

# 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

Copyright © 2008 Pearson Education, Inc. All rights reserved. No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form or by any means, electronic, mechanical, photocopying, recording, or otherwise, without the prior written permission of the publisher. Printed in the United States of America. For information on obtaining permission for use of material in this work, please submit a written request to Pearson Education, Inc., Rights and Contracts Department, 501 Boylston Street, Suite 900, Boston, MA 02116, fax (617) 671-3447, or online at <http://www.pearsoned.com/legal/permissions.htm>.

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.

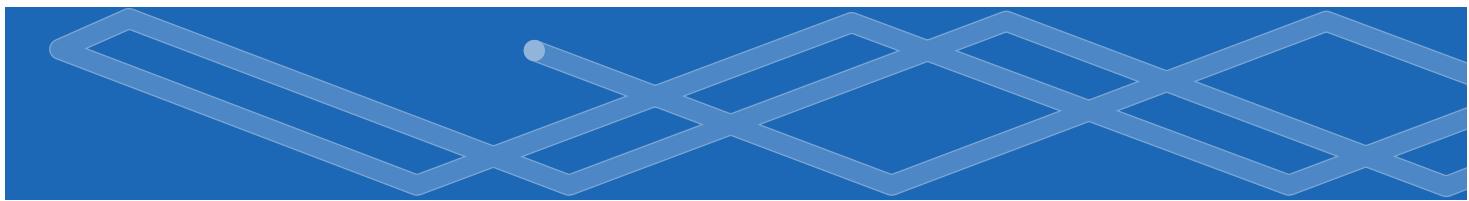
For *casual readers* (including instructors, teaching assistants, and students!), the booksite is a resource for accessing all manner of extra information associated with the book's content. All of the booksite content provides web links and other routes to pursue more information about the topic under 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*

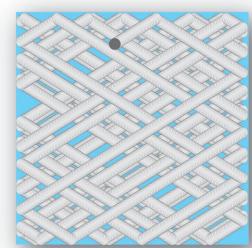
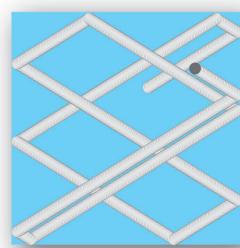
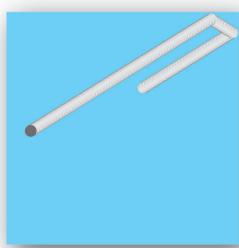
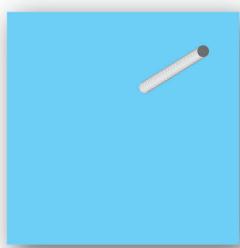




# Contents

<i>Preface</i> . . . . .	v
<i>Elements of Programming</i> . . . . .	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
<i>Functions and Modules</i> . . . . .	183
2.1 Static Methods	184
2.2 Libraries and Clients	218
2.3 Recursion	254
2.4 Case Study: Percolation	286
<i>Object-Oriented Programming</i> . . . . .	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
<i>Algorithms and Data Structures</i> . . . . .	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
<i>Context</i> . . . . .	695
<i>Index</i> . . . . .	699

# *Chapter One*

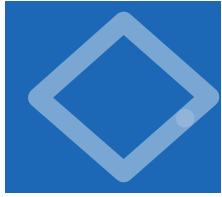


# *Elements of Programming*

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

<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.

1.1.1 Hello, World . . . . . 6

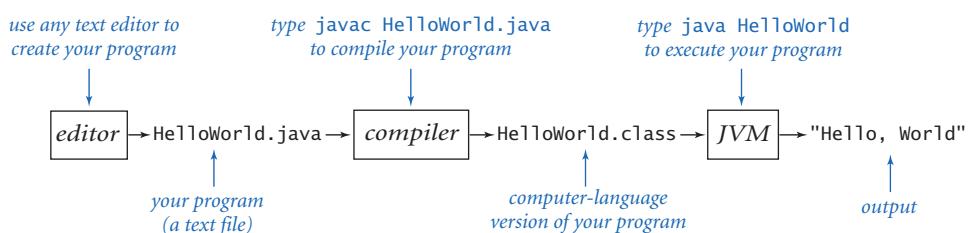
1.1.2 Using a command-line argument . . 8

*Programs in this section*

***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`.

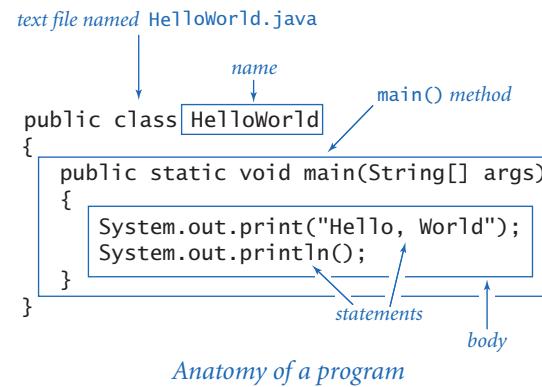
When you type `java` followed by a class name in your terminal application, the system calls the `main()` method that you defined in that class, and executes its statements in order, one by one. Thus, typing `java HelloWorld` causes the system to call on the `main()` method in PROGRAM 1.1.1 and execute 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.

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.*

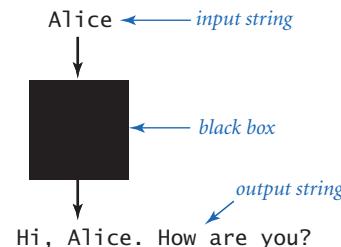
```
% javac UseArgument.java
% java UseArgument Alice
Hi, Alice. How are you?
% java UseArgument Bob
Hi, Bob. How are you?
```

**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`.

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.



A bird's-eye view of a Java program



## 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 `He1-`



World 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`

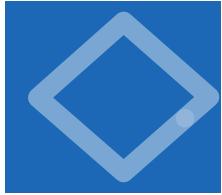
**1.1.3** Describe what happens if you misspell (by, say, omitting the second letter) the following in `HelloWorld.java`:

- 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.

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.

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*

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.

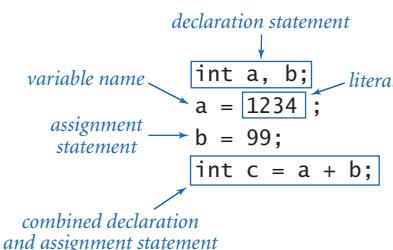
**Variables.** A variable is a name that we use to refer to a data-type value. We use variables to keep track of changing values as a computation unfolds. For example,

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.

**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.



Using a primitive data type

*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):

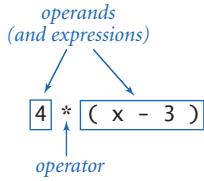
```
int t = a;
a = b;
b = t;
```

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

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,

<i>expression</i>	<i>value</i>
<code>"Hi, " + "Bob"</code>	<code>"Hi, Bob"</code>
<code>"1" + " 2 " + "1"</code>	<code>"1 2 1"</code>
<code>"1234" + " " + "99"</code>	<code>"1234 + 99"</code>
<code>"1234" + "99"</code>	<code>"123499"</code>

*Typical String expressions*

`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 it 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  $n$ th 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.

<i>values</i>	sequences of characters
<i>typical literals</i>	<code>"Hello," "1 " " * "</code>
<i>operation</i>	concatenate
<i>operator</i>	<code>+</code>

*Java's built-in String data type*

### 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.

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

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

*Typical int expressions*

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

*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.

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  $\pi$ . Alternatively, we can use a notation like scientific notation: the literal 6.022e23 represents the number  $6.022 \times 10^{23}$ . 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

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

*Typical double expressions*

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

*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  $\phi$  and  $1 - \phi$ , where  $\phi$  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.

