends upon precisely what you do. Such errors are called *syntax errors* and are usually caught by the compiler. For example, if you make a program `Bad` that is exactly the same as `HelloWorld` except that you omit the line containing the first left brace (and change the program name from `HelloWorld` to `Bad`), you get the following helpful message:

```
% javac Bad.java
Bad.java:2: '{' expected
   public static void main(String[] args)
   ^
1 error
```

From this message, you might correctly surmise that you need to insert a left brace. But the compiler may not be able to tell you exactly what mistake you made, so the error message may be hard to understand. For example, if you omit the second left brace instead of the first one, you get the following messages:

```
% javac Bad.java
Bad.java:4: ';' expected
      System.out.print("Hello, World");
      ^
Bad.java:7: 'class' or 'interface' expected
}
^
Bad.java:8: 'class' or 'interface' expected
^
3 errors
```

One way to get used to such messages is to intentionally introduce mistakes into a simple program and then see what happens. Whatever the error message says, you should treat the compiler as a friend, for it is just trying to tell you that something is wrong with your program.

Q. Can a program use more than one command-line argument?

A. Yes, you can use many arguments, though we normally use just a few. Note that the count starts at 0, so you refer to the first argument as `args[0]`, the second one as `args[1]`, the third one as `args[2]`, and so forth.

Q. What Java methods are available for me to use?

A. There are literally thousands of them. We introduce them to you in a deliberate fashion (starting in the next section) to avoid overwhelming you with choices.

Q. When I ran `UseArgument`, I got a strange error message. What's the problem?

A. Most likely, you forgot to include a command-line argument:

```
% java UseArgument
Hi, Exception in thread "main"
java.lang.ArrayIndexOutOfBoundsException: 0
        at UseArgument.main(UseArgument.java:6)
```

The JVM is complaining that you ran the program but did not type an argument as promised. You will learn more details about array indices in SECTION 1.4. Remember this error message: you are likely to see it again. Even experienced programmers forget to type arguments on occasion.

Exercises

1.1.1 Write a program that prints the `Hello, World` message 10 times.

1.1.2 Describe what happens if you omit the following in `HelloWorld.java`:

a. `public`
b. `static`
c. `void`
d. `args`

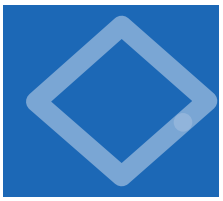
1.1.3 Describe what happens if you misspell (by, say, omitting the second letter) the following in `HelloWorld.java`:

a. `public`
b. `static`
c. `void`
d. `args`

1.1.4 Describe what happens if you try to execute `UseArgument` with each of the following command lines:

a. `java UseArgument java`
b. `java UseArgument @!&^%`
c. `java UseArgument 1234`
d. `java UseArgument.java Bob`
e. `java UseArgument Alice Bob`

1.1.5 Modify `UseArgument.java` to make a program `UseThree.java` that takes three names and prints out a proper sentence with the names in the reverse of the order given, so that, for example, `java UseThree Alice Bob Carol` gives `Hi Carol, Bob, and Alice`.



1.2 Built-in Types of Data

When programming in Java, you must always be aware of the type of data that your program is processing. The programs in Section 1.1 process strings of characters, many of the programs in this section process numbers, and we consider numerous other types later in the book. Understanding the distinctions among them is so important that we formally define the idea: a *data type* is a *set of values* and a *set of operations* defined on those values. You are familiar with various types of numbers, such as integers and real numbers, and with operations defined on them, such as addition and multiplication. In mathematics, we are accustomed to thinking of sets of numbers as being infinite; in computer programs we have to work with a finite number of possibilities. Each operation that we perform is well-defined *only* for the finite set of values in an associated data type.

1.2.1 String concatenation example . . . 20
1.2.2 Integer multiplication and division 22
1.2.3 Quadratic formula. 24
1.2.4 Leap year 27
1.2.5 Casting to get a random integer . . 33

Programs in this section

There are eight *primitive* types of data in Java, mostly for different kinds of numbers. Of the eight primitive types, we most often use these: `int` for integers; `double` for real numbers; and `boolean` for true-false values. There are other types of data available in Java libraries: for example, the programs in Section 1.1 use the type `String` for strings of characters. Java treats the `String` type differently from other types because its usage for input and output is essential. Accordingly, it shares some characteristics of the primitive types: for example, some of its operations are built in to the Java language. For clarity, we refer to primitive types and `String` collectively as *built-in* types. For the time being, we concentrate on programs that are based on computing with built-in types. Later, you will learn about Java library data types and building your own data types. Indeed, programming in Java is often centered on building data types, as you shall see in Chapter 3.

After defining basic terms, we consider several sample programs and code fragments that illustrate the use of different types of data. These code fragments do not do much real computing, but you will soon see similar code in longer programs. Understanding data types (values and operations on them) is an essential step in beginning to program. It sets the stage for us to begin working with more intricate programs in the next section. Every program that you write will use code like the tiny fragments shown in this section.

type	*set of values*	*common operators*	*sample literal values*
`int`	integers	`+ - * / %`	`99 -12 2147483647`
`double`	floating-point numbers	`+ - * /`	`3.14 -2.5 6.022e23`
`boolean`	boolean values	`&& || !`	`true false`
`char`	characters		`'A' '1' '%' '\n'`
`String`	sequences of characters	`+`	`"AB" Hello" "2.5"`

Basic built-in data types

Definitions To talk about data types, we need to introduce some terminology. To do so, we start with the following code fragment:

```
int a, b, c;
a = 1234;
b = 99;
c = a + b;
```

The first line is a *declaration* that declares the names of three *variables* to be the *identifiers* `a`, `b`, and `c` and their type to be `int`. The next three lines are *assignment statements* that change the values of the variables, using the *literals* `1234` and `99`, and the *expression* `a + b`, with the end result that `c` has the value `1333`.

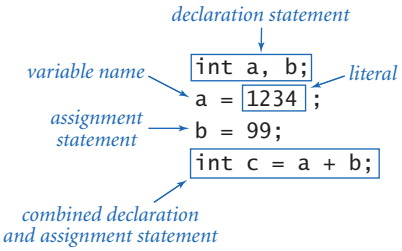
Identifiers. We use identifiers to name variables (and many other things) in Java. An identifier is a sequence of letters, digits, _, and $, the first of which is not a digit. The sequences of characters `abc`, `Ab$`, `abc123`, and `a_b` are all legal Java identifiers, but `Ab*`, `1abc`, and `a+b` are not. Identifiers are case-sensitive, so `Ab`, `ab`, and `AB` are all different names. You cannot use certain *reserved words*—such as `public`, `static`, `int`, `double`, and so forth—to name variables.

Literals. A literal is a source-code representation of a data-type value. We use strings of digits like `1234` or `99` to define `int` literal values, and add a decimal point as in `3.14159` or `2.71828` to define `double` literal values. To specify a `boolean` value, we use the keywords `true` or `false`, and to specify a `String`, we use a sequence of characters enclosed in quotes, such as `"Hello, World"`. We will consider other kinds of literals as we consider each data type in more detail.

Variables. A variable is a name that we use to refer to a data-type value. We use variables to keep track of changing values as a computation unfolds. For example,

we use the variable `n` in many programs to count things. We create a variable in a *declaration* that specifies its type and gives it a name. We compute with it by using the name in an *expression* that uses operations defined for its type. Each variable always stores one of the permissible data-type values.

Declaration statements. A declaration statement associates a variable name with a type at compile time. Java requires us to use declarations to specify the names and types of variables. By doing so, we are being explicit about any computation that we are specifying. Java is said to be a *strongly-typed* language, because the Java compiler can check for consistency at compile time (for example, it does not permit us to add a `String` to a `double`). This situation is precisely analogous to making sure that quantities have the proper units in a scientific application (for example, it does not make sense to add a quantity measured in inches to another measured in pounds). Declarations can appear anywhere before a variable is first used—most often, we put them *at* the point of first use.



Using a primitive data type

Assignment statements. An assignment statement associates a data-type value with a variable. When we write `c = a + b` in Java, we are not expressing mathematical equality, but are instead expressing an action: set the value of the variable `c` to be the value of `a` plus the value of `b`. It is true that `c` is mathematically equal to `a + b` immediately after the assignment statement has been executed, but the point of the statement is to change the value of `c` (if necessary). The left-hand side of an assignment statement must be a single variable; the right-hand side can be an arbitrary *expression* that produces values of the type. For example, we can say `discriminant = b*b - 4*a*c` in Java, but we cannot say `a + b = b + a` or `1 = a`. In short, *the meaning of = is decidedly not the same as in mathematical equations*. For example, `a = b` is certainly not the same as `b = a`, and while the value of `c` is the value of `a` plus the value of `b` after `c = a + b` has been executed, that may cease to be the case if subsequent statements change the values of any of the variables.

Initialization. In a simple declaration, the initial value of the variable is undefined. For economy, we can combine a declaration with an assignment statement to provide an initial value for the variable.

Tracing changes in variable values. As a final check on your understanding of the purpose of assignment statements, convince yourself that the following code *exchanges* the values of a and b (assume that a and b are `int` variables):

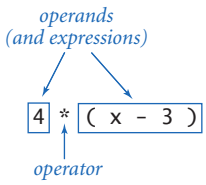
```
int t = a;
a = b;
b = t;
```

To do so, use a time-honored method of examining program behavior: study a table of the variable values after each statement (such a table is known as a *trace*).

	a	b	t
`int a, b;`	*undefined*	*undefined*	
`a = 1234;`	1234	*undefined*	
`b = 99;`	1234	99	
`int t = a;`	1234	99	1234
`a = b;`	99	99	1234
`b = t;`	99	1234	1234

Your first trace

Expressions. An expression is a literal, a variable, or a sequence of operations on literals and/or variables that produces a value. For primitive types, expressions look just like mathematical formulas, which are based on familiar symbols or *operators* that specify data-type operations to be performed on one or more *operands*. Each operand can be any expression. Most of the operators that we use are *binary operators* that take exactly two operands, such as `x + 1` or `y / 2`. An expression that is enclosed in parentheses is another expression with the same value. For example, we can write `4 * (x - 3)` or `4*x - 12` on the right-hand side of an assignment statement and the compiler will understand what we mean.



Anatomy of an expression

Precedence. Such expressions are shorthand for specifying a sequence of computations: in what order should they be performed? Java has natural and well-defined *precedence* rules (see the booksite) that fully specify this order. For arithmetic operations, multiplication and division are performed before addition and subtraction, so that `a-b*c` and `a-(b*c)` represent the same sequence of operations. When arithmetic operators have the same precedence, the order is determined by *left-associativity*, so that `a-b-c` and `(a-b)-c` represent the same sequence of operations. You can use parentheses to override the rules, so you should not need to worry about the details of precedence for most of the programs that you write. (Some of the programs that you *read* might depend subtly on precedence rules, but we avoid such programs in this book.)

Converting strings to primitive values for command-line arguments. Java provides the library methods that we need to convert the strings that we type as

command-line arguments into numeric values for primitive types. We use the Java library methods `Integer.parseInt()` and `Double.parseDouble()` for this purpose. For example, typing `Integer.parseInt("123")` in program text yields the literal value `123` (typing `123` has the same effect) and the code `Integer.parseInt(args[0])` produces the same result as the literal value typed as a string on the command line. You will see several examples of this usage in the programs in this section.

Converting primitive type values to strings for output. As mentioned at the beginning of this section, the Java built-in `String` type obeys special rules. One of these special rules is that you can easily convert any type of data to a `String`: whenever we use the + operator with a `String` as one of its operands, Java automatically converts the other to a `String`, producing as a result the `String` formed from the characters of the first operand followed by the characters of the second operand. For example, the result of these two code fragments

```
String a = "1234";
String b = "99";
String c = a + b;
```

```
String a = "1234";
int b = 99;
String c = a + b;
```

are both the same: they assign to `c` the value `"123499"`. We use this automatic conversion liberally to form `String` values for `System.out.print()` and `System.out.println()` for output. For example, we can write statements like this one:

```
System.out.println(a + " + " + b + " = " + c);
```

If `a`, `b`, and `c` are `int` variables with the values `1234`, `99`, and `1333`, respectively, then this statement prints out the string `1234 + 99 = 1333`.

With these mechanisms, our view of each Java program as a black box that takes string arguments and produces string results is still valid, but we can now interpret those strings as numbers and use them as the basis for meaningful computation. Next, we consider these details for the basic built-in types that you will use most often (strings, integers, floating-point numbers, and true–false values), along with sample code illustrating their use. To understand how to use a data type, you need to know not just its defined set of values, but also which operations you can perform, the language mechanism for invoking the operations, and the conventions for specifying literal values.

Characters and Strings A `char` is an alphanumeric character or symbol, like the ones that you type. There are 2^{16} different possible character values, but we usually restrict attention to the ones that represent letters, numbers, symbols, and whitespace characters such as tab and newline. Literals for `char` are characters enclosed in single quotes; for example, `'a'` represents the letter a. For tab, newline, backslash, single quote and double quote, we use the special *escape sequences* `'\t'`, `'\n'`, `'\\'`, `'\''`, and `'\"'`, respectively. The characters are encoded as 16-bit integers using an encoding scheme known as Unicode, and there are escape sequences for specifying special characters not found on your keyboard (see the booksite). We usually do not perform any operations directly on characters other than assigning values to variables.

values	sequences of characters
typical literals	`"Hello," "1 " " * "`
operation	concatenate
operator	+

Java's built-in `String` *data type*

A `String` is a sequence of characters. A literal `String` is a sequence of characters within double quotes, such as `"Hello, World"`. The `String` data type is *not* a primitive type, but Java sometimes treats it like one. For example, the *concatenation* operator (+) that we just considered is built in to the language as a binary operator in the same way as familiar operations on numbers.

The concatenation operation (along with the ability to declare `String` variables and to use them in expressions and assignment statements) is sufficiently powerful to allow us to attack some nontrivial computing tasks. As an example, `Ruler` (PROGRAM 1.2.1) computes a table of values of the *ruler function* that describes the relative lengths of the marks on a ruler. One noteworthy feature of this computation is that it illustrates how easy is is to craft short programs that produce huge amounts of output. If you extend this program in the obvious way to print five lines, six lines, seven lines, and so forth, you will see that each time you add just two statements to this program, you increase the size of its output by precisely one more than a factor of two. Specifically, if the program prints n lines, the nth line contains 2^n-1 numbers. For example, if you were to add statements in this way so that the program prints 30 lines, it would attempt to print more than 1 *billion* numbers.

expression	*value*
`"Hi, " + "Bob"`	`"Hi, Bob"`
`"1" + " 2 " + "1"`	`"1 2 1"`
`"1234" + " + " + "99"`	`"1234 + 99"`
`"1234" + "99"`	`"123499"`

Typical `String` *expressions*

Program 1.2.1 String concatenation example

```
public class Ruler
{
   public static void main(String[] args)
   {
      String ruler1 = "1";
      String ruler2 = ruler1 + " 2 " + ruler1;
      String ruler3 = ruler2 + " 3 " + ruler2;
      String ruler4 = ruler3 + " 4 " + ruler3;
      System.out.println(ruler1);
      System.out.println(ruler2);
      System.out.println(ruler3);
      System.out.println(ruler4);
   }
}
```

This program prints the relative lengths of the subdivisions on a ruler. The nth line of output is the relative lengths of the marks on a ruler subdivided in intervals of $1/2^n$ of an inch. For example, the fourth line of output gives the relative lengths of the marks that indicate intervals of one-sixteenth of an inch on a ruler.

```
% javac Ruler.java
% java Ruler
1
1 2 1
1 2 1 3 1 2 1
1 2 1 3 1 2 1 4 1 2 1 3 1 2 1
```

The ruler function for n = 4

As just discussed, our most frequent use (by far) of the concatenation operation is to put together results of computation for output with `System.out.print()` and `System.out.println()`. For example, we could simplify `UseArgument` (PROGRAM 1.1.2) by replacing its three statements with this single statement:

```
System.out.println("Hi, " + args[0] + ". How are you?");
```

We have considered the `String` type first precisely because we need it for output (and command-line input) in programs that process other types of data.

Integers An `int` is an integer (natural number) between `-2147483648` (-2^{31}) and `2147483647` ($2^{31}-1$). These bounds derive from the fact that integers are represented in binary with 32 binary digits: there are 2^{32} possible values. (The term *binary digit* is omnipresent in computer science, and we nearly always use the abbreviation *bit*: a bit is either 0 or 1.) The range of possible `int` values is asymmetric because zero is included with the positive values. See the booksite for more details about number representation, but in the present context it suffices to know that an `int` is one of the finite set of values in the range just given. Sequences of the characters `0` through `9`, possibly with a plus or minus sign at the beginning (that, when interpreted as decimal numbers, fall within the defined range), are integer literal values. We use `int`s frequently because they naturally arise when implementing programs.

expression	*value*	*comment*
`5 + 3`	`8`	
`5 - 3`	`2`	
`5 * 3`	`15`	
`5 / 3`	`1`	no fractional part
`5 % 3`	`2`	remainder
`1 / 0`		run-time error
`3 * 5 - 2`	`13`	`*` has precedence
`3 + 5 / 2`	`5`	`/` has precedence
`3 - 5 - 2`	`-4`	left associative
`( 3 - 5 ) - 2`	`-4`	better style
`3 - ( 5 - 2 )`	`0`	unambiguous

Typical `int` *expressions*

Standard arithmetic operators for addition/subtraction (`+` and `-`), multiplication (`*`), division (`/`), and remainder (`%`) for the `int` data type are built in to Java. These operators take two `int` operands and produce an `int` result, with one significant exception—division or remainder by zero is not allowed. These operations are defined just as in grade school (keeping in mind that all results must be integers): given two `int` values `a` and `b`, the value of `a / b` is the number of times `b` goes into `a` *with the fractional part discarded*, and the value of `a % b` is the remainder that you get when you divide `a` by `b`. For example, the value of `17 / 3` is `5`, and the value of `17 % 3` is `2`. The `int` results that we get from arithmetic operations are just what we expect, except that if the result is too large to fit into `int`'s 32-bit representation, then it will be truncated in a well-defined manner. This situation is known as *overflow*. In

values	integers between -2^{31} and $+2^{31}-1$				
typical literals	`1234`	`99`	`-99`	`0`	`1000000`
operations	add	subtract	multiply	divide	remainder
operators	`+`	`-`	`*`	`/`	`%`

Java's built-in `int` *data type*

Program 1.2.2 Integer multiplication and division

```
public class IntOps
{
   public static void main(String[] args)
   {
      int a = Integer.parseInt(args[0]);
      int b = Integer.parseInt(args[1]);
      int p = a * b;
      int q = a / b;
      int r = a % b;
      System.out.println(a + " * " + b + " = " + p);
      System.out.println(a + " / " + b + " = " + q);
      System.out.println(a + " % " + b + " = " + r);
      System.out.println(a + " = " + q + " * " + b + " + " + r);
   }
}
```

Arithmetic for integers is built in to Java. Most of this code is devoted to the task of getting the values in and out; the actual arithmetic is in the simple statements in the middle of the program that assign values to `p`, `q`, *and* `r`.

```
% javac IntOps.java
% java IntOps 1234 99
1234 * 99 = 122166
1234 / 99 = 12
1234 % 99 = 46
1234 = 12 * 99 + 46
```

general, we have to take care that such a result is not misinterpreted by our code. For the moment, we will be computing with small numbers, so you do not have to worry about these boundary conditions.

PROGRAM 1.2.2 illustrates basic operations for manipulating integers, such as the use of expressions involving arithmetic operators. It also demonstrates the use of `Integer.parseInt()` to convert `String` values on the command line to `int` values, as well as the use of automatic type conversion to convert `int` values to `String` values for output.

Three other built-in types are different representations of integers in Java. The `long`, `short`, and `byte` types are the same as `int` except that they use 64, 16, and 8 bits respectively, so the range of allowed values is accordingly different. Programmers use `long` when working with huge integers, and the other types to save space. You can find a table with the maximum and minimum values for each type on the booksite, or you can figure them out for yourself from the numbers of bits.

Floating-point numbers The `double` type is for representing *floating-point* numbers, for use in scientific and commercial applications. The internal representation is like scientific notation, so that we can compute with numbers in a huge range. We use floating-point numbers to represent real numbers, but they are decidedly not the same as real numbers! There are infinitely many real numbers, but we can only represent a finite number of floating-points in any digital computer representation. Floating-point numbers do approximate real numbers sufficiently well that we can use them in applications, but we often need to cope with the fact that we cannot always do exact computations.

expression	*value*
`3.141 + .03`	`3.171`
`3.141 - .03`	`3.111`
`6.02e23 / 2.0`	`3.01e23`
`5.0 / 3.0`	`1.6666666666666667`
`10.0 % 3.141`	`0.577`
`1.0 / 0.0`	`Infinity`
`Math.sqrt(2.0)`	`1.4142135623730951`
`Math.sqrt(-1.0)`	`NaN`

Typical `double` *expressions*

We can use a sequence of digits with a decimal point to type floating-point numbers. For example, `3.14159` represents a six-digit approximation to π. Alternatively, we can use a notation like scientific notation: the literal `6.022e23` represents the number 6.022×10^{23}. As with integers, you can use these conventions to write floating-point literals in your programs or to provide floating-point numbers as string parameters on the command line.

The arithmetic operators `+`, `-`, `*`, and `/` are defined for `double`. Beyond the built-in operators, the Java `Math` library defines the square root, trigonometric

values	real numbers (specified by IEEE 754 standard)			
typical literals	`3.14159`	`6.022e23`	`-3.0` `2.0`	`1.4142135623730951`
operations	add	subtract	multiply	divide
operators	`+`	`-`	`*`	`/`

Java's built-in `double` *data type*

Program 1.2.3 Quadratic formula

```
public class Quadratic
{
   public static void main(String[] args)
   {
      double b = Double.parseDouble(args[0]);
      double c = Double.parseDouble(args[1]);
      double discriminant = b*b - 4.0*c;
      double d = Math.sqrt(discriminant);
      System.out.println((-b + d) / 2.0);
      System.out.println((-b - d) / 2.0);
   }
}
```

This program prints out the roots of the polynomial $x^2 + bx + c$, using the quadratic formula. For example, the roots of $x^2 - 3x + 2$ are 1 and 2 since we can factor the equation as $(x - 1)(x - 2)$; the roots of $x^2 - x - 1$ are ϕ and $1 - \phi$, where ϕ is the golden ratio, and the roots of $x^2 + x + 1$ are not real numbers.

```
% javac Quadratic.java
% java Quadratic -3.0 2.0
2.0
1.0
```

```
% java Quadratic -1.0 -1.0
1.618033988749895
-0.6180339887498949

% java Quadratic 1.0 1.0
NaN
NaN
```

functions, logarithm/exponential functions, and other common functions for floating-point numbers. To use one of these values in an expression, we write the name of the function followed by its argument in parentheses. For example, you can use the code `Math.sqrt(2.0)` when you want to use the square root of 2 in an expression. We discuss in more detail the mechanism behind this arrangement in SECTION 2.1 and more details about the `Math` library at the end of this section.

When working with floating point numbers, one of the first things that you will encounter is the issue of *precision*: `5.0/2.0` is `2.5` but `5.0/3.0` is `1.6666666666666667`. In SECTION 1.5, you will learn Java's mechanism for control-

ling the number of significant digits that you see in output. Until then, we will work with the Java default output format.

The result of a calculation can be one of the special values `Infinity` (if the number is too large to be represented) or `NaN` (if the result of the calculation is undefined). Though there are myriad details to consider when calculations involve these values, you can use `double` in a natural way and begin to write Java programs instead of using a calculator for all kinds of calculations. For example, PROGRAM 1.2.3 shows the use of `double` values in computing the roots of a quadratic equation using the quadratic formula. Several of the exercises at the end of this section further illustrate this point.

As with `long`, `short`, and `byte` for integers, there is another representation for real numbers called `float`. Programmers sometimes use `float` to save space when precision is a secondary consideration. The `double` type is useful for about 15 significant digits; the `float` type is good for only about 7 digits. We do not use `float` in this book.

Booleans The `boolean` type has just two values: `true` and `false`. These are the two possible `boolean` literals. Every `boolean` variable has one of these two values, and every `boolean` operation has operands and a result that takes on just one of these two values. This simplicity is deceiving—`boolean` values lie at the foundation of computer science.

values	true or false		
literals	`true` `false`		
operations	and	or	not
operators	`&&`	`||`	`!`

Java's built-in `boolean` *data type*

The most important operations defined for `boolean`s are *and* (`&&`), *or* (`||`), and *not* (`!`), which have familiar definitions:

- `a && b` is `true` if both operands are `true`, and `false` if either is `false`.
- `a || b` is `false` if both operands are `false`, and `true` if either is `true`.
- `!a` is `true` if `a` is `false`, and `false` if `a` is `true`.

Despite the intuitive nature of these definitions, it is worthwhile to fully specify each possibility for each operation in tables known as *truth tables*. The *not* function

a	!a
true	false
false	true

a	b	a && b	a || b
false	false	false	false
false	true	false	true
true	false	false	true
true	true	true	true

Truth-table definitions of `boolean` *operations*

a	b	a && b	!a	!b	!a || !b	!(!a || !b)
false	false	false	true	true	true	false
false	true	false	true	false	true	false
true	false	false	false	true	true	false
true	true	true	false	false	false	true

Truth-table proof that `a && b` *and* `!(!a || !b)` *are identical*

has only one operand: its value for each of the two possible values of the operand is specified in the second column. The *and* and *or* functions each have two operands: there are four different possibilities for operand input values, and the values of the functions for each possibility are specified in the right two columns.

We can use these operators with parentheses to develop arbitrarily complex expressions, each of which specifies a well-defined boolean function. Often the same function appears in different guises. For example, the expressions `(a && b)` and `!(!a || !b)` are equivalent.

The study of manipulating expressions of this kind is known as *Boolean logic*. This field of mathematics is fundamental to computing: it plays an essential role in the design and operation of computer hardware itself, and it is also a starting point for the theoretical foundations of computation. In the present context, we are interested in `boolean` expressions because we use them to control the behavior of our programs. Typically, a particular condition of interest is specified as a boolean expression and a piece of program code is written to execute one set of statements if the expression is `true` and a different set of statements if the expression is `false`. The mechanics of doing so are the topic of SECTION 1.3.

Comparisons Some *mixed-type* operators take operands of one type and produce a result of another type. The most important operators of this kind are the comparison operators ==, !=, <, <=, >, and >=, which all are defined for each primitive numeric type and produce a `boolean` result. Since operations are defined only

non-negative discriminant?	`(b*b - 4.0*a*c) >= 0.0`
beginning of a century?	`(year % 100) == 0`
legal month?	`(month >= 1) && (month <= 12)`

Typical comparison expressions

Program 1.2.4 Leap year

```
public class LeapYear
{
   public static void main(String[] args)
   {
      int year = Integer.parseInt(args[0]);
      boolean isLeapYear;
      isLeapYear = (year % 4 == 0);
      isLeapYear = isLeapYear && (year % 100 != 0);
      isLeapYear = isLeapYear || (year % 400 == 0);
      System.out.println(isLeapYear);
   }
}
```

This program tests whether an integer corresponds to a leap year in the Gregorian calendar. A year is a leap year if it is divisible by 4 (2004), unless it is divisible by 100 in which case it is not (1900), unless it is divisible by 400 in which case it is (2000).

```
% javac LeapYear.java
% java LeapYear 2004
true
% java LeapYear 1900
false
% java LeapYear 2000
true
```

with respect to data types, each of these symbols stands for many operations, one for each data type. It is required that both operands be of the same type. The result is always `boolean`.

Even without going into the details of number representation, it is clear that the operations for the various types are really quite different: for example, it is one thing to compare two `int`s to check that `(2 <= 2)` is `true` but quite another to compare two `double`s to check whether `(2.0 <= 0.002e3)` is `true` or `false`. Still, these operations are well-defined and useful to write code that tests for conditions such as `(b*b - 4.0*a*c) >= 0.0`, which is frequently needed, as you will see.

The comparison operations have lower precedence than arithmetic operators and higher precedence than boolean operators, so you do not need the parentheses in an expression like `(b*b - 4.0*a*c) >= 0.0`, and you could write an expression like `month >= 1 && month <= 12` without parentheses to test whether the value of the `int` variable `month` is between 1 and 12. (It is better style to use the parentheses, however.)

op	*meaning*	`true`	`false`
`==`	*equal*	`2 == 2`	`2 == 3`
`!=`	*not equal*	`3 != 2`	`2 != 2`
`<`	*less than*	`2 < 13`	`2 < 2`
`<=`	*less than or equal*	`2 <= 2`	`3 <= 2`
`>`	*greater than*	`13 > 2`	`2 > 13`
`>=`	*greater than or equal*	`3 >= 2`	`2 >= 3`

Comparisons with `int` operands and a `boolean` result

Comparison operations, together with boolean logic, provide the basis for decision-making in Java programs. PROGRAM 1.2.4 is an example of their use, and you can find other examples in the exercises at the end of this section. More importantly, in SECTION 1.3 we will see the role that boolean expressions play in more sophisticated programs.

Library methods and APIs As we have seen, many programming tasks involve using Java library methods in addition to the built-in operations on data-type values. The number of available library methods is vast. As you learn to program, you will learn to use more and more library methods, but it is best at the beginning to restrict your attention to a relatively small set of methods. In this chapter, you have already used some of Java's methods for printing, for converting data from one type to another, and for computing mathematical functions (the Java `Math` library). In later chapters, you will learn not just how to use other methods, but how to create and use your own methods.

For convenience, we will consistently summarize the library methods that you need to know how to use in tables like this one:

`public class System.out`	
`void print(String s)`	*print* `s`
`void println(String s)`	*print* `s`, *followed by a newline*
`void println()`	*print a newline*

Note: Any type of data can be used (and will be automatically converted to `String`).

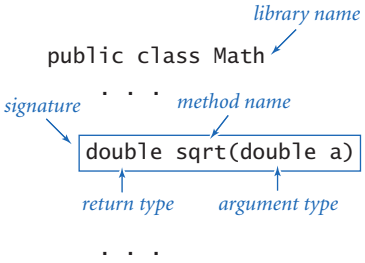
Excerpts from Java's library for standard output

Such a table is known as an *application programming interface* (*API*). It provides the information that you need to write an *application program* that uses the methods. Here is an API for the most commonly used methods in Java's `Math` library:

`public class Math`	
`double abs(double a)`	*absolute value of a*
`double max(double a, double b)`	*maximum of a and b*
`double min(double a, double b)`	*minimum of a and b*
Note 1: `abs()`, `max()`, *and* `min()` *are defined also for* `int`, `long`, *and* `float`.	
`double sin(double theta)`	*sine function*
`double cos(double theta)`	*cosine function*
`double tan(double theta)`	*tangent function*
Note 2: Angles are expressed in radians. Use `toDegrees()` *and* `toRadians()` *to convert.*	
Note 3: Use `asin()`, `acos()`, *and* `atan()` *for inverse functions.*	
`double exp(double a)`	*exponential* (e^a)
`double log(double a)`	*natural log* ($\log_e a$, *or* $\ln a$)
`double pow(double a, double b)`	*raise a to the bth power* (a^b)
`long round(double a)`	*round to the nearest integer*
`double random()`	*random number in* [0, 1)
`double sqrt(double a)`	*square root of a*
`double E`	*value of e (constant)*
`double PI`	*value of* π *(constant)*
See booksite for other available functions.	

Excerpts from Java's mathematics library

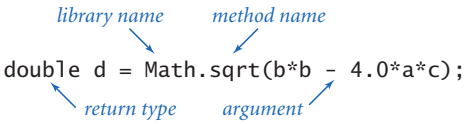
With the exception of `random()`, these methods implement mathematical functions—they use their arguments to compute a value of a specified type. Each method is described by a line in the API that specifies the information you need to know in order to use the method. The code in the tables is *not* the code that you type to use the method; it is known as the method's *signature*. The signature specifies the type of the arguments, the method name, and the type of the value that the method computes (the *return value*). When your program is executed, we say that it *calls* the system library code for the method, which *returns* the value for use in your code.



Anatomy of a method signature

Note that `random()` does not implement a mathematical function because it does not take an argument. On the other hand, `System.out.print()` and `System.out.println()` do not implement mathematical functions because they do not return values and therefore do not have a return type. (This condition is specified in the signature by the keyword `void`.)

In your code, you can use a library method by typing its name followed by arguments of the specified type, enclosed in parentheses and separated by commas. You can use this code in the same way as you use variables and literals in expressions. When you do so, you can expect that method to compute a value of the appropriate type, as documented in the left column of the API. For example, you can write expressions like `Math.sin(x) * Math.cos(y)` and so on. Method arguments may also be expressions, as in `Math.sqrt(b*b - 4.0*a*c)`.



Using a library method

The `Math` library also defines the precise constant values `PI` (for π) and `E` (for *e*), so that you can use those names to refer to those constants in your programs. For example, the value of `Math.sin(Math.PI/2)` is `1.0` and the value of `Math.log(Math.E)` is `1.0` (because `Math.sin()` takes its argument in radians and `Math.log()` implements the natural logarithm function).

To be complete, we also include here the following API for Java's conversion methods, which we use for command-line arguments:

`int`	`Integer.parseInt(String s)`	*convert* `s` *to an* `int` *value*
`double`	`Double.parseDouble(String s)`	*convert* `s` *to a* `double` *value*
`long`	`Long.parseLong(String s)`	*convert* `s` *to a* `long` *value*

Java library methods for converting strings to primitive types

You *do not* need to use methods like these to convert from `int`, `double`, and `long` values to `String` values for *output*, because Java automatically converts any value used as an argument to `System.out.print()` or `System.out.println()` to `String` for output.

expression	*library*	*type*	*value*
`Integer.parseInt("123")`	`Integer`	`int`	`123`
`Math.sqrt(5.0*5.0 - 4.0*4.0)`	`Math`	`double`	`3.0`
`Math.random()`	`Math`	`double`	*random in* [0, 1)
`Math.round(3.14159)`	`Math`	`long`	`3`

Typical expressions that use Java library methods

These APIs are typical of the online documentation that is the standard in modern programming. There is extensive online documentation of the Java APIs that is used by professional programmers, and it is available to you (if you are interested) directly from the Java website or through our booksite. You do not need to go to the online documentation to understand the code in this book or to write similar code, because we present and explain in the text all of the library methods that we use in APIs like these and summarize them in the endpapers. More important, in CHAPTERS 2 AND 3 you will learn in this book how to develop your own APIs and to implement functions for your own use.

Type conversion One of the primary rules of modern programming is that you should always be aware of the type of data that your program is processing. Only by knowing the type can you know precisely which set of values each variable can have, which literals you can use, and which operations you can perform. Typical programming tasks involve processing multiple types of data, so we often need to convert data from one type to another. There are several ways to do so in Java.

Explicit type conversion. You can use a method that takes an argument of one type (the value to be converted) and produces a result of another type. We have already used the `Integer.parseInt()` and `Double.parseDouble()` library methods to convert `String` values to `int` and `double` values, respectively. Many other methods are available for conversion among other types. For example, the library method `Math.round()` takes a `double` argument and returns a `long` result: the nearest integer to the argument. Thus, for example, `Math.round(3.14159)` and `Math.round(2.71828)` are both of type `long` and have the same value (`3`).

Explicit cast. Java has some built-in type conversion conventions for primitive types that you can take advantage of when you are aware that you might lose infor-

mation. You have to make your intention to do so explicit by using a device called a *cast*. You cast an expression from one primitive type to another by prepending the desired type name within parentheses. For example, the expression `(int) 2.71828` is a cast from `double` to `int` that produces an `int` with value `2`. The conversion methods defined for casts throw away information in a reasonable way (for a full list, see the booksite). For example, casting a floating-point number to an integer discards the fractional part by rounding towards zero. If you want a different result, such as rounding to the nearest integer, you must use the explicit conversion method `Math.round()`, as just discussed (but you then need to use an explicit cast to `int`, since that method returns a `long`). `RandomInt` (PROGRAM 1.2.5) is an example that uses a cast for a practical computation.

Automatic promotion for numbers. You can use data of any primitive numeric type where a value whose type has a larger range of values is expected, because Java automatically converts to the type with the larger range. This kind of conversion is called *promotion*. For example, we used numbers all of type `double` in PROGRAM 1.2.3, so there is no conversion. If we had chosen to make `b` and `c` of type `int` (using `Integer.parseInt()` to convert the command-line arguments), automatic promotion would be used to evaluate the expression `b*b - 4.0*c`. First, `c` is promoted to `double` to multiply by the `double` literal `4.0`, with a `double` result. Then, the `int` value `b*b` is promoted to `double` for the subtraction, leaving a `double` result. Or, we might have written `b*b - 4*c`. In that case, the expression `b*b - 4*c` would be evaluated as an `int` and then the result promoted to `double`, because that is what `Math.sqrt()` expects. Promotion is appropriate because your intent is clear and it can be done with no loss of information. On the other hand, a conversion that might involve loss of information (for example, assigning a `double` to an `int`) leads to a compile-time error.

expression	*expression type*	*expression value*
`"1234" + 99`	`String`	`"123499"`
`Integer.parseInt("123")`	`int`	`123`
`(int) 2.71828`	`int`	`2`
`Math.round(2.71828)`	`long`	`3`
`(int) Math.round(2.71828)`	`int`	`3`
`(int) Math.round(3.14159)`	`int`	`3`
`11 * 0.3`	`double`	`3.3`
`(int) 11 * 0.3`	`double`	`3.3`
`11 * (int) 0.3`	`int`	`0`
`(int) (11 * 0.3)`	`int`	`3`

Typical type conversions

Program 1.2.5 Casting to get a random integer

```
public class RandomInt
{
   public static void main(String[] args)
   {
      int N = Integer.parseInt(args[0]);
      double r = Math.random();  // uniform between 0 and 1
      int n = (int) (r * N);     // uniform between 0 and N-1
      System.out.println(n);
   }
}
```

This program uses the Java method `Math.random()` *to generate a random number* `r` *in the interval [0, 1), then multiplies* `r` *by the command-line argument* `N` *to get a random number greater than or equal to* `0` *and less than* `N`*, then uses a cast to truncate the result to be an integer* `n` *between* `0` *and* `N-1`*.*

```
% javac RandomInt.java

% java RandomInt 1000
548

% java RandomInt 1000
141

% java RandomInt 1000000
135032
```

Casting has higher precedence than arithmetic operations—any cast is applied to the value that immediately follows it. For example, if we write `int n = (int) 11 * 0.3`, the cast is no help: the literal `11` is already an integer, so the cast `(int)` has no effect. In this example, the compiler produces a `possible loss of precision` error message because there would be a loss of precision in converting the resulting value (`3.3`) to an `int` for assignment to `n`. The error is helpful because the intended computation for this code is likely `(int) (11 * 0.3)`, which has the value `3`, not `3.3`.

BEGINNING PROGRAMMERS TEND TO FIND TYPE conversion to be an annoyance, but experienced programmers know that paying careful attention to data types is a key to success in programming. It is well worth your while to take the time to understand what type conversion is all about. After you have written just a few programs, you will understand that these rules help you to make your intentions explicit and to avoid subtle bugs in your programs.

Summary *A data type is a set of values and a set of operations on those values.* Java has eight primitive data types: `boolean`, `char`, `byte`, `short`, `int`, `long`, `float`, and `double`. In Java code, we use operators and expressions like those in familiar mathematical expressions to invoke the operations associated with each type. The `boolean` type is for computing with the logical values `true` and `false`; the `char` type is the set of character values that we type; and the other six are numeric types, for computing with numbers. In this book, we most often use `boolean`, `int`, and `double`; we do not use `short` or `float`. Another data type that we use frequently, `String`, is not primitive, but Java has some built-in facilities for `String`s that are like those for primitive types.

When programming in Java, we have to be aware that every operation is defined only in the context of its data type (so we may need type conversions) and that all types can have only a finite number of values (so we may need to live with imprecise results).

The `boolean` type and its operations— `&&`, `||`, and `!` —are the basis for logical decision-making in Java programs, when used in conjunction with the mixed-type comparison operators `==`, `!=`, `<`, `>`, `<=`, and `>=`. Specifically, we use `boolean` expressions to control Java's conditional (`if`) and loop (`for` and `while`) constructs, which we will study in detail in the next section.

The numeric types and Java's libraries give us the ability to use Java as an extensive mathematical calculator. We write arithmetic expressions using the built-in operators `+`, `-`, `*`, `/`, and `%` along with Java methods from the `Math` library. Although the programs in this section are quite rudimentary by the standards of what we will be able to do after the next section, this class of programs is quite useful in its own right. You will use primitive types and basic mathematical functions extensively in Java programming, so the effort that you spend now understanding them will certainly be worthwhile.

Q&A

Q. What happens if I forget to declare a variable?

A. The compiler complains, as shown below for a program `IntOpsBad`, which is the same as PROGRAM 1.2.2 except that the `int` variable `p` is omitted from the declaration statement.

```
% javac IntOpsBad.java
IntOpsBad.java:7: cannot resolve symbol
symbol : variable p
location: class IntOpsBad
p = a * b;
          ^
IntOpsBad.java:10: cannot resolve symbol
symbol : variable p
location: class IntOpsBad
System.out.println(a + " * " + b + " = " + p);
                                           ^
2 errors
```

The compiler says that there are two errors, but there is really just one: the declaration of `p` is missing. If you forget to declare a variable that you use often, you will get quite a few error messages. A good strategy is to correct the *first* error and check that correction before addressing later ones.

Q. What happens if I forget to initialize a variable?

A. The compiler checks for this condition and will give you a `variable might not have been initialized` error message if you try to use the variable in an expression.

Q. Is there a difference between = and == ?

A. Yes, they are quite different! The first is an assignment operator that changes the value of a variable, and the second is a comparison operator that produces a `boolean` result. Your ability to understand this answer is a sure test of whether you understood the material in this section. Think about how you might explain the difference to a friend.

Q. Why do `int` values sometime become negative when they get large?

A. If you have not experienced this phenomenon, see EXERCISE 1.2.10. The problem has to do with the way integers are represented in the computer. You can learn the details on the booksite. In the meantime, a safe strategy is using the `int` type when you know the values to be less than ten digits and the `long` type when you think the values might get to be ten digits or more.

Q. It seems wrong that Java should just let `int`s overflow and give bad values. Shouldn't Java automatically check for overflow?

A. Yes, this issue is a contentious one among programmers. The short answer for now is that the lack of such checking is one reason such types are called *primitive* data types. A little knowledge can go a long way in avoiding such problems. Again, it is fine to use the `int` type for small numbers, but when values run into the billions, you cannot.

Q. What is the value of `Math.abs(-2147483648)`?

A. `-2147483648`. This strange (but true) result is a typical example of the effects of integer overflow.

Q. It is annoying to see all those digits when printing a `float` or a `double`. Can we get `System.out.println()` to print out just two or three digits after the decimal point?

A. That sort of task involves a closer look at the method used to convert from `double` to `String`. The Java library function `System.out.printf()` is one way to do the job, and it is similar to the basic printing method in the C programming language and many modern languages, as discussed in SECTION 1.5. Until then, we will live with the extra digits (which is not all bad, since doing so helps us to get used to the different primitive types of numbers).

Q. How can I initialize a `double` variable to infinity?

A. Java has built-in constants available for this purpose: `Double.POSITIVE_INFINITY` and `Double.NEGATIVE_INFINITY`.

Q. What is the value of `Math.round(6.022e23)`?

A. You should get in the habit of typing in a tiny Java program to answer such questions yourself (and trying to understand why your program produces the result that it does).

Q. Can you compare a `double` to an `int`?

A. Not without doing a type conversion, but remember that Java usually does the requisite type conversion automatically. For example, if `x` is an `int` with the value `3`, then the expression `(x < 3.1)` is `true`—Java converts `x` to `double` (because `3.1` is a `double` literal) before performing the comparison.

Q. Are expressions like `1/0` and `1.0/0.0` legal in Java?

A. No and yes. The first generates a run-time *exception* for division by zero (which stops your program because the value is undefined); the second is legal and has the value `Infinity`.

Q. Are there functions in Java's `Math` library for other trigonometric functions, like cosecant, secant, and cotangent?

A. No, because you could use `Math.sin()`, `Math.cos()`, and `Math.tan()` to compute them. Choosing which functions to include in an API is a tradeoff between the convenience of having every function that you need and the annoyance of having to find one of the few that you need in a long list. No choice will satisfy all users, and the Java designers have many users to satisfy. Note that there are plenty of redundancies even in the APIs that we have listed. For example, you could use `Math.sin(x)/Math.cos(x)` instead of `Math.tan(x)`.

Q. Can you use < and > to compare `String` variables?

A. No. Those operators are defined only for primitive types.

Q. How about == and `!=`?

A. Yes, but the result may not be what you expect, because of the meanings these operators have for non-primitive types. For example, there is a distinction between

a `String` and its value. The expression `"abc" == "ab" + x` is `false` when `x` is a `String` with value `"c"` because the two operands are stored in different places in memory (even though they have the same value). This distinction is essential, as you will learn when we discuss it in more detail in SECTION 3.1.

Q. What is the result of division and remainder for negative integers?

A. The quotient `a / b` rounds toward 0; the remainder `a % b` is defined such that `(a / b) * b + a % b` is always equal to `a`. For example, `-14/3` and `14/-3` are both `-4`, but `-14 % 3` is `-2` and `14 % -3` is `2`.

Q. Will `(a < b < c)` test whether three numbers are in order?

A. No, that will not compile. You need to say `(a < b && b < c)`.

Q. Fifteen digits for floating-point numbers certainly seems enough to me. Do I really need to worry much about precision?

A. Yes, because you are used to mathematics based on real numbers with infinite precision, whereas the computer always deals with approximations. For example, `(0.1 + 0.1 == 0.2)` is `true` but `(0.1 + 0.1 + 0.1 == 0.3)` is `false`! Pitfalls like this are not at all unusual in scientific computing. Novice programmers should avoid comparing two floating-point numbers for equality.

Q. Why do we say `(a && b)` and not `(a & b)`?

A. Java also has a `&` operator that we do not use in this book but which you may encounter if you pursue advanced programming courses.

Q. Why is the value of `10^6` not `1000000` but `12`?

A. The `^` operator is not an exponentiation operator, which you must have been thinking. Instead, it is an operator like `&` that we do not use in this book. You want the literal `1e6`. You could also use `Math.pow(10, 6)` but doing so is wasteful if you are raising `10` to a known power.

Exercises

1.2.1 Suppose that `a` and `b` are `int` values. What does the following sequence of statements do?

```
int t = a; b = t; a = b;
```

1.2.2 Write a program that uses `Math.sin()` and `Math.cos()` to check that the value of $\cos^2\theta + \sin^2\theta$ is approximately 1 for any θ entered as a command-line argument. Just print the value. Why are the values not always exactly 1?

1.2.3 Suppose that `a` and `b` are `int` values. Show that the expression

```
(!(a && b) && (a || b)) || ((a && b) || !(a || b))
```

is equivalent to `true`.

1.2.4 Suppose that `a` and `b` are `int` values. Simplify the following expression: `(!(a < b) && !(a > b))`.

1.2.5 The *exclusive or* operator `^` for `boolean` operands is defined to be `true` if they are different, `false` if they are the same. Give a truth table for this function.

1.2.6 Why does `10/3` give `3` and not `3.333333333`?

Solution. Since both `10` and `3` are integer literals, Java sees no need for type conversion and uses integer division. You should write `10.0/3.0` if you mean the numbers to be `double` literals. If you write `10/3.0` or `10.0/3`, Java does implicit conversion to get the same result.

1.2.7 What do each of the following print?

a.
```
System.out.println(2 + "bc");
```

b.
```
System.out.println(2 + 3 + "bc");
```

c.
```
System.out.println((2+3) + "bc");
```

d.
```
System.out.println("bc" + (2+3));
```

e.
```
System.out.println("bc" + 2 + 3);
```

Explain each outcome.

1.2.8 Explain how to use PROGRAM 1.2.3 to find the square root of a number.

1.2.9 What do each of the following print?

a. `System.out.println('b');`

b. `System.out.println('b' + 'c');`

c. `System.out.println((char) ('a' + 4));`

Explain each outcome.

1.2.10 Suppose that a variable `a` is declared as `int a = 2147483647` (or equivalently, `Integer.MAX_VALUE`). What do each of the following print?

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

b. `System.out.println(a+1);`

c. `System.out.println(2-a);`

d. `System.out.println(-2-a);`

e. `System.out.println(2*a);`

f. `System.out.println(4*a);`

Explain each outcome.

1.2.11 Suppose that a variable `a` is declared as `double a = 3.14159`. What do each of the following print?

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

b. `System.out.println(a+1);`

c. `System.out.println(8/(int) a);`

d. `System.out.println(8/a);`

e. `System.out.println((int) (8/a));`

Explain each outcome.

1.2.12 Describe what happens if you write `sqrt` instead of `Math.sqrt` in PROGRAM 1.2.3.

1.2.13 What is the value of `(Math.sqrt(2) * Math.sqrt(2) == 2)` ?

1.2.14 Write a program that takes two positive integers as command-line arguments and prints `true` if either evenly divides the other.

1.2.15 Write a program that takes three positive integers as command-line arguments and prints `true` if any one of them is greater than or equal to the sum of the other two and `false` otherwise. (*Note*: This computation tests whether the three numbers could be the lengths of the sides of some triangle.)

1.2.16 A physics student gets unexpected results when using the code

```
F = G * mass1 * mass2 / r * r;
```

to compute values according to the formula $F = Gm_1m_2 / r^2$. Explain the problem and correct the code.

1.2.17 Give the value of `a` after the execution of each of the following sequences:

```
int a = 1;
a = a + a;
a = a + a;
a = a + a;
```

```
boolean a = true;
a = !a;
a = !a;
a = !a;
```

```
int a = 2;
a = a * a;
a = a * a;
a = a * a;
```

1.2.18 Suppose that `x` and `y` are `double` values that represent the Cartesian coordinates of a point (x, y) in the plane. Give an expression whose value is the distance of the point from the origin.

1.2.19 Write a program that takes two `int` values `a` and `b` from the command line and prints a random integer between `a` and `b`.

1.2.20 Write a program that prints the sum of two random integers between `1` and `6` (such as you might get when rolling dice).

1.2.21 Write a program that takes a `double` value `t` from the command line and prints the value of $\sin(2t) + \sin(3t)$.

1.2.22 Write a program that takes three `double` values x_0, v_0, and t from the command line and prints the value of $x_0 + v_0t + gt^2/2$, where g is the constant 9.78033. (*Note*: This value the displacement in meters after t seconds when an object is thrown straight up from initial position x_0 at velocity v_0 meters per second.)

1.2.23 Write a program that takes two `int` values `m` and `d` from the command line and prints `true` if day d of month m is between 3/20 and 6/20, `false` otherwise.

Creative Exercises

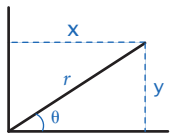
1.2.24 *Loan payments.* Write a program that calculates the monthly payments you would have to make over a given number of years to pay off a loan at a given interest rate compounded continuously, taking the number of years t, the principal P, and the annual interest rate r as command-line arguments. The desired value is given by the formula Pe^{rt}. Use `Math.exp()`.

1.2.25 *Wind chill.* Given the temperature t (in Fahrenheit) and the wind speed v (in miles per hour), the National Weather Service defines the effective temperature (the wind chill) to be:

$$w = 35.74 + 0.6215\,t + (0.4275\,t - 35.75)\,v^{0.16}$$

Write a program that takes two `double` command-line arguments `t` and `v` and prints out the wind chill. Use `Math.pow(a, b)` to compute a^b. *Note*: The formula is not valid if t is larger than 50 in absolute value or if v is larger than 120 or less than 3 (you may assume that the values you get are in that range).

1.2.26 *Polar coordinates.* Write a program that converts from Cartesian to polar coordinates. Your program should take two real numbers `x` and `y` on the command line and print the polar coordinates r and θ. Use the Java method `Math.atan2(y, x)` which computes the arctangent value of `y/x` that is in the range from $-\pi$ to π.



Polar coordinates

1.2.27 *Gaussian random numbers.* One way to generate a random number taken from the Gaussian distribution is to use the *Box-Muller* formula

$$w = \sin(2\,\pi\,v)\,(-2\ln u)^{1/2}$$

where u and v are real numbers between 0 and 1 generated by the `Math.random()` method. Write a program `StdGaussian` that prints out a standard Gaussian random variable.

1.2.28 *Order check.* Write a program that takes three `double` values `x`, `y`, and `z` as command-line arguments and prints `true` if the values are strictly ascending or descending ($x < y < z$ or $x > y > z$), and `false` otherwise.

1.2.29 *Day of the week.* Write a program that takes a date as input and prints the day of the week that date falls on. Your program should take three command line

parameters: `m` (month), `d` (day), and `y` (year). For `m`, use `1` for January, `2` for February, and so forth. For output, print `0` for Sunday, `1` for Monday, `2` for Tuesday, and so forth. Use the following formulas, for the Gregorian calendar:

$$
\begin{aligned}
y_0 &= y - (14 - m) / 12 \\
x &= y_0 + y_0/4 - y_0/100 + y_0/400 \\
m_0 &= m + 12 \times ((14 - m) / 12) - 2 \\
d_0 &= (d + x + (31 \times m_0)/12) \ \% \ 7
\end{aligned}
$$

Example: On what day of the week was February 14, 2000?

$$
\begin{aligned}
y_0 &= 2000 - 1 = 1999 \\
x &= 1999 + 1999/4 - 1999/100 + 1999/400 = 2483 \\
m_0 &= 2 + 12 \times 1 - 2 = 12 \\
d_0 &= (14 + 2483 + (31 \times 12) / 12) \ \% \ 7 = 2500 \ \% \ 7 = 1
\end{aligned}
$$

Answer: Monday.

1.2.30 *Uniform random numbers.* Write a program that prints five uniform random values between 0 and 1, their average value, and their minimum and maximum value. Use `Math.random()`, `Math.min()`, and `Math.max()`.

1.2.31 *Mercator projection.* The *Mercator projection* is a conformal (angle preserving) projection that maps latitude φ and longitude λ to rectangular coordinates (x, y). It is widely used—for example, in nautical charts and in the maps that you print from the web. The projection is defined by the equations $x = \lambda - \lambda_0$ and $y = 1/2 \ln ((1 + \sin \varphi) / (1 - \sin \varphi))$, where λ_0 is the longitude of the point in the center of the map. Write a program that takes λ_0 and the latitude and longitude of a point from the command line and prints its projection.

1.2.32 *Color conversion.* Several different formats are used to represent color. For example, the primary format for LCD displays, digital cameras, and web pages, known as the *RGB format,* specifies the level of red (R), green (G), and blue (B) on an integer scale from 0 to 255. The primary format for publishing books and magazines, known as the *CMYK format*, specifies the level of cyan (C), magenta (M), yellow (Y), and black (K) on a real scale from 0.0 to 1.0. Write a program `RGBtoCMYK` that converts RGB to CMYK. Take three integers—`r`, `g`, and `b`—from the

command line and print the equivalent CMYK values. If the RGB values are all 0, then the CMY values are all 0 and the K value is 1; otherwise, use these formulas:

$$
\begin{aligned}
w &= \max(\, r/255,\, g/255,\, b/255\,) \\
c &= (w - (r/255))/w \\
m &= (w - (g/255))/w \\
y &= (w - (b/255))/w \\
k &= 1 - w
\end{aligned}
$$

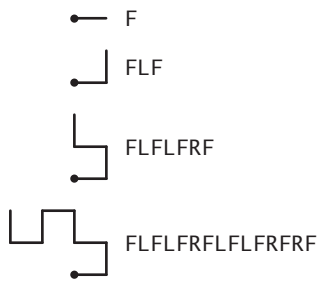
1.2.33 *Great circle.* Write a program `GreatCircle` that takes four command-line arguments—`x1`, `y1`, `x2`, and `y2`—(the latitude and longitude, in degrees, of two points on the earth) and prints out the great-circle distance between them. The great-circle distance (in nautical miles) is given by the equation:

$$d = 60 \arccos(\sin(x_1)\sin(x_2) + \cos(x_1)\cos(x_2)\cos(y_1 - y_2))$$

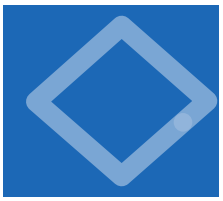
Note that this equation uses degrees, whereas Java's trigonometric functions use radians. Use `Math.toRadians()` and `Math.toDegrees()` to convert between the two. Use your program to compute the great-circle distance between Paris (48.87° N and −2.33° W) and San Francisco (37.8° N and 122.4° W).

1.2.34 *Three-sort.* Write a program that takes three `int` values from the command line and prints them in ascending order. Use `Math.min()` and `Math.max()`.

1.2.35 *Dragon curves.* Write a program to print the instructions for drawing the dragon curves of order 0 through 5. The instructions are strings of `F`, `L`, and `R` characters, where `F` means "draw line while moving 1 unit forward," `L` means "turn left," and `R` means "turn right." A dragon curve of order N is formed when you fold a strip of paper in half N times, then unfold to right angles. The key to solving this problem is to note that a curve of order N is a curve of order $N-1$ followed by an `L` followed by a curve of order $N-1$ traversed in reverse order, and then to figure out a similar description for the reverse curve .



Dragon curves of order 0, 1, 2, and 3



1.3 Conditionals and Loops

In the programs that we have examined to this point, each of the statements in the program is executed once, in the order given. Most programs are more complicated because the sequence of statements and the number of times each is executed can vary. We use the term *control flow* to refer to statement sequencing in a program. In this section, we introduce statements that allow us to change the control flow, using logic about the values of program variables. This feature is an essential component of programming.

1.3.1 Flipping a fair coin. 49
1.3.2 Your first while loop 51
1.3.3 Computing powers of two 53
1.3.4 Your first nested loops. 59
1.3.5 Harmonic numbers 61
1.3.6 Newton's method 62
1.3.7 Converting to binary 64
1.3.8 Gambler's ruin simulation 66
1.3.9 Factoring integers 69

Programs in this section

Specifically, we consider Java statements that implement *conditionals*, where some other statements may or may not be executed depending on certain conditions, and *loops*, where some other statements may be executed multiple times, again depending on certain conditions. As you will see in numerous examples in this section, conditionals and loops truly harness the power of the computer and will equip you to write programs to accomplish a broad variety of tasks that you could not contemplate attempting without a computer.

If statements Most computations require different actions for different inputs. One way to express these differences in Java is the `if` statement:

```
if (<boolean expression>) { <statements> }
```

This description introduces a formal notation known as a *template* that we will use to specify the format of Java constructs. We put within angle brackets (`< >`) a construct that we have already defined, to indicate that we can use any instance of that construct where specified. In this case, *<boolean expression>* represents an expression that has a boolean value, such as one involving a comparison operation, and *<statements>* represents a *statement block* (a sequence of Java statements, each terminated by a semicolon). This latter construct is familiar to you: the body of `main()` is such a sequence. If the sequence is a single statement, the curly braces are optional. It is possible to make formal definitions of *<boolean expression>* and *<statements>*, but we refrain from going into that level of detail. The meaning

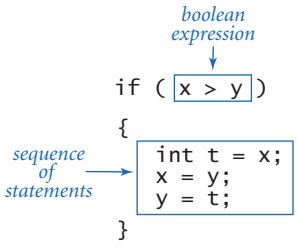
of an `if` statement is self-explanatory: the statement(s) in the sequence are to be executed if and only if the expression is `true`.

As a simple example, suppose that you want to compute the absolute value of an `int` value `x`. This statement does the job:

```
if (x < 0) x = -x;
```

As a second simple example, consider the following statement:

```
if (x > y)
{
   int t = x;
   x = y;
   y = t;
}
```



Anatomy of an `if` statement

This code puts `x` and `y` in ascending order by exchanging them if necessary.

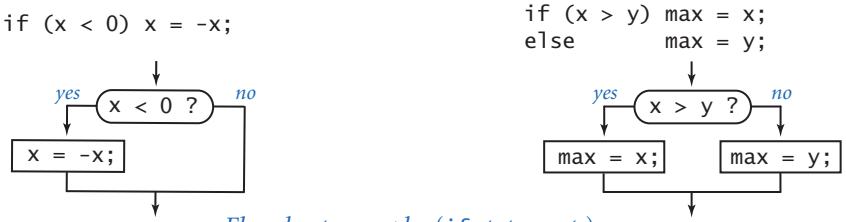
You can also add an `else` clause to an `if` statement, to express the concept of executing either one statement (or sequence of statements) or another, depending on whether the boolean expression is `true` or `false`, as in the following template:

```
if (<boolean expression>) <statements T>
else                      <statements F>
```

As a simple example of the need for an `else` clause, consider the following code, which assigns the maximum of two `int` values to the variable `max`:

```
if (x > y) max = x;
else       max = y;
```

One way to understand control flow is to visualize it with a diagram called a *flowchart*. Paths through the flowchart correspond to flow-of-control paths in the pro-



Flowchart examples (`if` statements)

absolute value	`if (x < 0) x = -x;`

put x *and* y *into sorted order*	`if (x > y)`
	`{`
	`  int t = x;`
	`  y = x;`
	`  x = t;`
	`}`

maximum of x *and* y	`if (x > y) max = x;` `else  max = y;`
error check for division operation	`if (den == 0) System.out.println("Division by zero");` `else  System.out.println("Quotient = " + num/den);`

error check for quadratic formula	`double discriminant = b*b - 4.0*c;`
	`if (discriminant < 0.0)`
	`{`
	`  System.out.println("No real roots");`
	`}`
	`else`
	`{`
	`  System.out.println((-b + Math.sqrt(discriminant))/2.0);`
	`  System.out.println((-b - Math.sqrt(discriminant))/2.0);`
	`}`

Typical examples of using `if` *statements*

gram. In the early days of computing, when programmers used low-level languages and difficult-to-understand flows of control, flowcharts were an essential part of programming. With modern languages, we use flowcharts just to understand basic building blocks like the `if` statement.

The accompanying table contains some examples of the use of `if` and `if-else` statements. These examples are typical of simple calculations you might need in programs that you write. Conditional statements are an essential part of programming. Since the *semantics* (meaning) of statements like these is similar to their meanings as natural-language phrases, you will quickly grow used to them.

PROGRAM 1.3.1 is another example of the use of the `if-else` statement, in this case for the task of simulating a coin flip. The body of the program is a single statement, like the ones in the table above, but it is worth special attention because it introduces an interesting philosophical issue that is worth contemplating: can a computer program produce *random* values? Certainly not, but a program *can* produce numbers that have many of the properties of random numbers.

Program 1.3.1 Flipping a fair coin

```
public class Flip
{
   public static void main(String[] args)
   {  // Simulate a coin flip.
      if (Math.random() < 0.5) System.out.println("Heads");
      else                     System.out.println("Tails");
   }
}
```

This program uses `Math.random()` *to simulate a coin flip. Each time you run it, it prints either heads or tails. A sequence of flips will have many of the same properties as a sequence that you would get by flipping a fair coin, but it is not a truly random sequence.*

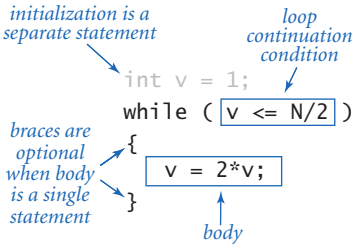
```
% java Flip
Heads
% java Flip
Tails
% java Flip
Tails
```

While loops Many computations are inherently repetitive. The basic Java construct for handling such computations has the following format:

```
while (<boolean expression>) { <statements> }
```

The `while` statement has the same form as the `if` statement (the only difference being the use of the keyword `while` instead of `if`), but the meaning is quite different. It is an instruction to the computer to behave as follows: if the expression is `false`, do nothing; if the expression is `true`, execute the sequence of statements (just as with `if`) but then check the expression again, execute the sequence of statements again if the expression is `true`, and *continue* as long as the expression is `true`. We often refer to the statement block in a loop as the *body* of the loop. As with the `if` statement, the braces are optional if a `while` loop body has just one statement. The `while` statement is equivalent to a sequence of identical `if` statements:

```
if (<boolean expression>) { <statements> }
if (<boolean expression>) { <statements> }
if (<boolean expression>) { <statements> }
...
```



Anatomy of a `while` *loop*

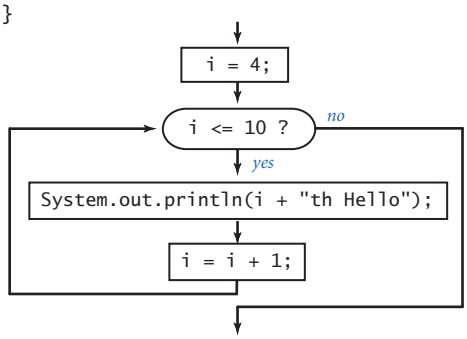
At some point, the code in one of the statements must change something (such as the value of some variable in the boolean expression) to make the boolean expression `false`, and then the sequence is broken.

A common programming paradigm involves maintaining an integer value that keeps track of the number of times a loop iterates. We start at some initial value, and then increment the value by 1 each time through the loop, testing whether it exceeds a predetermined maximum before deciding to continue. `TenHellos` (PROGRAM 1.3.2) is a simple example of this paradigm that uses a `while` statement. The key to the computation is the statement

```
i = i + 1;
```

As a mathematical equation, this statement is nonsense, but as a Java assignment statement it makes perfect sense: it says to compute the value `i + 1` and then assign the result to the variable `i`. If the value of `i` was `4` before the statement, it becomes `5` afterwards; if it was `5` it becomes `6`; and so forth. With the initial condition in `TenHellos` that the value of `i` starts at `4`, the statement block is executed five times until the sequence is broken, when the value of `i` becomes `11`.

```
int i = 4;
while (i <= 10)
{
   System.out.println(i + "th Hello");
   i = i + 1;
}
```



Flowchart example (while statement)

Using the `while` loop is barely worthwhile for this simple task, but you will soon be addressing tasks where you will need to specify that statements be repeated far too many times to contemplate doing it without loops. There is a profound difference between programs with `while` statements and programs without them, because `while` statements allow us to specify a potentially unlimited number of statements to be executed in a program. In particular, the `while` statement allows us to specify lengthy computations in short programs. This ability opens the door to writing programs for tasks that we could not contemplate addressing without a

Program 1.3.2 Your first while loop

```
public class TenHellos
{
   public static void main(String[] args)
   {  // Print 10 Hellos.
      System.out.println("1st Hello");
      System.out.println("2nd Hello");
      System.out.println("3rd Hello");
      int i = 4;
      while (i <= 10)
      {  // Print the ith Hello.
         System.out.println(i + "th Hello");
         i = i + 1;
      }
   }
}
```

This program uses a `while` *loop for the simple, repetitive task of printing the output shown below. After the third line, the lines to be printed differ only in the value of the index counting the line printed, so we define a variable* `i` *to contain that index. After initializing the value of* `i` *to* `4`*, we enter into a* `while` *loop where we use the value of* `i` *in the* `System.out.println()` *statement and increment it each time through the loop. After printing* `10th Hello`*, the value of* `i` *becomes* `11` *and the loop terminates.*

```
% java TenHellos
1st Hello
2nd Hello
3rd Hello
4th Hello
5th Hello
6th Hello
7th Hello
8th Hello
9th Hello
10th Hello
```

i	i <= 10	*output*
4	true	4th Hello
5	true	5th Hello
6	true	6th Hello
7	true	7th Hello
8	true	8th Hello
9	true	9th Hello
10	true	10th Hello
11	false	

Trace of `java TenHellos`

computer. But there is also a price to pay: as your programs become more sophisticated, they become more difficult to understand.

`PowersOfTwo` (PROGRAM 1.3.3) uses a `while` loop to print out a table of the powers of 2. Beyond the loop control counter `i`, it maintains a variable `v` that holds the powers of two as it computes them. The loop body contains three statements: one to print the current power of 2, one to compute the next (multiply the current one by 2), and one to increment the loop control counter.

There are many situations in computer science where it is useful to be familiar with powers of 2. You should know at least the first 10 values in this table and you should note that 2^{10} is about 1 thousand, 2^{20} is about 1 million, and 2^{30} is about 1 billion.

`PowersOfTwo` is the prototype for many useful computations. By varying the computations that change the accumulated value and the way that the loop control variable is incremented, we can print out tables of a variety of functions (see EXERCISE 1.3.11).

It is worthwhile to carefully examine the behavior of programs that use loops by studying a *trace* of the program. For example, a trace of the operation of `PowersOfTwo` should show the value of each variable before each iteration of the loop and the value of the conditional expression that controls the loop. Tracing the operation of a loop can be very tedious, but it is nearly always worthwhile to run a trace because it clearly exposes what a program is doing.

`PowersOfTwo` is nearly a self-tracing program, because it prints the values of its variables each time through the loop. Clearly, you can make any program produce a trace of itself by adding appropriate `System.out.println()` statements. Modern programming environments provide sophisticated tools for tracing, but

i	v	i <= N
0	1	true
1	2	true
2	4	true
3	8	true
4	16	true
5	32	true
6	64	true
7	128	true
8	256	true
9	512	true
10	1024	true
11	2048	true
12	4096	true
13	8192	true
14	16384	true
15	32768	true
16	65536	true
17	131072	true
18	262144	true
19	524288	true
20	1048576	true
21	2097152	true
22	4194304	true
23	8388608	true
24	16777216	true
25	33554432	true
26	67108864	true
27	134217728	true
28	268435456	true
29	536870912	true
30	1073741824	false

Trace of `java PowersOfTwo 29`

Program 1.3.3 Computing powers of two

```
public class PowersOfTwo
{
   public static void main(String[] args)
   {  // Print the first N powers of 2.
      int N = Integer.parseInt(args[0]);
      int v = 1;
      int i = 0;
      while (i <= N)
      {  // Print ith power of 2.
         System.out.println(i + " " + v);
         v = 2 * v;
         i = i + 1;
      }
   }
}
```

N	*loop termination value*
i	*loop control counter*
v	*current power of 2*

This program takes a command-line argument N *and prints a table of the powers of 2 that are less than or equal to* 2^N. *Each time through the loop, we increment the value of* i *and double the value of* v. *We show only the first three and the last three lines of the table; the program prints* N+1 *lines.*

```
% java PowersOfTwo 5
0 1
1 2
2 4
3 8
4 16
5 32
```

```
% java PowersOfTwo 29
0 1
1 2
2 4
...
27 134217728
28 268435456
29 536870912
```

this tried-and-true method is simple and effective. You certainly should add print statements to the first few loops that you write, to be sure that they are doing precisely what you expect.

There is a hidden trap in PowersOfTwo, because the largest integer in Java's int data type is $2^{31} - 1$ and the program does not test for that possibility. If you

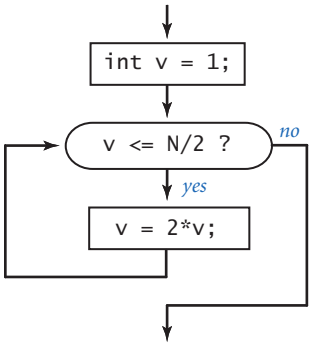
invoke it with `java PowersOfTwo 31`, you may be surprised by the last line of output:

```
...
1073741824
-2147483648
```

The variable `v` becomes too large and takes on a negative value because of the way Java represents integers. The maximum value of an `int` is available for us to use as `Integer.MAX_VALUE`. A better version of PROGRAM 1.3.3 would use this value to test for overflow and print an error message if the user types too large a value, though getting such a program to work properly for all inputs is trickier than you might think. (For a similar challenge, see EXERCISE 1.3.14.)

As a more complicated example, suppose that we want to compute the largest power of two that is less than or equal to a given positive integer `N`. If `N` is `13` we want the result `8`; if `N` is `1000`, we want the result `512`; if `N` is `64`, we want the result `64`; and so forth. This computation is simple to perform with a `while` loop:

```
int v = 1;
while (v <= N/2)
   v = 2*v;
```



Flowchart for the statements

```
int v = 1;
while (v <= N/2)
   v = 2*v;
```

It takes some thought to convince yourself that this simple piece of code produces the desired result. You can do so by making these observations:

- `v` is always a power of `2`.
- `v` is never greater than `N`.
- `v` increases each time through the loop, so the loop must terminate.
- After the loop terminates, `2*v` is greater than `N`.

Reasoning of this sort is often important in understanding how `while` loops work. Even though many of the loops you will write are much simpler than this one, you should be sure to convince yourself that each loop you write is going to behave as you expect.

The logic behind such arguments is the same whether the loop iterates just a few times, as in `TenHellos`, dozens of times, as in `PowersOfTwo`, or millions of times, as in several examples that we will soon consider. That leap from a few tiny cases to a huge computation is profound. When writing loops, understanding how

the values of the variables change each time through the loop (and checking that understanding by adding statements to trace their values and running for a small number of iterations) is essential. Having done so, you can confidently remove those training wheels and truly unleash the power of the computer.

For loops As you will see, the `while` loop allows us to write programs for all manner of applications. Before considering more examples, we will look at an alternate Java construct that allows us even more flexibility when writing programs with loops. This alternate notation is not fundamentally different from the basic `while` loop, but it is widely used because it often allows us to write more compact and more readable programs than if we used only `while` statements.

For notation. Many loops follow this scheme: initialize an index variable to some value and then use a `while` loop to test a loop continuation condition involving the index variable, where the last statement in the `while` loop increments the index variable. You can express such loops directly with Java's `for` notation:

```
for (<initialize>; <boolean expression>; <increment>)
{
   <statements>
}
```

This code is, with only a few exceptions, equivalent to

```
<initialize>;
while (<boolean expression>)
{
   <statements>
   <increment>;
}
```

Your Java compiler might even produce identical results for the two loops. In truth, *<initialize>* and *<increment>* can be any statements at all, but we nearly always use `for` loops to support this typical initialize-and-increment programming idiom. For example, the following two lines of code are equivalent to the corresponding lines of code in `TenHellos` (PROGRAM 1.3.2):

```
for (int i = 4; i <= 10; i = i + 1)
   System.out.println(i + "th Hello");
```

Typically, we work with a slightly more compact version of this code, using the shorthand notation discussed next.

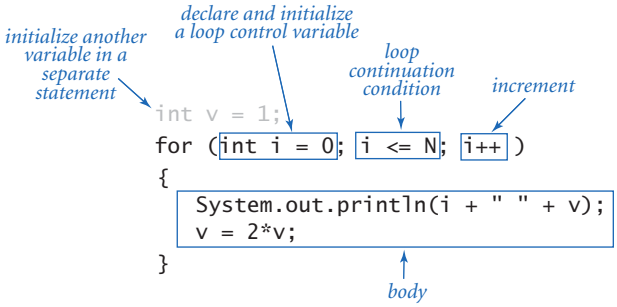
Compound assignment idioms. Modifying the value of a variable is something that we do so often in programming that Java provides a variety of different shorthand notations for the purpose. For example, the following four statements all increment the value of `i` by `1` in Java:

```
i = i + 1;   i++;   ++i;   i += 1;
```

You can also say `i--` or `--i` or `i -= 1` or `i = i-1` to decrement that value of `i` by 1. Most programmers use `i++` or `i--` in `for` loops, though any of the others would do. The `++` and `--` constructs are normally used for integers, but the *compound assignment* constructs are useful operations for any arithmetic operator in any primitive numeric type. For example, you can say `v *= 2` or `v += v` instead of `v = 2*v`. All of these idioms are for notational convenience, nothing more. This combination of shortcuts came into widespread use with the C programming language in the 1970s and have become standard. They have survived the test of time because they lead to compact, elegant, and easily understood programs. When you learn to write (and to read) programs that use them, you will be able to transfer that skill to programming in numerous modern languages, not just Java.

Scope. The scope of a variable is the part of the program where it is defined. Generally the scope of a variable is comprised of the statements that follow the declaration in the same block as the declaration. For this purpose, the code in the `for` loop header is considered to be in the same block as the `for` loop body. Therefore, the `while` and `for` formulations of loops are not quite equivalent: in a typical `for` loop, the incrementing variable is *not* available for use in later statements; in the corresponding `while` loop, it is. This distinction is often a reason to use a `while` instead of a `for` loop.

Choosing among different formulations of the same computation is a matter of each programmer's taste, as when a writer picks from among synonyms or chooses between using active and passive voice when composing a sentence. You will not find good hard-and-fast rules on how to compose a program any more than you will find such rules on how to compose a paragraph. Your goal should be to find a style that suits you, gets the computation done, and can be appreciated by others.