The most important operations defined for `booleans` 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

<i>values</i>	true or false		
<i>literals</i>	<code>true</code> <code>false</code>		
<i>operations</i>	<code>and</code>	<code>or</code>	<code>not</code>
<i>operators</i>	<code>&amp;&amp;</code>	<code>  </code>	<code>!</code>

*Java's built-in boolean data type*

<code>a</code>	<code>!a</code>	<code>a</code>	<code>b</code>	<code>a &amp;&amp; b</code>	<code>a    b</code>
<code>true</code>	<code>false</code>	<code>false</code>	<code>false</code>	<code>false</code>	<code>false</code>
<code>false</code>	<code>true</code>	<code>false</code>	<code>true</code>	<code>false</code>	<code>true</code>
		<code>true</code>	<code>false</code>	<code>false</code>	<code>true</code>
		<code>true</code>	<code>true</code>	<code>true</code>	<code>true</code>

*Truth-table definitions of boolean operations*

a	b	<code>a &amp;&amp; b</code>	<code>!a</code>	<code>!b</code>	<code>!a    !b</code>	<code>!(a    b)</code>
<code>false</code>	<code>false</code>	<code>false</code>	<code>true</code>	<code>true</code>	<code>true</code>	<code>false</code>
<code>false</code>	<code>true</code>	<code>false</code>	<code>true</code>	<code>false</code>	<code>true</code>	<code>false</code>
<code>true</code>	<code>false</code>	<code>false</code>	<code>false</code>	<code>true</code>	<code>true</code>	<code>false</code>
<code>true</code>	<code>true</code>	<code>true</code>	<code>false</code>	<code>false</code>	<code>false</code>	<code>true</code>

*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

<i>non-negative discriminant?</i>	<code>(b*b - 4.0*a*c) &gt;= 0.0</code>
<i>beginning of a century?</i>	<code>(year % 100) == 0</code>
<i>legal month?</i>	<code>(month &gt;= 1) &amp;&amp; (month &lt;= 12)</code>

*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 `ints` to check that `(2 <= 2)` is `true` but quite another to compare two `doubles` 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.)

<i>op</i>	<i>meaning</i>	<i>true</i>	<i>false</i>
<code>==</code>	<i>equal</i>	<code>2 == 2</code>	<code>2 == 3</code>
<code>!=</code>	<i>not equal</i>	<code>3 != 2</code>	<code>2 != 2</code>
<code>&lt;</code>	<i>less than</i>	<code>2 &lt; 13</code>	<code>2 &lt; 2</code>
<code>&lt;=</code>	<i>less than or equal</i>	<code>2 &lt;= 2</code>	<code>3 &lt;= 2</code>
<code>&gt;</code>	<i>greater than</i>	<code>13 &gt; 2</code>	<code>2 &gt; 13</code>
<code>&gt;=</code>	<i>greater than or equal</i>	<code>3 &gt;= 2</code>	<code>2 &gt;= 3</code>

*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)
    void println(String s)
    void println()
```

*print s*  
*print s, followed by a newline*  
*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:

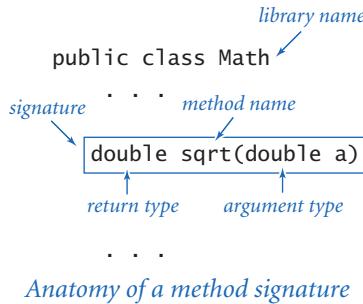
---

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

*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.



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)`.

The `Math` library also defines the precise constant values `PI` (for  $\pi$ ) 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:

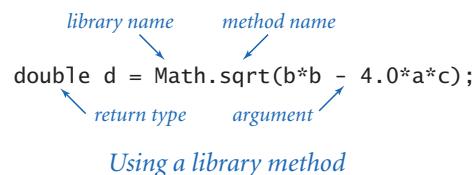
<code>int Integer.parseInt(String s)</code>	<i>convert s to an int value</i>
<code>double Double.parseDouble(String s)</code>	<i>convert s to a double value</i>
<code>long Long.parseLong(String s)</code>	<i>convert s to a long value</i>

*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*.

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



<i>expression</i>	<i>library</i>	<i>type</i>	<i>value</i>
<code>Integer.parseInt("123")</code>	<code>Integer</code>	<code>int</code>	123
<code>Math.sqrt(5.0*5.0 - 4.0*4.0)</code>	<code>Math</code>	<code>double</code>	3.0
<code>Math.random()</code>	<code>Math</code>	<code>double</code>	<i>random in [0, 1)</i>
<code>Math.round(3.14159)</code>	<code>Math</code>	<code>long</code>	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
<code>"1234" + 99</code>	String	"123499"
<code>Integer.parseInt("123")</code>	int	123
<code>(int) 2.71828</code>	int	2
<code>Math.round(2.71828)</code>	long	3
<code>(int) Math.round(2.71828)</code>	int	3
<code>(int) Math.round(3.14159)</code>	int	3
<code>11 * 0.3</code>	double	3.3
<code>(int) 11 * 0.3</code>	double	3.3
<code>11 * (int) 0.3</code>	int	0
<code>(int) (11 * 0.3)</code>	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 `Strings` 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 `ints` 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  $\theta$  entered as a command-line argument. Just print the value. Why are the values not always exactly 1?

`boolean`

**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:

<code>int a = 1;</code>	<code>boolean a = true;</code>	<code>int a = 2;</code>
<code>a = a + a;</code>	<code>a = !a;</code>	<code>a = a * a;</code>
<code>a = a + a;</code>	<code>a = !a;</code>	<code>a = a * a;</code>
<code>a = a + a;</code>	<code>a = !a;</code>	<code>a = a * a;</code>

**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

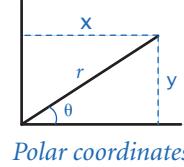
**1.2.24** *Loan payments.* Write a program that calculates the monthly payments you would have to make over a given number of years to pay off a loan at a given interest rate compounded continuously, taking the number of years  $t$ , the principal  $P$ , and the annual interest rate  $r$  as command-line arguments. The desired value is given by the formula  $P e^{rt}$ . 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  $\theta$ . Use the Java method `Math.atan2(y, x)` which computes the arctangent value of  $y/x$  that is in the range from  $-\pi$  to  $\pi$ .



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  $\varphi$  and longitude  $\lambda$  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  $\lambda_0$  is the longitude of the point in the center of the map. Write a program that takes  $\lambda_0$  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:

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

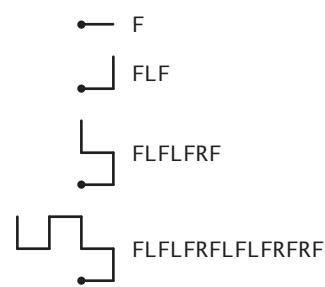
**1.2.33** *Great circle.* Write a program `GreatCircle` that takes four command-line arguments—`x1`, `y1`, `x2`, and `y2`—(the latitude and longitude, in degrees, of two points on the earth) and prints out the great-circle distance between them. The great-circle distance (in nautical miles) is given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

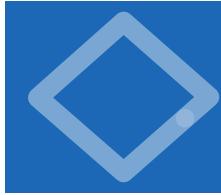
**1.2.34** *Three-sort.* Write a program that takes three `int` values from the command line and prints them in ascending order. Use `Math.min()` and `Math.max()`.

**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.

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

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*

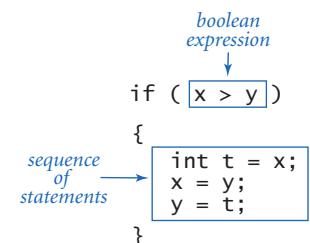
of an `if` statement is self-explanatory: the statement(s) in the sequence are to be executed if and only if the expression is `true`.

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.

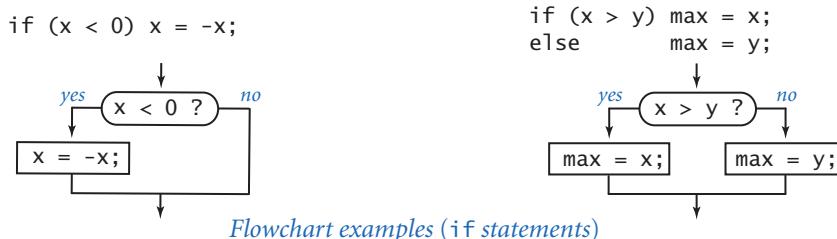
You can also add an `else` clause to an `if` statement, to express the concept of executing either one statement (or sequence of statements) or another, depending on whether the boolean expression is `true` or `false`, as in the following template:

```
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)

<i>absolute value</i>	<pre>if (x &lt; 0) x = -x;</pre>
<i>put x and y into sorted order</i>	<pre>if (x &gt; y) {     int t = x;     y = x;     x = t; }</pre>
<i>maximum of x and y</i>	<pre>if (x &gt; y) max = x; else         max = y;</pre>
<i>error check for division operation</i>	<pre>if (den == 0) System.out.println("Division by zero"); else           System.out.println("Quotient = " + num/den);</pre>
<i>error check for quadratic formula</i>	<pre>double discriminant = b*b - 4.0*c; if (discriminant &lt; 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); }</pre>

#### 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.

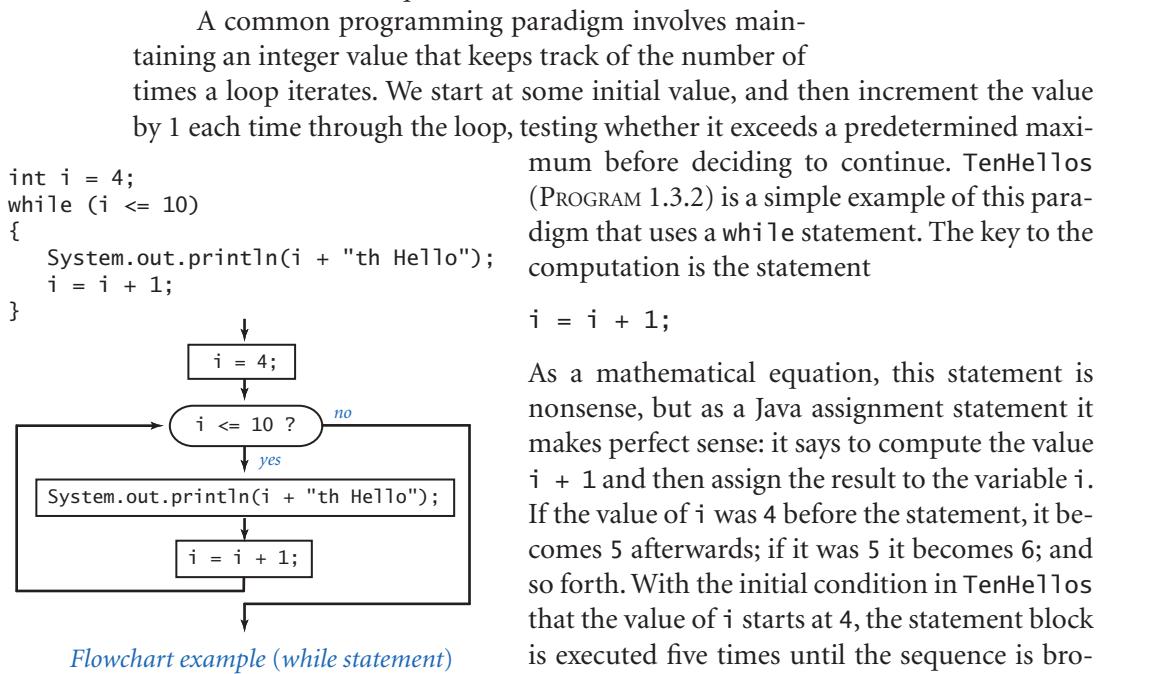
The `while` statement is equivalent to a sequence of identical `if` statements:

```

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

```

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.



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

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. *TenHello* (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 *TenHello* 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.

**Program 1.3.2 Your first while loop**

```
public class TenHello
{
    public static void main(String[] args)
    { // Print 10 Hello.
        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 TenHello
1st Hello
2nd Hello
3rd Hello
4th Hello
5th Hello
6th Hello
7th Hello
8th Hello
9th Hello
10th Hello
```

<i>i</i>	<i>i</i> <= 10	<i>output</i>
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 TenHello

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

<code>i</code>	<code>v</code>	<code>i &lt;= N</code>
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:

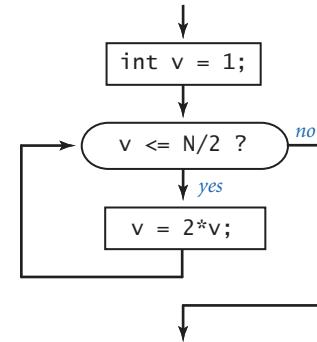
```
int v = 1;
while (v <= N/2)
    v = 2*v;
```

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 `TenHello`, 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



Flowchart for the statements

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

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 `TenHello` (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:

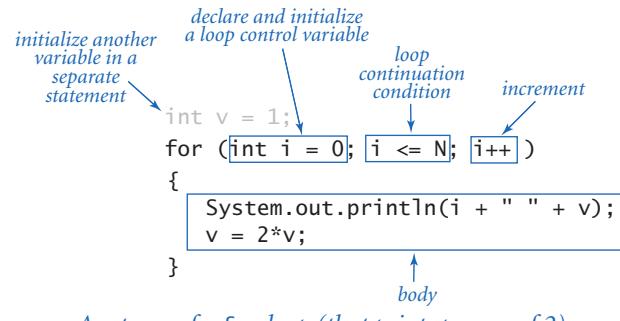
```
i = i + 1;    i++;    ++i;    i += 1;
```

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.

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.



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

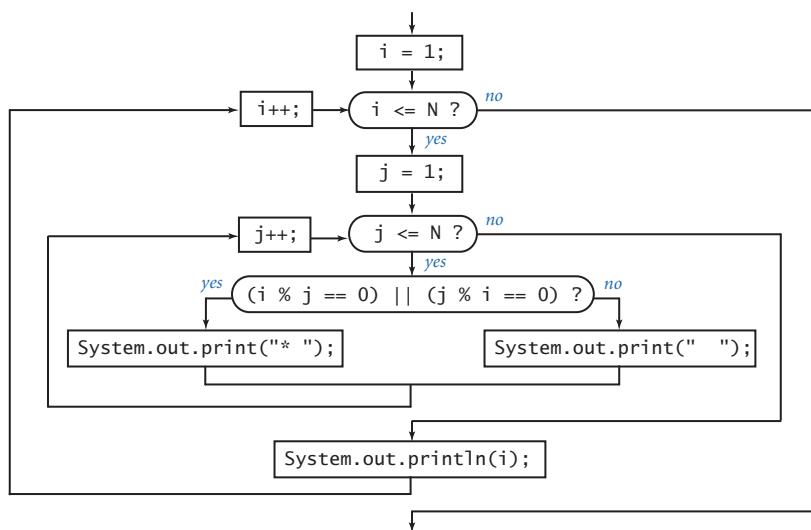
<i>print largest power of two less than or equal to N</i>	<pre>int v = 1; while (v &lt;= N/2)     v = 2*v; System.out.println(v);</pre>
<i>compute a finite sum (<math>1 + 2 + \dots + N</math>)</i>	<pre>int sum = 0; for (int i = 1; i &lt;= N; i++)     sum += i; System.out.println(sum);</pre>
<i>compute a finite product (<math>N! = 1 \times 2 \times \dots \times N</math>)</i>	<pre>int product = 1; for (int i = 1; i &lt;= N; i++)     product *= i; System.out.println(product);</pre>
<i>print a table of function values</i>	<pre>for (int i = 0; i &lt;= N; i++)     System.out.println(i + ' ' + 2*Math.PI*i/N);</pre>
<i>print the ruler function (see Program 1.2.1)</i>	<pre>String ruler = " "; for (int i = 1; i &lt;= N; i++)     ruler = ruler + i + ruler; System.out.println(ruler);</pre>