Anatomy of a `for` *loop (that prints powers of 2)*

The accompanying table includes several code fragments with typical examples of loops used in Java code. Some of these relate to code that you have already seen; others are new code for straightforward computations. To cement your understanding of loops in Java, put these code snippets into a class's code that takes an integer `N` from the command line (like `PowersOfTwo`) and *compile and run them*. Then, write some loops of your own for similar computations of your own invention, or do some of the early exercises at the end of this section. There is no substitute for the experience gained by running code that you create yourself, and it is imperative that you develop an understanding of how to write Java code that uses loops.

print largest power of two less than or equal to N	`int v = 1;`
	`while (v <= N/2)`
	`  v = 2*v;`
	`System.out.println(v);`

compute a finite sum (1 + 2 + . . . + N)	`int sum = 0;`
	`for (int i = 1; i <= N; i++)`
	`  sum += i;`
	`System.out.println(sum);`

compute a finite product (N! = 1 × 2 × . . . × N)	`int product = 1;`
	`for (int i = 1; i <= N; i++)`
	`  product *= i;`
	`System.out.println(product);`

print a table of function values	`for (int i = 0; i <= N; i++)` `  System.out.println(i + " " + 2*Math.PI*i/N);`

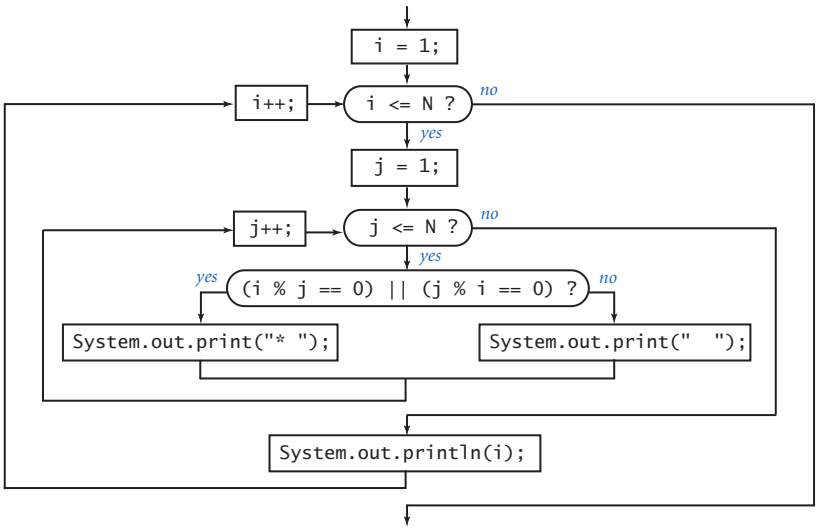
print the ruler function (see Program 1.2.1)	`String ruler = " ";`
	`for (int i = 1; i <= N; i++)`
	`  ruler = ruler + i + ruler;`
	`System.out.println(ruler);`

Typical examples of using `for` *and* `while` *statements*

Nesting The `if`, `while`, and `for` statements have the same status as assignment statements or any other statements in Java. That is, we can use them whenever a statement is called for. In particular, we can use one or more of them in the *<body>* of another to make compound statements. As a first example, `DivisorPattern` (PROGRAM 1.3.4) has a `for` loop whose statements are a `for` loop (whose statement is an `if` statement) and a print statement. It prints a pattern of asterisks where the *i*th row has an asterisk in each position corresponding to divisors of *i* (the same holds true for the columns).

To emphasize the nesting, we use indentation in the program code. We refer to the `i` loop as the *outer* loop and the `j` loop as the *inner* loop. The inner loop iterates all the way through for each iteration of the outer loop. As usual, the best way to understand a new programming construct like this is to study a trace.

`DivisorPattern` has a complicated control structure, as you can see from its flowchart. A diagram like this illustrates the importance of using a limited number of simple control structures in programming. With nesting, you can compose loops and conditionals to build programs that are easy to understand even though they may have a complicated control structure. A great many useful computations can be accomplished with just one or two levels of nesting. For example, many programs in this book have the same general structure as `DivisorPattern`.



Flowchart for `DivisorPattern`

Program 1.3.4 Your first nested loops

```
public class DivisorPattern
{
   public static void main(String[] args)
   {  // Print a square that visualizes divisors.
      int N = Integer.parseInt(args[0]);
      for (int i = 1; i <= N; i++)
      {  // Print the ith line
         for (int j = 1; j <= N; j++)
         {  // Print the jth entry in the ith line.
            if ((i % j == 0) || (j % i == 0))
               System.out.print("* ");
            else
               System.out.print("  ");
         }
         System.out.println(i);
      }
   }
}
```

N	*number of rows and columns*
i	*row index*
j	*column index*

This program takes an integer N *as the command-line argument and uses nested* for *loops to print an* N*-by-*N *table with an asterisk in row* i *and column* j *if either* i *divides* j *or* j *divides* i*. The loop control variables* i *and* j *control the computation.*

```
% java DivisorPattern 3
* * *  1
* *    2
*   *  3

% java DivisorPattern 16
* * * * * * * * * * * * * * * * 1
* *   *   *   *   *   *   *   * 2
*   *     *     *     *     *   3
* *   *       *       *       * 4
*       *         *         *   5
* * *     *           *         6
*           *               *   7
* *   *       *               * 8
*   *           *               9
* *     *         *             10
*                   *           11
* * * *   *           *         12
*                       *       13
* *         *               *   14
*   *   *                   *   15
* *   *       *               * 16
```

i	j	i % j	j % i	*output*
1	1	0	0	*
1	2	1	0	*
1	3	1	0	*
				1
2	1	0	1	*
2	2	0	0	*
2	3	2	1	
				2
3	1	0	1	*
3	2	1	2	
3	3	0	0	*
				3

Trace of java DivisorPattern 3

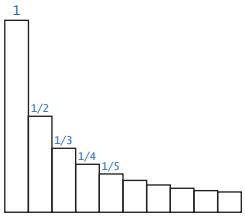
As a second example of nesting, consider the following program fragment, which a tax preparation program might use to compute income tax rates:

```
if      (income <      0) rate = 0.0;
else if (income <  47450) rate = .22;
else if (income < 114650) rate = .25;
else if (income < 174700) rate = .28;
else if (income < 311950) rate = .33;
else                      rate = .35;
```

In this case, a number of `if` statements are nested to test from among a number of mutually exclusive possibilities. This construct is a special one that we use often. Otherwise, it is best to use braces to resolve ambiguities when nesting `if` statements. This issue and more examples are addressed in the Q&A and exercises.

Applications The ability to program with loops immediately opens up the full world of computation. To emphasize this fact, we next consider a variety of examples. These examples all involve working with the types of data that we considered in SECTION 1.2, but rest assured that the same mechanisms serve us well for any computational application. The sample programs are carefully crafted, and by studying and appreciating them, you will be prepared to write your own programs containing loops, as requested in many of the exercises at the end of this section.

The examples that we consider here involve computing with numbers. Several of our examples are tied to problems faced by mathematicians and scientists throughout the past several centuries. While computers have existed for only 50 years or so, many of the computational methods that we use are based on a rich mathematical tradition tracing back to antiquity.



Finite sum. The computational paradigm used by `PowersOfTwo` is one that you will use frequently. It uses two variables—one as an index that controls a loop and the other to accumulate a computational result. `Harmonic` (PROGRAM 1.3.5) uses the same paradigm to evaluate the finite sum $H_N = 1 + 1/2 + 1/3 + ... + 1/N$. These numbers, which are known as the *Harmonic numbers*, arise frequently in discrete mathematics. Harmonic numbers are the discrete analog of the logarithm. They also approximate the area under the curve $y = 1/x$. You can use PROGRAM 1.3.5 as a model for computing the values of other sums (see EXERCISE 1.3.16).

Program 1.3.5 Harmonic numbers

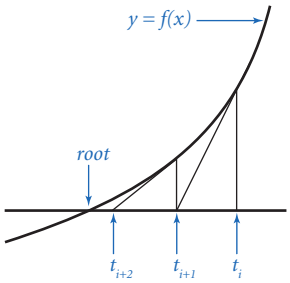
```
public class Harmonic
{
   public static void main(String[] args)
   {  // Compute the Nth Harmonic number.
      int N = Integer.parseInt(args[0]);
      double sum = 0.0;
      for (int i = 1; i <= N; i++)
      {  // Add the ith term to the sum
         sum += 1.0/i;
      }
      System.out.println(sum);
   }
}
```

N	*number of terms in sum*
i	*loop index*
sum	*cumulated sum*

This program computes the value of the Nth Harmonic number. The value is known from mathematical analysis to be about ln(N) + 0.57721 for large N. Note that ln(10000) ≈ 9.21034.

```
% java Harmonic 2
1.5
% java Harmonic 10
2.9289682539682538
% java Harmonic 10000
9.787606036044348
```

Computing the square root. How are functions in Java's `Math` library, such as `Math.sqrt()`, implemented? `Sqrt` (PROGRAM 1.3.6) illustrates one technique. To compute the square root function, it uses an iterative computation that was known to the Babylonians over 4,000 years ago. It is also a special case of a general computational technique that was developed in the 17th century by Isaac Newton and Joseph Raphson and is widely known as *Newton's method*. Under generous conditions on a given function $f(x)$, Newton's method is an effective way to find roots (values of x for which the function is 0). Start with an initial estimate, t_0. Given the



Newton's method

Program 1.3.6 Newton's method

```
public class Sqrt
{
   public static void main(String[] args)
   {
      double c = Double.parseDouble(args[0]);
      double epsilon = 1e-15;
      double t = c;
      while (Math.abs(t - c/t) > epsilon * t)
      {  // Replace t by the average of t and c/t.
         t = (c/t + t) / 2.0;
      }
      System.out.println(t);
   }
}
```

c	*argument*
epsilon	*error tolerance*
t	*estimate of c*

This program computes the square root of its command-line argument to 15 decimal places of accuracy, using Newton's method (see text).

```
% java Sqrt 2.0
1.414213562373095
% java Sqrt 2544545
1595.1630010754388
```

iteration	t	c/t
	2.0000000000000000	1.0
1	1.5000000000000000	1.3333333333333333
2	1.4166666666666665	1.4117647058823530
3	1.4142156862745097	1.4142114384748700
4	1.4142135623746899	1.4142135623715002
5	1.4142135623730950	1.4142135623730951

Trace of `java Sqrt 2.0`

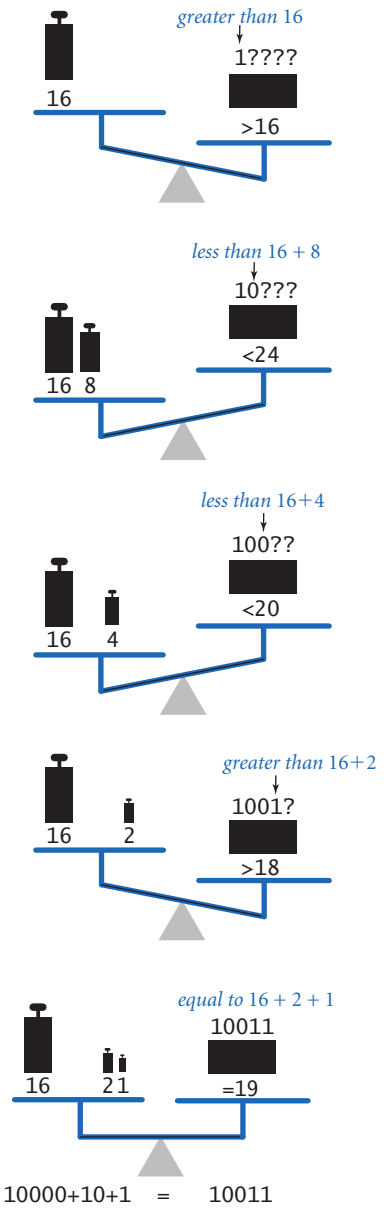
estimate t_i, compute a new estimate by drawing a line tangent to the curve $y = f(x)$ at the point $(t_i, f(t_i))$ and set t_{i+1} to the x-coordinate of the point where that line hits the x-axis. Iterating this process, we get closer to the root.

Computing the square root of a positive number c is equivalent to finding the positive root of the function $f(x) = x^2 - c$. For this special case, Newton's method amounts to the process implemented in `Sqrt` (see EXERCISE 1.3.17). Start with the estimate $t = c$. If t is equal to c/t, then t is equal to the square root of c, so the computation is complete. If not, refine the estimate by replacing t with the average of t

and *c*/*t*. With Newton's method, we get the value of the square root of 2 accurate to 15 places in just 5 iterations of the loop.

Newton's method is important in scientific computing because the same iterative approach is effective for finding the roots of a broad class of functions, including many for which analytic solutions are not known (so the Java `Math` library would be no help). Nowadays, we take for granted that we can find whatever values we need of mathematical functions; before computers, scientists and engineers had to use tables or computed values by hand. Computational techniques that were developed to enable calculations by hand needed to be very efficient, so it is not surprising that many of those same techniques are effective when we use computers. Newton's method is a classic example of this phenomenon. Another useful approach for evaluating mathematical functions is to use Taylor series expansions (see EXERCISES 1.3.35–36).

Number conversion. `Binary` (PROGRAM 1.3.7) prints the binary (base 2) representation of the decimal number typed as the command-line argument. It is based on decomposing a number into a sum of powers of two. For example, the binary representation of 19 is `10011`, which is the same as saying that 19 = 16 + 2 + 1. To compute the binary representation of `N`, we consider the powers of 2 less than or equal to `N` in decreasing order to determine which belong in the binary decomposition (and therefore correspond to a 1 bit in the binary representation). The process corresponds precisely to using a balance scale to weigh an object, using weights whose values are powers of two. First, we find largest weight not heavier than the object. Then, considering the weights in decreasing order, we add each weight to test whether the object is lighter. If so, we remove the



Scale analog to binary conversion

Program 1.3.7 Converting to binary

```
public class Binary
{
   public static void main(String[] args)
   {  // Print binary representation of N.
      int N = Integer.parseInt(args[0]);
      int v = 1;
      while (v <= N/2)
         v = 2*v;
      // Now v is the largest power of 2 <= N.

      int n = N;
      while (v > 0)
      {  // Cast out powers of 2 in decreasing order.
         if (n < v) { System.out.print(0);         }
         else       { System.out.print(1); n -= v; }
         v = v/2;
      }
      System.out.println();
   }
}
```

N	*integer to convert*
v	*current power of 2*
n	*current excess*

This program prints the binary representation of a positive integer given as the command-line argument, by casting out powers of 2 in decreasing order (see text).

```
% java Binary 19
10011
% java Binary 100000000
101111101011110000100000000
```

weight; if not, we leave the weight and try the next one. Each weight corresponds to a bit in the binary representation of the weight of the object: leaving a weight corresponds to a 1 bit in the binary representation of the object's weight, and removing a weight corresponds to a 0 bit in the binary representation of the object's weight.

In `Binary`, the variable `v` corresponds to the current weight being tested, and the variable `n` accounts for the excess (unknown) part of the object's weight (to

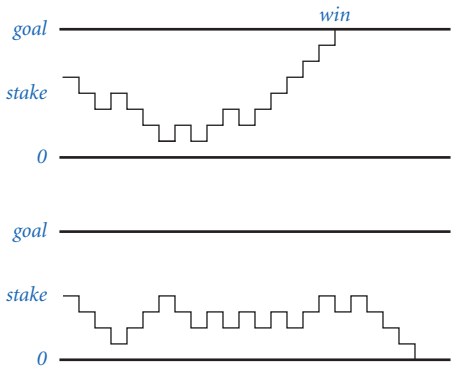
n	*binary representation*	v	v > 0	*binary representation*	n < v	*output*
19	10011	16	true	10000	false	1
3	0011	8	true	1000	true	0
3	011	4	true	100	true	0
3	01	2	true	10	false	1
1	1	1	true	1	false	1
0		0	false			

Trace of casting-out-powers-of-two loop for `java Binary 19`

simulate leaving a weight on the balance, we just subtract that weight from `n`). The value of `v` decreases through the powers of two. When it is larger than `n`, `Binary` prints `0`; otherwise, it prints `1` and subtracts `v` from `n`. As usual, a trace (of the values of `n`, `v`, `n < v`, and the output bit for each loop iteration) can be very useful in helping you to understand the program. Read from top to bottom in the rightmost column of the trace, the output is `10011`, the binary representation of 19.

Converting data from one representation to another is a frequent theme in writing computer programs. Thinking about conversion emphasizes the distinction between an abstraction (an integer like the number of hours in a day) and a representation of that abstraction (24 or 11000). The irony here is that the computer's representation of an integer is actually based on its binary representation.

Simulation. Our next example is different in character from the ones we have been considering, but it is representative of a common situation where we use computers to simulate what might happen in the real world so that we can make informed decisions. The specific example that we consider now is from a thoroughly studied class of problems known as *gambler's ruin*. Suppose that a gambler makes a series of fair $1 bets, starting with some given initial stake. The gambler always goes broke eventually, but when we set other limits on the game, various questions arise. For example, suppose that the gam-



Gambler simulation sequences

Program 1.3.8 Gambler's ruin simulation

```
public class Gambler
{
   public static void main(String[] args)
   {  // Run T experiments that start with $stake
      // and terminate on $0 or $goal.
      int stake = Integer.parseInt(args[0]);
      int goal  = Integer.parseInt(args[1]);
      int T     = Integer.parseInt(args[2]);
      int bets = 0;
      int wins = 0;
      for (int t = 0; t < T; t++)
      {  // Run one experiment.
         int cash = stake;
         while (cash > 0 && cash < goal)
         {  // Simulate one bet.
             bets++;
             if (Math.random() < 0.5) cash++;
             else                     cash--;
         }  // Cash is either 0 (ruin) or $goal (win).
         if (cash == goal) wins++;
      }
      System.out.println(100*wins/T + "% wins");
      System.out.println("Avg # bets: " + bets/T);
   }
}
```

stake	*initial stake*
goal	*walkaway goal*
T	*number of trials*
bets	*bet count*
wins	*win count*
cash	*cash on hand*

The inner `while` *loop in this program simulates a gambler with* `$stake` *who makes a series of* `$1` *bets, continuing until going broke or reaching* `$goal`*. The running time of this program is proportional to T times the average number of bets. For example, the third command below causes nearly 100 million random numbers to be generated.*

```
% java Gambler 10 20 1000
50% wins
Avg # bets: 100
% java Gambler 50 250 100
19% wins
Avg # bets: 11050
% java Gambler 500 2500 100
21% wins
Avg # bets: 998071
```

bler decides ahead of time to walk away after reaching a certain goal. What are the chances that the gambler will win? How many bets might be needed to win or lose the game? What is the maximum amount of money that the gambler will have during the course of the game?

`Gambler` (PROGRAM 1.3.8) is a simulation that can help answer these questions. It does a sequence of trials, using `Math.random()` to simulate the sequence of bets, continuing until the gambler is broke or the goal is reached, and keeping track of the number of wins and the number of bets. After running the experiment for the specified number of trials, it averages and prints out the results. You might wish to run this program for various values of the command-line arguments, not necessarily just to plan your next trip to the casino, but to help you think about the following questions: Is the simulation an accurate reflection of what would happen in real life? How many trials are needed to get an accurate answer? What are the computational limits on performing such a simulation? Simulations are widely used in applications in economics, science, and engineering, and questions of this sort are important in any simulation.

In the case of `Gambler`, we are verifying classical results from probability theory, which say the *probability of success is the ratio of the stake to the goal* and that the *expected number of bets is the product of the stake and the desired gain* (the difference between the goal and the stake). For example, if you want to go to Monte Carlo to try to turn $500 into $2,500, you have a reasonable (20%) chance of success, but you should expect to make a million $1 bets! If you try to turn $1 into $1,000, you have a .1% chance and can expect to be done (ruin, most likely) in about 999 bets.

Simulation and analysis go hand-in-hand, each validating the other. In practice, the value of simulation is that it can suggest answers to questions that might be too difficult to resolve with analysis. For example, suppose that our gambler, recognizing that there will never be enough time to make a million bets, decides ahead of time to set an upper limit on the number of bets. How much money can the gambler expect to take home in that case? You can address this question with an easy change to PROGRAM 1.3.8 (see EXERCISE 1.3.24), but addressing it with mathematical analysis is not so easy.

Factoring. A *prime* is an integer greater than one whose only positive divisors are one and itself. The prime factorization of an integer is the multiset of primes whose product is the integer. For example, `3757208 = 2*2*2*7*13*13*397`. `Factors` (PROGRAM 1.3.9) computes the prime factorization of any given positive integer. In contrast to many of the other programs that we have seen (which we could do in a few minutes with a calculator or even a pencil and paper), this computation would not be feasible without a computer. How would you go about trying to find the factors of a number like `287994837222311`? You might find the factor `17` quickly, but even with a calculator it would take you quite a while to find `1739347`.

i	N	*output*
2	3757208	2 2 2
3	469651	
4	469651	
5	469651	
6	469651	
7	469651	7
8	67093	
9	67093	
10	67093	
11	67093	
12	67093	
13	67093	13 13
14	397	
15	397	
16	397	
17	397	
18	397	
19	397	
20	397	
		397

Trace of `java Factors 3757208`

Although `Factors` is compact and straightforward, it certainly will take some thought to for you to convince yourself that it produces the desired result for any given integer. As usual, following a trace that shows the values of the variables at the beginning of each iteration of the outer `for` loop is a good way to understand the computation. For the case where the initial value of `N` is `3757208`, the inner `while` loop iterates three times when `i` is `2`, to remove the three factors of `2`; then zero times when `i` is `3`, `4`, `5`, and `6`, since none of those numbers divide `469651`; and so forth. Tracing the program for a few example inputs clearly reveals its basic operation. To convince ourselves that the program will behave as expected for all inputs, we reason about what we expect each of the loops to do. The `while` loop clearly prints and removes from `n` all factors of `i`, but the key to understanding the program is to see that the following fact holds at the beginning of each iteration of the `for` loop: `n` has no factors between `2` and `i-1`. Thus, if `i` is not prime, it will not divide `n`; if `i` is prime, the `while` loop will do its job. Once we know that `n` has no factors less than or equal to `i`, we also know that it has no factors greater than `n/i`, so we need look no further when `i` is greater than `n/i`.

In a more naïve implementation, we might simply have used the condition `(i < n)` to terminate the `for` loop. Even given the blinding speed of modern computers, such a decision would have a dramatic effect on the size of the numbers that we could factor. EXERCISE 1.3.26 encourages you to experiment with the program to

Program 1.3.9 Factoring integers

```
public class Factors
{
   public static void main(String[] args)
   {  // Print the prime factors of N.
      long N = Long.parseLong(args[0]);
      long n = N;
      for (long i = 2; i <= n/i; i++)
      {  // Test whether i is a factor.
         while (n % i == 0)
         {  // Cast out and print i factors.
            n /= i;
            System.out.print(i + " ");
         }  // Any factors of n are greater than i.
      }
      if (n > 1) System.out.print(n);
      System.out.println();
   }
}
```

N	*integer to factor*
n	*unfactored part*
i	*potential factor*

This program prints the prime factorization of any positive integer in Java's `long` *data type. The code is simple, but it takes some thought to convince oneself that it is correct (see text).*

```
% java Factors 3757208
2 2 2 7 13 13 397
```

```
% java Factors 287994837222311
17 1739347 9739789
```

learn the effectiveness of this simple change. On a computer that can do billions of operations per second, we could factor numbers on the order of 10^9 in a few seconds; with the `(i <= n/i)` test we can factor numbers on the order of 10^{18} in a comparable amount of time. Loops give us the ability to solve difficult problems, but they also give us the ability to construct simple programs that run slowly, so we must always be cognizant of performance.

In modern applications in cryptography, there are important situations where we wish to factor truly huge numbers (with, say, hundreds or thousands of digits). Such a computation is prohibitively difficult even *with* the use of a computer.

Other conditional and loop constructs To more fully cover the Java language, we consider here four more control-flow constructs. You need not think about using these constructs for every program that you write, because you are likely to encounter them much less frequently than the `if`, `while`, and `for` statements. You certainly do not need to worry about using these constructs until you are comfortable using `if`, `while`, and `for`. You might encounter one of them in a program in a book or on the web, but many programmers do not use them at all and we do not use any of them outside this section.

Break statement. In some situations, we want to immediately exit a loop without letting it run to completion. Java provides the `break` statement for this purpose. For example, the following code is an effective way to test whether a given integer `N>1` is prime:

```
int i;
for (i = 2; i <= N/i; i++)
   if (N % i == 0) break;
if (i > N/i) System.out.println(N + " is prime");
```

There are two different ways to leave this loop: either the `break` statement is executed (because `i` divides `N`, so `N` is not prime) or the `for` loop condition is not satisfied (because no `i` with `i <= N/i` was found that divides `N`, which implies that `N` is prime). Note that we have to declare `i` outside the `for` loop instead of in the initialization statement so that its scope extends beyond the loop.

Continue statement. Java also provides a way to skip to the next iteration of a loop: the `continue` statement. When a `continue` is executed within a loop body, the flow of control transfers directly to the increment statement for the next iteration of the loop.

Switch statement. The `if` and `if-else` statements allow one or two alternatives in directing the flow of control. Sometimes, a computation naturally suggests more than two mutually exclusive alternatives. We could use a sequence or a chain of `if-else` statements, but the Java `switch` statement provides a direct solution. Let us move right to a typical example. Rather than printing an `int` variable `day` in a program that works with days of the weeks (such as a solution to EXERCISE 1.2.29), it is easier to use a `switch` statement, as follows:

```
switch (day)
{
   case 0: System.out.println("Sun"); break;
   case 1: System.out.println("Mon"); break;
   case 2: System.out.println("Tue"); break;
   case 3: System.out.println("Wed"); break;
   case 4: System.out.println("Thu"); break;
   case 5: System.out.println("Fri"); break;
   case 6: System.out.println("Sat"); break;
}
```

When you have a program that seems to have a long and regular sequence of `if` statements, you might consider consulting the booksite and using a `switch` statement, or using an alternate approach described in SECTION 1.4.

Do-while loop. Another way to write a loop is to use the template

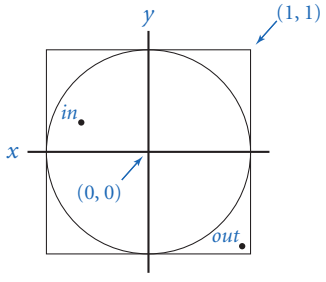
```
do { <statements> } while (<boolean expression>);
```

The meaning of this statement is the same as

```
while (<boolean expression>) { <statements> }
```

except that the first test of the condition is omitted. If the condition initially holds, there is no difference. For an example in which `do-while` is useful, consider the problem of generating points that are randomly distributed in the unit disk. We can use `Math.random()` to generate x and y coordinates independently to get points that are randomly distributed in the 2-by-2 square centered on the origin. Most points fall within the unit disk, so we just reject those that do not. We always want to generate at least one point, so a `do-while` loop is ideal for this computation. The following code sets x and y such that the point (x, y) is randomly distributed in the unit disk:

```
do
{  // Scale x and y to be random in (-1, 1).
   x = 2.0*Math.random() - 1.0;
   y = 2.0*Math.random() - 1.0;
} while (Math.sqrt(x*x + y*y) > 1.0);
```



Since the area of the disk is π and the area of the square is 4, the expected number of times the loop is iterated is $4/\pi$ (about 1.27).

Infinite loops Before you write programs that use loops, you need to think about the following issue: what if the loop-continuation condition in a `while` loop is always satisfied? With the statements that you have learned so far, one of two bad things could happen, both of which you need to learn to cope with.

First, suppose that such a loop calls `System.out.println()`. For example, if the condition in `TenHellos` were `(i > 3)` instead of `(i <= 10)`, it would always be `true`. What happens? Nowadays, we use *print* as an abstraction to mean *display in a terminal window* and the result of attempting to display an unlimited number of lines in a terminal window is dependent on operating-system conventions. If your system is set up to have *print* mean *print characters on a piece of paper*, you might run out of paper or have to unplug the printer. In a terminal window, you need a *stop printing* operation. Before running programs with loops on your own, you make sure that you know what to do to "pull the plug" on an infinite loop of `System.out.println()` calls and then test out the strategy by making the change to `TenHellos` indicated above and trying to stop it. On most systems, `<ctrl-c>` means *stop the current program*, and should do the job.

```
public class BadHellos
...
int i = 4;
while (i > 3)
{
   System.out.println
      (i + "th Hello");
   i = i + 1;
}
...
```

```
% java BadHellos
1st Hello
2nd Hello
3rd Hello
5th Hello
6th Hello
7th Hello
...
```

An infinite loop

Second, *nothing* might happen. If your program has an infinite loop that does not produce any output, it will spin through the loop and you will see no results at all. When you find yourself in such a situation, you can inspect the loops to make sure that the loop exit condition always happens, but the problem may not be easy to identify. One way to locate such a bug is to insert calls to `System.out.println()` to produce a trace. If these calls fall within an infinite loop, this strategy reduces the problem to the case discussed in the previous paragraph, but the output might give you a clue about what to do.

You might not know (or it might not matter) whether a loop is infinite or just very long. Even `BadHellos` eventually would terminate after printing over a billion lines because of overflow. If you invoke PROGRAM 1.3.8 with arguments such as `java Gambler 100000 200000 100`, you may not want to wait for the answer. You will learn to be aware of and to estimate the running time of your programs.

Why not have Java detect infinite loops and warn us about them? You might be surprised to know that it is not possible to do so, in general. This counterintuitive fact is one of the fundamental results of theoretical computer science.

Summary For reference, the accompanying table lists the programs that we have considered in this section. They are representative of the kinds of tasks we can address with short programs comprised of `if`, `while`, and `for` statements processing built-in types of data. These types of computations are an appropriate way to become familiar with the basic Java flow-of-control constructs. The time that you spend now working with as many such programs as you can will certainly pay off for you in the future.

program	*description*
`Flip`	simulate a coin flip
`TenHellos`	your first loop
`PowersOfTwo`	compute and print a table of values
`DivisorPattern`	your first nested loop
`Harmonic`	compute finite sum
`Sqrt`	classic iterative algorithm
`Binary`	basic number conversion
`Gambler`	simulation with nested loops
`Factors`	`while` loop within a `for` loop

Summary of programs in this section

To learn how to use conditionals and loops, you must practice writing and debugging programs with `if`, `while`, and `for` statements. The exercises at the end of this section provide many opportunities for you to begin this process. For each exercise, you will write a Java program, then run and test it. All programmers know that it is unusual to have a program work as planned the first time it is run, so you will want to have an understanding of your program and an expectation of what it should do, step by step. At first, use explicit traces to check your understanding and expectation. As you gain experience, you will find yourself thinking in terms of what a trace might produce as you compose your loops. Ask yourself the following kinds of questions: What will be the values of the variables after the loop iterates the first time? The second time? The final time? Is there any way this program could get stuck in an infinite loop?

Loops and conditionals are a giant step in our ability to compute: `if`, `while`, and `for` statements take us from simple straight-line programs to arbitrarily complicated flow of control. In the next several chapters, we will take more giant steps that will allow us to process large amounts of input data and allow us to define and process types of data other than simple numeric types. The `if`, `while`, and `for` statements of this section will play an essential role in the programs that we consider as we take these steps.

Q&A

Q. What is the difference between = and ==?

A. We repeat this question here to remind you to be sure not to use = when you mean == in a conditional expression. The expression `(x = y)` assigns the value of `y` to `x`, whereas the expression `(x == y)` tests whether the two variables currently have the same values. In some programming languages, this difference can wreak havoc in a program and be difficult to detect, but Java's type safety usually will come to the rescue. For example, if we make the mistake of typing `(t = goal)` instead of `(t == goal)` in PROGRAM 1.3.8, the compiler finds the bug for us:

```
javac Gambler.java
Gambler.java:18: incompatible types
found : int
required: boolean
if (t = goal) wins++;
    ^
1 error
```

Be careful about writing `if (x = y)` when `x` and `y` are `boolean` variables, since this will be treated as an assignment statement, which assigns the value of `y` to `x` and evaluates to the truth value of `y`. For example, instead of writing `if (isPrime = false)`, you should write `if (!isPrime)`.

Q. So I need to pay attention to using == instead of = when writing loops and conditionals. Is there something else in particular that I should watch out for?

A. Another common mistake is to forget the braces in a loop or conditional with a multi-statement body. For example, consider this version of the code in `Gambler`:

```
for (int t = 0; t < T; t++)
   for (cash = stake; cash > 0 && cash < goal; bets++)
      if (Math.random() < 0.5) cash++;
      else                     cash--;
   if (cash == goal) wins++;
```

The code appears correct, but it is dysfunctional because the second `if` is outside both `for` loops and gets executed just once. Our practice of using explicit braces for long statements is precisely to avoid such insidious bugs.

Q. Anything else?

A. The third classic pitfall is ambiguity in nested `if` statements:

```
if <expr1> if <expr2> <stmntA> else <stmntB>
```

In Java this is equivalent to

```
if <expr1> { if <expr2> <stmntA> else <stmntB> }
```

even if you might have been thinking

```
if <expr1> { if <expr2> <stmntA> } else <stmntB>
```

Again, using explicit braces is a good way to avoid this pitfall.

Q. Are there cases where I must use a `for` loop but not a `while`, or vice versa?

A. No. Generally, you should use a `for` loop when you have an initialization, an increment, and a loop continuation test (if you do not need the loop control variable outside the loop). But the equivalent `while` loop still might be fine.

Q. What are the rules on where we declare the loop-control variables?

A. Opinions differ. In older programming languages, it was required that all variables be declared at the beginning of a *<body>*, so many programmers are in this habit and there is a lot of code out there that follows this convention. But it makes a lot of sense to declare variables where they are first used, particularly in `for` loops, when it is normally the case that the variable is not needed outside the loop. However, it is not uncommon to need to test (and therefore declare) the loop-control variable outside the loop, as in the primality-testing code we considered as an example of the `break` statement.

Q. What is the difference between `++i` and `i++`?

A. As statements, there is no difference. In expressions, both increment `i`, but `++i` has the value after the increment and `i++` the value before the increment. In this book, we avoid statements like `x = ++i` that have the side effect of changing variable values. So, it is safe to not worry much about this distinction and just use `i++`

in `for` loops and as a statement. When we do use `++i` in this book, we will call attention to it and say why we are using it.

Q. So, <*initialize*> and <*increment*> can be any statements whatsoever in a `for` loop. How can I take advantage of that?

A. Some experts take advantage of this ability to create compact code fragments, but, as a beginner, it is best for you to use a `while` loop in such situations. In fact, the situation is even more complicated because <*initialize*> and <*increment*> can be *sequences* of statements, separated by commas. This notation allows for code that initializes and modifies other variables besides the loop index. In some cases, this ability leads to compact code. For example, the following two lines of code could replace the last eight lines in the body of the `main()` method in `PowersOfTwo` (PROGRAM 1.3.3):

```
for (int i = 0, v = 1; i <= n; i++, v *= 2)
   System.out.println(i + " " + v);
```

Such code is rarely necessary and better avoided, particularly by beginners.

Q Can I use a `double` value as an index in a `for` loop?

A It is legal, but generally bad practice to do so. Consider the following loop:

```
for (double x = 0.0; x <= 1.0; x += 0.1)
   System.out.println(x + " " + Math.sin(x));
```

How many times does it iterate? The number of iterations depends on an equality test between `double` values, which may not always give the result that you expect.

Q. Anything else tricky about loops?

A. Not all parts of a `for` loop need to be filled in with code. The initialization statement, the boolean expression, the increment statement, and the loop body can each be omitted. It is generally better style to use a `while` statement than null statements in a `for` loop. In the code in this book, we avoid null statements.

```
int v = 1;
while (v <= N/2)
   v *= 2;

for (int v = 1; v <= N/2; )
   v *= 2;

for (int v = 1; v <= N/2; v *= 2)
   ;
```

null increment statement

null loop body

Three equivalent loops

Exercises

1.3.1 Write a program that takes three integer command-line arguments and prints `equal` if all three are equal, and `not equal` otherwise.

1.3.2 Write a more general and more robust version of `Quadratic` (PROGRAM 1.2.3) that prints the roots of the polynomial $ax^2 + bx + c$, prints an appropriate message if the discriminant is negative, and behaves appropriately (avoiding division by zero) if a is zero.

1.3.3 What (if anything) is wrong with each of the following statements?

a.
```
if (a > b) then c = 0;
```

b.
```
if a > b { c = 0; }
```

c.
```
if (a > b) c = 0;
```

d.
```
if (a > b) c = 0 else b = 0;
```

1.3.4 Write a code fragment that prints `true` if the `double` variables `x` and `y` are both strictly between `0` and `1` and `false` otherwise.

1.3.5 Improve your solution to EXERCISE 1.2.25 by adding code to check that the values of the command-line arguments fall within the ranges of validity of the formula, and also adding code to print out an error message if that is not the case.

1.3.6 Suppose that `i` and `j` are both of type `int`. What is the value of `j` after each of the following statements is executed?

a.
```
for (i = 0, j = 0; i < 10; i++)  j += i;
```

b.
```
for (i = 0, j = 1; i < 10; i++)  j += j;
```

c.
```
for (j = 0; j < 10; j++) j += j;
```

d.
```
for (i = 0, j = 0; i < 10; i++) j += j++;
```

1.3.7 Rewrite `TenHellos` to make a program `Hellos` that takes the number of lines to print as a command-line argument. You may assume that the argument is less than `1000`. Hint: Use `i % 10` and `i % 100` to determine when to use `st`, `nd`, `rd`, or `th` for printing the `i`th `Hello`.

1.3.8 Write a program that, using one `for` loop and one `if` statement, prints the

integers from 1,000 to 2,000 with five integers per line. Hint: Use the % operation.

1.3.9 Write a program that takes an integer N as a command-line argument, uses Math.random() to print N uniform random values between 0 and 1, and then prints their average value (see EXERCISE 1.2.30).

1.3.10 Describe what happens when you try to print a ruler function (see the table on page 57) with a value of N that is too large, such as 100.

1.3.11 Write a program FunctionGrowth that prints a table of the values $\log N$, N, $N \log N$, N^2, N^3, and 2^N for $N = 16, 32, 64, \ldots, 2048$. Use tabs (\t characters) to line up columns.

1.3.12 What are the values of m and n after executing the following code?

```
int n = 123456789;
int m = 0;
while (n != 0)
{
   m = (10 * m) + (n % 10);
   n = n / 10;
}
```

1.3.13 What does the following program print ?

```
int f = 0, g = 1;
for (int i = 0; i <= 15; i++)
{
   System.out.println(f);
   f = f + g;
   g = f - g;
}
```

Solution. Even an expert programmer will tell you that the only way to understand a program like this is to trace it. When you do, you will find that it prints the values 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 134, 233, 377, and 610. These numbers are the first sixteen of the famous *Fibonacci sequence*, which are defined by the following formulas: $F_0 = 0$, $F_1 = 1$, and $F_n = F_{n-1} + F_{n-2}$ for $n > 1$. The Fibonacci sequence arises in a surprising variety of contexts, they have been studied for centuries, and

many of their properties are well-known. For example, the ratio of successive numbers approaches the *golden ratio* ϕ (about 1.618) as n approaches infinity.

1.3.14 Write a program that takes a command-line argument N and prints all the positive powers of two less than or equal to N. Make sure that your program works properly for all values of N. (`Integer.parseInt()` will generate an error if N is too large, and your program should print nothing if N is negative.)

1.3.15 Expand your solution to EXERCISE 1.2.24 to print a table giving the total amount paid and the remaining principal after each monthly payment.