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:

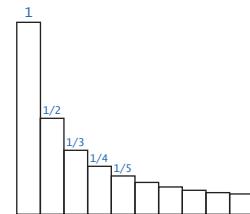
```
if      (income <      0) rate = 0.0;
else if (income < 47450) rate = .22;
else if (income < 114650) rate = .25;
else if (income < 174700) rate = .28;
else if (income < 311950) rate = .33;
else                      rate = .35;
```

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  $H_N = 1 + 1/2 + 1/3 + \dots + 1/N$ . 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**

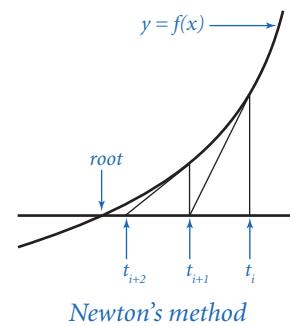
```
public class Harmonic
{
    public static void main(String[] args)
    { // Compute the Nth Harmonic number.
        int N = Integer.parseInt(args[0]);
        double sum = 0.0;
        for (int i = 1; i <= N; i++)
        { // Add the ith term to the sum
            sum += 1.0/i;
        }
        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  $\ln(10000) \approx 9.21034$ .

```
% 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



### 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
	2.000000000000000	1.0
1	1.500000000000000	1.333333333333333
2	1.416666666666666	1.4117647058823530
3	1.4142156862745097	1.4142114384748700
4	1.4142135623746899	1.4142135623715002
5	1.4142135623730950	1.4142135623730951

Trace of java Sqrt 2.0

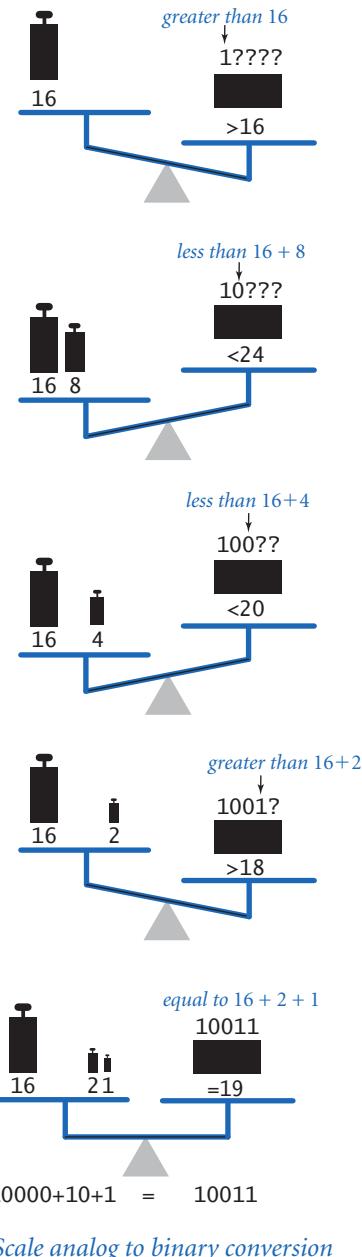
estimate  $t_i$ , compute a new estimate by drawing a line tangent to the curve  $y = f(x)$  at the point  $(t_i, f(t_i))$  and set  $t_{i+1}$  to the  $x$ -coordinate of the point where that line hits the  $x$ -axis. Iterating this process, we get closer to the root.

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
10111110101110000100000000
```

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

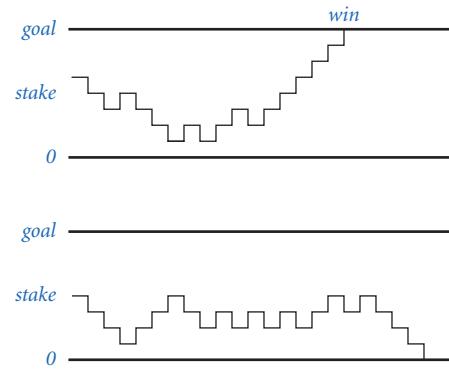
<i>n</i>	<i>binary representation</i>	<i>v</i>	<i>v &gt; 0</i>	<i>binary representation</i>	<i>n &lt; v</i>	<i>output</i>
19	10011	16	true	10000	false	1
3	0011	8	true	1000	true	0
3	011	4	true	100	true	0
3	01	2	true	10	false	1
1	1	1	true	1	false	1
0		0	false			

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

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

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

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.

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

**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 ( $i \leq n/i$ ) 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> }
```

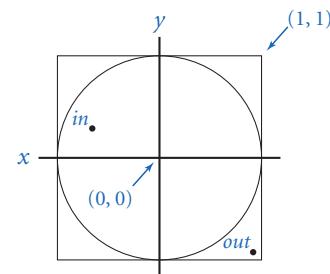
except that the first test of the condition is omitted. If the condition initially holds, there is no difference. For an example in which `do-while` is useful, consider the problem of generating points that are randomly distributed in the unit disk. We can use `Math.random()` to generate  $x$  and  $y$  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  $\pi$  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 `TenHello`s 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

```
public class BadHello
...
int i = 4;
while (i > 3)
{
    System.out.println
        (i + "th Hello");
    i = i + 1;
}
...
% java BadHello
1st Hello
2nd Hello
3rd Hello
5th Hello
6th Hello
7th Hello
...
```

*An infinite loop*

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 `TenHello`s indicated above and trying to stop it. On most systems, `<ctrl-c>` means *stop the current program*, and should do the job.

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 `BadHello`s 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.

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.

<i>program</i>	<i>description</i>
Flip	simulate a coin flip
TenHello	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	<code>while</code> loop within a <code>for</code> loop

*Summary of programs in this section*

**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> <stmtA> else <stmtB>
```

In Java this is equivalent to

```
if <expr1> { if <expr2> <stmtA> else <stmtB> }
```

even if you might have been thinking

```
if <expr1> { if <expr2> <stmtA> } else <stmtB>
```

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)      null increment
    v *= 2;           statement
    ↓
for (int v = 1; v <= N/2; )
    v *= 2;
for (int v = 1; v <= N/2; v *= 2)
    ; ← 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  $N = 16, 32, 64, \dots, 2048$ . 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*  $\phi$  (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  $1/1^2 + 1/2^2 + \dots + 1/N^2$  *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  $\pi$ .) 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 ( $i \leq N/i$ ) 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  $d_1 + 2d_2 + 3d_3 + \dots + 10d_{10}$  must be a multiple of 11 (here  $d_i$  denotes the  $i$ th digit from the right). The checksum digit  $d_{10}$  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  $e^x = 1 + x + x^2/2! + x^3/3! + \dots$ .

*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++) dem *= i;
double term = num/dem;
```

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  $\sin x = x - x^3/3! + x^5/5! - \dots$  and  $\cos x = 1 - x^2/2! + x^4/4! - \dots$ .

**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.

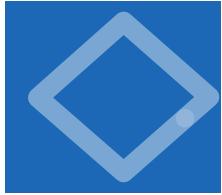
**1.3.41** *Median-of-5.* Write a program that takes five distinct integers from the command line and prints the median value (the value such that two of the others are smaller and two are larger). *Extra credit:* Solve the problem with a program that compares values fewer than seven times for any given 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  $n$   $n$ th powers are needed to obtain a sum that is itself an  $n$ th 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.

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  $i$ th 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*.

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.

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*

a	a[0]
	a[1]
	a[2]
	a[3]
	a[4]
	a[5]
	a[6]
	a[7]

*An array*

**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];
```

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.

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

*Trace of dot product computation*

<i>create an array with random values</i>	<pre>double[] a = new double[N]; for (int i = 0; i &lt; N; i++)     a[i] = Math.random();</pre>
<i>print the array values, one per line</i>	<pre>for (int i = 0; i &lt; N; i++)     System.out.println(a[i]);</pre>
<i>find the maximum of the array values</i>	<pre>double max = Double.NEGATIVE_INFINITY; for (int i = 0; i &lt; N; i++)     if (a[i] &gt; max) max = a[i];</pre>
<i>compute the average of the array values</i>	<pre>double sum = 0.0; for (int i = 0; i &lt; N; i++)     sum += a[i]; double average = sum / N;</pre>
<i>copy to another array</i>	<pre>double[] b = new double[N]; for (int i = 0; i &lt; N; i++)     b[i] = a[i];</pre>
<i>reverse the elements within an array</i>	<pre>for (int i = 0; i &lt; N/2; i++) {     double temp = b[i];     b[i] = b[N-1-i];     b[N-i-1] = temp; }</pre>

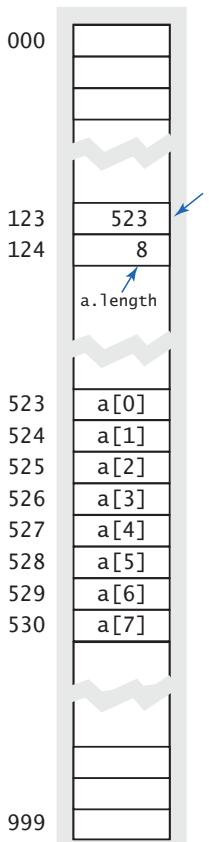
*Typical array-processing code (for arrays of N double values)*

**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 `ArrayIndexOutOfBoundsException` 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	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 `ArrayOutOfBoundsException` 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 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



*Coupon collection*

### 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  $N$ .

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.

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

*Trace for a typical run of  
java CouponCollector 6*

### 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
2	T	T	F	T	F	T	F	T	F	T	F	T	F	T	F	T	F	T	F	T	F	T	F	
3	T	T	F	T	F	T	F	F	F	T	F	F	F	T	F	T	F	F	T	F	T	F	T	
5	T	T	F	T	F	T	F	F	F	T	F	F	F	T	F	T	F	F	F	T	F	F	F	
	T	T	F	T	F	T	F	F	F	T	F	T	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  $i \leq N/i$ :

- 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.

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:

		a[1][2]
99	85	98
row 1 →	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

↑  
column 2

Anatomy of a  
two-dimensional array

```

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[][]`

`a[5] →`

a[0][0]	a[0][1]	a[0][2]
a[1][0]	a[1][1]	a[1][2]
a[2][0]	a[2][1]	a[2][2]
a[3][0]	a[3][1]	a[3][2]
a[4][0]	a[4][1]	a[4][2]
a[5][0]	a[5][1]	a[5][2]
a[6][0]	a[6][1]	a[6][2]
a[7][0]	a[7][1]	a[7][2]
a[8][0]	a[8][1]	a[8][2]
a[9][0]	a[9][1]	a[9][2]

*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.

*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

				row averages in column $N$								
				$N = 3$	99	85	98	94	<u>92+77+76</u>			
				98	57	78	77	<u>3</u>	92	77	76	81
				$M = 10$	94	32	11	45	99	34	22	51
					99	34	22	51	90	46	54	63
					76	59	88	74	92	66	89	82
					92	66	89	82	97	71	24	64
					89	29	38	52	92	55	57	<u>10</u>
					<u>85+57+...+29</u>							

#### 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

```
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

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.

$a[][]$	$.70 .20 .10$	$a[1][2]$
	$.30 .60 .10$	
	$.50 .10 .40$	
$b[][]$	$.80 .30 .50$	$b[1][2]$
	$.10 .40 .10$	
	$.10 .30 .40$	
$c[][]$	$1.5 .50 .60$	$c[1][2]$
	$.40 1.0 .20$	
	$.60 .40 .80$	

#### 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];
    }
}
```

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

$a[][]$	$.70 .20 .10$	$row 1$
	$.30 .60 .10$	
	$.50 .10 .40$	
$b[][]$	$.80 .30 .50$	$column 2$
	$.10 .40 .10$	
	$.10 .30 .40$	
$c[][]$	$c[1][2] = .3 * .5$	
	$+ .6 * .1$	
	$+ .1 * .4$	
	$= .25$	
	$.59 .32 .41$	
	$.31 .36 .25$	
	$.45 .31 .42$	

#### Matrix multiplication

*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  $a[]\cdot x[] = b[]$*

```
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[][]
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

x[]
.33
.33
.33

b[]
94
77
81
45
51
63
74
82
64
52
```

*row averages*

*Vector-matrix multiplication  $y[]\cdot a[] = c[]$*

```
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 .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
```

*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.

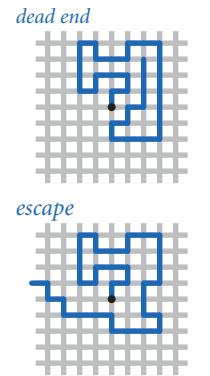
**Multidimensional arrays.** The same notation extends to allow us to write code using arrays that have any number of dimensions. For instance, we can declare and initialize a three-dimensional array with the code

```
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



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

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.

```

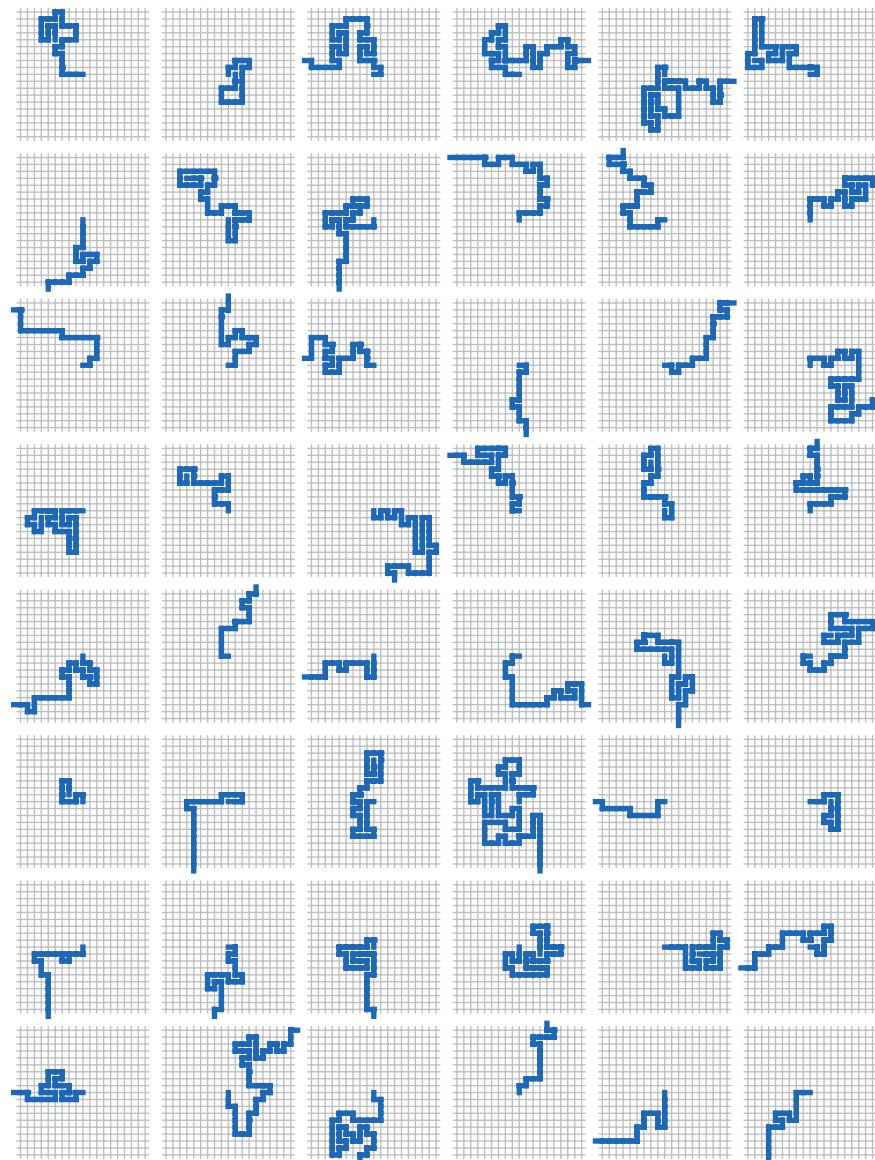
% java SelfAvoidingWalk 5 100
0% dead ends
% java SelfAvoidingWalk 20 100
36% dead ends
% java SelfAvoidingWalk 40 100
80% dead ends
% java SelfAvoidingWalk 80 100
98% dead ends
% java SelfAvoidingWalk 160 100
100% dead ends

```