1.3.16 Unlike the harmonic numbers, the sum $1/1^2 + 1/2^2 + ... + 1/N^2$ *does* converge to a constant as N grows to infinity. (Indeed, the constant is $\pi^2/6$, so this formula can be used to estimate the value of π.) Which of the following `for` loops computes this sum? Assume that `N` is an `int` initialized to `1000000` and `sum` is a `double` initialized to `0.0`.

a.
```
for (int i = 1; i <= N; i++) sum += 1 / (i*i);
```

b.
```
for (int i = 1; i <= N; i++) sum += 1.0 / i*i;
```

c.
```
for (int i = 1; i <= N; i++) sum += 1.0 / (i*i);
```

d.
```
for (int i = 1; i <= N; i++) sum += 1 / (1.0*i*i);
```

1.3.17 Show that PROGRAM 1.3.6 implements Newton's method for finding the square root of c. *Hint*: Use the fact that the slope of the tangent to a (differentiable) function $f(x)$ at $x = t$ is $f'(t)$ to find the equation of the tangent line and then use that equation to find the point where the tangent line intersects the x-axis to show that you can use Newton's method to find a root of any function as follows: at each iteration, replace the estimate t by $t - f(t) / f'(t)$.

1.3.18 Using Newton's method, develop a program that takes integers `N` and `k` as command-line arguments and prints the `k`th root of `N` (*Hint*: see EXERCISE 1.3.17).

1.3.19 Modify `Binary` to get a program `Kary` that takes `i` and `k` as command-line arguments and converts `i` to base `k`. Assume that `i` is an integer in Java's `long` data type and that `k` is an integer between 2 and 16. For bases greater than 10, use the letters `A` through `F` to represent the 11th through 16th digits, respectively.

1.3.20 Write a code fragment that puts the binary representation of a positive integer `N` into a `String s`.

Solution. Java has a built-in method `Integer.toBinaryString(N)` for this job, but the point of the exercise is to see how such a method might be implemented. Working from PROGRAM 1.3.7, we get the solution

```
String s = "";
int v = 1;
while (v <= n/2) v = 2*v;
while (v > 0)
{
   if (n < v) { s += 0;         }
   else       { s += 1; n -= v; }
   v = v/2;
}
```

A simpler option is to work from right to left:

```
String s = "";
for (int n = N; n > 0; n /= 2)
   s = (n % 2) + s;
```

Both of these methods are worthy of careful study.

1.3.21 Write a version of `Gambler` that uses two nested `while` loops or two nested `for` loops instead of a `while` loop inside a `for` loop.

1.3.22 Write a program `GamblerPlot` that traces a gambler's ruin simulation by printing a line after each bet in which one asterisk corresponds to each dollar held by the gambler.

1.3.23 Modify `Gambler` to take an extra command-line argument that specifies the (fixed) probability that the gambler wins each bet. Use your program to try to learn how this probability affects the chance of winning and the expected number of bets. Try a value of p close to .5 (say, .48).

1.3.24 Modify `Gambler` to take an extra command-line argument that specifies the number of bets the gambler is willing to make, so that there are three possible

ways for the game to end: the gambler wins, loses, or runs out of time. Add to the output to give the expected amount of money the gambler will have when the game ends. *Extra credit*: Use your program to plan your next trip to Monte Carlo.

1.3.25 Modify `Factors` to print just one copy each of the prime divisors.

1.3.26 Run quick experiments to determine the impact of using the termination condition `(i <= N/i)` instead of `(i < N)` in `Factors` in PROGRAM 1.3.9. For each method, find the largest n such that when you type in an n digit number, the program is sure to finish within 10 seconds.

1.3.27 Write a program `Checkerboard` that takes one command-line argument `N` and uses a loop within a loop to print out a two-dimensional `N`-by-`N` checkerboard pattern with alternating spaces and asterisks.

1.3.28 Write a program `GCD` that finds the greatest common divisor (gcd) of two integers using *Euclid's algorithm*, which is an iterative computation based on the following observation: if `x` is greater than `y`, then if `y` divides `x`, the gcd of `x` and `y` is `y`; otherwise, the gcd of `x` and `y` is the same as the gcd of `x % y` and `y`.

1.3.29 Write a program `RelativelyPrime` that takes one command-line argument `N` and prints out an `N`-by-`N` table such that there is an `*` in row `i` and column `j` if the gcd of `i` and `j` is `1` (`i` and `j` are relatively prime) and a space in that position otherwise.

1.3.30 Write a program `PowersOfK` that takes an integer `k` as command-line argument and prints all the positive powers of `k` in the Java `long` data type. *Note*: The constant `Long.MAX_VALUE` is the value of the largest integer in `long`.

1.3.31 Generate a random point (x, y, z) on the surface of a sphere using Marsaglia's method: Pick a random point (a, b) in the unit disk using the method described at the end of this section. Then, set $x = 2\,a\,\sqrt{1 - a^2 - b^2}$. $y = 2\,b\,\sqrt{1 - a^2 - b^2}$. and $z = 1 - 2\,(a^2 + b^2)$.

Creative Exercises

1.3.32 *Ramanujan's taxi.* Srinivasa Ramanujan was an Indian mathematician who became famous for his intuition for numbers. When the English mathematician G. H. Hardy came to visit him one day, Hardy remarked that the number of his taxi was 1729, a rather dull number. To which Ramanujan replied, "No, Hardy! No, Hardy! It is a very interesting number. It is the smallest number expressible as the sum of two cubes in two different ways." Verify this claim by writing a program that takes a command-line argument `N` and prints out all integers less than or equal to `N` that can be expressed as the sum of two cubes in two different ways. In other words, find distinct positive integers a, b, c, and d such that $a^3 + b^3 = c^3 + d^3$. Use four nested `for` loops.

1.3.33 *Checksum.* The International Standard Book Number (ISBN) is a 10-digit code that uniquely specifies a book. The rightmost digit is a checksum digit that can be uniquely determined from the other 9 digits, from the condition that $d_1 + 2d_2 + 3d_3 + ... + 10d_{10}$ must be a multiple of 11 (here d_i denotes the ith digit from the right). The checksum digit d_i can be any value from 0 to 10. The ISBN convention is to use the character `'X'` to denote 10. Example: the checksum digit corresponding to `020131452` is `5` since 5 is the only value of x between `0` and `10` for which

$$10 \cdot 0 + 9 \cdot 2 + 8 \cdot 0 + 7 \cdot 1 + 6 \cdot 3 + 5 \cdot 1 + 4 \cdot 4 + 3 \cdot 5 + 2 \cdot 2 + 1 \cdot x$$

is a multiple of 11. Write a program that takes a 9-digit integer as a command-line argument, computes the checksum, and prints out the the ISBN number.

1.3.34 *Counting primes.* Write a program `PrimeCounter` that takes a command-line argument `N` and finds the number of primes less than or equal to `N`. Use it to print out the number of primes less than or equal to 10 million. *Note*: if you are not careful, your program may not finish in a reasonable amount of time!

1.3.35 *2D random walk.* A two-dimensional random walk simulates the behavior of a particle moving in a grid of points. At each step, the random walker moves north, south, east, or west with probability equal to 1/4, independent of previous moves. Write a program `RandomWalker` that takes a command-line argument `N` and estimates how long it will take a random walker to hit the boundary of a `2N`-by-`2N` square centered at the starting point.

1.3.36 *Exponential function.* Assume that `x` is a positive variable of type `double`. Write a code fragment that uses the Taylor series expansion to set the value of `sum` to $e^x = 1 + x + x^2/2! + x^3/3! + \ldots$.

Solution. The purpose of this exercise is to get you to think about how a library function like `Math.exp()` might be implemented in terms of elementary operators. Try solving it, then compare your solution with the one developed here.

We start by considering the problem of computing one term. Suppose that `x` and `term` are variables of type `double` and `n` is a variable of type `int`. The following code fragment sets `term` to $x^N / N!$ using the direct method of having one loop for the numerator and another loop for the denominator, then dividing the results:

```
double num = 1.0, dem = 1.0;
for (int i = 1; i <= n; i++) num *= x;
for (int i = 1; i <= n; i++) den *= i;
double term = num/den;
```

A better approach is to use just a single `for` loop:

```
double term = 1.0;
for (i = 1; i <= n; i++) term *= x/i;
```

Besides being more compact and elegant, the latter solution is preferable because it avoids inaccuracies caused by computing with huge numbers. For example, the two-loop approach breaks down for values like $x = 10$ and $N = 100$ because 100! is too large to represent as a `double`.

To compute e^x, we nest this `for` loop within another `for` loop:

```
double term = 1.0;
double sum = 0.0;
for (int n = 1; sum != sum + term; n++)
{
   sum += term;
   term = 1.0;
   for (int i = 1; i <= n; i++) term *= x/i;
}
```

The number of times the loop iterates depends on the relative values of the next term and the accumulated sum. Once the value of the sum stops changing, we

leave the loop. (This strategy is more efficient than using the termination condition `(term > 0)` because it avoids a significant number of iterations that do not change the value of the sum.) This code is effective, but it is inefficient because the inner `for` loop recomputes all the values it computed on the previous iteration of the outer `for` loop. Instead, we can make use of the term that was added in on the previous loop iteration and solve the problem with a single `for` loop:

```
double term = 1.0;
double sum = 0.0;
for (int n = 1; sum != sum + term; n++)
{
   sum += term;
   term *= x/n;
}
```

1.3.37 *Trigonometric functions.* Write two programs, `Sin` and `Cos`, that compute the sine and cosine functions using their Taylor series expansions $\sin x = x - x^3/3! + x^5/5! - \ldots$ and $\cos x = 1 - x^2/2! + x^4/4! - \ldots$.

1.3.38 *Experimental analysis.* Run experiments to determine the relative costs of `Math.exp()` and the methods from EXERCISE 1.3.36 for computing e^x: the direct method with nested `for` loops, the improvement with a single `for` loop, and the latter with the termination condition `(term > 0)`. Use trial-and-error with a command-line argument to determine how many times your computer can perform each computation in 10 seconds.

1.3.39 *Pepys problem.* In 1693 Samuel Pepys asked Isaac Newton which is more likely: getting 1 at least once when rolling a fair die six times or getting 1 at least twice when rolling it 12 times. Write a program that could have provided Newton with a quick answer.

1.3.40 *Game simulation.* In the 1970s game show *Let's Make a Deal*, a contestant is presented with three doors. Behind one of them is a valuable prize. After the contestant chooses a door, the host opens one of the other two doors (never revealing the prize, of course). The contestant is then given the opportunity to switch to the other unopened door. Should the contestant do so? Intuitively, it might seem that

the contestant's initial choice door and the other unopened door are equally likely to contain the prize, so there would be no incentive to switch. Write a program `MonteHall` to test this intuition by simulation. Your program should take a command-line argument `N`, play the game `N` times using each of the two strategies (switch or do not switch), and print the chance of success for each of the two strategies.

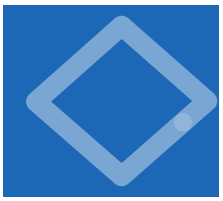
1.3.41 *Median-of-5.* Write a program that takes five distinct integers from the command line and prints the median value (the value such that two of the others are smaller and two are larger). *Extra credit*: Solve the problem with a program that compares values fewer than seven times for any given input.

1.3.42 *Sorting three numbers.* Suppose that the variables `a`, `b`, `c`, and `t` are all of the same numeric primitive type. Prove that the following code puts `a`, `b`, and `c` in ascending order:

```
if (a > b) { t = a; a = b; b = t; }
if (a > c) { t = a; a = c; c = t; }
if (b > c) { t = b; b = c; c = t; }
```

1.3.43 *Chaos.* Write a program to study the following simple model for population growth, which might be applied to study fish in a pond, bacteria in a test tube, or any of a host of similar situations. We suppose that the population ranges from 0 (extinct) to 1 (maximum population that can be sustained). If the population at time t is x, then we suppose the population at time $t + 1$ to be $rx(1-x)$, where the argument r, known as the *fecundity parameter*, controls the rate of growth. Start with a small population—say, $x = 0.01$—and study the result of iterating the model, for various values of r. For which values of r does the population stabilize at $x = 1 - 1/r$? Can you say anything about the population when r is 3.5? 3.8? 5?

1.3.44 *Euler's sum-of-powers conjecture.* In 1769 Leonhard Euler formulated a generalized version of Fermat's Last Theorem, conjecturing that at least n nth powers are needed to obtain a sum that is itself an nth power, for $n > 2$. Write a program to disprove Euler's conjecture (which stood until 1967), using a quintuply nested loop to find four positive integers whose 5th power sums to the 5th power of another positive integer. That is, find a, b, c, d, and e such that $a^5 + b^5 + c^5 + d^5 = e^5$. Use the `long` data type.



1.4 Arrays

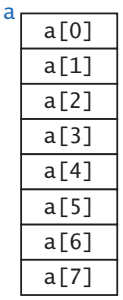
In this section, we consider a fundamental programming construct known as the *array*. The primary purpose of an array is to facilitate storing and manipulating large quantities of data. Arrays play an essential role in many data processing tasks. They also correspond to vectors and matrices, which are widely used in science and in scientific programming. We will consider basic properties of array processing in Java, with many examples illustrating why they are useful.

1.4.1 Sampling without replacement . . . 94
1.4.2 Coupon collector simulation 98
1.4.3 Sieve of Eratosthenes 100
1.4.4 Self-avoiding random walks 109

Programs in this section

An array stores a sequence of values that are all of the same type. Processing such a set of values is very common. We might have exam scores, stock prices, nucleotides in a DNA strand, or characters in a book. Each of these examples involve a large number of values that are all of the same type.

We want not only to store values but also directly access each individual value. The method that we use to refer to individual values in an array is numbering and then *indexing* them. If we have N values, we think of them as being numbered from 0 to $N-1$. Then, we can unambiguously specify one of them by referring to the ith value for any value of i from 0 to $N-1$. To refer to the `i`th value in an array `a`, we use the notation `a[i]`, pronounced *a sub i*. This Java construct is known as a *one-dimensional* array.



An array

The one-dimensional array is our first example in this book of a *data structure* (a method for organizing data). We also consider in this section a more complicated data structure known as *a two-dimensional array*. Data structures play an essential role in modern programming—Chapter 4 is largely devoted to the topic.

Typically, when we have a large amount of data to process, we first put all of the data into one or more arrays. Then we use array indexing to refer to individual values and to process the data. We consider such applications when we discuss data input in Section 1.5 and in the case study that is the subject of Section 1.6. In this section, we expose the basic properties of arrays by considering examples where our programs first populate arrays with computed values from experimental studies and then process them.

Arrays in Java Making an array in a Java program involves three distinct steps:

- Declare the array name and type.
- Create the array.
- Initialize the array values.

To declare the array, you need to specify a name and the type of data it will contain. To create it, you need to specify its size (the number of values). For example, the following code makes an array of `N` numbers of type `double`, all initialized to `0.0`:

```
double[] a;
a = new double[N];
for (int i = 0; i < N; i++)
   a[i] = 0.0;
```

The first statement is the array declaration. It is just like a declaration of a variable of the corresponding primitive type except for the square brackets following the type name, which specify that we are declaring an array. The second statement creates the array. This action is unnecessary for variables of a primitive type (so we have not seen a similar action before), but it is needed for all other types of data in Java (see SECTION 3.1). In the code in this book, we normally keep the array length in an integer variable `N`, but any integer-valued expression will do. The `for` statement initializes the `N` array values. We refer to each value by putting its index in brackets after the array name. This code sets all of the array entries to the value `0.0`.

When you begin to write code that uses an array, you must be sure that your code declares, creates, and initializes it. Omitting one of these steps is a common programming mistake. For economy in code, we often take advantage of Java's default array initialization convention and combine all three steps into a single statement. For example, the following statement is equivalent to the code above:

```
double[] a = new double[N];
```

The code to the left of the equal sign constitutes the declaration; the code to the right constitutes the creation. The `for` loop is unnecessary in this case because the default initial value of variables of type `double` in a Java array is `0.0`, but it would be required if a nonzero value were desired. The default initial value is zero for all numbers and `false` for type `boolean`. For `String` and other non-primitive types, the default is the value `null`, which you will learn about in CHAPTER 3.

After declaring and creating an array, you can refer to any individual value anywhere you would use a variable name in a program by enclosing an integer in-

dex in braces after the array name. We refer to the `i`th item with the code `a[i]`. The explicit initialization code shown earlier is an example of such a use. The obvious advantage of using arrays is to avoid explicitly naming each variable individually. Using an array index is virtually the same as appending the index to the array name: for example, if we wanted to process eight variables of type `double`, we could declare each of them individually with the declaration

```
double a0, a1, a2, a3, a4, a5, a6, a7;
```

and then refer to them as `a0`, `a1` and so forth instead of declaring them with `double[] a = new double[8]` and referring to them as `a[0]`, `a[1]`, and so forth. But naming dozens of individual variables in this way would be cumbersome and naming millions is untenable.

As an example of code that uses arrays, consider using arrays to represent *vectors*. We consider vectors in detail in SECTION 3.3; for the moment, think of a vector as a sequence of real numbers. The *dot product* of two vectors (of the same length) is the sum of the products of their corresponding components. The dot product of two vectors that are represented as one-dimensional arrays `x[]` and `y[]` that are each of length 3 is the expression `x[0]*y[0] + x[1]*y[1] + x[2]*y[2]`. If we represent the two vectors as one-dimensional arrays `x[]` and `y[]` that are each of length `N` and of type `double`, the dot product is easy to compute:

```
double sum = 0.0;
for (int i = 0; i < N; i++)
   sum += x[i]*y[i];
```

i	x[i]	y[i]	x[i]*y[i]	sum
				0
0	.30	.50	.15	.15
1	.60	.10	.06	.21
2	.10	.40	.04	.25
				.25

Trace of dot product computation

The simplicity of coding such computations makes the use of arrays the natural choice for all kinds of applications. (Note that when we use the notation `x[]`, we are referring to the whole array, as opposed to `x[i]`, which is a reference to the `i`th entry.)

The accompanying table has many examples of array-processing code, and we will consider even more examples later in the book, because arrays play a central role in processing data in many applications. Before considering more sophisticated examples, we describe a number of important characteristics of programming with arrays.

create an array with random values	`double[] a = new double[N];`
	`for (int i = 0; i < N; i++)`
	`  a[i] = Math.random();`

print the array values, one per line	`for (int i = 0; i < N; i++)` `  System.out.println(a[i]);`

find the maximum of the array values	`double max = Double.NEGATIVE_INFINITY;`
	`for (int i = 0; i < N; i++)`
	`  if (a[i] > max) max = a[i];`

compute the average of the array values	`double sum = 0.0;`
	`for (int i = 0; i < N; i++)`
	`  sum += a[i];`
	`double average = sum / N;`

copy to another array	`double[] b = new double[N];`
	`for (int i = 0; i < N; i++)`
	`  b[i] = a[i];`

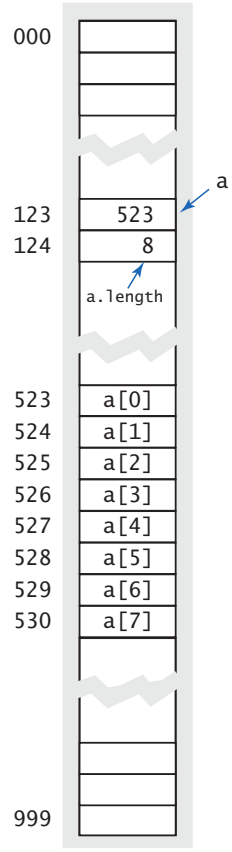
reverse the elements within an array	`for (int i = 0; i < N/2; i++)`
	`{`
	`  double temp = b[i];`
	`  b[i] = b[N-1-i];`
	`  b[N-i-1] = temp;`
	`}`

Typical array-processing code (for arrays of N `double` *values)*

Zero-based indexing. We always refer to the first element of an array as `a[0]`, the second as `a[1]`, and so forth. It might seem more natural to you to refer to the first element as `a[1]`, the second value as `a[2]`, and so forth, but starting the indexing with 0 has some advantages and has emerged as the convention used in most modern programming languages. Misunderstanding this convention often leads to *off-by one-errors* that are notoriously difficult to avoid and debug, so be careful!

Array length. Once we create an array, its size is fixed. The reason that we need to explicitly create arrays at runtime is that the Java compiler cannot know how much space to reserve for the array at compile time (as it can for primitive-type values). Our convention is to keep the size of the array in a variable `N` whose value can be set at runtime (usually it is the value of a command-line argument). Java's standard mechanism is to allow a program to refer to the length of an array `a[]` with the code `a.length`; we normally use `N` to create the array, or set the value of `N` to `a.length`. Note that the last element of an array is always `a[a.length-1]`.

Memory representation. Arrays are fundamental data structures in that they have a direct correspondence with memory systems on virtually all computers. The elements of an array are stored consecutively in memory, so that it is easy to quickly access any array value. Indeed, we can view memory itself as a giant array. On modern computers, memory is implemented in hardware as a sequence of indexed memory locations that each can be quickly accessed with an appropriate index. When referring to computer memory, we normally refer to a location's index as its *address*. It is convenient to think of the name of the array—say, `a`—as storing the memory address of the first element of the array `a[0]`. For the purposes of illustration, suppose that the computer's memory is organized as 1,000 values, with addresses from 000 to 999. (This simplified model ignores the fact that array elements can occupy differing amounts of memory depending on their type, but you can ignore such details for the moment.) Now, suppose that an array of eight elements is stored in memory locations 523 through 530. In such a situation, Java would store the memory address (index) of the first array value somewhere else in memory, along with the array length. We refer to the address as a *pointer* and think of it as *pointing to* the referenced memory location. When we specify `a[i]`, the compiler generates code that accesses the desired value by adding the index `i` to the memory address of the array `a[]`. For example, the Java code `a[4]` would generate machine code that finds the value at memory location `523 + 4 = 527`. Accessing element `i` of an array is an efficient operation because it simply requires adding two integers and then referencing memory—just two elementary operations. Extending the model to handle different-sized array elements just involves multiplying the index by the element size before adding to the array address.



Memory representation

Memory allocation. When you use `new` to create an array, Java reserves space in memory for it. This process is called *memory allocation*. The same process is required for all variables that you use in a program. We call attention to it now because it is your responsibility to use `new` to allocate memory for an array before accessing any of its elements. If you fail to adhere to this rule, you will get a compile-time *uninitialized variable* error. Java automatically initializes all of the values in an array when it is created. You should remember that the time required to create an array is proportional to its length.

Bounds checking. As already indicated, you must be careful when programming with arrays. It is your responsibility to use legal indices when accessing an array element. If you have created an array of size `N` and use an index whose value is less than `0` or greater than `N-1`, your program will terminate with an `ArrayIndexOutOfBounds` run-time exception. (In many programming languages, such *buffer overflow* conditions are not checked by the system. Such unchecked errors can and do lead to debugging nightmares, but it is also not uncommon for such an error to go unnoticed and remain in a finished program. You might be surprised to know that such a mistake can be exploited by a hacker to take control of a system, even your personal computer, to spread viruses, steal personal information, or wreak other malicious havoc.) The error messages provided by Java may seem annoying to you at first, but they are small price to pay to have a more secure program.

Setting array values at compile time. When we have a small number of literal values that we want to keep in array, we can declare and initialize it by listing the values between curly braces, separated by commas. For example, we might use the following code in a program that processes playing cards.

```
String[] suit = { "Clubs", "Diamonds", "Hearts", "Spades" };

String[] rank =
{
   "2", "3", "4", "5", "6", "7", "8", "9", "10",
   "Jack", "Queen", "King", "Ace"
};
```

After creating the two arrays, we can use them to print out a random card name, such as `Queen of Clubs`, as follows:

```
int i = (int) (Math.random() * rank.length);
int j = (int) (Math.random() * suit.length);
System.out.println(rank[i] + " of " + suit[j]);
```

This code uses the idiom introduced in SECTION 1.2 to generate random indices and then uses the indices to pick strings out of the arrays. Whenever the values of all array entries are known at compile time (and the size of the array is not too large) it makes sense to use this method of initializing the array—just put all the values in braces on the right hand side of an assignment in the array declaration. Doing so implies array creation, so the `new` keyword is not needed.

Setting array values at runtime. A more typical situation is when we wish to compute the values to be stored in an array. In this case, we can use array names with indices in the same way we use variable names on the left side of assignment statements. For example, we might use the following code to initialize an array of size 52 that represents a deck of playing cards, using the two arrays just defined:

```
String[] deck = new String[suit.length * rank.length];
for (int i = 0; i < suit.length; i++)
   for (int j = 0; j < rank.length; j++)
      deck[rank.length*i + j] = rank[i] + " of " + suit[j];
```

After this code has been executed, if you were to print out the contents of deck in order from `deck[0]` through `deck[51]` using `System.out.println()`, you would get the sequence

```
2 of Clubs
2 of Diamonds
2 of Hearts
2 of Spades
3 of Clubs
3 of Diamonds
...
Ace of Hearts
Ace of Spades
```

Exchange. Frequently, we wish to exchange two values in an array. Continuing our example with playing cards, the following code exchanges the cards at position `i` and `j` using the same idiom that we traced as our first example of the use of assignment statements in SECTION 1.2:

```
String t = deck[i];
deck[i] = deck[j];
deck[j] = t;
```

When we use this code, we are assured that we are perhaps changing the *order* of the values in the array but not the *set* of values in the array. When `i` and `j` are equal, the array is unchanged. When `i` and `j` are not equal, the values `a[i]` and `a[j]` are found in different places in the array. For example, if we were to use this code with `i` equal to `1` and `j` equal to `4` in the `deck` array of the previous example, it would leave `3 of Clubs` in `deck[1]` and `2 of Diamonds` in `deck[4]`.

Shuffle. The following code shuffles our deck of cards:

```
int N = deck.length;
for (int i = 0; i < N; i++)
{
   int r = i + (int) (Math.random() * (N-i));
   String t = deck[i];
   deck[i] = deck[r];
   deck[r] = t;
}
```

Proceeding from left to right, we pick a random card from `deck[i]` through `deck[N-1]` (each card equally likely) and exchange it with `deck[i]`. This code is more sophisticated than it might seem: First, we ensure that the cards in the deck after the shuffle are the same as the cards in the deck before the shuffle by using the exchange idiom. Second, we ensure that the shuffle is random by choosing uniformly from the cards not yet chosen.

Sampling without replacement. In many situations, we want to draw a random sample from a set such that each member of the set appears at most once in the sample. Drawing numbered ping-pong balls from a basket for a lottery is an example of this kind of sample, as is dealing a hand from a deck of cards. `Sample` (PROGRAM 1.4.1) illustrates how to sample, using the basic operation underlying shuffling. It takes command-line arguments `M` and `N` and creates a *permutation* of size `N` (a rearrangement of the integers from `0` to `N-1`) whose first `M` entries com-

i	r	perm														
		0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
		0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	9	9	1	2	3	4	5	6	7	8	0	10	11	12	13	14	15
1	5	9	5	2	3	4	1	6	7	8	0	10	11	12	13	14	15
2	13	9	5	13	3	4	1	6	7	8	0	10	11	12	2	14	15
3	5	9	5	13	1	4	3	6	7	8	0	10	11	12	2	14	15
4	11	9	5	13	1	11	3	6	7	8	0	10	4	12	2	14	15
5	8	9	5	13	1	11	8	6	7	3	0	10	4	12	2	14	15
		9	5	13	1	11	8	6	7	3	0	10	4	12	2	14	15

Trace of `java Sample 6 16`

Program 1.4.1 Sampling without replacement

```
public class Sample
{
   public static void main(String[] args)
   {  // Print a random sample of M integers
      // from 0 ... N-1 (no duplicates).
      int M = Integer.parseInt(args[0]);
      int N = Integer.parseInt(args[1]);
      int[] perm = new int[N];

      // Initialize perm[].
      for (int j = 0; j < N; j++)
          perm[j] = j;

      // Take sample.
      for (int i = 0; i < M; i++)
      {  // Exchange perm[i] with a random element to its right.
         int r = i + (int) (Math.random() * (N-i));
         int t = perm[r];
         perm[r] = perm[i];
         perm[i] = t;
      }

      // Print sample.
      for (int i = 0; i < M; i++)
         System.out.print(perm[i] + " ");
      System.out.println();
   }
}
```

M	*sample size*
N	*range*
perm[]	*permutation of 0 to N-1*

This program takes two command-line arguments M *and* N *and produces a sample of* M *of the integers from* 0 *to* N-1*. This process is useful, not just in state and local lotteries, but in scientific applications of all sorts. If the first argument is equal to the second, the result is a random permutation of the integers from* 0 *to* N-1*. If the first argument is greater than the second, the program will terminate with an* ArrayOutOfBounds *exception.*

```
% java Sample 6 16
9 5 13 1 11 8

% java Sample 10 1000
656 488 298 534 811 97 813 156 424 109

% java Sample 20 20
6 12 9 8 13 19 0 2 4 5 18 1 14 16 17 3 7 11 10 15
```

prise a random sample. The accompanying trace of the contents of the `perm[]` array at the end of each iteration of the main loop (for a run where the values of M and N are 6 and 16, respectively) illustrates the process.

If the values of `r` are chosen such that each value in the given range is equally likely, then `perm[0]` through `perm[M-1]` are a random sample at the end of the process (even though some elements might move multiple times) because each element in the sample is chosen by taking each item not yet sampled, with equal probability for each choice. One important reason to explicitly compute the permutation is that we can use it to print out a random sample of *any* array by using the elements of the permutation as indices into the array. Doing so is often an attractive alternative to actually rearranging the array because it may need to be in order for some other reason (for instance, a company might wish to draw a random sample from a list of customers that is kept in alphabetical order). To see how this trick works, suppose that we wish to draw a random poker hand from our `deck[]` array, constructed as just described. We use the code in `Sample` with $N = 52$ and $M = 5$ and replace `perm[i]` with `deck[perm[i]]` in the `System.out.print()` statement (and change it to `println()`), resulting in output such as the following:

```
3 of Clubs
Jack of Hearts
6 of Spades
Ace of Clubs
10 of Diamonds
```

Sampling like this is widely used as the basis for statistical studies in polling, scientific research, and many other applications, whenever we want to draw conclusions about a large population by analyzing a small random sample.

Precomputed values. One simple application of arrays is to save values that you have computed, for later use. As an example, suppose that you are writing a program that performs calculations using small values of the harmonic numbers (see PROGRAM 1.3.5). An efficient approach is to save the values in an array, as follows:

```
double[] H = new double[N];
for (int i = 1; i < N; i++)
   H[i] = H[i-1] + 1.0/i;
```

Then you can just use the code `H[i]` to refer to any of the values. Precomputing values in this way is an example of a *space-time tradeoff*: by investing in space (to save

the values) we save time (since we do not need to recompute them). This method is not effective if we need values for huge N, but it is very effective if we need values for small N many different times.

Simplifying repetitive code. As an example of another simple application of arrays, consider the following code fragment, which prints out the name of a month given its number (1 for January, 2 for February, and so forth):

```
if      (m ==  1) System.out.println("Jan");
else if (m ==  2) System.out.println("Feb");
else if (m ==  3) System.out.println("Mar");
else if (m ==  4) System.out.println("Apr");
else if (m ==  5) System.out.println("May");
else if (m ==  6) System.out.println("Jun");
else if (m ==  7) System.out.println("Jul");
else if (m ==  8) System.out.println("Aug");
else if (m ==  9) System.out.println("Sep");
else if (m == 10) System.out.println("Oct");
else if (m == 11) System.out.println("Nov");
else if (m == 12) System.out.println("Dec");
```

We could also use a `switch` statement, but a much more compact alternative is to use a `String` array consisting of the names of each month:

```
String[] months =
{
   "", "Jan", "Feb", "Mar", "Apr", "May", "Jun",
       "Jul", "Aug", "Sep", "Oct", "Nov", "Dec"
};
System.out.println(months[m]);
```

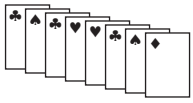
This technique would be especially useful if you needed to access the name of a month by its number in several different places in your program. Note that we intentionally waste one slot in the array (element `0`) to make `months[1]` correspond to January, as required.

Assignments and equality tests. Suppose that you have created the two arrays `a[]` and `b[]`. What does it mean to assign one to the other with the code `a = b;`? Similarly, what does it mean to test whether the two arrays are equal with the code `(a == b)`? The answers to these questions may not be what you first assume, but if you think about the array memory representation, you will see that Java's interpretation

of these operations makes sense: An assignment makes the names `a` and `b` refer to the same array. The alternative would be to have an implied loop that assigns each value in `b` to the corresponding value in `a`. Similarly, an equality test checks whether the two names refer to the same array. The alternative would be to have an implied loop that tests whether each value in one array is equal to the corresponding value in the other array. In both cases, the implementation in Java is very simple: it just performs the standard operation as if the array name were a variable whose value is the memory address of the array. Note that there are many other operations that you might want to perform on arrays: for example, it would be nice in some applications to say `a = a + b` and have it mean "add the corresponding element in `b[]` to each element in `a[]`," but that statement is not legal in Java. Instead, we write an explicit loop to perform all the additions. We will consider in detail Java's mechanism for satisfying such higher-level programming needs in SECTION 3.2. In typical applications, we use this mechanism, so we rarely need to use Java's assignments and equality tests with arrays.

WITH THESE BASIC DEFINITIONS AND EXAMPLES out of the way, we can now consider two applications that both address interesting classical problems and illustrate the fundamental importance of arrays in efficient computation. In both cases, the idea of using data to index into an array plays a central role and enables a computation that would not otherwise be feasible.

Coupon collector Suppose that you have a shuffled deck of cards and you turn them face up, one by one. How many cards do you need to turn up before you have seen one of each suit? How many cards do you need to turn up before seeing one of each value? These are examples of the famous *coupon collector* problem. In general, suppose that a trading card company issues trading cards with N different possible cards: how many do you have to collect before you have all N possibilities, assuming that each possibility is equally likely for each card that you collect?



Coupon collection

Coupon collecting is no toy problem. For example, it is very often the case that scientists want to know whether a sequence that arises in nature has the same characteristics as a random sequence. If so, that fact might be of interest; if not, further investigation may be warranted to look for patterns that might be of importance. For example, such tests are used by scientists to decide which parts of genomes are worth studying. One effective test for whether a sequence is truly random is

Program 1.4.2 Coupon collector simulation

```
public class CouponCollector
{
   public static void main(String[] args)
   {  // Generate random values in (0..N] until finding each one.
      int N = Integer.parseInt(args[0]);
      boolean[] found = new boolean[N];
      int cardcnt = 0, valcnt = 0;
      while (valcnt < N)
      {  // Generate another value.
         int val = (int) (Math.random() * N);
         cardcnt++;
         if (!found[val])
         {
            valcnt++;
            found[val] = true;
         }
      }  // N different values found.
      System.out.println(cardcnt);
   }
}
```

N	*range*
cardcnt	*values generated*
valcnt	*different values found*
found[]	*table of found values*

This program simulates coupon collection by taking a command-line argument N *and generating random numbers between* 0 *and* N-1 *until getting every possible value.*

```
% java CouponCollector 1000
6583
% java CouponCollector 1000
6477
% java CouponCollector 1000000
12782673
```

the *coupon collector test*: compare the number of elements that need to be examined before all values are found against the corresponding number for a uniformly random sequence. `CouponCollector` (PROGRAM 1.4.2) is an example program that simulates this process and illustrates the utility of arrays. It takes the value of *N* from the command line and generates a sequence of random integer values between 0

and $N-1$ using the code `(int) (Math.random() * N)` (see PROGRAM 1.2.5). Each value represents a card: for each card, we want to know if we have seen that value before. To maintain that knowledge, we use an array `found[]`, which uses the card value as an index: `found[i]` is `true` if we have seen a card with value `i` and `false` if we have not. When we get a new card that is represented by the integer `val`, we check whether we have seen its value before simply by accessing `found[val]`. The computation consists of keeping count of the number of distinct values seen and the number of cards generated and printing the latter when the former gets to `N`.

val	found 0	1	2	3	4	5	valcnt	cardcnt
	F	F	F	F	F	F	0	0
2	F	F	T	F	F	F	1	1
0	T	F	T	F	F	F	2	2
4	T	F	T	F	T	F	3	3
0	T	F	T	F	T	F	3	4
1	T	T	T	F	T	F	4	5
2	T	T	T	F	T	F	4	6
5	T	T	T	F	T	T	5	7
0	T	T	T	F	T	T	5	8
1	T	T	T	F	T	T	5	9
3	T	T	T	T	T	T	6	10

Trace for a typical run of `java CouponCollector 6`

As usual, the best way to understand a program is to consider a trace of the values of its variables for a typical run. It is easy to add code to `CouponCollector` that produces a trace that gives the values of the variables at the end of the `while` loop for a typical run. In the accompanying figure, we use `F` for the value `false` and `T` for the value `true` to make the trace easier to follow. Tracing programs that use large arrays can be a challenge: when you have an array of size N in your program, it represents N variables, so you have to list them all. Tracing programs that use `Math.random()` also can be a challenge because you get a different trace every time you run the program. Accordingly, we check relationships among variables carefully. Here, note that `valcnt` always is equal to the number of `true` values in `found[]`.

Without arrays, we could not contemplate simulating the coupon collector process for huge N; with arrays it is easy to do so. We will see many examples of such processes throughout the book.

Sieve of Eratosthenes Prime numbers play an important role in mathematics and computation, including cryptography. A *prime number* is an integer greater than one whose only positive divisors are one and itself. The prime counting function $\pi(N)$ is the number of primes less than or equal to N. For example, $\pi(25) = 9$ since the first nine primes are 2, 3, 5, 7, 11, 13, 17, 19, and 23. This function plays a central role in number theory.