```

% 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, your 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).

$H(1)$	$H(2)$	$H(4)$
<code>T</code>	<code>T T</code>	<code>T T T T</code>
	<code>T F</code>	<code>T F T F</code>
		<code>T T F F</code>
		<code>T F F T</code>

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.

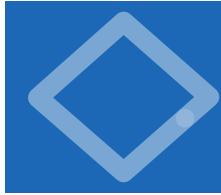
**1.4.34 Bridge hands.** In the game of bridge, four players are dealt hands of 13 cards each. An important statistic is the distribution of the number of cards in each suit in a hand. Which is the most likely, 5-3-3-2, 4-4-3-2, or 4-3-3-3?

**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  $a[N][k] = (a[N-1][k] + a[N-1][k-1])/2$ .

<i>Pascal's triangle</i>	<i>binomial distribution</i>
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.

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.

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*

**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.

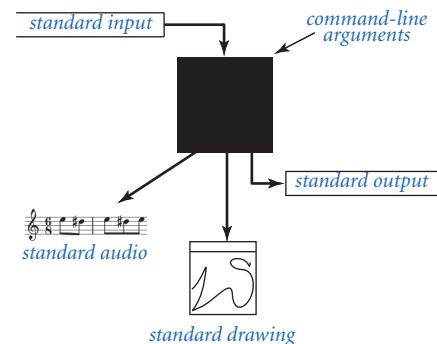
**Standard audio.** Our class `StdAudio` allows you to create sound with your programs. It uses a standard format to convert arrays of numbers into sound.

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).

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



*A bird's-eye view of a Java program (revisited)*

```

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

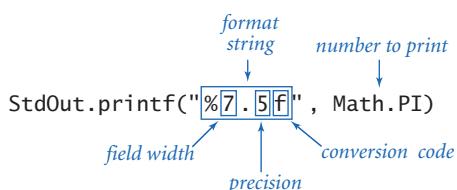
```
StdOut.printf("The square root of %.1f is %.6f", c, t);
```

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	1595.1680010754388	"%14.2f"	"1595.17"
	e		"%.7f" "%14.4e"	"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:

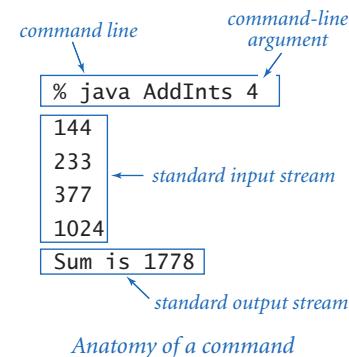
```
public class AddInts
{
    public static void main(String[] args)
    {
        int N = Integer.parseInt(args[0]);
        int sum = 0;
        for (int i = 0; i < N; i++)
        {
            int value = StdIn.readInt();
            sum += value;
        }
        StdOut.println("Sum is " + sum);
    }
}
```

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.

**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.

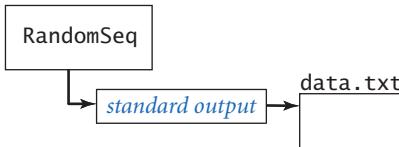
**Redirection and piping** For many applications, typing input data as a standard input stream from the terminal window is untenable because our program's processing power is then limited by the amount of data that we can type (and our typing speed). Similarly, we often want to save the information printed 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

```
java RandomSeq 1000 > data.txt
```



*Redirecting standard output to a file*

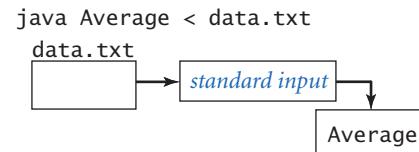
have to change `RandomSeq` (PROGRAM 1.5.1) in any way for this mechanism to work—it is using the standard output abstraction and is unaffected by our use of a different implementation of that abstraction. You can use this mechanism to save output from any program that you write. Once we have expended 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 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.

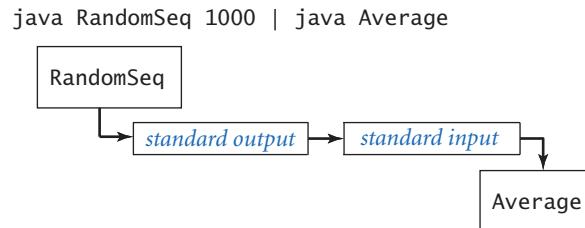
**Connecting two programs.** The most flexible way to implement the standard input and standard output abstractions is to specify that they are implemented by our own programs! This mechanism is called *piping*. For example, the command

```
% 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	<i>lower bound of range</i>
hi	<i>upper bound of range</i>
t	<i>current number</i>

*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.

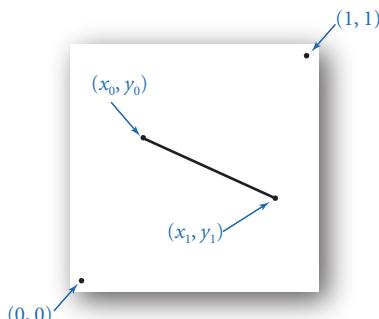
PROCESSING LARGE AMOUNTS OF INFORMATION PLAYS an essential role in many applications of computing. A scientist may need to analyze data collected from a series of experiments, a stock trader may wish to analyze information about recent financial transactions, or a student may wish to maintain collections 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



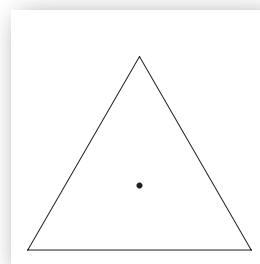
```
StdDraw.line(x0, y0, x1, y1);
```

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 $x_0$ , double $x_1$ )	<i>reset x range to <math>(x_0, x_1)</math></i>
void setYscale(double $y_0$ , double $y_1$ )	<i>reset y range to <math>(y_0, y_1)</math></i>
void setPenRadius(double $r$ )	<i>set pen radius to <math>r</math></i>

*Note: Methods with the same names but no arguments reset to default values.*

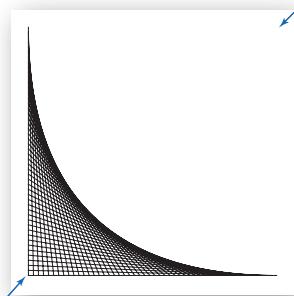
For example, when we issue the command `StdDraw.setXscale(0, N)`, we are telling the drawing device that we will be using  $x$ -coordinates between 0 and  $N$ . Note that the two-call sequence

```
StdDraw.setXscale( $x_0$ ,  $x_1$ );  
StdDraw.setYscale( $y_0$ ,  $y_1$ );
```

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.