Program 1.4.3 Sieve of Eratosthenes

```
public class PrimeSieve
{
   public static void main(String[] args)
   {  // Print the number of primes <= N.
      int N = Integer.parseInt(args[0]);
      boolean[] isPrime = new boolean[N+1];
      for (int i = 2; i <= N; i++)
         isPrime[i] = true;

      for (int i = 2; i <= N/i; i++)
      {  if (isPrime[i])
         {  // Mark multiples of i as nonprime.
            for (int j = i; j <= N/i; j++)
               isPrime[i * j] = false;
         }
      }

      // Count the primes.
      int primes = 0;
      for (int i = 2; i <= N; i++)
         if (isPrime[i]) primes++;
      System.out.println(primes);
   }
}
```

N	*argument*
isPrime[i]	*is* i *prime?*
primes	*prime counter*

This program takes a command-line argument N *and computes the number of primes less than or equal to* N*. To do so, it computes an array of boolean values with* isPrime[i] *set to* true *if* i *is prime, and to* false *otherwise. First, it sets to* true *all array elements in order to indicate that no numbers are initially known to be nonprime. Then it sets to* false *array elements corresponding to indices that are known to be nonprime (multiples of known primes). If* a[i] *is still* true *after all multiples of smaller primes have been set to* false*, then we know* i *to be prime. The termination test in the second* for *loop is* i <= N/i *instead of the naive* i <= N *because any number with no factor less than* N/i *has no factor greater than* N/i*, so we do not have to look for such factors. This improvement makes it possible to run the program for large* N*.*

```
% java PrimeSieve 25
9
% java PrimeSieve 100
25
% java PrimeSieve 1000000000
50847534
```

i	isPrime																							
	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25
	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T
2	T	T	F	T	F	T	F	T	F	T	F	T	F	T	F	T	F	T	F	T	F	T	F	T
3	T	T	F	T	F	T	F	F	F	T	F	T	F	F	F	T	F	T	F	F	F	T	F	T
5	T	T	F	T	F	T	F	F	F	T	F	T	F	F	F	T	F	T	F	F	F	T	F	F
	T	T	F	T	F	T	F	F	F	T	F	T	F	F	F	T	F	T	F	F	F	T	F	F

Trace of `java PrimeSieve 25`

One approach to counting primes is to use a program like `Factors` (PROGRAM 1.3.9). Specifically, we could modify the code in `Factors` to set a boolean value to be `true` if a given number is prime and `false` otherwise (instead of printing out factors), then enclose that code in a loop that increments a counter for each prime number. This approach is effective for small N, but becomes too slow as N grows.

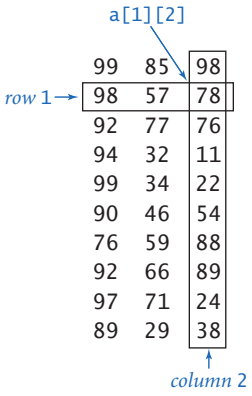
`PrimeSieve` (PROGRAM 1.4.3) takes a command-line integer `N` and computes the prime count using a technique known as the *Sieve of Eratosthenes*. The program uses a `boolean` array `isPrime[]` to record which integers are prime. The goal is to set `isPrime[i]` to `true` if `i` is prime, and to `false` otherwise. The sieve works as follows: Initially, set all array elements to `true`, indicating that no factors of any integer have yet been found. Then, repeat the following steps as long as `i <= N/i`:

- Find the next smallest `i` for which no factors have been found.
- Leave `isPrime[i]` as `true` since `i` has no smaller factors.
- Set the `isPrime[]` entries for all multiples of `i` to be `false`.

When the nested `for` loop ends, we have set the `isPrime[]` entries for all nonprimes to be `false` and have left the `isPrime[]` entries for all primes as `true`. With one more pass through the array, we can count the number of primes less than or equal to `N`. As usual, it is easy to add code to print a trace. For programs such as `PrimeSieve`, you have to be a bit careful—it contains a nested `for-if-for`, so you have to pay attention to the braces in order to put the print code in the correct place. Note that we stop when `i > N/i`, just as we did for `Factors`.

With `PrimeSieve`, we can compute $\pi(N)$ for large N, limited primarily by the maximum array size allowed by Java. This is another example of a space-time tradeoff. Programs like `PrimeSieve` play an important role in helping mathematicians to develop the theory of numbers, which has many important applications.

Two-dimensional arrays In many applications, a convenient way to store information is to use a table of numbers organized in a rectangular table and refer to *rows* and *columns* in the table. For example, a teacher might need to maintain a table with a row corresponding to each student and a column corresponding to each assignment, a scientist might need to maintain a table of experimental data with rows corresponding to experiments and columns corresponding to various outcomes, or a programmer might want to prepare an image for display by setting a table of pixels to various grayscale values or colors.



Anatomy of a two-dimensional array

The mathematical abstraction corresponding to such tables is a *matrix*; the corresponding Java construct is a *two-dimensional array*. You are likely to have already encountered many applications of matrices and two-dimensional arrays, and you will certainly encounter many others in science, in engineering, and in computing applications, as we will demonstrate with examples throughout this book. As with vectors and one-dimensional arrays, many of the most important applications involve processing large amounts of data, and we defer considering those applications until we consider input and output, in SECTION 1.5.

Extending Java array constructs to handle two-dimensional arrays is straightforward. To refer to the element in row `i` and column `j` of a two-dimensional array `a[][]`, we use the notation `a[i][j]`; to declare a two-dimensional array, we add another pair of brackets; and to create the array, we specify the number of rows followed by the number of columns after the type name (both within brackets), as follows:

```
double[][] a = new double[M][N];
```

We refer to such an array as an *M*-by-*N* array. By convention, the first dimension is the number of rows and the second is the number of columns. As with one-dimensional arrays, Java initializes all entries in arrays of numbers to zero and in arrays of boolean values to `false`.

Initialization. Default initialization of two-dimensional arrays is useful because it masks more code than for one-dimensional arrays. The following code is equivalent to the single-line create-and-initialize idiom that we just considered:

```
double[][] a;
a = new double[M][N];
for (int i = 0; i < M; i++)
{  // Initialize the ith row.
   for (int j = 0; j < N; j++)
      a[i][j] = 0.0;
}
```

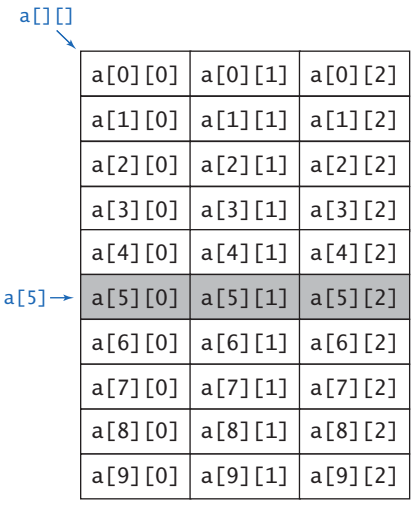
This code is superfluous when initializing to zero, but the nested `for` loops are needed to initialize to some other value(s). As you will see, this code is a model for the code that we use to access or modify each element of a two-dimensional array.

Output. We use nested `for` loops for many array-processing operations. For example, to print an M-by-N array in the familiar tabular format, we would use the following code

```
for (int i = 0; i < M; i++)
{  // Print the ith row.
   for (int j = 0; j < N; j++)
      System.out.print(a[i][j] + " ");
   System.out.println();
}
```

regardless of the array elements' type. If desired, we could add code to embellish the output with row and column numbers (see EXERCISE 1.4.6), but Java programmers typically tabulate arrays with row numbers running top to bottom from 0 and column number running left to right from 0. Generally, we also do so and do not bother to use labels.

Memory representation. Java represents a two-dimensional array as an array of arrays. A matrix with M rows and N columns is actually an array of length M, each entry of which is an array of length N. In a two-dimensional Java array `a[][]`, we can use the code `a[i]` to refer to the `i`th row (which is a one-dimensional array), but we have no corresponding way to refer to a column.



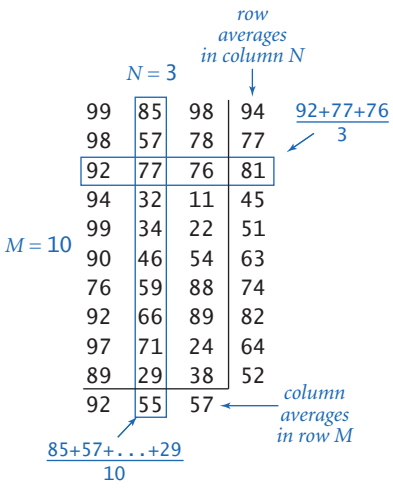
A 10-by-3 array

Setting values at compile time. The Java method for initializing an array of values at compile time follows immediately from the representation. A two-dimensional array is an array of rows, each row initialized as a one-dimensional array. To initialize a two-dimensional array, we enclose in braces a list of terms to initialize the rows, separated by commas. Each term in the list is itself a list: the values for the array elements in the row, enclosed in braces and separated by commas.

```
int[][] a =
{
   { 99, 85, 98,  0 },
   { 98, 57, 78,  0 },
   { 92, 77, 76,  0 },
   { 94, 32, 11,  0 },
   { 99, 34, 22,  0 },
   { 90, 46, 54,  0 },
   { 76, 59, 88,  0 },
   { 92, 66, 89,  0 },
   { 97, 71, 24,  0 },
   { 89, 29, 38,  0 },
   {  0,  0,  0,  0 }
};
```

Compile-time initialization of a two-dimensional array

Spreadsheets. One familiar use of arrays is a *spreadsheet* for maintaining a table of numbers. For example, a teacher with M students and N test grades for each student might maintain an $(M+1)$-by-$(N+1)$ array, reserving the last column for each student's average grade and the last row for the average test grades. Even though we typically do such computations within specialized applications, it is worthwhile to study the underlying code as an introduction to array processing. To compute the average grade for each student (average values for each row), sum the entries for each row and divide by N. The row-by-row order in which this code processes the matrix



Compute row averages

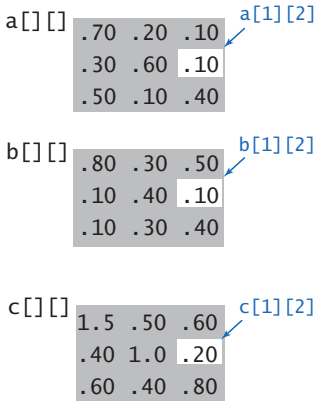
```
for (int i = 0; i < M; i++)
{  // Compute average for row i
   double sum = 0.0;
   for (int j = 0; j < N; j++)
      sum += a[i][j];
   a[i][N] = (int) Math.round(sum/N);
}
```

Compute column averages

```
for (int j = 0; j < N; j++)
{  // Compute average for column j
   double sum = 0.0;
   for (int i = 0; i < M; i++)
      sum += a[i][j];
   a[M][j] = (int) Math.round(sum/M);
}
```

Typical spreadsheet calculations

entries is known as *row-major* order. Similarly, to compute the average test grade (average values for each column), sum the entries for each column and divide by *M*. The column-by-column order in which this code processes the matrix entries is known as *column-major* order.



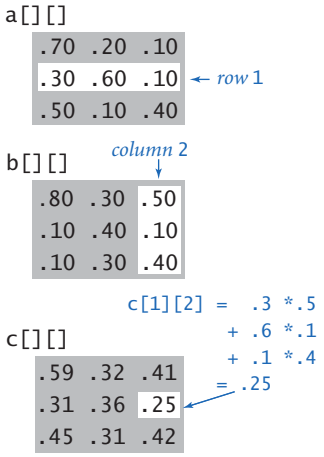
Matrix addition

Matrix operations. Typical applications in science and engineering involve representing matrices as two-dimensional arrays and then implementing various mathematical operations with matrix operands. Again, even though such processing is often done within specialized applications, it is worthwhile for you to understand the underlying computation. For example, we can *add* two *N*-by-*N* matrices as follows:

```
double[][] c = new double[N][N];
for (int i = 0; i < N; i++)
   for (int j = 0; j < N; j++)
      c[i][j] = a[i][j] + b[i][j];
```

Similarly, we can *multiply* two matrices. You may have learned matrix multiplication, but if you do not recall or are not familiar with it, the Java code below for square matrices is essentially the same as the mathematical definition. Each entry `c[i][j]` in the product of `a[]` and `b[]` is computed by taking the dot product of row `i` of `a[]` with column `j` of `b[]`.

```
double[][] c = new double[N][N];
for (int i = 0; i < N; i++)
{
   for (int j = 0; j < N; j++)
   {
      // Compute dot product of row i and column j.
      for (int k = 0; k < N; k++)
         c[i][j] += a[i][k]*b[k][j];
   }
}
```



Matrix multiplication

The definition extends to matrices that are not necessarily square (see Exercise 1.4.17).

Special cases of matrix multiplication. Two special cases of matrix multiplication are important. These special cases occur when one of the dimensions of one of the matrices is 1, so it may be viewed as a vector. We have *matrix-vector multiplication*, where we multiply an M-by-N matrix by a *column vector* (an N-by-1 matrix) to get an M-by-1 column vector result (each entry in the result is the dot product of the corresponding row in the matrix with the operand vector). The second case is *vector-matrix multiplication*, where we multiply a *row vector* (a 1-by-M matrix) by an M-by-N matrix to get a 1-by-N row vector result (each entry in the result is the dot product of the operand vector with the corresponding column in the matrix). These operations provide a succinct way to express numerous matrix calculations. For example, the row-average computation for such a spreadsheet with M rows and N columns is equivalent to a matrix-vector multiplication where the column vector has M entries all equal to $1/M$. Similarly, the column-average computation in such a spreadsheet is equivalent to a vector-matrix multiplication where the row vector has N entries all equal to $1/N$. We return to vector-matrix multiplication in the context of an important application at the end of this chapter.

Matrix-vector multiplication `a[][]*x[] = b[]`

```
for (int i = 0; i < M; i++)
{  // Dot product of row i and x[].
   for (int j = 0; j < N; j++)
      b[i] += a[i][j]*x[j];
}
```

a[][]			x[]	b[]
99	85	98		94
98	57	78		77
92	77	76		81
94	32	11		45
99	34	22	.33	51
90	46	54	.33	63
76	59	88	.33	74
92	66	89		82
97	71	24		64
89	29	38		52

row averages (b[])

Vector-matrix multiplication `y[]*a[][] = c[]`

```
for (int j = 0; j < N; j++)
{  // Dot product of y[] and column j.
   for (int i = 0; i < M; i++)
      c[j] += y[i]*a[i][j];
}
```

y[] [.1 .1 .1 .1 .1 .1 .1 .1 .1 .1]

	a[][]		
	99	85	98
	98	57	78
	92	77	76
	94	32	11
	99	34	22
	90	46	54
	76	59	88
	92	66	89
	97	71	24
	89	29	38
c[]	92	55	57

column averages (c[])

Matrix-vector and vector-matrix multiplication

Ragged arrays. There is actually no requirement that all rows in a two-dimensional array have the same length—an array with rows of nonuniform length is known as a *ragged array* (see Exercise 1.4.32 for an example application). The possibility of ragged arrays creates the need for more care in crafting array-processing code. For example, this code prints the contents of a ragged array:

```
for (int i = 0; i < a.length; i++)
{
   for (int j = 0; j < a[i].length; j++)
      System.out.print(a[i][j] + " ");
   System.out.println();
}
```

This code tests your understanding of Java arrays, so you should take the time to study it. In this book, we normally use square or rectangular arrays, whose dimension is given by a variable `M` or `N`. Code that uses `a[i].length` in this way is a clear signal to you that an array is ragged.

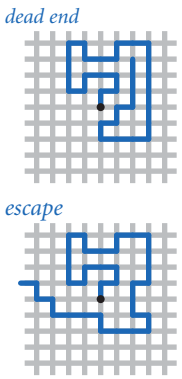
Multidimensional arrays. The same notation extends to allow us to write code using arrays that have any number of dimensions. For instance, we can declare and initialize a three-dimensional array with the code

```
double[][][] a = new double[N][N][N];
```

and then refer to an entry with code like `a[i][j][k]`, and so forth.

TWO-DIMENSIONAL ARRAYS PROVIDE A NATURAL REPRESENTATION for matrices, which are omnipresent in science, mathematics, and engineering. They also provide a natural way to organize large amounts of data, a key factor in spreadsheets and many other computing applications. Through Cartesian coordinates, two- and three-dimensional arrays also provide the basis for a models of the physical world. We consider their use in all three arenas throughout this book.

Example: self-avoiding random walks Suppose that you leave your dog in the middle of a large city whose streets form a familiar grid pattern. We assume that there are N north-south streets and N east-west streets all regularly spaced and fully intersecting in a pattern known as a *lattice*. Trying to escape the city, the dog makes a random choice of which way to go at each intersection, but knows by scent to avoid visiting any place previously visited. But it is possible for the dog to get stuck in a dead end where there is no choice but to revisit some intersection. What is the chance that this will happen? This amusing problem is a simple example of a famous model known as the *self-avoiding random walk*, which has important scientific applications in the study of polymers and in statistical mechanics, among many others. For example, you can see



Self-avoiding walks

that this process models a chain of material growing a bit at a time, until no growth is possible. To better understand such processes, scientists seek to understand the properties of self-avoiding walks.

The dog's escape probability is certainly dependent on the size of the city. In a tiny 5-by-5 city, it is easy to convince yourself that the dog is certain to escape. But what are the chances of escape when the city is large? We are also interested in other parameters. For example, how long is the dog's path, on the average? How often does the dog come within one block of a previous position other than the one just left, on the average? How often does the dog come within one block of escaping? These sorts of properties are important in the various applications just mentioned.

`SelfAvoidingWalk` (PROGRAM 1.4.4) is a simulation of this situation that uses a two-dimensional `boolean` array, where each entry represents an intersection. The value `true` indicates that the dog has visited the intersection; `false` indicates that the dog has not visited the intersection. The path starts in the center and takes random steps to places not yet visited until getting stuck or escaping at a boundary. For simplicity, the code is written so that if a random choice is made to go to a spot that has already been visited, it takes no action, trusting that some subsequent random choice will find a new place (which is assured because the code explicitly tests for a dead end and leaves the loop in that case).

Note that the code depends on Java initializing all of the array entries to `false` for each experiment. It also exhibits an important programming technique where we code the loop exit test in the `while` statement as a *guard* against an illegal statement in the body of the loop. In this case, the `while` loop continuation test serves as a guard against an out-of-bounds array access within the loop. This corresponds to checking whether the dog has escaped. Within the loop, a successful dead-end test results in a `break` out of the loop.

As you can see from the sample runs, the unfortunate truth is that your dog is nearly certain to get trapped in a dead end in a large city. If you are interested in learning more about self-avoiding walks, you can find several suggestions in the exercises. For example, the dog is virtually certain to escape in the three-dimensional version of the problem. While this is an intuitive result that is confirmed by our tests, the development of a mathematical model that explains the behavior of self-avoiding walks is a famous open problem: despite extensive research, no one knows a succinct mathematical expression for the escape probability, the average length of the path, or any other important parameter.

Program 1.4.4 Self-avoiding random walks

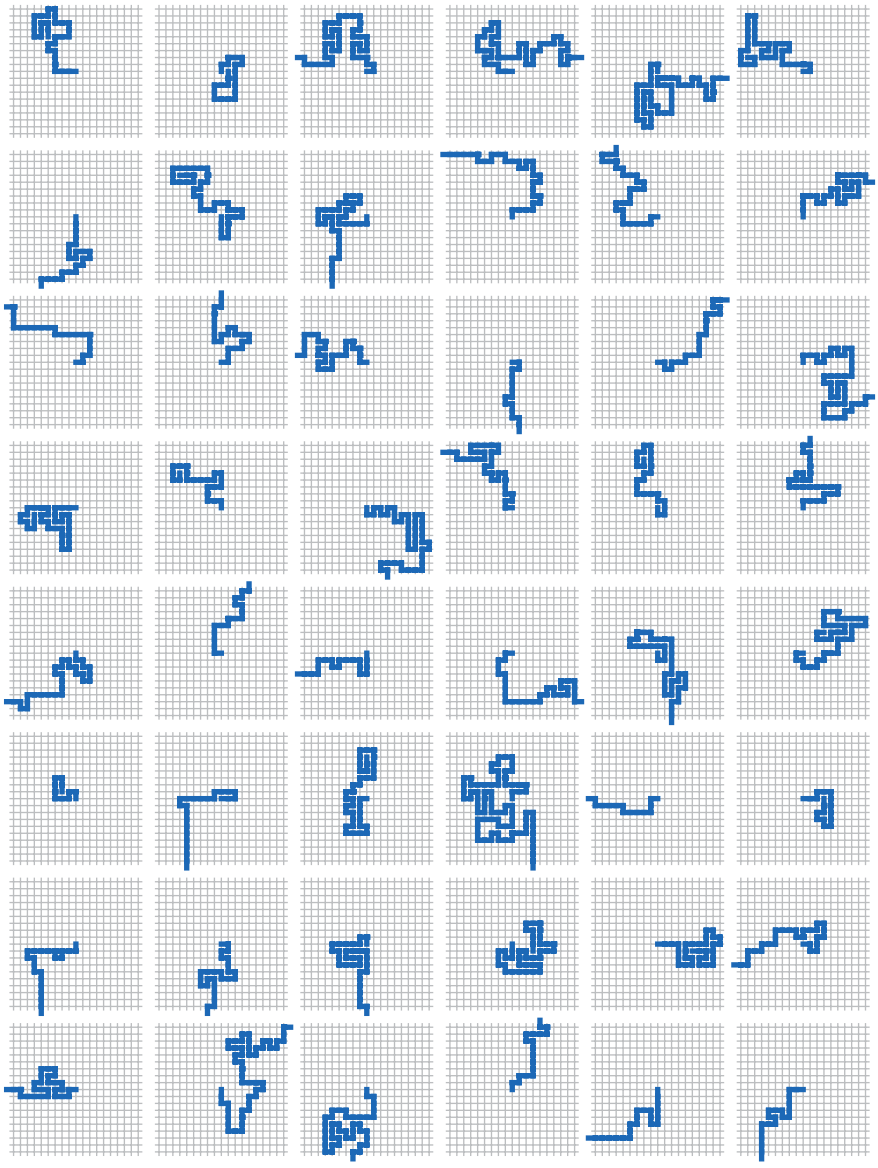
```
public class SelfAvoidingWalk
{
   public static void main(String[] args)
   {  // Do T random self-avoiding walks
      //   in an N-by-N lattice
      int N = Integer.parseInt(args[0]);
      int T = Integer.parseInt(args[1]);
      int deadEnds = 0;
      for (int t = 0; t < T; t++)
      {
         boolean[][] a = new boolean[N][N];
         int x = N/2, y = N/2;
         while (x > 0 && x < N-1 && y > 0 && y < N-1)
         {  // Check for dead end and make a random move.
            a[x][y] = true;
            if (a[x-1][y] && a[x+1][y] && a[x][y-1] && a[x][y+1])
            {  deadEnds++; break;  }
            double r = Math.random();
            if      (r < 0.25) { if (!a[x+1][y]) x++; }
            else if (r < 0.50) { if (!a[x-1][y]) x--; }
            else if (r < 0.75) { if (!a[x][y+1]) y++; }
            else if (r < 1.00) { if (!a[x][y-1]) y--; }
         }
      }
      System.out.println(100*deadEnds/T + "% dead ends");
   }
}
```

N	*lattice size*
T	*number of trials*
deadEnds	*trials resulting in a dead end*
a[][]	*intersections visited*
x, y	*current position*
r	*random number in (0, 1)*

This program takes command-line arguments N *and* T *and computes* T *self-avoiding walks in an* N*-by-*N *lattice. For each walk, it creates a boolean array, starts the walk in the center, and continues until either a dead end or a boundary is reached. The result of the computation is the percentage of dead ends. As usual, increasing the number of experiments increases the precision of the results.*

```
% java SelfAvoidingWalk 5 100
0% dead ends
% java SelfAvoidingWalk 20 100
36% dead ends
% java SelfAvoidingWalk 40 100
80% dead ends
% java SelfAvoidingWalk 80 100
98% dead ends
% java SelfAvoidingWalk 160 100
100% dead ends
```

```
% java SelfAvoidingWalk 5 1000
0% dead ends
% java SelfAvoidingWalk 20 1000
32% dead ends
% java SelfAvoidingWalk 40 1000
70% dead ends
% java SelfAvoidingWalk 80 1000
95% dead ends
% java SelfAvoidingWalk 160 1000
100% dead ends
```



Self-avoiding random walks in a 21-by-21 grid

Summary Arrays are the fourth basic element (after assignments, conditionals, and loops) found in virtually every programming language, completing our coverage of basic Java constructs. As you have seen with the sample programs that we have presented, you can write programs that can solve all sorts of problems using just these constructs.

Arrays are prominent in many of the programs that we consider, and the basic operations that we have discussed here will serve you well in addressing many programming tasks. When you are not using arrays explicitly (and you are sure to be doing so frequently), you will be using them implicitly, because all computers have a memory that is conceptually equivalent to an indexed array.

The fundamental ingredient that arrays add to our programs is a potentially huge increase in the size of a program's *state*. The state of a program can be defined as the information you need to know to understand what a program is doing. In a program without arrays, if you know the values of the variables and which statement is the next to be executed, you can normally determine what the program will do next. When we trace a program, we are essentially tracking its state. When a program uses arrays, however, there can be too huge a number of values (each of which might be changed in each statement) for us to effectively track them all. This difference makes writing programs with arrays more of a challenge than writing programs without them.

Arrays directly represent vectors and matrices, so they are of direct use in computations associated with many basic problems in science and engineering. Arrays also provide a succinct notation for manipulating a potentially huge amount of data in a uniform way, so they play a critical role in any application that involves processing large amounts of data, as you will see throughout this book.

Q&A

Q. Some Java programmers use `int a[]` instead of `int[] a` to declare arrays. What's the difference?

A. In Java, both are legal and equivalent. The former is how arrays are declared in C. The latter is the preferred style in Java since the type of the variable `int[]` more clearly indicates that it is an *array* of integers.

Q. Why do array indices start at `0` instead of `1`?

A. This convention originated with machine-language programming, where the address of an array element would be computed by adding the index to the address of the beginning of an array. Starting indices at 1 would entail either a waste of space at the beginning of the array or a waste of time to subtract the 1.

Q. What happens if I use a negative number to index an array?

A. The same thing as when you use an index that is too big. Whenever a program attempts to index an array with an index that is not between zero and the array length minus one, Java will issue an `ArrayIndexOutOfBoundsException` and terminate the program.

Q. What happens when I compare two arrays with `(a == b)`?

A. The expression evaluates to `true` only if `a[]` and `b[]` refer to the same array, not if they have the same sequence of elements. Unfortunately, this is rarely what you want.

Q. If `a[]` is an array, why does `System.out.println(a)` print out a hexadecimal integer, like `@f62373` , instead of the elements of the array?

A. Good question. It is printing out the memory address of the array, which, unfortunately, is rarely what you want.

Q. What other pitfalls should I watch out for when using arrays?

A. It is very important to remember that Java *always* initializes arrays when you create them, so that *creating an array takes time proportional to the size of the array*.

Exercises

1.4.1 Write a program that declares and initializes an array `a[]` of size `1000` and accesses `a[1000]`. Does your program compile? What happens when you run it?

1.4.2 Describe and explain what happens when you try to compile a program with the following statement:

```
int N = 1000;
int[] a = new int[N*N*N*N];
```

1.4.3 Given two vectors of length `N` that are represented with one-dimensional arrays, write a code fragment that computes the *Euclidean distance* between them (the square root of the sums of the squares of the differences between corresponding entries).

1.4.4 Write a code fragment that reverses the order of a one-dimensional array `a[]` of `String` values. Do not create another array to hold the result. *Hint*: Use the code in the text for exchanging two elements.

1.4.5 What is wrong with the following code fragment?

```
int[] a;
for (int i = 0; i < 10; i++)
   a[i] = i * i;
```

Solution. It does not allocate memory for `a[]` with `new`. This code results in a `variable a might not have been initialized` compile-time error.

1.4.6 Write a code fragment that prints the contents of a two-dimensional boolean array, using * to represent `true` and a space to represent `false`. Include row and column numbers.

1.4.7 What does the following code fragment print?

```
int[] a = new int[10];
for (int i = 0; i < 10; i++)
   a[i] = 9 - i;
for (int i = 0; i < 10; i++)
   a[i] = a[a[i]];
for (int i = 0; i < 10; i++)
   System.out.println(a[i]);
```

1.4.8 What values does the following code put in the array `a[]`?

```
int N = 10;
int[] a = new int[N];
a[0] = 1;
a[1] = 1;
for (int i = 2; i < N; i++)
   a[i] = a[i-1] + a[i-2];
```

1.4.9 What does the following code fragment print?

```
int[] a = { 1, 2, 3 };
int[] b = { 1, 2, 3 };
System.out.println(a == b);
```

1.4.10 Write a program `Deal` that takes an command-line argument `N` and prints `N` poker hands (five cards each) from a shuffled deck, separated by blank lines.

1.4.11 Write code fragments to create a two-dimensional array `b[][]` that is a copy of an existing two-dimensional array `a[][]`, under each of the following assumptions:

a. `a[][]` is square

b. `a[][]` is rectangular

c. `a[][]` may be ragged

Your solution to *b* should work for *a*, and your solution to *c* should work for both *b* and *a*, but your code should get progressively more complicated.

1.4.12 Write a code fragment to print the *transposition* (rows and columns changed) of a square two-dimensional array. For the example spreadsheet array in the text, you code would print the following:

```
99  98  92  94  99  90  76  92  97  89
85  57  77  32  34  46  59  66  71  29
98  78  76  11  22  54  88  89  24  38
```

1.4.13 Write a code fragment to transpose a square two-dimensional array *in place* without creating a second array.

1.4.14 Write a program that takes an integer *N* from the command line and creates an *N*-by-*N* boolean array `a[][]` such that `a[i][j]` is `true` if `i` and `j` are relatively prime (have no common factors), and `false` otherwise. Use your solution to EXERCISE 1.4.6 to print the array. *Hint*: Use sieving.

1.4.15 Write a program that computes the product of two square matrices of boolean values, using the *or* operation instead of + and the *and* operation instead of *.

1.4.16 Modify the spreadsheet code fragment in the text to compute a *weighted* average of the rows, where the weights of each test score are in a one-dimensional array `weights[]`. For example, to assign the last of the three tests in our example to be twice the weight of the others, you would use

```
double[] weights = { .25, .25, .50 };
```

Note that the weights should sum to 1.

1.4.17 Write a code fragment to multiply two rectangular matrices that are not necessarily square. *Note*: For the dot product to be well-defined, the number of columns in the first matrix must be equal to the number of rows in the second matrix. Print an error message if the dimensions do not satisfy this condition.

1.4.18 Modify `SelfAvoidingWalk` (PROGRAM 1.4.4) to calculate and print the average length of the paths as well as the dead-end probability. Keep separate the average lengths of escape paths and dead-end paths.

1.4.19 Modify `SelfAvoidingWalk` to calculate and print the average area of the smallest axis-oriented rectangle that encloses the path. Keep separate statistics for escape paths and dead-end paths.

Creative Exercises

1.4.20 *Dice simulation.* The following code computes the exact probability distribution for the sum of two dice:

```
double[] dist = new double[13];
for (int i = 1; i <= 6; i++)
   for (int j = 1; j <= 6; j++)
      dist[i+j] += 1.0;

for (int k = 1; k <= 12; k++)
   dist[k] /= 36.0;
```

The value `dist[k]` is the probability that the dice sum to `k`. Run experiments to validate this calculation simulating N dice throws, keeping track of the frequencies of occurrence of each value when you compute the sum of two random integers between 1 and 6. How large does N have to be before your empirical results match the exact results to three decimal places?

1.4.21 *Longest plateau.* Given an array of integers, find the length and location of the longest contiguous sequence of equal values where the values of the elements just before and just after this sequence are smaller.

1.4.22 *Empirical shuffle check.* Run computational experiments to check that our shuffling code works as advertised. Write a program `ShuffleTest` that takes command-line arguments M and N, does N shuffles of an array of size M that is initialized with `a[i] = i` before each shuffle, and prints an M-by-M table such that row `i` gives the number of times `i` wound up in position `j` for all `j`. All entries in the array should be close to N/M.

1.4.23 *Bad shuffling.* Suppose that you choose a random integer between `0` and `N-1` in our shuffling code instead of one between `i` and `N-1`. Show that the resulting order is *not* equally likely to be one of the $N!$ possibilities. Run the test of the previous exercise for this version.

1.4.24 *Music shuffling.* You set your music player to shuffle mode. It plays each of the N songs before repeating any. Write a program to estimate the likelihood that you will not hear any sequential pair of songs (that is, song 3 does not follow song 2, song 10 does not follow song 9, and so on).

1.4.24 *Minima in permutations.* Write a program that takes an integer `N` from the command line, generates a random permutation, prints the permutation, and prints the number of left-to-right minima in the permutation (the number of times an element is the smallest seen so far). Then write a program that takes integers `M` and `N` from the command line, generates `M` random permutations of size `N`, and prints the average number of left-to-right minima in the permutations generated. *Extra credit*: Formulate a hypothesis about the number of left-to-right minima in a permutation of size `N`, as a function of `N`.

1.4.25 *Inverse permutation.* Write a program that reads in a permutation of the integers `0` to `N-1` from `N` command-line arguments and prints the inverse permutation. (If the permutation is in an array `a[]`, its inverse is the array `b[]` such that `a[b[i]] = b[a[i]] = i`.) Be sure to check that the input is a valid permutation.

1.4.26 *Hadamard matrix.* The N-by-N Hadamard matrix $H(N)$ is a boolean matrix with the remarkable property that any two rows differ in exactly $N/2$ entries. (This property makes it useful for designing error-correcting codes.) $H(1)$ is a 1-by-1 matrix with the single entry `true`, and for $N>1$, $H(2N)$ is obtained by aligning four copies of $H(N)$ in a large square, and then inverting all of the entries in the lower right N-by-N copy, as shown in the following examples (with `T` representing `true` and `F` representing `false`, as usual).

$H(1)$	$H(2)$	$H(4)$
T	T T	T T T T
	T F	T F T F
		T T F F
		T F F T

Write a program that takes one command-line argument N and prints $H(N)$. Assume that N is a power of 2.

1.4.27 *Rumors.* Alice is throwing a party with N other guests, including Bob. Bob starts a rumor about Alice by telling it to one of the other guests. A person hearing this rumor for the first time will immediately tell it to one other guest, chosen at random from all the people at the party except Alice and the person from whom

they heard it. If a person (including Bob) hears the rumor for a second time, he or she will not propagate it further. Write a program to estimate the probability that everyone at the party (except Alice) will hear the rumor before it stops propagating. Also calculate an estimate of the expected number of people to hear the rumor.

1.4.28 *Find a duplicate.* Given an array of `N` elements with each element between `1` and `N`, write an algorithm to determine whether there are any duplicates. You do not need to preserve the contents of the given array, but do not use an extra array.

1.4.29 *Counting primes.* Compare `PrimeSieve` with the method that we used to demonstrate the `break` statement, at the end of SECTION 1.3. This is a classic example of a time-space tradeoff: `PrimeSieve` is fast, but requires a `boolean` array of size N; the other approach uses only two integer variables, but is substantially slower. Estimate the magnitude of this difference by finding the value of N for which this second approach can complete the computation in about the same time as `java PrimeSeive 1000000`.

1.4.30 *Minesweeper.* Write a program that takes 3 command-line arguments M, N, and p and produces an M-by-N boolean array where each entry is occupied with probability p. In the minesweeper game, occupied cells represent bombs and empty cells represent safe cells. Print out the array using an asterisk for bombs and a period for safe cells. Then, replace each safe square with the number of neighboring bombs (above, below, left, right, or diagonal) and print out the solution.

```
* * . . .        * * 1 0 0
. . . . .        3 3 2 0 0
. * . . .        1 * 1 0 0
```

Try to write your code so that you have as few special cases as possible to deal with, by using an $(M+2)$-by-$(N+2)$ boolean array.

1.4.31 *Self-avoiding walk length.* Suppose that there is no limit on the size of the grid. Run experiments to estimate the average walk length.

1.4.32 *Three-dimensional self-avoiding walks.* Run experiments to verify that the dead-end probability is 0 for a three-dimensional self-avoiding walk and to compute the average walk length for various values of N.

1.4.33 *Random walkers.* Suppose that N random walkers, starting in the center of an N-by-N grid, move one step at a time, choosing to go left, right, up, or down with equal probability at each step. Write a program to help formulate and test a hypothesis about the number of steps taken before all cells are touched.

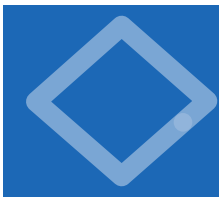
1.4.34 *Bridge hands.* In the game of bridge, four players are dealt hands of 13 cards each. An important statistic is the distribution of the number of cards in each suit in a hand. Which is the most likely, 5-3-3-2, 4-4-3-2, or 4-3-3-3?

1.4.35 *Birthday problem.* Suppose that people enter an empty room until a pair of people share a birthday. On average, how many people will have to enter before there is a match? Run experiments to estimate the value of this quantity. Assume birthdays to be uniform random integers between 0 and 364.

1.4.36 *Coupon collector.* Run experiments to validate the classical mathematical result that the expected number of coupons needed to collect N values is about NH_N. For example, if you are observing the cards carefully at the blackjack table (and the dealer has enough decks randomly shuffled together), you will wait until about 235 cards are dealt, on average, before seeing every card value.

1.4.37 *Binomial coefficients.* Write a program that builds and prints a two-dimensional ragged array `a` such that `a[N][k]` contains the probability that you get exactly `k` heads when you toss a coin `N` times. Take a command-line argument to specify the maximum value of `N`. These numbers are known as the *binomial distribution*: if you multiply each entry in row i by 2^N, you get the *binomial coefficients* (the coefficients of x^k in $(x+1)^N$) arranged in *Pascal's triangle*. To compute them, start with `a[N][0] = 0` for all `N` and `a[1][1] = 1`, then compute values in successive rows, left to right, with `a[N][k] = (a[N-1][k] + a[N-1][k-1])/2`.

Pascal's triangle	*binomial distribution*
1	1
1 1	1/2 1/2
1 2 1	1/4 1/2 1/4
1 3 3 1	1/8 3/8 3/8 1/8
1 4 6 4 1	1/16 1/4 3/8 1/4 1/16



1.5 Input and Output

In this section we extend the set of simple abstractions (command-line input and standard output) that we have been using as the interface between our Java programs and the outside world to include *standard input*, *standard drawing*, and *standard audio*. Standard input makes it convenient for us to write programs that process arbitrary amounts of input and to interact with our programs; standard drawing makes it possible for us to work with graphical representations of images, freeing us from having to encode everything as text; and standard audio adds sound. These extensions are easy to use, and you will find that they bring you to yet another new world of programming.

1.5.1 Generating a random sequence. . . 122
1.5.2 Interactive user input 129
1.5.3 Averaging a stream of numbers. . . 130
1.5.4 A simple filter 134
1.5.5 Input-to-drawing filter 139
1.5.6 Bouncing ball 145
1.5.7 Digital signal processing 150

Programs in this section

The abbreviation *I/O* is universally understood to mean *input/output*, a collective term that refers to the mechanisms by which programs communicate with the outside world. Your computer's operating system controls the physical devices that are connected to your computer. To implement the standard I/O abstractions, we use libraries of methods that interface to the operating system.

You have already been accepting argument values from the command line and printing strings in a terminal window; the purpose of this section is to provide you with a much richer set of tools for processing and presenting data. Like the `System.out.print()` and `System.out.println()` methods that you have been using, these methods do not implement mathematical functions—their purpose is to cause some side effect, either on an input device or an output device. Our prime concern is using such devices to get information into and out of our programs.

An essential feature of standard I/O mechanisms is that there is no limit on the amount of input or output data, from the point of view of the program. Your programs can consume input or produce output indefinitely.

One use of standard I/O mechanisms is to connect your programs to *files* on your computer's disk. It is easy to connect standard input, standard output, standard drawing, and standard audio to files. Such connections make it easy to have your Java programs save or load results to files for archival purposes or for later reference by other programs or other applications.

Bird's-eye view The conventional model that we have been using for Java programming has served us since SECTION 1.1. To build context, we begin by briefly reviewing the model.

A Java program takes input values from the command line and prints a string of characters as output. By default, both *command-line input* and *standard output* are associated with the application that takes commands (the one in which you have been typing the `java` and `javac` commands). We use the generic term *terminal window* to refer to this application. This model has proven to be a convenient and direct way for us to interact with our programs and data.

Command-line input. This mechanism, which we have been using to provide input values to our programs, is a standard part of Java programming. All classes have a `main()` method that takes a `String` array `args[]` as its argument. That array is the sequence of command-line arguments that we type, provided to Java by the operating system. By convention, both Java and the operating system process the arguments as strings, so if we intend for an argument to be a number, we use a method such as `Integer.parseInt()` or `Double.parseDouble()` to convert it from `String` to the appropriate type.

Standard output. To print output values in our programs, we have been using the system methods `System.out.println()` and `System.out.print()`. Java puts the results of a program's sequence of calls on these methods into the form of an abstract stream of characters known as *standard output*. By default, the operating system connects standard output to the terminal window. All of the output in our programs so far has been appearing in the terminal window.

For reference, and as a starting point, `RandomSeq` (PROGRAM 1.5.1) is a program that uses this model. It takes a command-line argument N and produces an output sequence of N random numbers between 0 and 1.

NOW WE ARE GOING TO COMPLEMENT command-line input and standard output with three additional mechanisms that address their limitations and provide us with a far more useful programming model. These mechanisms give us a new bird's-eye view of a Java program in which the program converts a standard input stream and a sequence of command-line arguments into a standard output stream, a standard drawing, and a standard audio stream.

Program 1.5.1 Generating a random sequence

```
public class RandomSeq
{
   public static void main(String[] args)
   {  // Print a random sequence of N real values in [0, 1)
      int N = Integer.parseInt(args[0]);
      for (int i = 0; i < N; i++)
         System.out.println(Math.random());
   }
}
```

This program illustrates the conventional model that we have been using so far for Java programming. It takes a command-line argument N and prints N random numbers between 0 and 1. From the program's point of view, there is no limit on the length of the output sequence.

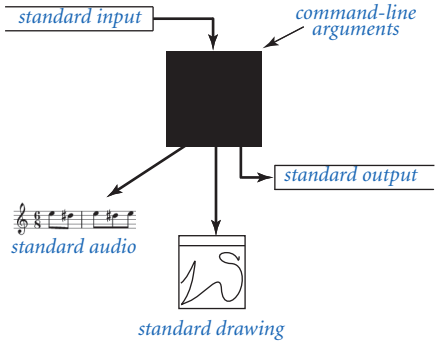
```
% java RandomSeq 1000000
0.2498362534343327
0.5578468691774513
0.5702167639727175
0.32191774192688727
0.6865902823177537
...
```

Standard input. Our class `StdIn` is a library that implements a standard input abstraction to complement the standard output abstraction. Just as you can print a value to standard output at any time during the execution of your program, you can read a value from a standard input stream at any time.

Standard drawing. Our class `StdDraw` allows you to create drawings with your programs. It uses a simple graphics model that allows you to create drawings consisting of points and lines in a window on your computer. `StdDraw` also includes facilities for text, color, and animation.

Standard audio. Our class `StdAudio` allows you to create sound with your programs. It uses a standard format to convert arrays of numbers into sound.

To use both command-line input and standard output, you have been using built-in Java facilities. Java also has built-in facilities that support abstractions like standard input, standard draw, and standard audio, but they are somewhat more complicated to use, so we have developed a simpler interface to them in our `StdIn`, `StdDraw`, and `StdAudio` libraries. To logically complete our programming model, we also include a `StdOut` library. To use these libraries, download `StdIn.java`, `StdOut.java`, `StdDraw.java`, and `StdAudio.java` and place them in the same directory as your program (or use one of the other mechanisms for sharing libraries described on the booksite).



A bird's-eye view of a Java program (revisited)

The standard input and standard output abstractions date back to the development of the UNIX operating system in the 1970s and are found in some form on all modern systems. Although they are primitive by comparison to various mechanisms developed since, modern programmers still depend on them as a reliable way to connect data to programs. We have developed for this book standard draw and standard audio in the same spirit as these earlier abstractions to provide you with an easy way to produce visual and aural output.

Standard output Java's `System.out.print()` and `System.out.println()` methods implement the basic standard output abstraction that we need. Nevertheless, to treat standard input and standard output in a uniform manner (and to provide a few technical improvements), starting in this section and continuing through the rest of the book, we use similar methods that are defined in our `StdOut` library. `StdOut.print()` and `StdOut.println()` are nearly the same as the Java methods that you have been using (see the booksite for a discussion of the differences, which need not concern you now). The `StdOut.printf()` method is a main topic of this section and will be of interest to you now because it gives you more control over the appearance of the output. It was a feature of the C language of the early 1970s that still survives in modern languages because it is so useful.

Since the first time that we printed `double` values, we have been distracted by excessive precision in the printed output. For example, when we use `System.out.print(Math.PI)` we get the output `3.141592653589793`, even though we might

public class StdOut	
void print(String s)	*print* s
void println(String s)	*print* s*, followed by newline*
void println()	*print a new line*
void printf(String f, ...)	*formatted print*

API for our library of static methods for standard output

prefer to see 3.14 or 3.14159. The print() and println() methods present each number to 15 decimal places even when we would be happy with just a few digits of precision. The printf() method is more flexible: it allows us to specify the number of digits and the precision when converting data type values to strings for output. With printf(), we can write StdOut.printf("%7.5f", Math.PI) to get 3.14159, and we can replace System.out.print(t) with

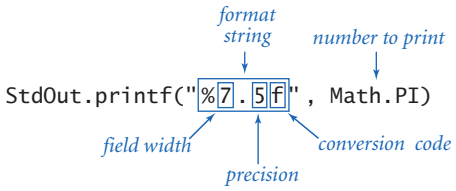
```
StdOut.printf("The square root of %.1f is %.6f", c, t);
```

in Newton (PROGRAM 1.3.6) to get output like

```
The square root of 2.0 is 1.414214
```

Next, we describe the meaning and operation of these statements, along with extensions to handle the other built-in types of data.

Formatted printing basics. In its simplest form, printf() takes two arguments. The first argument is a *format string* that describes how to convert the second argument into a string for output. The simplest type of format string begins with % and ends with a one-letter *conversion code*. The conversion codes that we use most frequently are d (for decimal values from Java's integer types), f (for floating-point values), and s (for String values). Between the % and the conversion code is an integer that specifies the *field width* of the converted value (the number of characters in the converted output string). By default, blanks are added on the left to make the length of the converted output equal to the field width; if we want the blanks on the right, we can insert a minus sign before the field width. (If the



Anatomy of a formatted print statement

converted output string is larger than the field width, the field width is ignored.) Following the width, we have the option of including a period followed by the number of digits to put after the decimal point (the precision) for a `double` value or the number of characters to take from the beginning of the string for a `String` value. The most important thing to remember about using `printf()` is that *the conversion code in the format and the type of the corresponding argument must match.* That is, Java must be able to convert from the type of the argument to the type required by the conversion code. Every type of data can be converted to `String`, but if you write `StdOut.printf("%12d", Math.PI)` or `StdOut.printf("%4.2f", 512)`, you will get an `IllegalFormatConversionException` run-time error.

Format string. The first argument of `printf()` is a `String` that may contain characters other than a format string. Any part of the argument that is not part of a format string passes through to the output, with the format string replaced by the argument value (converted to a string as specified). For example, the statement

```
StdOut.printf("PI is approximately %.2f\n", Math.PI);
```

prints the line

```
PI is approximately 3.14
```

Note that we need to explicitly include the newline character `\n` in the argument in order to print a new line with `printf()`.

type	*code*	*typical literal*	*sample format strings*	*converted string values for output*
`int`	`d`	`512`	`"%14d"` `"%-14d"`	`"           512"` `"512           "`

`double`	`f`
	`e`
		`1595.1680010754388`	`"%14.2f"`
			`"%.7f"`
			`"%14.4e"`
				`"       1595.17"`
				`"1595.1680011"`
				`"    1.5952e+03"`

`String`	`s`	`"Hello, World"`	`"%14s"`
			`"%-14s"`
			`"%-14.5s"`
				`"  Hello, World"`
				`"Hello, World  "`
				`"Hello          "`

Format conventions for `printf()` (see the booksite for many other options)

Multiple arguments. The `printf()` function can take more than two arguments. In this case, the format string will have a format specifier for each additional argument, perhaps separated by other characters to pass through to the output. For example, if you were making payments on a loan, you might use code whose inner loop contains the statements

```
String formats = "%3s  $%6.2f   $%7.2f   $%5.2f\n";
StdOut.printf(formats, month[i], pay, balance, interest);
```

to print the second and subsequent lines in a table like this (see EXERCISE 1.5.14):

```
     payment    balance  interest
Jan  $299.00   $9742.67   $41.67
Feb  $299.00   $9484.26   $40.59
Mar  $299.00   $9224.78   $39.52
...
```

Formatted printing is convenient because this sort of code is much more compact than the string-concatenation code that we have been using.

Standard input Our `StdIn` library takes data from a *standard input stream* that may be empty or may contain a sequence of values separated by whitespace (spaces, tabs, newline characters, and the like). Each value is a `String` or a value from one of Java's primitive types. One of the key features of the standard input stream is that your program *consumes* values when it reads them. Once your program has read a value, it cannot back up and read it again. This assumption is restrictive, but it reflects physical characteristics of some input devices and simplifies implementing the abstraction. The library consists of the nine methods: `isEmpty()`, `readInt()`, `readDouble()`, `readLong()`, `readBoolean()`, `readChar()`, `readString()`, `readLine()`, and `readAll()`. Within the input stream model, these methods are largely self-documenting (the names describe their effect), but their precise operation is worthy of careful consideration, so we will consider several examples in detail.

Typing input. When you use the `java` command to invoke a Java program from the command line, you actually are doing three things: issuing a command to start executing your program, specifying the values of the command line arguments, and beginning to define the standard input stream. The string of characters that you type in the terminal window after the command line *is* the standard input stream. When you type characters, you are interacting with your program. The

public class StdIn	
boolean isEmpty()	`true` *if no more values,* `false` *otherwise*
int readInt()	*read a value of type* `int`
double readDouble()	*read a value of type* `double`
long readLong()	*read a value of type long*
boolean readBoolean()	*read a value of type* `boolean`
char readChar()	*read a value of type* `char`
String readString()	*read a value of type* `String`
String readLine()	*read the rest of the line*
String readAll()	*read the rest of the text*

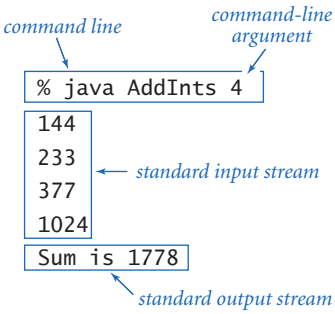
API for our library of static methods for standard input

program *waits* for you to create the standard input stream. For example, consider the following program, which takes a command-line argument *N*, then reads *N* numbers from standard input and adds them:

```
public class AddInts
{
   public static void main(String[] args)
   {
      int N = Integer.parseInt(args[0]);
      int sum = 0;
      for (int i = 0; i < N; i++)
      {
         int value = StdIn.readInt();
         sum += value;
      }
      StdOut.println("Sum is " + sum);
   }
}
```

When you type `java AddInts 4`, after the program takes the command-line argument, it calls the method `StdIn.readInt()` and waits for you to type an integer. Suppose that you want `144` to be the first value. As you type `1`, then `4`, and then `4`, nothing happens, because `StdIn` does not know that you are done typing the integer. But when you then type `<return>` to signify the end of your integer, `StdIn.readInt()` immediately returns the value `144`, which your program adds to `sum`

and then calls `StdIn.readInt()` again. Again, nothing happens until you type the second value: if you type `2`, then `3`, then `3`, and then `<return>` to end the number, `StdIn.readInt()` returns the value `233`, which your program again adds to `sum`. After you have typed four numbers in this way, `AddInts` expects no more input and prints out the sum, as desired.



Anatomy of a command

Input format. If you type `abc` or `12.2` or `true` when `StdIn.readInt()` is expecting an `int`, it will respond with a `NumberFormatException`. The format for each type is the same as you have been using for literal values within Java programs. For convenience, `StdIn` treats strings of consecutive whitespace characters as identical to one space and allows you to delimit your numbers with such strings. It does not matter how many spaces you put between numbers, or whether you enter numbers on one line or separate them with tab characters or spread them out over several lines, (except that your terminal application processes standard input one line at a time, so it will wait until you type `<return>` before sending all of the numbers on that line to standard input). You can mix values of different types in an input stream, but whenever the program expects a value of a particular type, the input stream must have a value of that type.

Interactive user input. `TwentyQuestions` (PROGRAM 1.5.2) is a simple example of a program that interacts with its user. The program generates a random integer and then gives clues to a user trying to guess the number. (As a side note: by using *binary search*, you can always get to the answer in at most twenty questions. See SECTION 4.2.) The fundamental difference between this program and others that we have written is that the user has the ability to change the control flow *while* the program is executing. This capability was very important in early applications of computing, but we rarely write such programs nowadays because modern applications typically take such input through the graphical user interface, as discussed in CHAPTER 3. Even a simple program like `TwentyQuestions` illustrates that writing programs that support user interaction is potentially very difficult because you have to plan for all possible user inputs.

Program 1.5.2 Interactive user input

```
public class TwentyQuestions
{
   public static void main(String[] args)
   {  // Generate a number and answer questions
      // while the user tries to guess the value.
      int N = 1 + (int) (Math.random() * 1000000);
      StdOut.print("I'm thinking of a number ");
      StdOut.println("between 1 and 1,000,000");
      int m = 0;
      while (m != N)
      {  // Solicit one guess and provide one answer
         StdOut.print("What's your guess? ");
         m = StdIn.readInt();
         if (m == N) StdOut.println("You win!");
         if (m < N)  StdOut.println("Too low ");
         if (m > N)  StdOut.println("Too high");
      }
   }
}
```

N	*hidden value*
m	*user's guess*

This program plays a simple guessing game. You type numbers, each of which is an implicit question ("Is this the number?") and the program tells you whether your guess is too high or too low. You can always get it to print `You win!` *with less than twenty questions. To use this program, you need to first download* `StdIn.java` *and* `StdOut.java` *into the same directory as this code (which is in a file named* `TwentyQuestions.java`*).*

```
% java TwentyQuestions
I'm thinking of a number between 1 and 1,000,000
What's your guess? 500000
Too high
What's your guess? 250000
Too low
What's your guess? 375000
Too high
What's your guess? 312500
Too high
What's your guess? 300500
Too low
...
```

Program 1.5.3 Averaging a stream of numbers

```
public class Average
{
   public static void main(String[] args)
   {  // Average the numbers on the input stream.
      double sum = 0.0;
      int cnt = 0;
      while (!StdIn.isEmpty())
      {  // Read a number and cumulate the sum.
         double value = StdIn.readDouble();
         sum += value;
         cnt++;
      }
      double average = sum / cnt;
      StdOut.println("Average is " + average);
   }
}
```

cnt	*count of numbers read*
sum	*cumulated sum*

This program reads in a sequence of real numbers from standard input and prints their average on standard output (provided that the sum does not overflow). From its point of view, there is no limit on the size of the input stream. The commands on the right below use redirection and piping (discussed in the next subsection) to provide 100,000 numbers to average.

```
% java Average
10.0 5.0 6.0
3.0
7.0 32.0
<ctrl-d>
Average is 10.5
```

```
% java RandomSeq 100000 > data.txt
% java Average < data.txt
Average is 0.5010473676174824

% java RandomSeq 100000 | java Average
Average is 0.5000499417963857
```

Processing an arbitrary-size input stream. Typically, input streams are finite: your program marches through the input stream, consuming values until the stream is empty. But there is no restriction of the size of the input stream, and some programs simply process all the input presented to them. `Average` (PROGRAM 1.5.3) is an example that reads in a sequence of real numbers from standard input and prints their average. It illustrates a key property of using an input stream: the length

of the stream is not known to the program. We type all the numbers that we have, and then the program averages them. Before reading each number, the program uses the method `StdIn.isEmpty()` to check whether there are any more numbers in the input stream. How do we signal that we have no more data to type? By convention, we type a special sequence of characters known as the *end-of-file* sequence. Unfortunately, the terminal applications that we typically encounter on modern operating systems use different conventions for this critically important sequence. In this book, we use `<ctrl-d>` (many systems require `<ctrl-d>` to be on a line by itself); the other widely used convention is `<ctrl-z>` on a line by itself. `Average` is a simple program, but it represents a profound new capability in programming: with standard input, we can write programs that process an unlimited amount of data. As you will see, writing such programs is an effective approach for numerous data-processing applications.

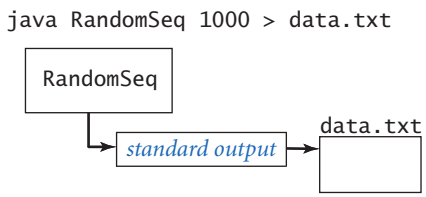
STANDARD INPUT IS A SUBSTANTIAL STEP up from the command-line input model that we have been using, for two reasons, as illustrated by `TwentyQuestions` and `Average`. First, we can interact with our program—with command-line arguments, we can only provide data to the program *before* it begins execution. Second, we can read in large amounts of data—with command-line arguments, we can only enter values that fit on the command line. Indeed, as illustrated by `Average`, the amount of data can be potentially unlimited, and many programs are made simpler by that assumption. A third *raison d'être* for standard input is that your operating system makes it possible to change the source of standard input, so that you do not have to type all the input. Next, we consider the mechanisms that enable this possibility.

Redirection and piping For many applications, typing input data as a standard input stream from the terminal window is untenable because our program's processing power is then limited by the amount of data that we can type (and our typing speed). Similarly, we often want to save the information printed on the standard output stream for later use. To address such limitations, we next focus on the idea that standard input is an *abstraction*—the program just expects its input and has no dependence on the source of the input stream. Standard output is a similar abstraction. The power of these abstractions derives from our ability (through the operating system) to specify various other sources for standard input and standard output, such as a file, the network, or another program. All modern operating systems implement these mechanisms.

Redirecting standard output to a file. By adding a simple directive to the command that invokes a program, we can *redirect* its standard output to a file, either for permanent storage or for input to another program at a later time. For example,

```
% java RandomSeq 1000 > data.txt
```

specifies that the standard output stream is not to be printed in the terminal window, but instead is to be written to a text file named `data.txt`. Each call to `System.out.print()` or `System.out.println()` appends text at the end of that file. In this example, the end result is a file that contains 1,000 random values. No output appears in the terminal window: it goes directly into the file named after the > symbol. Thus, we can save away information for later retrieval. Note that we do not have to change `RandomSeq` (PROGRAM 1.5.1) in any way for this mechanism to work—it is using the standard output abstraction and is unaffected by our use of a different implementation of that abstraction. You can use this mechanism to save output from any program that you write. Once we have expended a significant amount of effort to obtain a result, we often want to save the result for later reference. In a modern system, you can save some information by using cut-and-paste or some similar mechanism that is provided by the operating system, but cut-and-paste is inconvenient for large amounts of data. By contrast, redirection is specifically designed to make it easy to handle large amounts of data.

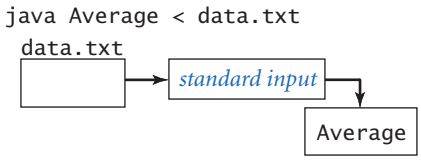


Redirecting standard output to a file

Redirecting from a file to standard input. Similarly, we can redirect standard input so that `StdIn` reads data from a file instead of the terminal application:

```
% java Average < data.txt
```

This command reads a sequence of numbers from the file `data.txt` and computes their average value. Specifically, the < symbol is a directive that tells the operating system to implement the standard input stream by reading from the text file `data.txt` instead of waiting for the user to type something into the terminal window. When the program calls `StdIn.readDouble()`, the operating system reads the value from the file. The file `data.txt` could



Redirecting from a file to standard input

have been created by any application, not just a Java program—virtually every application on your computer can create text files. This facility to redirect from a file to standard input enables us to create *data-driven code* where we can change the data processed by a program without having to change the program at all. Instead, we keep data in files and write programs that read from standard input.

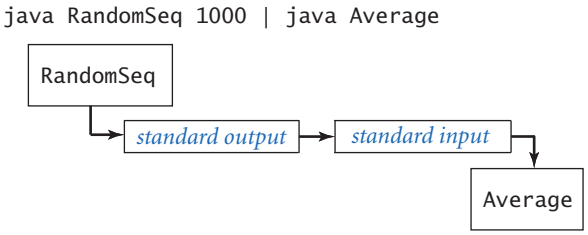
Connecting two programs. The most flexible way to implement the standard input and standard output abstractions is to specify that they are implemented by our own programs! This mechanism is called *piping*. For example, the command

```
% java RandomSeq 1000 | java Average
```

specifies that the standard output for `RandomSeq` and the standard input stream for `Average` are the *same* stream. The effect is as if `RandomSeq` were typing the numbers it generates into the terminal window while `Average` is running. This example also has the same effect as the following sequence of commands:

```
% java RandomSeq 1000 > data.txt
% java Average < data.txt
```

In this case, the file `data.txt` is not created. This difference is profound, because it removes another limitation on the size of the input and output streams that we can process. For example, we could replace `1000` in our example with `1000000000`, even though we might not have the space to save a billion numbers on our computer (we do need the *time* to process them, however). When `RandomSeq` calls `System.out.println()`, a string is added to the end of the stream; when `Average` calls `StdIn.readInt()`, a string is removed from the beginning of the stream. The timing of precisely what happens is up to the operating system: it might run `RandomSeq` until it produces some output, and then run `Average` to consume that output, or it might run `Average` until it needs some output, and then run `RandomSeq` until it produces the needed output. The end result is the same, but our programs are freed from worrying about such details because they work solely with the standard input and standard output abstractions.



Piping the output of one program to the input of another

Program 1.5.4 A simple filter

```
public class RangeFilter
{
   public static void main(String[] args)
   {  // Filter out numbers not between lo and hi.
      int lo = Integer.parseInt(args[0]);
      int hi = Integer.parseInt(args[1]);
      while (!StdIn.isEmpty())
      {  // Process one number.
         int t = StdIn.readInt();
         if (t >= lo && t <= hi) StdOut.print(t + " ");
      }
      StdOut.println();
   }
}
```

lo	*lower bound of range*
hi	*upper bound of range*
t	*current number*

This filter copies to the output stream the numbers from the input stream that fall inside the range given by the command-line parameters. There is no limit on the length of the streams.

```
% java RangeFilter 100 400
358 1330 55 165 689 1014 3066 387 575 843 203 48 292 877 65 998
<ctrl-d>
358 165 387 203 292
```

Filters. Piping, a core feature of the original UNIX system of the early 1970s, still survives in modern systems because it is a simple abstraction for communicating among disparate programs. Testimony to the power of this abstraction is that many UNIX programs are still being used today to process files that are thousands or millions of times larger than imagined by the programs' authors. We can communicate with other Java programs via calls on methods, but standard input and standard output allow us to communicate with programs that were written at another time and, perhaps, in another language. With standard input and standard output, we are agreeing on a simple interface to the outside world. For many common tasks, it is convenient to think of each program as a *filter* that converts a standard input stream to a standard output stream in some way, with piping as the command

mechanism to connect programs together. For example, `RangeFilter` (PROGRAM 1.5.4) takes two command-line arguments and prints on standard output those numbers from standard input that fall within the specified range. You might imagine standard input to be measurement data from some instrument, with the filter being used to throw away data outside the range of interest for the experiment at hand. Several standard filters that were designed for UNIX still survive (sometimes with different names) as commands in modern operating systems. For example, the `sort` filter puts the lines on standard input in sorted order:

```
% java RandomSeq 6 | sort
0.035813305516568916
0.14306638757584322
0.348292877655532103
0.5761644592016527
0.7234592733392126
0.9795908813988247
```

We discuss sorting in SECTION 4.2. A second useful filter is `grep`, which prints the lines from standard input that match a given pattern. For example, if you type

```
% grep lo < RangeFilter.java
```

you get the result

```
        // Filter out numbers not between lo and hi.
        int lo = Integer.parseInt(args[0]);
            if (t >= lo && t <= hi) StdOut.print(t + " ");
```

Programmers often use tools such as `grep` to get a quick reminder of variable names or language usage details. A third useful filter is `more`, which reads data from standard input and displays it in your terminal window one screenful at a time. For example, if you type

```
% java RandomSeq 1000 | more
```

you will see as many numbers as fit in your terminal window, but `more` will wait for you to hit the space bar before displaying each succeeding screenful. The term *filter* is perhaps misleading: it was meant to describe programs like `RangeFilter` that write some subsequence of standard input to standard output, but it is now often used to describe any program that reads from standard input and writes to standard output.

Multiple streams. For many common tasks, we want to write programs that take input from multiple sources and/or produce output intended for multiple destinations. In SECTION 3.1 we discuss our `Out` and `In` libraries, which generalize `StdOut` and `StdIn` to allow for multiple input and output streams. These libraries include provisions not just for redirecting these streams to and from files, but also from arbitrary web pages.

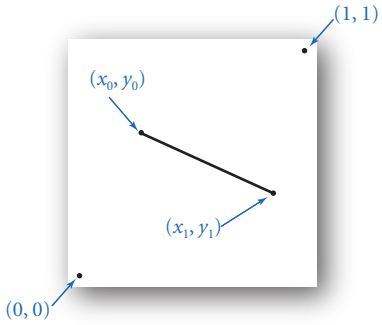
PROCESSING LARGE AMOUNTS OF INFORMATION PLAYS an essential role in many applications of computing. A scientist may need to analyze data collected from a series of experiments, a stock trader may wish to analyze information about recent financial transactions, or a student may wish to maintain collections of music and movies. In these and countless other applications, data-driven programs are the norm. Standard output, standard input, redirection, and piping provides us with the capability to address such applications with our Java programs. We can collect data into files on our computer through the web or any of the standard devices and use redirection and piping to connect data to our programs. Many (if not most) of the programming examples that we consider throughout this book have this ability.

Standard drawing Up to this point, our input/output abstractions have focused exclusively on text strings. Now we introduce an abstraction for producing drawings as output. This library is easy to use and allows us to take advantage of a visual medium to cope with far more information than is possible with just text.

As with standard input, our standard drawing abstraction is implemented in a library that you need to download from the booksite, `StdDraw.java`. Standard drawing is very simple: we imagine an abstract drawing device capable of drawing lines and points on a two-dimensional canvas. The device is capable of responding to the commands that our programs issue in the form of calls to methods in `StdDraw` such as the following:

```
public class StdDraw (basic drawing commands)
   void line(double x0, double y0, double x1, double y1)
   void point(double x, double y)
```

Like the methods for standard input and standard output, these methods are nearly self-documenting: `StdDraw.line()` draws a straight line segment connecting the



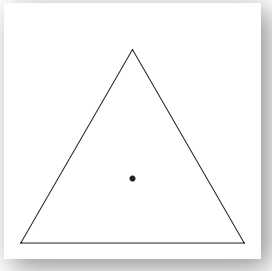

```
StdDraw.line(x0, y0, x1, y1);
```

point (x_0, y_0) with the point (x_1, y_1) whose coordinates are given as arguments. `StdDraw.point()` draws a spot centered on the point (x, y) whose coordinates are given as arguments. The default scale is the unit square (all coordinates between 0 and 1). The standard implementation displays the canvas in a window on your computer's screen, with black lines and points on a white background. The window includes a menu option to save your drawing to a file, in a format suitable for publishing on paper or on the web.

Your first drawing. The `HelloWorld` equivalent for graphics programming with `StdDraw` is to draw a triangle with a point inside. To form the triangle, we draw three lines: one from the point (0, 0) at the lower left corner to the point (1, 0), one from that point to the third point at (1/2, $\sqrt{3}$/2), and one from that point back to (0, 0). As a final flourish, we draw a spot in the middle of the triangle. Once you have successfully downloaded `StdDraw.java` and then compiled and run `Triangle`, you are off and running to write your own programs that draw figures comprised of lines and points. This ability literally adds a new dimension to the output that you can produce.

When you use a computer to create drawings, you get immediate feedback (the drawing) so that you can refine and improve your program quickly. With a computer program, you can create drawings that you could not contemplate making by hand. In particular, instead of viewing our data as just numbers, we can use pictures, which are far more expressive. We will consider other graphics examples after we discuss a few other drawing commands.

```
public class Triangle
{
   public static void main(String[] args)
   {
      double t = Math.sqrt(3.0)/2.0;
      StdDraw.line(0.0, 0.0, 1.0, 0.0);
      StdDraw.line(1.0, 0.0, 0.5,   t);
      StdDraw.line(0.5,   t, 0.0, 0.0);
      StdDraw.point(0.5, t/3.0);
   }
}
```



Your first drawing

Control commands. The default coordinate system for standard drawing is the unit square, but we often want to draw plots at different scales. For example, a typical situation is to use coordinates in some range for the x-coordinate, or the y-coordinate, or both. Also, we often want to draw lines of different thickness and points of different size from the standard. To accommodate these needs, `StdDraw` has the following methods:

`public class StdDraw` (*basic control commands*)	
`void setXscale(double x0, double x1)`	*reset x range to* (x_0, x_1)
`void setYscale(double y0, double y1)`	*reset y range to* (y_0, y_1)
`void setPenRadius(double r)`	*set pen radius to r*

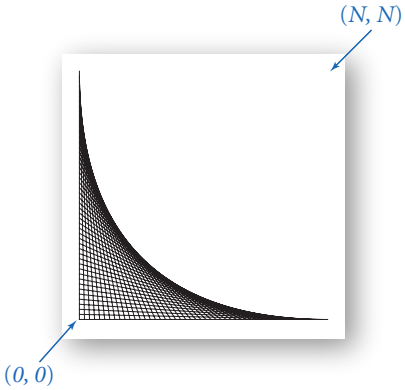
Note: Methods with the same names but no arguments reset to default values.

For example, when we issue the command `StdDraw.setXscale(0, N)`, we are telling the drawing device that we will be using x-coordinates between `0` and `N`. Note that the two-call sequence

```
StdDraw.setXscale(x0, x1);
StdDraw.setYscale(y0, y1);
```

sets the drawing coordinates to be within a *bounding box* whose lower left corner is at (x_0, y_0) and whose upper right corner is at (x_1, y_1). If you use integer coordinates, Java casts them to `double`, as expected. Scaling is the simplest of the transformations commonly used in graphics. In the applications that we consider in this chapter, we use it in a straightforward way to match our drawings to our data.

```
int N = 50;
StdDraw.setXscale(0, N);
StdDraw.setYscale(0, N);
for (int i = 0; i <= N; i++)
   StdDraw.line(0, N-i, i, 0);
```



Scaling to integer coordinates

The pen is circular, so that lines have rounded ends, and when you set the pen radius to r and draw a point, you get a circle of radius r. The default pen radius is .002 and is not affected by coordinate scaling. This default is about 1/500 the width of the default window, so that if you draw 200 points equally spaced along a horizontal or vertical line, you will

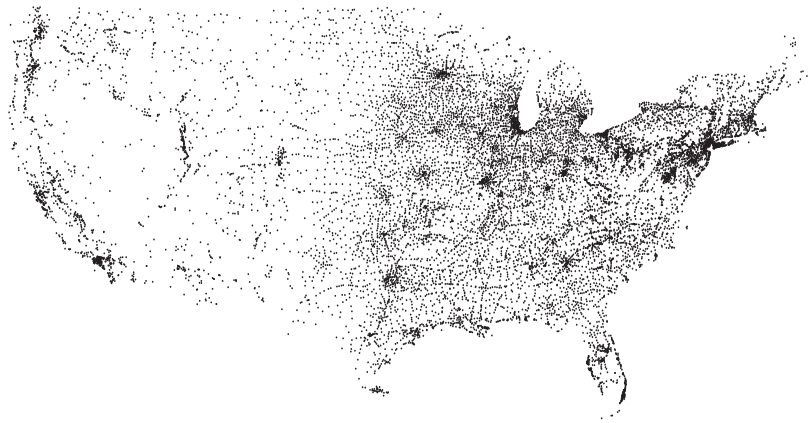
Program 1.5.5 Input-to-drawing filter

```
public class PlotFilter
{
   public static void main(String[] args)
   {  // Plot points in standard input.

      // Scale as per first four values.
      double x0 = StdIn.readDouble();
      double y0 = StdIn.readDouble();
      double x1 = StdIn.readDouble();
      double y1 = StdIn.readDouble();
      StdDraw.setXscale(x0, x1);
      StdDraw.setYscale(y0, y1);

     // Read and plot the rest of the points.
      while (!StdIn.isEmpty())
      {  // Read and plot a point.
         double x = StdIn.readDouble();
         double y = StdIn.readDouble();
         StdDraw.point(x, y);
      }
   }
}
```

x0	*left bound*
y0	*bottom bound*
x1	*right bound*
y1	*top bound*
x, y	*current point*

Some data is inherently visual. The file USA.txt *on the booksite has the coordinates of the US cities with populations over 500 (by convention, the first four numbers are the minimum and maximum x and y values).*

```
% java PlotFilter < USA.txt
```



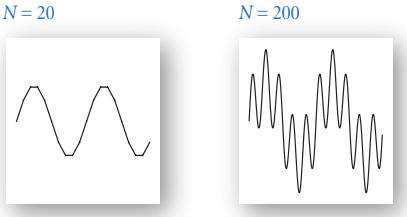
be able to see individual circles, but if you draw 250 such points, the result will look like a line. When you issue the command `StdDraw.setPenRadius(.01)`, you are saying that you want the thickness of the lines and the size of the points to be five times the .002 standard.

Filtering data to a standard drawing. One of the simplest applications of standard draw is to plot data, by filtering it from standard input to the drawing. `PlotFilter` (PROGRAM 1.5.5) is such a filter: it reads a sequence of points defined by (x, y) coordinates and draws a spot at each point. It adopts the convention that the first four numbers on standard input specify the bounding box, so that it can scale the plot without having to make an extra pass through all the points to determine the scale (this kind of convention is typical with such data files). The graphical representation of points plotted in this way is far more expressive (and far more compact) than the numbers themselves or anything that we could create with the standard output representation that our programs have been limited to until now. The plotted image that is produced by PROGRAM 1.5.5 makes it far easier for us to infer properties of the cities (such as, for example, clustering of population centers) than does a list of the coordinates. Whenever we are processing data that represents the physical world, a visual image is likely to be one of the most meaningful ways that we can use to display output. `PlotFilter` illustrates just how easily you can create such an image.

Plotting a function graph. Another important use of `StdDraw` is to plot experimental data or the values of a mathematical function. For example, suppose that we want to plot values of the function $y = \sin(4x) + \sin(20x)$ in the interval $[0, \pi]$. Accomplishing this task is a prototypical example of *sampling*: there are an infinite number of points in the interval, so we have to make do with evaluating the function at a finite number of points within the interval. We sample the function by choosing a set of x-values, then computing y-values by evaluating the function at each x-value. Plotting the function by connecting successive points with lines produces what is known as a *piecewise linear approximation*. The simplest way to proceed is to regularly space the x values: we decide ahead of time on a sample size, then space the x-coordinates by the interval size divided by the sample size. To make sure that the values we plot fall in the visible canvas, we scale the x-axis corresponding to the interval and the y-axis corresponding to the maximum and minimum values of the function within the interval. The smoothness of the curve depends on properties

of the function and the size of the sample. If the sample size is too small, the rendition of the function may not be at all accurate (it might not be very smooth, and it might miss major fluctuations); if the sample is too large, producing the plot may be time-consuming, since some functions are time-consuming to compute. (In SECTION 2.4, we will look at a method for plotting a smooth curve without using an excessive number of points.) You can use this same technique to plot the function graph of any function you choose: decide on an x-interval where you want to plot the function, compute function values evenly spaced through that interval and store them in an array, determine and set the y-scale, and draw the line segments.

```
double[] x = new double[N+1];
double[] y = new double[N+1];
for (int i = 0; i <= N; i++)
   x[i] = Math.PI * i / N;
for (int i = 0; i <= N; i++)
   y[i] = Math.sin(4*x[i]) + Math.sin(20*x[i]);
StdDraw.setXscale(0, Math.PI);
StdDraw.setYscale(-2.0, 2.0);
for (int i = 1; i <= N; i++)
   StdDraw.line(x[i-1], y[i-1], x[i], y[i]);
```



Plotting a function graph

Outline and filled shapes. `StdDraw` also includes methods to draw circles, rectangles, and arbitrary polygons. Each shape defines an outline. When the method name is just the shape name, that outline is traced by the drawing pen. When the name begins with `filled`, the named shape is instead filled solid, not traced. As usual, we summarize the available methods in an API:

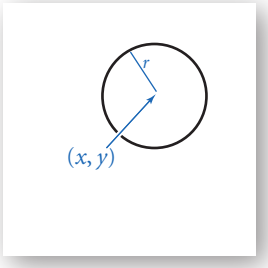
`public class StdDraw` *(shapes)*
`void circle(double x, double y, double r)`
`void filledCircle(double x, double y, double r)`
`void square(double x, double y, double r)`
`void filledSquare(double x, double y, double r)`
`void polygon(double[] x, double[] y)`
`void filledPolygon(double[] x, double[] y)`

The arguments for `circle()` and `filledCircle()` define a circle of radius r centered at (x, y); the arguments for `square()` and `filledSquare()` define a square

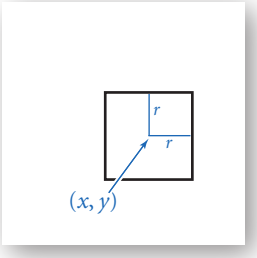
of side length $2r$ centered on (x, y); and the arguments for `polygon()` and `filledPolygon()` define a sequence of points that we connect by lines, including one from the last point to the first point. If you want to define shapes other than squares or circles, use one of these methods. For example,

```
double[] xd = { x-r, x, x+r, x };
double[] yd = { y, y+r, y, y-r };
StdDraw.polygon(xd, yd);
```

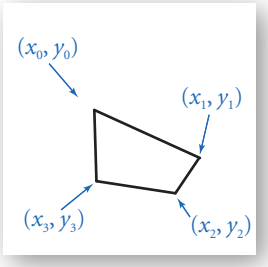
plots a diamond (a rotated $2r$-by-$2r$ square) centered on the point (x,y).