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

```
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);
```



*(N, N)*  
*(0, 0)*  
*Scaling to integer coordinates*

**Program 1.5.5 Input-to-drawing filter**

```
public class PlotFilter
{
    public static void main(String[] args)
    { // Plot points in standard input.

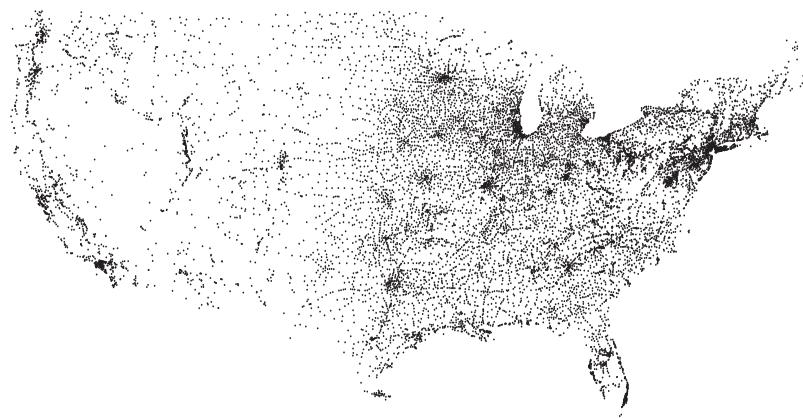
        // Scale as per first four values.
        double x0 = StdIn.readDouble();
        double y0 = StdIn.readDouble();
        double x1 = StdIn.readDouble();
        double y1 = StdIn.readDouble();
        StdDraw.setXscale(x0, x1);
        StdDraw.setYscale(y0, y1);

        // 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.

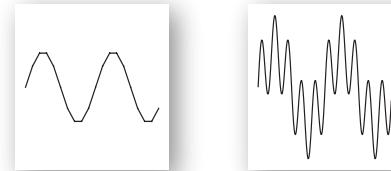
**Plotting a function graph.** Another important use of `StdDraw` is to plot experimental data or the values of a mathematical function. For example, suppose that we want to plot values of the function  $y = \sin(4x) + \sin(20x)$  in the interval  $[0, \pi]$ . Accomplishing this task is a prototypical 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]);
```

$N = 20$

$N = 200$



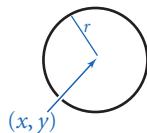
Plotting a function graph

**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 {  
      
    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

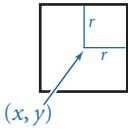
of side length  $2r$  centered on  $(x, y)$ ; and the arguments for `polygon()` and `filledPolygon()` define a sequence of points that we connect by lines, including one from the last point to the first point. If you want to define shapes other than squares or circles, use one of these methods. For example,



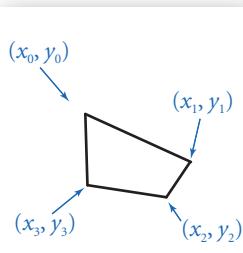
```
double[] xd = { x-r, x, x+r, x };
double[] yd = { y, y+r, y, y-r };
StdDraw.polygon(xd, yd);
```

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);
```

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

*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)

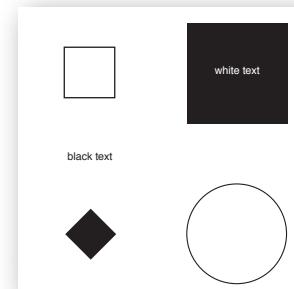
```
void text(double x, double y, String s)
void setFont(Font f)
void setPenColor(Color c)
```

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.

**Animation.** The `StdDraw` library supplies additional methods that provide limitless opportunities for creating interesting effects.

```
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*

public class <code>StdDraw</code> ( <i>advanced control commands</i> )	
<code>void setCanvasSize(int w, int h)</code>	<i>create canvas in screen window of width from w and height h (in pixels)</i>
<code>void clear()</code>	<i>clear the canvas to white (default)</i>
<code>void clear(Color c)</code>	<i>clear the canvas; color it c</i>
<code>void show(int dt)</code>	<i>draw, then pause dt milliseconds</i>
<code>void show()</code>	<i>draw, turn off pause mode</i>

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).

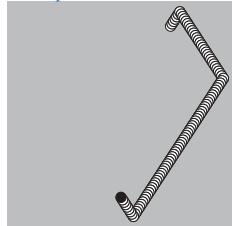
**Program 1.5.6 Bouncing ball**

```
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);
        }
    }
}
```

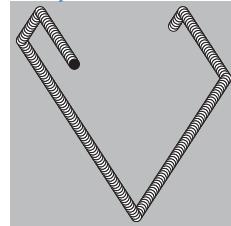
rx, ry	position
vx, vy	velocity
dt	wait time
radius	ball radius

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).

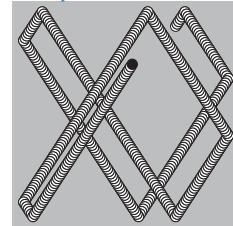
100 steps



200 steps



500 steps



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 book-site 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)

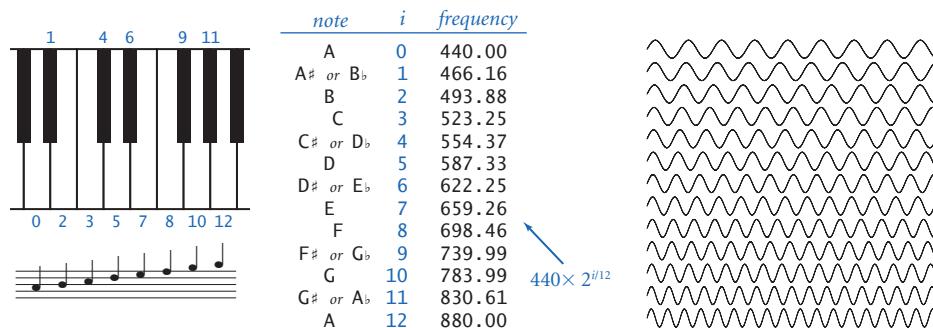
    void setXscale(double x0, double x1)      reset x range to (x0, x1)
    void setYscale(double y0, double y1)      reset y range to (y0, y1)
    void setPenRadius(double r)                set pen radius to r
    void setPenColor(Color c)                 set pen color to c
    void setFont(Font f)                     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.

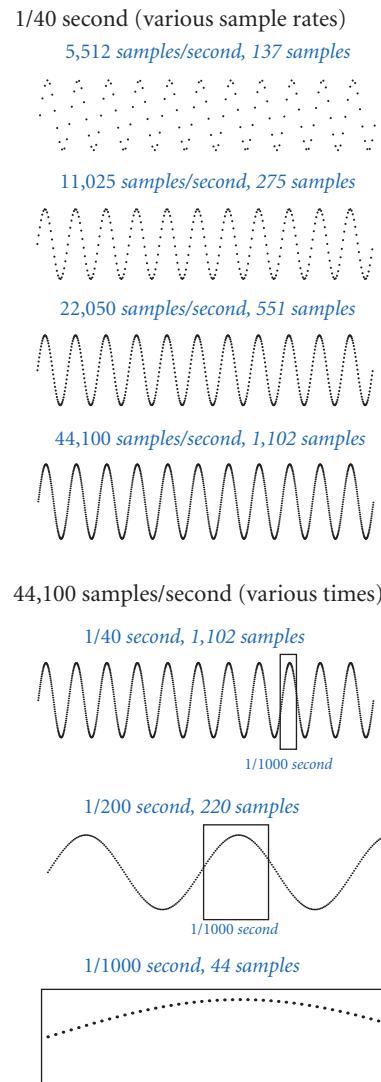
**Concert A.** Sound is the perception of the vibration of molecules, in particular, the vibration of our eardrums. Therefore, oscillation is the key to understanding sound. Perhaps the simplest place to start is to consider the musical note *A* above middle *C*, which is known as *concert A*. This note 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  $2\pi$  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.