`StdDraw.circle(x, y, r);`



`StdDraw.square(x, y, r);`




```
double[] x = {x0, x1, x2, x3};
double[] y = {y0, y1, y2, y3};
StdDraw.polygon(x, y);
```

Text and color. Occasionally, you may wish to annotate or highlight various elements in your drawings. `StdDraw` has a method for drawing text, another for setting parameters associated with text, and another for changing the color of the ink in the pen. We make scant use of these features in this book, but they can be very useful, particularly for drawings on your computer screen. You will find many examples of their use on the booksite.

`public class StdDraw` *(text and color commands)*

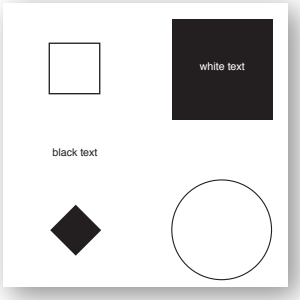
```
void text(double x, double y, String s)
void setFont(Font f)
void setPenColor(Color c)
```

In this code, `Font` and `Color` are non-primitive types that you will learn about in SECTION 3.1. Until then, we leave the details to `StdDraw`. The available pen colors are `BLACK`, `BLUE`, `CYAN`, `DARK_GRAY`, `GRAY`, `GREEN`, `LIGHT_GRAY`, `MAGENTA`, `ORANGE`, `PINK`, `RED`, `WHITE`, and `YELLOW`, defined as constants within `StdDraw`. For example, the call `StdDraw.setPenColor(StdDraw.GRAY)` changes to gray ink. The default ink color is `BLACK`. The default font in `StdDraw` suffices for most of the drawings that you need (and you can find information on using other fonts on

the booksite). For example, you might wish to use these methods to annotate function plots to highlight relevant values, and you might find it useful to develop similar methods to annotate other parts of your drawings.

Shapes, color, and text are basic tools that you can use to produce a dizzying variety of images, but you should use them sparingly. Use of such artifacts usually presents a design challenge, and our `StdDraw` commands are crude by the standards of modern graphics libraries, so that you are likely to need an extensive number of calls to them to produce the beautiful images that you may imagine. On the other hand, using color or labels to help focus on important information in drawings is often worthwhile, as is using color to represent data values.

```
StdDraw.square(.2, .8, .1);
StdDraw.filledSquare(.8, .8, .2);
StdDraw.circle(.8, .2, .2);
double[] xd = { .1, .2, .3, .2 };
double[] yd = { .2, .3, .2, .1 };
StdDraw.filledPolygon(xd, yd);
StdDraw.text(.2, .5, "black text");
StdDraw.setPenColor(StdDraw.WHITE);
StdDraw.text(.8, .8, "white text");
```



Shape and text examples

Animation. The `StdDraw` library supplies additional methods that provide limitless opportunities for creating interesting effects.

`public class StdDraw` *(advanced control commands)*

`void setCanvasSize(int w, int h)`	*create canvas in screen window of width from* `w` *and height* `h` *(in pixels)*
`void clear()`	*clear the canvas to white (default)*
`void clear(Color c)`	*clear the canvas; color it* `c`
`void show(int dt)`	*draw, then pause* `dt` *milliseconds*
`void show()`	*draw, turn off pause mode*

The default canvas size is 512-by-512 pixels; if you want to change it, call `setCanvasSize()` before any drawing commands. The `clear()` and `show()` methods support dynamic changes in the images on the computer screen. Such effects can provide compelling visualizations. We give an example next that also works for the printed page. There are more examples in the booksite that are likely to capture your imagination.

Bouncing ball. The `HelloWorld` of animation is to produce a black ball that appears to move around on the canvas. Suppose that the ball is at position (r_x, r_y) and we want to create the impression of moving it to a new position nearby, such as, for example, $(r_x + .01, r_y + .02)$. We do so in two steps:

- Erase the drawing.
- Draw a black ball at the new position.

To create the illusion of movement, we iterate these steps for a whole sequence of positions (one that will form a straight line, in this case). But these two steps do not suffice, because the computer is so quick at drawing that the image of the ball will rapidly flicker between black and white instead of creating an animated image. Accordingly, `StdDraw` has a `show()` method that allows us to control when the results of drawing actions are actually shown on the display. You can think of it as collecting all of the lines, points, shapes, and text that we tell it to draw, and then immediately drawing them all when we issue the `show()` command. To control the apparent speed, `show()` takes an argument `dt` that tells `StdDraw` to wait `dt` milliseconds after doing the drawing. By default, `StdDraw` issues a `show()` after each `line()`, `point()`, or other drawing command; we turn that option off when we call `StdDraw.show(t)` and turn it back on when we call `StdDraw.show()` with no arguments. With these commands, we can create the illusion of motion with the following steps:

- Erase the drawing (but do not show the result).
- Draw a black ball at the new position.
- Show the result of both commands, and wait for a brief time.

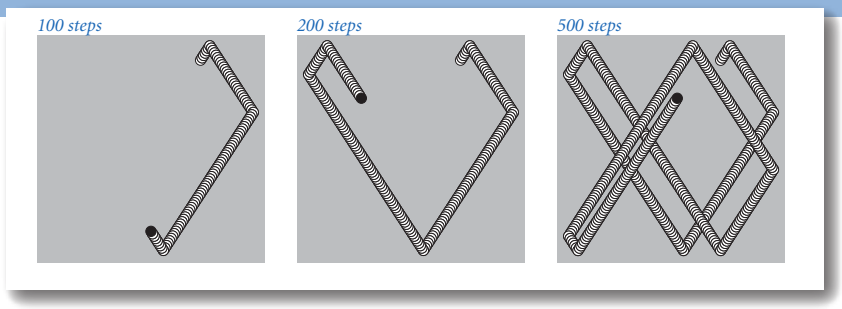
`BouncingBall` (PROGRAM 1.5.6) implements these steps to create the illusion of a ball moving in the 2-by-2 box centered on the origin. The current position of the ball is (r_x, r_y), and we compute the new position at each step by adding v_x to r_x and v_y to r_y. Since (v_x, v_y) is the fixed distance that the ball moves in each time unit, it represents the *velocity*. To keep the ball in the drawing, we simulate the effect of the ball bouncing off the walls according to the laws of elastic collision. This effect is easy to implement: when the ball hits a vertical wall, we just change the velocity in the x-direction from v_x to $-v_x$, and when the ball hits a horizontal wall, we change the velocity in the y-direction from v_y to $-v_y$. Of course, you have to download the code from the booksite and run it on your computer to see motion. To make the image clearer on the printed page, we modified `BouncingBall` to use a gray background that also shows the track of the ball as it moves (see EXERCISE 1.5.34).

Program 1.5.6 Bouncing ball

```
public class BouncingBall
{
   public static void main(String[] args)
   {  // Simulate the movement of a bouncing ball.
      StdDraw.setXscale(-1.0, 1.0);
      StdDraw.setYscale(-1.0, 1.0);
      double rx = .480, ry = .860;
      double vx = .015, vy = .023;
      double radius = .05;
      while(true)
      {  // Update ball position and draw it there.
         if (Math.abs(rx + vx) + radius > 1.0) vx = -vx;
         if (Math.abs(ry + vy) + radius > 1.0) vy = -vy;
         rx = rx + vx;
         ry = ry + vy;
         StdDraw.clear();
         StdDraw.filledCircle(rx, ry, radius);
         StdDraw.show(20);
      }
   }
}
```

rx, ry	*position*
vx, vy	*velocity*
dt	*wait time*
radius	*ball radius*

This program simulates the movement of a bouncing ball in the box with coordinates between -1 and +1. The ball bounces off the boundary according to the laws of elastic collision. The 20-millisecond wait for `StdDraw.show()` *keeps the black image of the ball persistent on the screen, even though most of the ball's pixels alternate between black and white. If you modify this code to take the wait time* `dt` *as a command-line argument, you can control the speed of the ball. The images below, which show the track of the ball, are produced by a modified version of this code (see Exercise 1.5.34).*



STANDARD DRAWING COMPLETES OUR PROGRAMMING MODEL by adding a "picture is worth a thousand words" component. It is a natural abstraction that you can use to better open up your programs to the outside world. With it, you can easily produce the function plots and visual representations of data that are commonly used in science and engineering. We will put it to such uses frequently throughout this book. Any time that you spend now working with the sample programs on the last few pages will be well worth the investment. You can find many useful examples on the booksite and in the exercises, and you are certain to find some outlet for your creativity by using `StdDraw` to meet various challenges. Can you draw an *N*-pointed star? Can you make our bouncing ball actually bounce (add gravity)? You may be surprised at how easily you can accomplish these and other tasks.

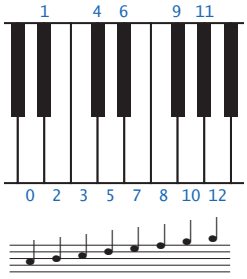
public class StdDraw	
void line(double x0, double y0, double x1, double y1)	
void point(double x, double y)	
void text(double x, double y, String s)	
void circle(double x, double y, double r)	
void filledCircle(double x, double y, double r)	
void square(double x, double y, double r)	
void filledSquare(double x, double y, double r)	
void polygon(double[] x, double[] y)	
void filledPolygon(double[] x, double[] y)	
void setXscale(double x0, double x1)	*reset x range to* (x_0, x_1)
void setYscale(double y0, double y1)	*reset y range to* (y_0, y_1)
void setPenRadius(double r)	*set pen radius to* `r`
void setPenColor(Color c)	*set pen color to* `c`
void setFont(Font f)	*set text font to* `f`
void setCanvasSize(int w, int h)	*set canvas to* `w`*-by-*`h` *window*
void clear(Color c)	*clear the canvas; color it* `c`
void show(int dt)	*show all; pause* `dt` *milliseconds*
void save(String filename)	*save to a* `.jpg` *or* `w.png` *file*

Note: Methods with the same names but no arguments reset to default values.

API for our library of static methods for standard drawing

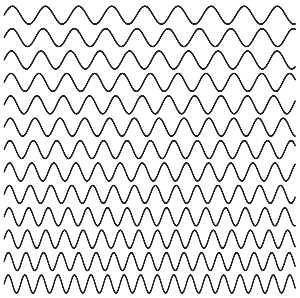
Standard audio As a final example of a basic abstraction for output, we consider `StdAudio`, a library that you can use to play, manipulate, and synthesize sound files. You probably have used your computer to process music. Now you can write programs to do so. At the same time, you will learn some concepts behind a venerable and important area of computer science and scientific computing: *digital signal processing*. We will only scratch the surface of this fascinating subject, but you may be surprised at the simplicity of the underlying concepts.

Concert A. Sound is the perception of the vibration of molecules, in particular, the vibration of our eardrums. Therefore, oscillation is the key to understanding sound. Perhaps the simplest place to start is to consider the musical note *A* above middle *C*, which is known as *concert A*. This note is nothing more than a sine wave, scaled to oscillate at a frequency of 440 times per second. The function $\sin(t)$ repeats itself once every 2π units on the x-axis, so if we measure t in seconds and plot the function $\sin(2\pi t \times 440)$, we get a curve that oscillates 440 times per second. When you play an *A* by plucking a guitar string, pushing air through a trumpet, or causing a small cone to vibrate in a speaker, this sine wave is the prominent part of the sound that you hear and recognize as concert *A*. We measure frequency in *hertz* (cycles per second). When you double or halve the frequency, you move up or down one octave on the scale. For example, 880 hertz is one octave above concert *A* and 110 hertz is two octaves below concert *A*. For reference, the frequency range of human hearing is about 20 to 20,000 hertz. The amplitude (y-value) of a sound corresponds to the volume. We plot our curves between -1 and $+1$ and assume that any devices that record and play sound will scale as appropriate, with further scaling controlled by you when you turn the volume knob.



note	*i*	*frequency*
A	0	440.00
A♯ *or* B♭	1	466.16
B	2	493.88
C	3	523.25
C♯ *or* D♭	4	554.37
D	5	587.33
D♯ *or* E♭	6	622.25
E	7	659.26
F	8	698.46
F♯ *or* G♭	9	739.99
G	10	783.99
G♯ *or* A♭	11	830.61
A	12	880.00

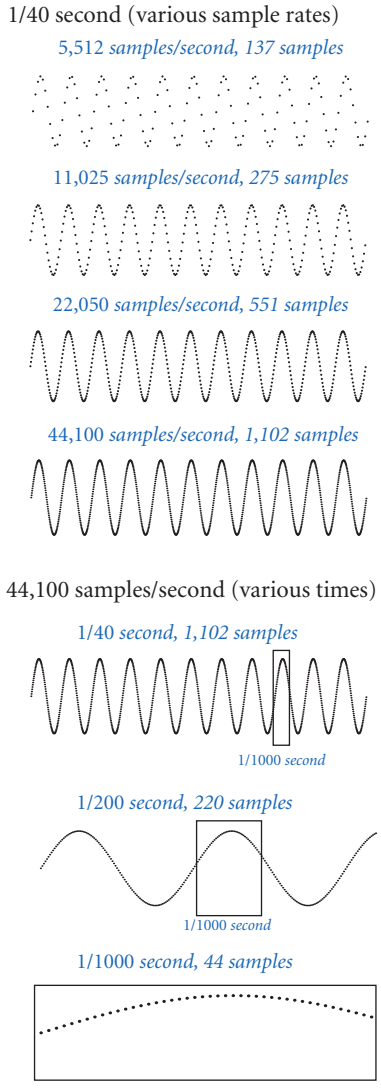
$440 \times 2^{i/12}$



Notes, numbers, and waves

Other notes. A simple mathematical formula characterizes the other notes on the chromatic scale. There are twelve notes on the chromatic scale, divided equally on a logarithmic (base 2) scale. We get the ith note above a given note by multiplying its frequency by the $(i/12)$th power of 2. In other words, the frequency of each note in the chromatic scale is precisely the frequency of the previous note in the scale multiplied by the twelfth root of two (about 1.06). This information suffices to create music! For example, to play the tune *Frère Jacques,* we just need to play each of the notes *A B C# A* by producing sine waves of the appropriate frequency for about half a second and then repeat the pattern. The primary method in the `StdAudio` library, `StdAudio.play()`, allows you to do just that.

Sampling. For digital sound, we represent a curve by sampling it at regular intervals, in precisely the same manner as when we plot function graphs. We sample sufficiently often that we have an accurate representation of the curve—a widely used sampling rate for digital sound is 44,100 samples per second. For concert *A*, that rate corresponds to plotting each cycle of the sine wave by sampling it at about 100 points. Since we sample at regular intervals, we only need to compute the *y*-coordinates of the sample points. It is that simple: *we represent sound as an array of numbers* (`double` values that are between −1 and +1). Our standard sound library method `StdAudio.play()` takes an array as its argument and plays the sound represented by that array on your computer. For example, suppose that you want to play concert *A* for 10 seconds. At 44,100 samples per second, you need an array of 441,001 `double` values. To fill in the array, use a `for` loop that samples the function $\sin(2\pi t \times 440)$ at $t = 0/44100$,



Sampling a sine wave

1/44100, 2/44100, 3/44100, . . . 441000/44100. Once we fill the array with these values, we are ready for `StdAudio.play()`, as in the following code:

```
int sps = 44100;                  // samples per second
int hz = 440;                     // concert A
double duration = 10.0;           // ten seconds
int N = (int) (sps * duration);   // total number of samples
double[] a = new double[N+1];
for (int i = 0; i <= N; i++)
   a[i] = Math.sin(2*Math.PI * i * hz / sps);
StdAudio.play(a);
```

This code is the `HelloWorld` of digital audio. Once you use it to get your computer to play this note, you can write code to play other notes and make music! The difference between creating sound and plotting an oscillating curve is nothing more than the output device. Indeed, it is instructive and entertaining to send the same numbers to both standard draw and standard audio (see EXERCISE 1.5.27).

Saving to a file. Music can take up a lot of space on your computer. At 44,100 samples per second, a four-minute song corresponds to 4×60×44100=10,584,000 numbers. Therefore, it is common to represent the numbers corresponding to a song in a binary format that uses less space than the string-of-digits representation that we use for standard input and output. Many such formats have been developed in recent years—`StdAudio` uses the `.wav` format. You can find some information about the `.wav` format on the booksite, but you do not need to know the details, because `StdAudio` takes care of the conversions for you. Our standard library for audio allows you to play `.wav` files, to write programs to create and manipulate arrays of `double` values, and to read and write them as `.wav` files.

public class StdAudio	
void play(String file)	*play the given* `.wav` *file*
void play(double[] a)	*play the given sound wave*
void play(double x)	*play sample for 1/44100 second*
void save(String file, double[] a)	*save to a* `.wav` *file*
double[] read(String file)	*read from a* `.wav` *file*

API for our library of static methods for standard audio

Program 1.5.7 Digital signal processing

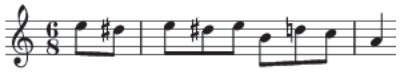
```
public class PlayThatTune
{
   public static void main(String[] args)
   {  // Read a tune from StdIn and play it.
      int sps = 44100;
      while (!StdIn.isEmpty())
      {  // Read and play one note.
         int pitch = StdIn.readInt();
         double duration = StdIn.readDouble();
         double hz = 440 * Math.pow(2, pitch / 12.0);
         int N = (int) (sps * duration);
         double[] a = new double[N+1];
         for (int i = 0; i <= N; i++)
            a[i] = Math.sin(2*Math.PI * i * hz / sps);
         StdAudio.play(a);
      }
   }
}
```

pitch	*distance from A*
duration	*note play time*
hz	*frequency*
N	*number of samples*
a[]	*sampled sine wave*

This is a data-driven program that plays pure tones from the notes on the chromatic scale, specified on standard input as a pitch (distance from concert A) and a duration (in seconds). The test client reads the notes from standard input, creates an array by sampling a sine wave of the specified frequency and duration at 44100 samples per second, and then plays each note by calling `StdAudio.play()`.

```
% more elise.txt
7 .25
6 .25
7 .25
6 .25
7 .25
2 .25
5 .25
3 .25
0 .50
```

```
% java PlayThatTune < elise.txt
```



`PlayThatTune` (PROGRAM 1.5.7) is an example that shows how easily we can create music with `StdAudio`. It takes notes from standard input, indexed on the chromatic scale from concert *A*, and plays them on standard audio. You can imagine all sorts of extensions on this basic scheme, some of which are addressed in the exercises. We include `StdAudio` in our basic arsenal of programming tools because sound processing is one important application of scientific computing that is certainly familiar to you. Not only has the commercial application of digital signal processing had a phenomenal impact on modern society, but the science and engineering behind it combines physics and computer science in interesting ways. We will study more components of digital signal processing in some detail later in the book. (For example, you will learn in SECTION 2.1 how to create sounds that are more musical than the pure sounds produced by `PlayThatTune`.)

I/O IS A PARTICULARLY CONVINCING EXAMPLE of the power of abstraction because standard input, standard output, standard draw, and standard audio can be tied to different physical devices at different times without making any changes to programs. Although devices may differ dramatically, we can write programs that can do I/O without depending on the properties of specific devices. From this point forward, we will use methods from `StdOut`, `StdIn`, `StdDraw`, and/or `StdAudio` in nearly every program in this book, and you will use them in nearly all of your programs, so make sure to download copies of these libraries. For economy, we collectively refer to these libraries as `Std*`. One important advantage of using such libraries is that you can switch to new devices that are faster, cheaper, or hold more data without changing your program at all. In such a situation, the details of the connection are a matter to be resolved between your operating system and the `Std*` implementations. On modern systems, new devices are typically supplied with software that resolves such details automatically for both the operating system and for Java.

Conceptually, one of the most significant features of the standard input, standard output, standard draw, and standard audio data streams is that they are *infinite*: from the point of view of your program, there is no limit on their length. This point of view not only leads to programs that have a long useful life (because they are less sensitive to changes in technology than programs with built-in limits). It also is related to the *Turing machine*, an abstract device used by theoretical computer scientists to help us understand fundamental limitations on the capabilities of real computers. One of the essential properties of the model is the idea of a finite discrete device that works with an unlimited amount of input and output.

Q&A

Q. Why are we not using the standard Java libraries for input, graphics, and sound?

A. We *are* using them, but we prefer to work with simpler abstract models. The Java libraries behind `StdIn`, `StdDraw`, and `StdAudio` are built for production programming, and the libraries and their APIs are a bit unwieldy. To get an idea of what they are like, look at the code in `StdIn.java`, `StdDraw.java`, and `StdAudio.java`.

Q. So, let me get this straight. If I use the format `%2.4f` for a `double` value, I get two digits before the decimal point and four digits after, right?

A. No, that specifies just four digits after the decimal point. The first value is the width of the whole field. You want to use the format `%7.2f` to specify seven characters in total, four before the decimal point, the decimal point itself, and two digits after the decimal point.

Q. What other conversion codes are there for `printf()`?

A. For integer values, there is `o` for octal and `x` for hexadecimal. There are also numerous formats for dates and times. See the booksite for more information.

Q. Can my program re-read data from standard input?

A. No. You only get one shot at it, in the same way that you cannot undo a `println()` command.

Q. What happens if my program attempts to read data from standard input after it is exhausted?

A. You will get an error. `StdIn.isEmpty()` allows you to avoid such an error by checking whether there is more input available.

Q. What does the error message `Exception in thread "main" java.lang.NoClassDefFoundError: StdIn` mean?

A. You probably forgot to put `StdIn.java` in your working directory.

Q. I have a different working directory for each project that I am working on, so I

have copies of `StdOut.java`, `StdIn.java`, `StdDraw.java`, and `StdAudio.java` in each of them. Is there some better way?

A. Yes. You can put them all in one directory and use the "classpath" mechanism to tell Java where to find them. This mechanism is operating-system dependent—you can find instructions on how to use it on the booksite.

Q. My terminal window hangs at the end of a program using `StdAudio`. How can I avoid having to use `<ctrl-c>` to get a command prompt?

A. Add a call to `System.exit(0)` as the last line in `main()`. Don't ask why.

Q. So I use negative integers to go below concert *A* when making input files for `PlayThatTune`?

A. Right. Actually, our choice to put concert *A* at 0 is arbitrary. A popular standard, known as the *MIDI Tuning Standard*, starts numbering at the *C* five octaves below concert *A*. By that convention, concert *A* is 69 and you do not need to use negative numbers.

Q. Why do I hear weird results on standard audio when I try to sonify a sine wave with a frequency of 30,000 Hertz (or more)?

A. The *Nyquist frequency*, defined as one-half the sampling frequency, represents the highest frequency that can be reproduced. For standard audio, the sampling frequency is 44,100, so the Nyquist frequency is 22,050.

Exercises

1.5.1 Write a program that reads in integers (as many as the user enters) from standard input and prints out the maximum and minimum values.

1.5.2 Modify your program from the previous exercise to insist that the integers must be positive (by prompting the user to enter positive integers whenever the value entered is not positive).

1.5.3 Write a program that takes an integer N from the command line, reads N `double` values from standard input, and prints their mean (average value) and standard deviation (square root of the sum of the squares of their differences from the average, divided by $N-1$).

1.5.4 Extend your program from the previous exercise to create a filter that prints all the values that are further than 1.5 standard deviations from the mean. Use an array.

1.5.5 Write a program that reads in a sequence of integers and prints out both the integer that appears in a longest consecutive run and the length of the run. For example, if the input is `1 2 2 1 5 1 1 7 7 7 7 1 1`, then your program should print `Longest run: 4 consecutive 7s`.

1.5.6 Write a filter that reads in a sequence of integers and prints out the integers, removing repeated values that appear consecutively. For example, if the input i `1 2 2 1 5 1 1 7 7 7 7 1 1 1 1 1 1 1 1 1`, your program should print out `1 2 1 5 1 7 1`.

1.5.7 Write a program that takes a command-line argument `N`, reads in `N-1` distinct integers between `1` and `N`, and determines the missing value.

1.5.8 Write a program that reads in positive real numbers from standard input and prints out their geometric and harmonic means. The *geometric mean* of N positive numbers $x_1, x_2, ..., x_N$ is $(x_1 \times x_2 \times ... \times x_N)^{1/N}$. The *harmonic mean* is $(1/x_1 + 1/x_2 + ... + 1/x_N) / (1/N)$. *Hint*: For the geometric mean, consider taking logs to avoid overflow.

1.5.9 Suppose that the file `input.txt` contains the two strings `F` and `F`. What

does the following command do (see EXERCISE 1.2.35)?

```
java Dragon < input.txt | java Dragon | java Dragon

public class Dragon
{
   public static void main(String[] args)
   {
      String dragon = StdIn.readString();
      String nogard = StdIn.readString();
      StdOut.print(dragon + "L" + nogard);
      StdOut.print(" ");
      StdOut.print(dragon + "R" + nogard);
      StdOut.println();
   }
}
```

1.5.10 Write a filter `TenPerLine` that takes a sequence of integers between 0 and 99 and prints 10 integers per line, with columns aligned. Then write a program `RandomIntSeq` that takes two command-line arguments `M` and `N` and outputs `N` random integers between 0 and `M-1`. Test your programs with the command `java RandomIntSeq 200 100 | java TenPerLine`.

1.5.11 Write a program that reads in text from standard input and prints out the number of words in the text. For the purpose of this exercise, a word is a sequence of non-whitespace characters that is surrounded by whitespace.

1.5.12 Write a program that reads in lines from standard input with each line containing a name and two integers and then uses `printf()` to print a table with a column of the names, the integers, and the result of dividing the first by the second, accurate to three decimal places. You could use a program like this to tabulate batting averages for baseball players or grades for students.

1.5.13 Which of the following *require* saving all the values from standard input (in an array, say), and which could be implemented as a filter using only a fixed number of variables? For each, the input comes from standard input and consists of N real numbers between 0 and 1.

- Print the maximum and minimum numbers.
- Print the kth smallest value.
- Print the sum of the squares of the numbers.
- Print the average of the N numbers.
- Print the percentage of numbers greater than the average.
- Print the N numbers in increasing order.
- Print the N numbers in random order.

1.5.14 Write a program that prints a table of the monthly payments, remaining principal, and interest paid for a loan, taking three numbers as command-line arguments: the number of years, the principal, and the interest rate (see EXERCISE 1.2.24).

1.5.15 Write a program that takes three command-line arguments x, y, and z, reads from standard input a sequence of point coordinates (x_i, y_i, z_i), and prints the coordinates of the point closest to (x, y, z). Recall that the square of the distance between (x, y, z) and (x_i, y_i, z_i) is $(x - x_i)^2 + (y - y_i)^2 + (z - z_i)^2$. For efficiency, do not use `Math.sqrt()` or `Math.pow()`.

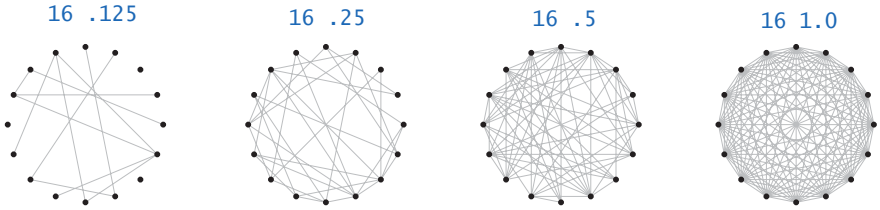
1.5.16 Given the positions and masses of a sequence of objects, write a program to compute their center-of-mass, or *centroid*. The centroid is the average position of the N objects, weighted by mass. If the positions and masses are given by (x_i, y_i, m_i), then the centroid (x, y, m) is given by:

$$
\begin{aligned}
m &= m_1 + m_2 + \ldots + m_N \\
x &= (m_1 x_1 + \ldots + m_n x_N) / m \\
y &= (m_1 y_1 + \ldots + m_n y_N) / m
\end{aligned}
$$

1.5.17 Write a program that reads in a sequence of real numbers between -1 and $+1$ and prints out their average magnitude, average power, and the number of zero crossings. The *average magnitude* is the average of the absolute values of the data values. The *average power* is the average of the squares of the data values. The number of *zero crossings* is the number of times a data value transitions from a strictly negative number to a strictly positive number, or vice versa. These three statistics are widely used to analyze digital signals.

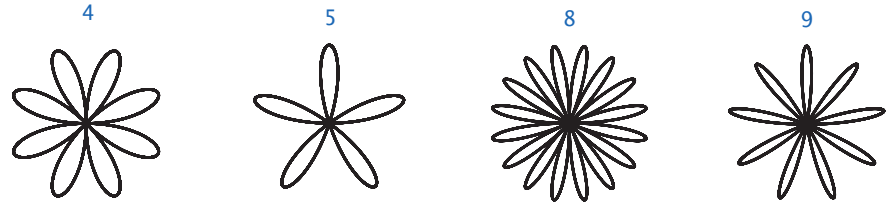
1.5.18 Write a program that takes a command-line argument `N` and plots an `N`-by-`N` checkerboard with red and black squares. Color the lower left square red.

1.5.19 Write a program that takes as command-line arguments an integer `N` and a double value `p` (between 0 and 1), plots `N` equally spaced points of size on the circumference of a circle, and then, with probability `p` for each pair of points, draws a gray line connecting them.



1.5.20 Write code to draw hearts, spades, clubs, and diamonds. To draw a heart, draw a diamond, then attach two semicircles to the upper left and upper right sides.

1.5.21 Write a program that takes a command-line argument N and plots a rose with N petals (if N is odd) or $2N$ petals (if N is even), by plotting the polar coordinates (r, θ) of the function $r = \sin(N\,\theta)$ for θ ranging from 0 to 2π radians.

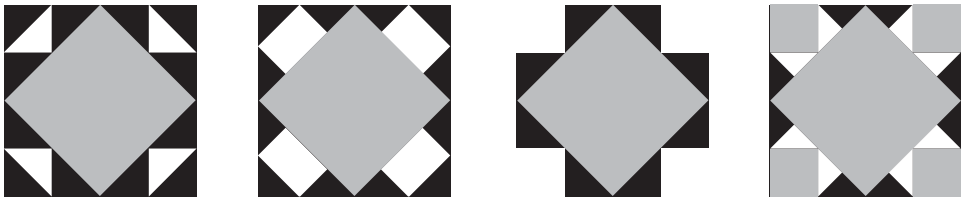


1.5.22 Write a program that takes a string `s` from the command line and displays it in banner style on the screen, moving from left to right and wrapping back to the beginning of the string as the end is reached. Add a second command-line argument to control the speed.

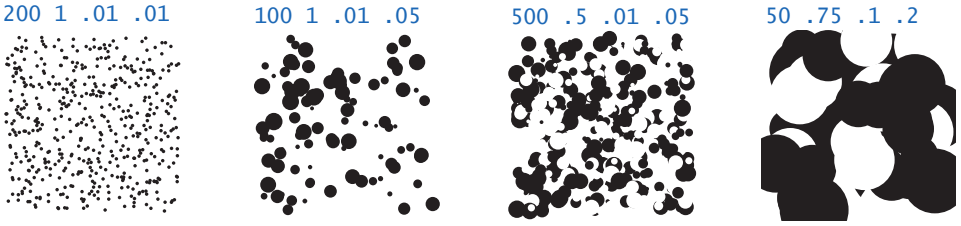
1.5.23 Modify `PlayThatTune` to take additional command-line arguments that control the volume (multiply each sample value by the volume) and the tempo (multiply each note's duration by the tempo).

1.5.24 Write a program that takes the name of a `.wav` file and a playback rate r as command-line arguments and plays the file at the given rate. First, use `StdAudio.read()` to read the file into an array `a[]`. If $r = 1$, just play `a[]`; otherwise create a new array `b[]` of approximate size r times `a.length`. If $r < 1$, populate `b[]` by *sampling* from the original; if $r > 1$, populate `b[]` by *interpolating* from the original. Then play `b[]`.

1.5.25 Write programs that uses `StdDraw` to create each of the following designs.



1.5.26 Write a program `Circles` that draws filled circles of random size at random positions in the unit square, producing images like those below. Your program should take four command-line arguments: the number of circles, the probability that each circle is black, the minimum radius, and the maximum radius.



Creative Exercises

1.5.27 *Visualizing audio.* Modify `PlayThatTune` to send the values played to standard drawing, so that you can watch the sound waves as they are played. You will have to experiment with plotting multiple curves in the drawing canvas to synchronize the sound and the picture.

1.5.28 *Statistical polling.* When collecting statistical data for certain political polls, it is very important to obtain an unbiased sample of registered voters. Assume that you have a file with `N` registered voters, one per line. Write a filter that prints out a random sample of size `M` (see PROGRAM 1.4.1).

1.5.29 *Terrain analysis.* Suppose that a terrain is represented by a two-dimensional grid of elevation values (in meters). A *peak* is a grid point whose four neighboring cells (left, right, up, and down) have strictly lower elevation values. Write a program `Peaks` that reads a terrain from standard input and then computes and prints the number of peaks in the terrain.

1.5.30 *Histogram.* Suppose that the standard input stream is a sequence of `double` values. Write a program that takes an integer N and two `double` values l and r from the command line and uses `StdDraw` to plot a histogram of the count of the numbers in the standard input stream that fall in each of the N intervals defined by dividing (l, r) into N equal-sized intervals.

1.5.31 *Spirographs.* Write a program that takes three parameters R, r, and a from the command line and draws the resulting *spirograph*. A spirograph (technically, an epicycloid) is a curve formed by rolling a circle of radius r around a larger fixed circle of radius R. If the pen offset from the center of the rolling circle is $(r+a)$, then the equation of the resulting curve at time t is given by

$$x(t) = (R + r)\cos(t) - (r + a)\cos((R + r)t/r)$$
$$y(t) = (R + r)\sin(t) - (r + a)\sin((R + r)t/r)$$

Such curves were popularized by a best-selling toy that contains discs with gear teeth on the edges and small holes that you could put a pen in to trace spirographs.

1.5.32 *Clock.* Write a program that displays an animation of the second, minute, and hour hands of an analog clock. Use the method `StdDraw.show(1000)` to update the display roughly once per second.

1.5.33 *Oscilloscope.* Write a program to simulate the output of an oscilloscope and produce Lissajous patterns. These patterns are named after the French physicist, Jules A. Lissajous, who studied the patterns that arise when two mutually perpendicular periodic disturbances occur simultaneously. Assume that the inputs are sinusoidal, so that the following parametric equations describe the curve:

$$x(t) = A_x \sin(w_x t + \theta_x)$$
$$y(t) = A_y \sin(w_y t + \theta_y)$$

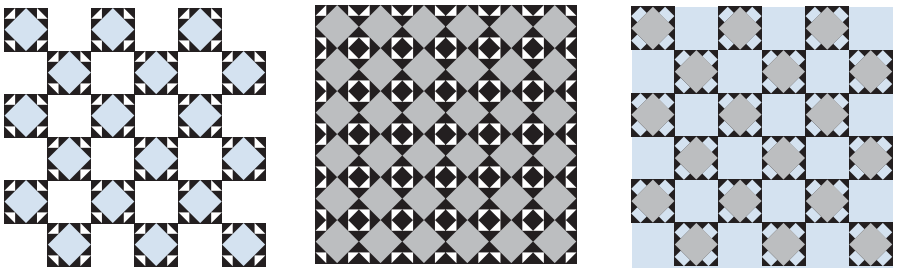
Take the six parameters A_x, w_x, θ_x, A_y, w_y, and θ_y from the command line.

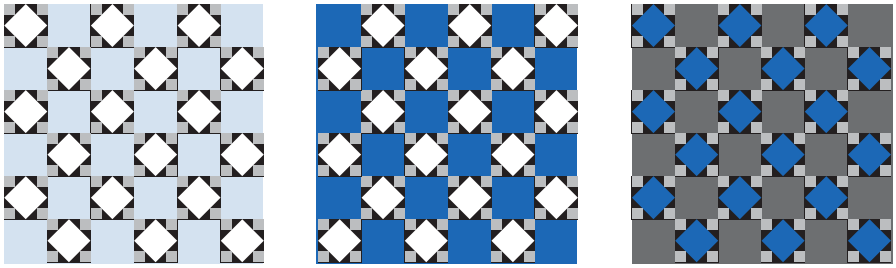
1.5.34 *Bouncing ball with tracks.* Modify `BouncingBall` to produce images like the ones shown in the text, which show the track of the ball on a gray background.

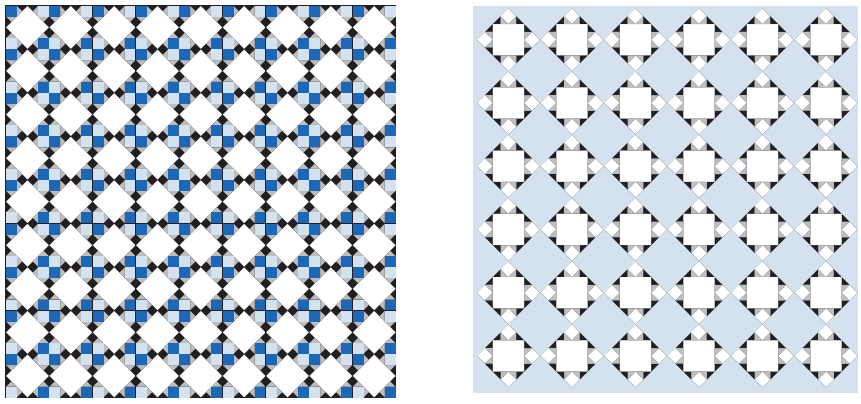
1.5.35 *Bouncing ball with gravity.* Modify `BouncingBall` to incorporate gravity in the vertical direction. Add calls to `StdAudio.play()` to add one sound effect when the ball hits a wall and a different one when it hits the floor.

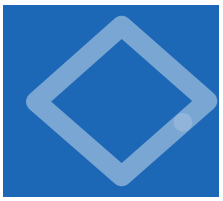
1.5.36 *Random tunes.* Write a program that uses `StdAudio` to play random tunes. Experiment with keeping in key, assigning high probabilities to whole steps, repetition, and other rules to produce reasonable melodies.

1.5.37 *Tile patterns.* Using your solution to EXERCISE 1.5.25, write a program `TilePattern` that takes a command-line argument `N` and draws an `N`-by-`N` pattern, using the tile of your choice. Add a second command-line argument that adds a checkerboard option. Add a third command-line argument for color selection. Using the patterns on the facing page as a starting point, design a tile floor. Be creative! *Note*: These are all designs from antiquity that you can find in many ancient (and modern) buildings.









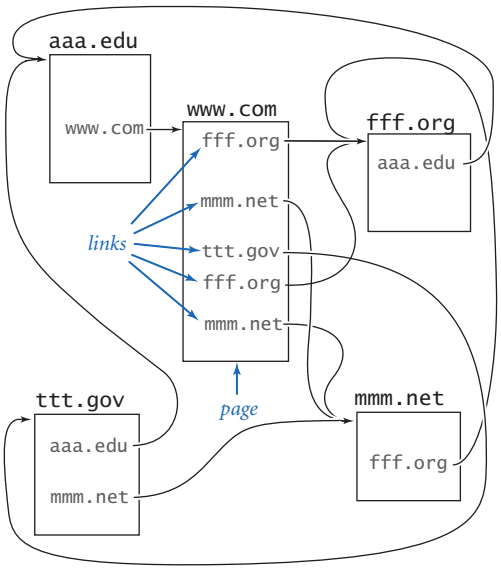
1.6 Case Study: Random Web Surfer

COMMUNICATING ACROSS THE WEB HAS BECOME an integral part of everyday life. This communication is enabled in part by scientific studies of the structure of the web, a subject of active research since its inception. We next consider a simple model of the web that has proven to be a particularly successful approach to understanding some of its properties. Variants of this model are widely used and have been a key factor in the explosive growth of search applications on the web.

1.6.1 Computing the transition matrix . 165
1.6.2 Simulating a random surfer. 167
1.6.3 Mixing a Markov chain 174
Programs in this section

The model is known as the *random surfer* model, and is simple to describe. We consider the web to be a fixed set of *pages*, with each page containing a fixed set of *hyperlinks* (for brevity, we use the term *links*), and each link a reference to some other page. We study what happens to a person (the random surfer) who randomly moves from page to page, either by typing a page name into the address bar or by clicking a link on the current page.

The underlying mathematical model behind the web model is known as the *graph*, which we will consider in detail at the end of the book (in SECTION 4.5). We defer discussion of details about processing graphs until then. Instead, we concentrate on calculations associated with a natural and well-studied probabilistic model that accurately describes the behavior of the random surfer.



Pages and links

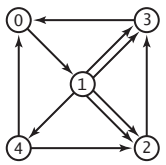
The first step in studying the random surfer model is to formulate it more precisely. The crux of the matter is to specify what it means to randomly move from page to page. The following intuitive *90-10 rule* captures both methods of moving to a new page: *Assume that 90 per cent of the time the random surfer clicks a random link on the current page (each link chosen with equal probability) and that 10 percent of the time the random surfer goes directly to a random page (all pages on the web chosen with equal probability).*

You can immediately see that this model has flaws, because you know from your own experience that the behavior of a real web surfer is not quite so simple:

- No one chooses links or pages with equal probability.
- There is no real potential to surf directly to each page on the web.
- The 90-10 (or any fixed) breakdown is just a guess.
- It does not take the back button or bookmarks into account.
- We can only afford to work with a small sample of the web.

Despite these flaws, the model is sufficiently rich that computer scientists have learned a great deal about properties of the web by studying it. To appreciate the model, consider the small example on the previous page. Which page do you think the random surfer is most likely to visit?

Each person using the web behaves a bit like the random surfer, so understanding the fate of the random surfer is of intense interest to people building web infrastructure and web applications. The model is a tool for understanding the experience of each of the hundreds of millions of web users. In this section, you will use the basic programming tools from this chapter to study the model and its implications.




```
% more tiny.txt
5          <- N
0 1
1 2  1 2
1 3  1 3  1 4
2 3
3 0                 <- links
4 0  4 2
```

Random surfer input format

Input format We want to be able to study the behavior of the random surfer on various web models, not just one example. Consequently, we want to write *data-driven code*, where we keep data in files and write programs that read the data from standard input. The first step in this approach is to define an *input format* that we can use to structure the information in the input files. We are free to define any convenient input format.

Later in the book, you will learn how to read web pages in Java programs (SECTION 3.1) and to convert from names to numbers (SECTION 4.4) as well as other techniques for efficient graph processing. For now, we assume that there are N web pages, numbered from `0` to `N-1`, and we represent links with ordered pairs of such numbers, the first specifying the page containing the link and the second specifying the page to which it refers. Given these conventions, a straightforward input format for the random surfer problem is an input stream consisting of an integer (the value of N) followed by a sequence of pairs of integers (the representations of all the links). `StdIn` treats all sequences of whitespace characters as a single delimiter, so we are free to either put one link per line or arrange them several to a line.

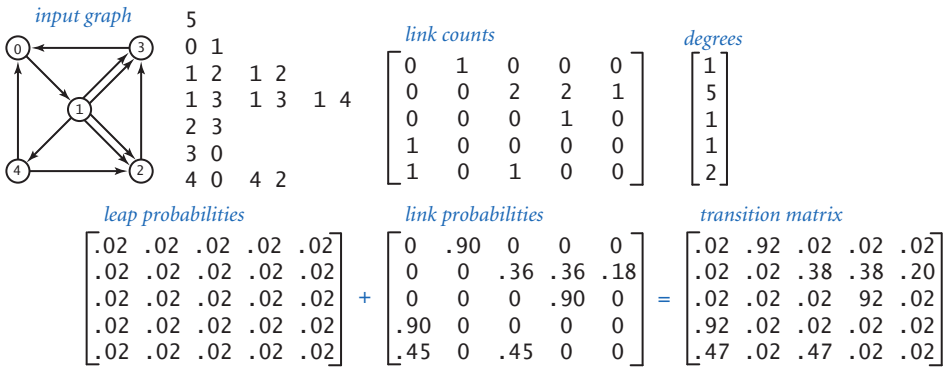
Transition matrix We use a two-dimensional matrix, that we refer to as the *transition matrix*, to completely specify the behavior of the random surfer. With N web pages, we define an N-by-N matrix such that the entry in row `i` and column `j` is the probability that the random surfer moves to page `j` when on page `i`. Our first task is to write code that can create such a matrix for any given input. By the 90-10 rule, this computation is not difficult. We do so in three steps:

- Read N, and then create arrays `counts[][]` and `outDegree[]`.
- Read the links and accumulate counts so that `counts[i][j]` counts the links from `i` to `j` and `outDegree[i]` counts the links from `i` to anywhere.
- Use the 90-10 rule to compute the probabilities.

The first two steps are elementary, and the third is not much more difficult: multiply `counts[i][j]` by `.90/degree[i]` if there is a link from `i` to `j` (take a random link with probability .9), and then add `.10/N` to each entry (go to a random page with probability .1). `Transition` (PROGRAM 1.6.1) performs this calculation: It is a filter that converts the list-of-links representation of a web model into a transition-matrix representation.

The transition matrix is significant because each row represents a *discrete probability distribution*—the entries fully specify the behavior of the random surfer's next move, giving the probability of surfing to each page. Note in particular that the entries sum to 1 (the surfer always goes somewhere).

The output of `Transition` defines another file format, one for matrices of `double` values: the numbers of rows and columns followed by the values for matrix entries. Now, we can write programs that read and process transition matrices.



Transition matrix computation

Program 1.6.1 Computing the transition matrix

```
public class Transition
{
   public static void main(String[] args)
   {  // Print random-surfer probabilites.
      int N = StdIn.readInt();
      int[][] counts = new int[N][N];
      int[] outDegree = new int[N];
      while (!StdIn.isEmpty())
      {  // Accumulate link counts.
         int i = StdIn.readInt();
         int j = StdIn.readInt();
         outDegree[i]++;
         counts[i][j]++;
      }

      StdOut.println(N + " " + N);
      for (int i = 0; i < N; i++)
      {  // Print probability distribution for row i.
         for (int j = 0; j < N; j++)
         {  // Print probability for column j.
            double p = .90*counts[i][j]/outDegree[i] + .10/N;
            StdOut.printf("%8.5f", p);
         }
         StdOut.println();
      }
   }
}
```

N	*number of pages*
counts[i][j]	*count of links from page i to page j*
outDegree[i]	*count of links from page i to anywhere*
p	*transition probability*

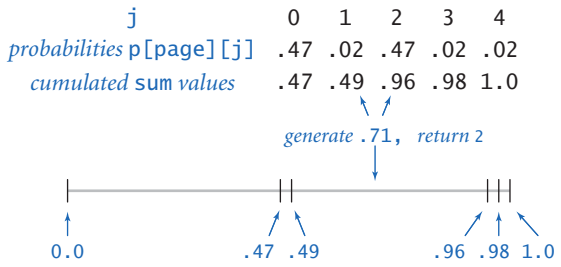
This program is a filter that reads links from standard input and produces the corresponding transition matrix on standard output. First, it processes the input to count the outlinks from each page. Then it applies the 90-10 rule to compute the transition matrix (see text). It assumes that there are no pages that have no outlinks in the input (see Exercise 1.6.3).

```
% more tiny.txt
5
0 1
1 2  1 2
1 3  1 3  1 4
2 3
3 0
4 0  4 2
```

```
% java Transition < tiny.txt
5 5
 0.02000 0.92000 0.02000 0.02000 0.02000
 0.02000 0.02000 0.38000 0.38000 0.20000
 0.02000 0.02000 0.02000 0.92000 0.02000
 0.92000 0.02000 0.02000 0.02000 0.02000
 0.47000 0.02000 0.47000 0.02000 0.02000
```

Simulation Given the transition matrix, simulating the behavior of the random surfer involves surprisingly little code, as you can see in `RandomSurfer` (PROGRAM 1.6.2). This program reads a transition matrix and surfs according to the rules, starting at page `0` and taking the number of moves as a command-line argument. It counts the number of times that the surfer visits each page. Dividing that count by the number of moves yields an estimate of the probability that a random surfer winds up on the page. This probability is known as the page's *rank*. In other words, `RandomSurfer` computes an estimate of all page ranks.

One random move. The key to the computation is the random move, which is specified by the transition matrix. We maintain a variable `page` whose value is the current location of the surfer. Row `page` of the matrix gives, for each `j`, the probability that the surfer next goes to `j`. In other words, when the surfer is at `page`, our task is to generate a random integer between `0` and `N-1` according to the distribution given by row `page` in the transition matrix (the one-dimensional array `p[page]`). How can we accomplish this task? We can use `Math.random()` to generate a random number `r` between `0` and `1`, but how does that help us get to a random page? One way to answer this question is to think of the probabilities in row `page` as defining a set of N intervals in (0, 1) with each probability corresponding to an interval length. Then our random variable `r` falls into one of the intervals, with probability precisely specified by the interval length. This reasoning leads to the following code:



Generating a random integer from a discrete distribution

```
double sum = 0.0;
for (int j = 0; j < N; j++)
{  // Find interval containing r.
   sum += p[page][j];
   if (r < sum) { page = j; break; }
}
```

The variable `sum` tracks the endpoints of the intervals defined in row `p[page]`, and the `for` loop finds the interval containing the random value `r`. For example, suppose that the surfer is at page `4` in our example. The transition probabilities are .47,

Program 1.6.2 Simulating a random surfer

```
public class RandomSurfer
{
   public static void main(String[] args)
   {  // Simulate random-surfer leaps and links.
      int T = Integer.parseInt(args[0]);
      int N = StdIn.readInt();
      StdIn.readInt();

      // Read transition matrix.
      double[][] p = new double[N][N];
      for (int i = 0; i < N; i++)
         for (int j = 0; j < N; j++)
            p[i][j] = StdIn.readDouble();

      int page = 0; // Start at page 0.
      int[] freq = new int[N];
      for (int t = 0; t < T; t++)
      {  // Make one random move.
         double r = Math.random();
         double sum = 0.0;
         for (int j = 0; j < N; j++)
         {  // Find interval containing r.
            sum += p[page][j];
            if (r < sum) { page = j; break; }
         }
         freq[page]++;
      }

      for (int i = 0; i < N; i++)     // Print page ranks.
         StdOut.printf("%8.5f", (double) freq[i] / T);
      StdOut.println();
    }
}
```

T	*number of moves*
N	*number of pages*
page	*current page*
p[i][j]	*probability that the surfer moves from page i to page j*
freq[i]	*number of times the surfer hits page i*

This program uses a transition matrix to simulate the behavior of a random surfer. It takes the number of moves as a command-line argument, reads the transition matrix, performs the indicated number of moves as prescribed by the matrix, and prints the relative frequency of hitting each page. The key to the computation is the random move to the next page (see text).

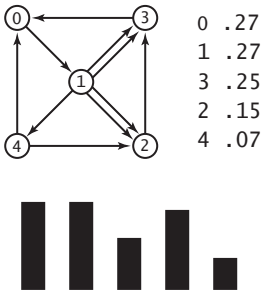
```
% java Transition < tiny.txt | java RandomSurfer 100
 0.24000 0.23000 0.16000 0.25000 0.12000
% java Transition < tiny.txt | java RandomSurfer 10000
 0.27280 0.26530 0.14820 0.24830 0.06540
% java Transition < tiny.txt | java RandomSurfer 1000000
0.27324 0.26568 0.14581 0.24737 0.06790
```

.02, .47, .02, and .02, and `sum` takes on the values `0.0`, `0.47`, `0.49`, `0.96`, `0.98`, and `1.0`. These values indicate that the probabilities define the five intervals (0, .47), (.47, .49), (.49, .96), (.96, .98), and (.98, 1), one for each page. Now, suppose that `Math.random()` returns the value `.71`. We increment `j` from 0 to 1 to 2 and stop there, which indicates that `.71` is in the interval (.49, .96), so we send the surfer to the third page (page `2`). Then, we perform the same computation for `p[2]`, and the random surfer is off and surfing. For large *N*, we can use *binary search* to substantially speed up this computation (see Exercise 4.2.36). Typically, we are interested in speeding up the search in this situation because we are likely to need a huge number of random moves, as you will see.

Markov chains. The random process that describes the surfer's behavior is known as a *Markov chain*, named after the Russian mathematician Andrey Markov, who developed the concept in the early 20th century. Markov chains are widely applicable, well-studied, and have many remarkable and useful properties. For example, you may have wondered why `RandomSurfer` starts the random surfer at page 0 whereas you might have expected a random choice. A basic limit theorem for Markov chains says that the surfer could start *anywhere*, because the probability that a random surfer eventually winds up on any particular page is the same for all starting pages! No matter where the surfer starts, the process eventually stabilizes to a point where further surfing provides no further information. This phenomenon is known as *mixing*. Though this phenomenon is perhaps counterintuitive at first, it explains coherent behavior in a situation that might seem chaotic. In the present context, it captures the idea that the web looks pretty much the same to everyone after surfing for a sufficiently long time. However, not all Markov chains have this mixing property. For example, if we eliminate the random leap from our model, certain configurations of web pages can present problems for the surfer. Indeed, there exist on the web sets of pages known as *spider traps*, which are designed to attract incoming links but have no outgoing links. Without the random leap, the surfer could get stuck in a spider trap. The primary purpose of the 90-10 rule is to guarantee mixing and eliminate such anomalies.

Page ranks. The `RandomSurfer` simulation is straightforward: it loops for the indicated number of moves, randomly surfing through the graph. Because of the mixing phenomenon, increasing the number of iterations gives increasingly accurate estimates of the probability that the surfer lands on each page (the page

ranks). How do the results compare with your intuition when you first thought about the question? You might have guessed that page 4 was the lowest-ranked page, but did you think that pages 0 and 1 would rank higher than page 3? If we want to know which page is the highest rank, we need more precision and more accuracy. `RandomSurfer` needs 10^n moves to get answers precise to n decimal places and many more moves for those answers to stabilize to an accurate value. For our example, it takes tens of thousands of iterations to get answers accurate to two decimal places and millions of iterations to get answers accurate to three places (see EXERCISE 1.6.5). The end result is that page 0 beats page 1 by 27.3% to 26.6%. That such a tiny difference would appear in such a small problem is quite surprising: if you guessed that page 0 is the most likely spot for the surfer to end up, you were lucky! Accurate page rank estimates for the web are valuable in practice for many reasons. First, using them to put in order the pages that match the search criteria for web searches proved to be vastly more in line with people's expectations than previous methods. Next, this measure of confidence and reliability led to the investment of huge amounts of money in web advertising based on page ranks. Even in our tiny example, page ranks might be used to convince advertisers to pay up to four times as much to place an ad on page 0 as on page 4. Computing page ranks is mathematically sound, an interesting computer science problem, and big business, all rolled into one.



Page ranks with histogram

Visualizing the histogram. With `StdDraw`, it is also easy to create a visual representation that can give you a feeling for how the random surfer visit frequencies converge to the page ranks. Simply add

```
StdDraw.clear();
StdDraw.setXscale(-1, N);
StdDraw.setYscale(0, t);
StdDraw.setPenRadius(.5/N);
for (int i = 0; i < N; i++)
   StdDraw.line(i, 0, i, freq[i]);
StdDraw.show(20);
```

to the random move loop, run `RandomSurfer` for large values of `T`, and you will see a drawing of the frequency histogram that eventually stabilizes to the page ranks. After you have used this tool once, you are likely to find yourself using it *every* time

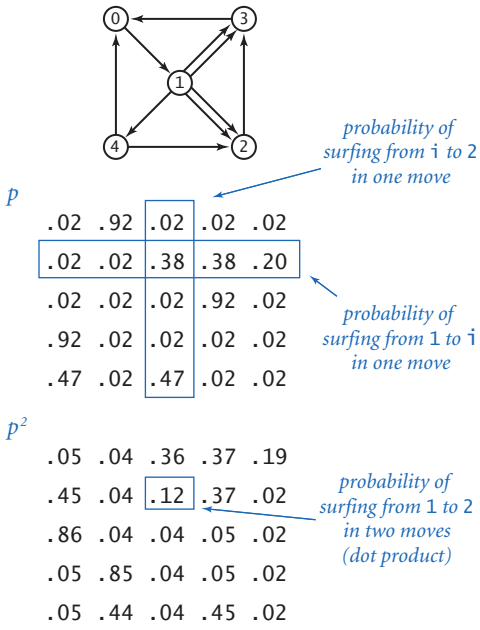
you want to study a new model (perhaps with some minor adjustments to handle larger models).

Studying other models. `RandomSurfer` and `Transition` are excellent examples of data-driven programs. You can easily create a data model just by creating a file like `tiny.txt` that starts with an integer `N` and then specifies pairs of integers between `0` and `N-1` that represent links connecting pages. You are encouraged to run it for various data models as suggested in the exercises, or to make up some models of your own to study. If you have ever wondered how web page ranking works, this calculation is your chance to develop better intuition about what causes one page to be ranked more highly than another. What kind of page is likely to be rated highly? One that has many links to other pages, or one that has just a few links to other pages? The exercises in this section present many opportunities to study the behavior of the random surfer. Since `RandomSurfer` uses standard input, you can write simple programs that generate large input models, pipe their output to `RandomSurfer`, and therefore study the random surfer on large models. Such flexibility is an important reason to use standard input and standard output.

Directly simulating the behavior of a random surfer to understand the structure of the web is appealing, but it has limitations. Think about the following question: Could you use it to compute page ranks for a web model with millions (or billions!) of web pages and links? The quick answer to this question is *no*, because you cannot even afford to store the transition matrix for such a large number of pages. A matrix for millions of pages would have *trillions* of entries. Do you have that much space on your computer? Could you use `RandomSurfer` to find page ranks for a smaller model with, say, thousands of pages? To answer this question, you might run multiple simulations, record the results for a large number of trials, and then interpret those experimental results. We do use this approach for many scientific problems (the gambler's ruin problem is one example; Section 2.4 is devoted to another), but it can be very time-consuming, as a huge number of trials may be necessary to get the desired accuracy. Even for our tiny example, we saw that it takes millions of iterations to get the page ranks accurate to three or four decimal places. For larger models, the required number of iterations to obtain accurate estimates becomes truly huge.

Mixing a Markov chain It is important to remember that the page ranks are a property of the web model, not any particular approach for computing it. That is, `RandomSurfer` is just *one* way to compute page ranks. Fortunately, a simple computational model based on a well-studied area of mathematics provides a far more efficient approach than simulation to the problem of computing page ranks. That model makes use of the basic arithmetic operations on two-dimensional matrices that we considered in SECTION 1.4.

Squaring a Markov chain. What is the probability that the random surfer will move from page `i` to page `j` in *two* moves? The first move goes to an intermediate page `k`, so we calculate the probability of moving from `i` to `k` and then from `k` to `j` for all possible `k` and add up the results. For our example, the probability of moving from `1` to `2` in two moves is the probability of moving from `1` to `0` to `2` (.02×.02), plus the probability of moving from `1` to `1` to `2` (.02×.38), plus the probability of moving from `1` to `2` to `2` (.38×.02), plus the probability of moving from `1` to `3` to `2` (.38×.02), plus the probability of moving from `1` to `4` to `2` (.20×.47), which adds up to a grand total of .1172. The same process works for each pair of pages. *This calculation is one that we have seen before*, in the definition of matrix multiplication: the entry in row `i` and column `j` in the result is the dot product of row `i` and column `j` in the original. In other words, the result of multiplying `p[][]` by itself is a matrix where the entry in row `i` and column `j` is the probability that the random surfer moves from page `i` to page `j` in two moves. Studying the entries of the two-move transition matrix for our example is well worth your time and will help you better understand the movement of the random surfer. For instance, the largest entry in the square is the one in row 2 and column 0, reflecting the fact that a surfer starting on page 2 has only one link out, to page 3, where there is also only one link out, to page 0. Therefore, by far the most likely outcome for a surfer start-



Squaring a Markov chain

ing on page 2 is to end up in page 0 after two moves. All of the other two-move routes involve more choices and are less probable. It is important to note that this is an exact computation (up to the limitations of Java's floating-point precision), in contrast to `RandomSurfer`, which produces an estimate and needs more iterations to get a more accurate estimate.

The power method. We might then calculate the probabilities for three moves by multiplying by `p[][]` again, and for four moves by multiplying by `p[][]` yet again, and so forth. However, matrix-matrix multiplication is expensive, and we are actually interested in a *vector*-matrix calculation. For our example, we start with the vector

```
[1.0 0.0 0.0 0.0 0.0 ]
```

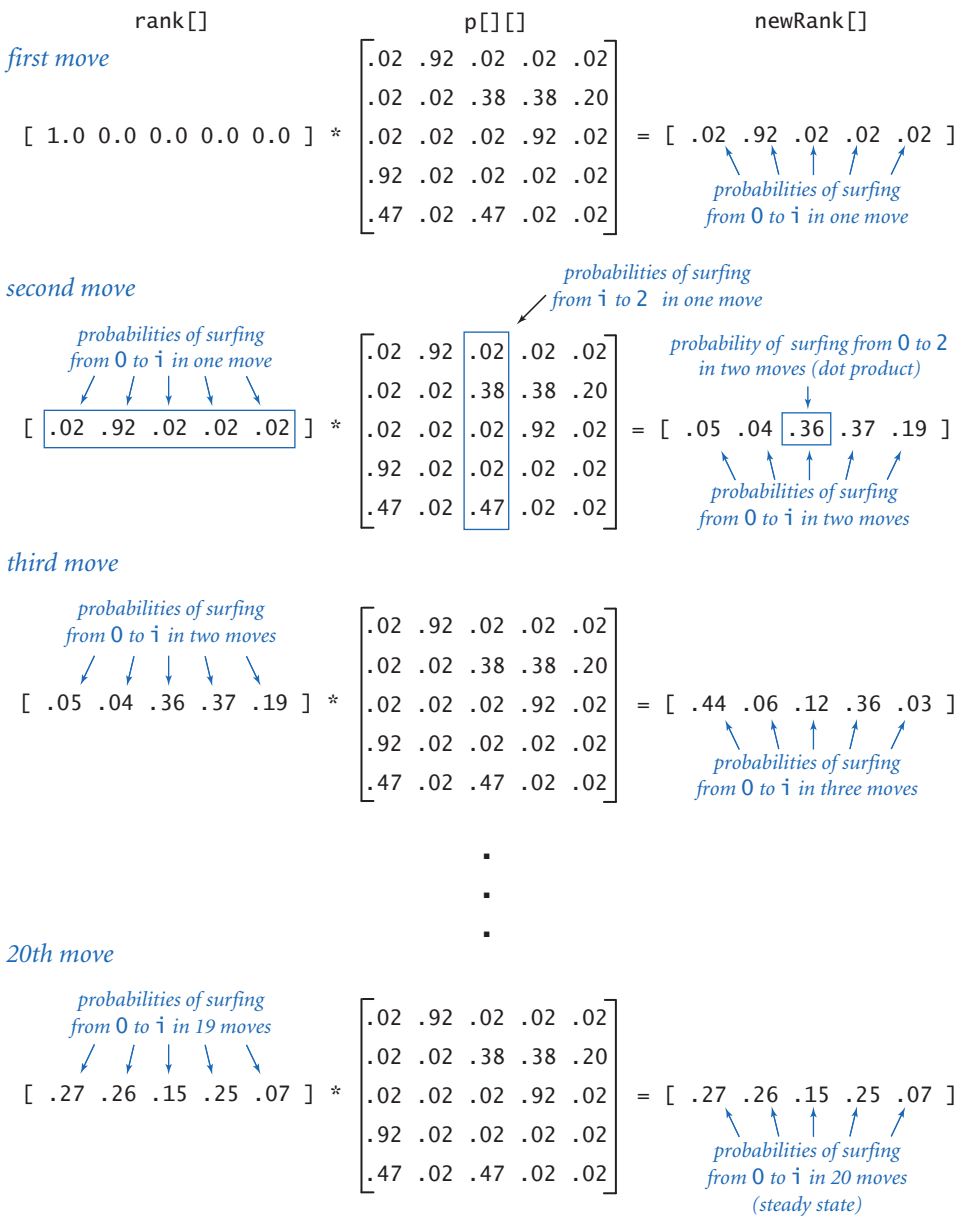
which specifies that the random surfer starts on page 0. Multiplying this vector by the transition matrix gives the vector

```
[.02 .92 .02 .02 .02 ]
```

which is the probabilities that the surfer winds up on each of the pages after one step. Now, multiplying *this* vector by the transition matrix gives the vector

```
[.05 .04 .36 .37 .19 ]
```

which contains the probabilities that the surfer winds up on each of the pages after *two* steps. For example, the probability of moving from 0 to 2 in two moves is the probability of moving from 0 to 0 to 2 (.02×.02), plus the probability of moving from 0 to 1 to 2 (.92×.38), plus the probability of moving from 0 to 2 to 2 (.02×.02), plus the probability of moving from 0 to 3 to 2 (.02×.02), plus the probability of moving from 0 to 4 to 2 (.02×.47), which adds up to a grand total of .36 . From these initial calculations, the pattern is clear: *The vector giving the probabilities that the random surfer is at each page after t steps is precisely the product of the corresponding vector for t − 1 steps and the transition matrix.* By the basic limit theorem for Markov chains, this process converges to the same vector no matter where we start; in other words, after a sufficient number of moves, the probability that the surfer ends up on any given page is independent of the starting point. `Markov` (PROGRAM 1.6.3) is an implementation that you can use to check convergence for our example. For instance, it gets the same results (the page ranks accurate to two decimal places) as `RandomSurfer`, but with just 20 matrix-vector multiplications



The power method for computing page ranks (limit values of transition probabilities)

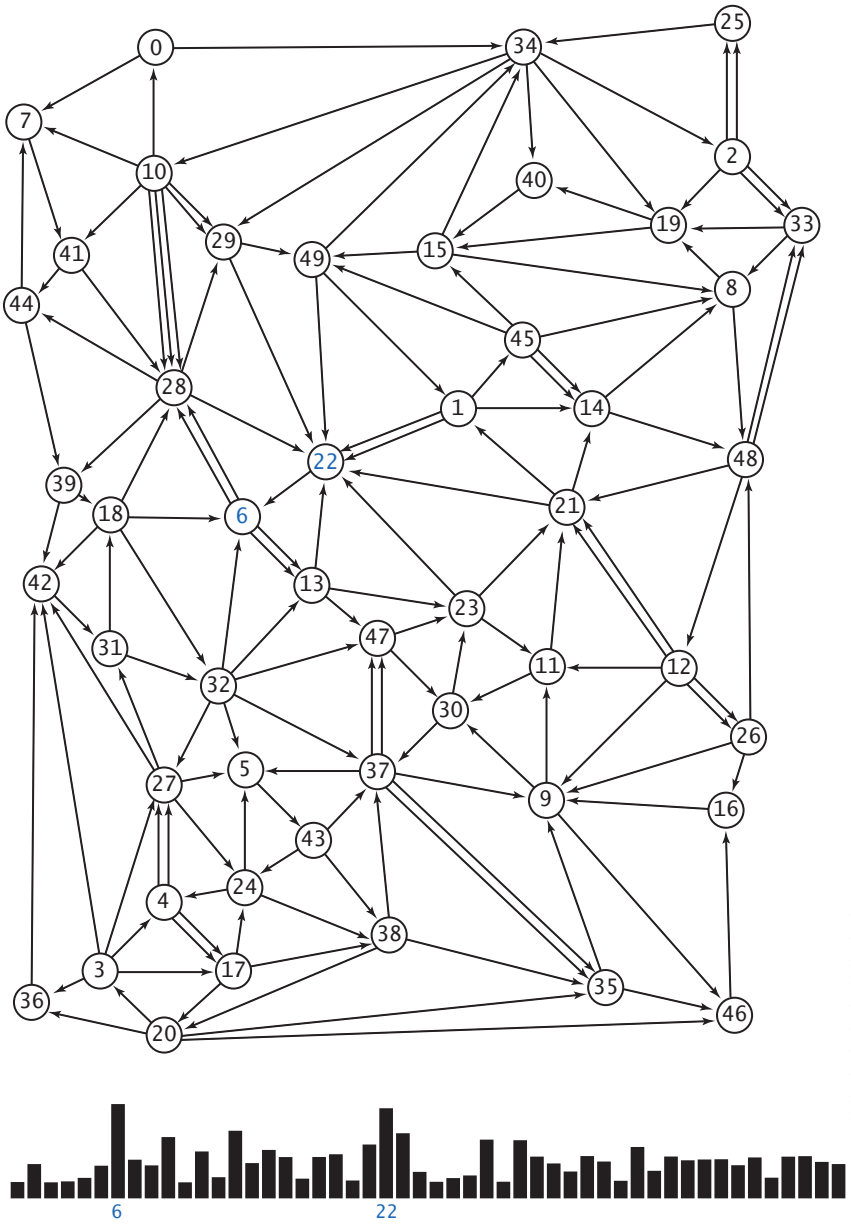
Program 1.6.3 Mixing a Markov chain

```
public class Markov
{  //  Compute page ranks after T moves.
   public static void main(String[] args)
   {
      int T = Integer.parseInt(args[0]);
      int N = StdIn.readInt();
      StdIn.readInt();

      // Read p[][] from StdIn.
      double[][] p = new double[N][N];
      for (int i = 0; i < N; i++)
         for (int j = 0; j < N; j++)
            p[i][j] = StdIn.readDouble();

      // Use the power method to compute page ranks.
      double[] rank = new double[N];
      rank[0] = 1.0;
      for (int t = 0; t < T; t++)
      {  // Compute effect of next move on page ranks.
         double[] newRank = new double[N];
         for (int j = 0; j < N; j++)
         {  //  New rank of page j is dot product
            //  of old ranks and column j of p[][].
            for (int k = 0; k < N; k++)
               newRank[j] += rank[k]*p[k][j];
         }

         for (int j = 0; j < N; j++)
            rank[j] = newRank[j];
      }

      for (int i = 0; i < N; i++)   // Print page ranks.
         StdOut.printf("%8.5f", rank[i]);
      StdOut.println();
   }
}
```

T	*number of iterations*
N	*number of pages*
p[][]	*transition matrix*
rank[]	*page ranks*
newRank[]	*new page ranks*

This program reads a transition matrix from standard input and computes the probabilities that a random surfer lands on each page (page ranks) after the number of steps specified as command-line argument.

```
% java Transition < tiny.txt | java Markov 20
 0.27245 0.26515 0.14669 0.24764 0.06806

% java Transition < tiny.txt | java Markov 40
 0.27303 0.26573 0.14618 0.24723 0.06783
```



Page ranks with histogram for a larger example

instead of the tens of thousands of iterations needed by `RandomSurfer`. Another 20 multiplications gives the results accurate to three decimal places, as compared with millions of iterations for `RandomSurfer`, and just a few more give the results to full precision (see EXERCISE 1.6.6).

MARKOV CHAINS ARE WELL-STUDIED, BUT THEIR impact on the web was not truly felt until 1998, when two graduate students, Sergey Brin and Lawrence Page, had the audacity to build a Markov chain and compute the probabilities that a random surfer hits each page for *the whole web*. Their work revolutionized web search and is the basis for the page ranking method used by GOOGLE, the highly successful web search company that they founded. Specifically, the company periodically recomputes the random surfer's probability for each page. Then, when you do a search, it lists the pages related to your search keywords in order of these ranks. Such page ranks now predominate because they somehow correspond to the expectations of typical web users, reliably providing them with *relevant* web pages for typical searches. The computation that is involved is enormously time-consuming, due to the huge number of pages on the web, but the result has turned out to be enormously profitable and well worth the expense. The method used in `Markov` is far more efficient than simulating the behavior of a random surfer, but it is still too slow to actually compute the probabilities for a huge matrix corresponding to all the pages on the web. That computation is enabled by better data structures for graphs (see CHAPTER 4).

Lessons Developing a full understanding of the random surfer model is beyond the scope of this book. Instead, our purpose is to show you an application that involves writing a bit more code than the short programs that we have been using to teach specific concepts. What specific lessons can we learn from this case study?

We already have a full computational model. Primitive types of data and strings, conditionals and loops, arrays, and standard input/output enable you to address interesting problems of all sorts. Indeed, it is a basic precept of theoretical computer science that this model suffices to specify any computation that can be performed on any reasonable computing device. In the next two chapters, we discuss two critical ways in which the model has been extended to drastically reduce the amount of time and effort required to develop large and complex programs.

Data-driven code is prevalent. The concept of using standard input and output streams and saving data in files is a powerful one. We write filters to convert from one kind of input to another, generators that can produce huge input files for study, and programs that can handle a wide variety of different models. We can save data for archiving or later use. We can also process data derived from some other source and then save it in a file, whether it is from a scientific instrument or a distant website. The concept of data-driven code is an easy and flexible way to support this suite of activities.

Accuracy can be elusive. It is a mistake to assume that a program produces accurate answers simply because it can print numbers to many decimal places of precision. Often, the most difficult challenge that we face is ensuring that we have accurate answers.

Uniform random numbers are only a start. When we speak informally about random behavior, we often are thinking of something more complicated than the "every value equally likely" model that `Math.random()` gives us. Many of the problems that we consider involve working with random numbers from other distributions, such as `RandomSurfer`.

Efficiency matters. It is also a mistake to assume that your computer is so fast that it can do *any* computation. Some problems require much more computational effort than others. CHAPTER 4 is devoted to a thorough discussion of evaluating the performance of the programs that you write. We defer detailed consideration of such issues until then, but remember that you always need to have some general idea of the performance requirements of your programs.

PERHAPS THE MOST IMPORTANT LESSON TO learn from writing programs for complicated problems like the example in this section is that *debugging is difficult.* The polished programs in the book mask that lesson, but you can rest assured that each one is the product of a long bout of testing, fixing bugs, and running the programs on numerous inputs. Generally we avoid describing bugs and the process of fixing them in the text because that makes for a boring account and overly focuses attention on bad code, but you can find some examples and descriptions in the exercises and on the booksite.

Exercises

1.6.1 Modify `Transition` to take the leap probability from the command line and use your modified version to examine the effect on page ranks of switching to an 80-20 rule or a 95-5 rule.

1.6.2 Modify `Transition` to ignore the effect of multiple links. That is, if there are multiple links from one page to another, count them as one link. Create a small example that shows how this modification can change the order of page ranks.

1.6.3 Modify `Transition` to handle pages with no outgoing links, by filling rows corresponding to such pages with the value 1/*N*.

1.6.4 The code fragment in `RandomSurfer` that generates the random move fails if the probabilities in the row `p[page]` do not add up to 1. Explain what happens in that case, and suggest a way to fix the problem.

1.6.5 Determine, to within a factor of 10, the number of iterations required by `RandomSurfer` to compute page ranks to four decimal places and to five decimal places for `tiny.txt`.

1.6.6. Determine the number of iterations required by `Markov` to compute page ranks to three decimal places, to four decimal places, and to ten decimal places for `tiny.txt`.

1.6.7 Download the file `medium.txt` from the booksite (which reflects the 50-page example depicted in this section) and add to it links *from* page 23 *to* every other page. Observe the effect on the page ranks, and discuss the result.

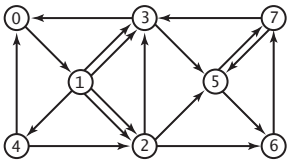
1.6.8 Add to `medium.txt` (see the previous exercise) links *to* page 23 *from* every other page, observe the effect on the page ranks, and discuss the result.

1.6.9 Suppose that your page is page `23` in `medium.txt`. Is there a link that you could add from your page to some other page that would *raise* the rank of *your* page?

1.6.10 Suppose that your page is page `23` in `medium.txt`. Is there a link that you could add from your page to some other page that would *lower* the rank of *that* page?

1.6.11 Use `Transition` and `RandomSurfer` to determine the transition probabilities for the eight-page example shown below.

1.6.12 Use `Transition` and `Markov` to determine the transition probabilities for the eight-page example shown below.



Eight-page example

Creative Exercises

1.6.13 *Matrix squaring.* Write a program like `Markov` that computes page ranks by repeatedly squaring the matrix, thus computing the sequence p, p^2, p^4, p^8, p^{16}, and so forth. Verify that all of the rows in the matrix converge to the same values.

1.6.14 *Random web.* Write a generator for `Transition` that takes as input a page count `N` and a link count `M` and prints to standard output `N` followed by `M` random pairs of integers from `0` to `N-1`. (See SECTION 4.5 for a discussion of more realistic web models.)

1.6.15 *Hubs and authorities.* Add to your generator from the previous exercise a fixed number of *hubs*, which have links pointing to them from 10% of the pages, chosen at random, and *authorities*, which have links pointing from them to 10% of the pages. Compute page ranks. Which rank higher, hubs or authorities?

1.6.16 *Page ranks.* Design an array of pages and links where the highest-ranking page has fewer links pointing to it than some other page.

1.6.17 *Hitting time.* The hitting time for a page is the expected number of moves between times the random surfer visits the page. Run experiments to estimate page hitting times for `tiny.txt`, compare with page ranks, formulate a hypothesis about the relationship, and test your hypothesis on `medium.txt`.

1.6.18 *Cover time.* Write a program that estimates the time required for the random surfer to visit every page at least once, starting from a random page.

1.6.19 *Graphical simulation.* Create a graphical simulation where the size of the dot representing each page is proportional to its rank. To make your program data-driven, design a file format that includes coordinates specifying where each page should be drawn. Test your program on `medium.txt`.