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  $i$ th 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  $4 \times 60 \times 44100 = 10,584,000$  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 <code>.wav</code> 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 <code>.wav</code> file
double[] read(String file)	read from a <code>.wav</code> 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);
        }
    }
}
```

<b>pitch</b>	<i>distance from A</i>
<b>duration</b>	<i>note play time</i>
<b>hz</b>	<i>frequency</i>
<b>N</b>	<i>number of samples</i>
<b>a[]</b>	<i>sampled sine wave</i>

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, 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 is 1 2 2 1 5 1 1 7 7 7 7 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  $x_1, x_2, \dots, x_N$  is  $(x_1 \times x_2 \times \dots \times x_N)^{1/N}$ . The *harmonic mean* is  $(1/x_1 + 1/x_2 + \dots + 1/x_N) / (1/N)$ . 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  $k$ th 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:

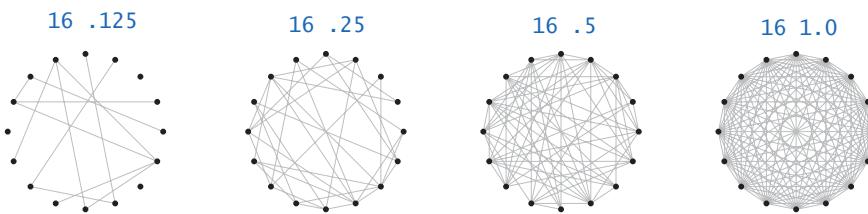
$$\begin{aligned}m &= m_1 + m_2 + \dots + m_N \\x &= (m_1 x_1 + \dots + m_N x_N) / m \\y &= (m_1 y_1 + \dots + m_N y_N) / m\end{aligned}$$

**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.



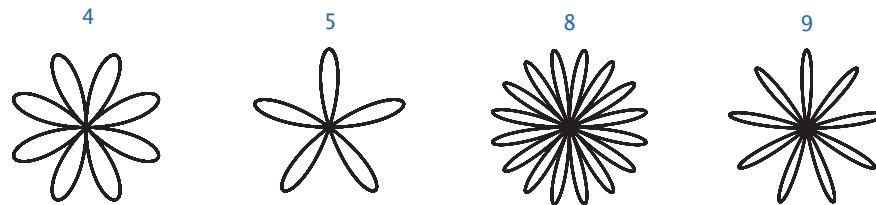
**1.5.18** Write a program that takes a command-line argument  $N$  and plots an  $N$ -by- $N$  checkerboard with red and black squares. Color the lower left square red.

**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  $(r, \theta)$  of the function  $r = \sin(N \theta)$  for  $\theta$  ranging from 0 to  $2\pi$  radians.



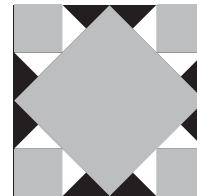
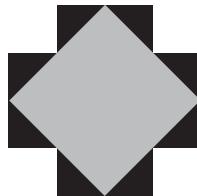
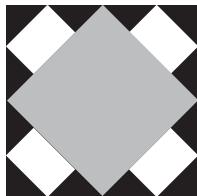
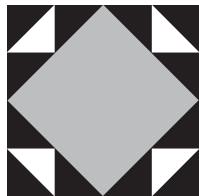
**1.5.22** Write a program that takes a string  $s$  from the command line and displays it in banner style on the screen, moving from left to right and wrapping back to the beginning of the string as the end is reached. Add a second command-line argument to control the speed.



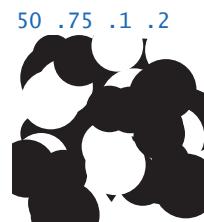
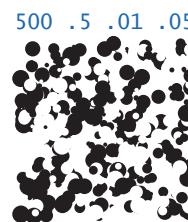
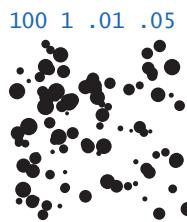
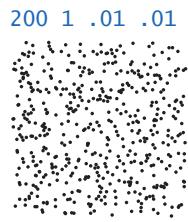
**1.5.23** Modify `PlayThatTune` to take additional command-line arguments that control the volume (multiply each sample value by the volume) and the tempo (multiply each note's duration by the tempo).

**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:

$$\begin{aligned}x(t) &= A_x \sin (w_x t + \theta_x) \\y(t) &= A_y \sin (w_y t + \theta_y)\end{aligned}$$

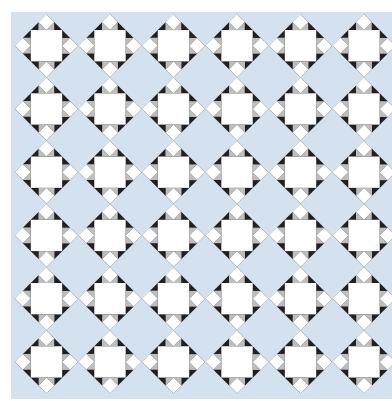
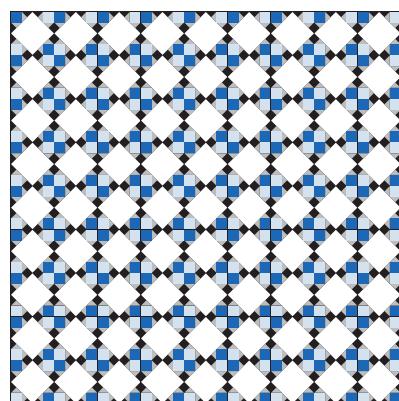
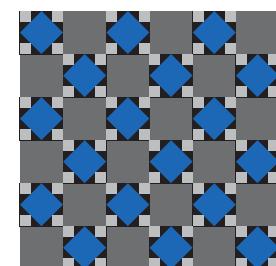
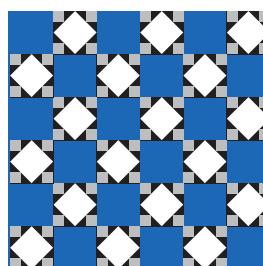
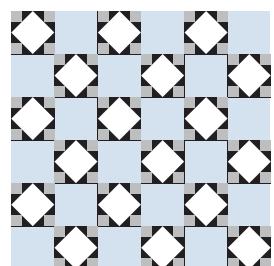
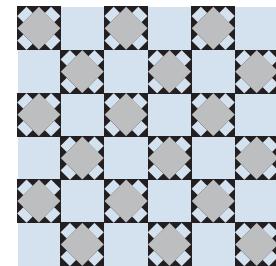
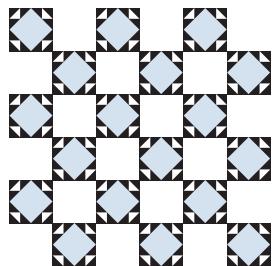
Take the six parameters  $A_x, w_x, \theta_x, A_y, w_y$ , and  $\theta_y$  from the command line.

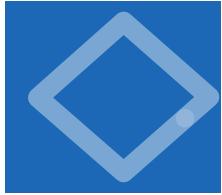
**1.5.34 Bouncing ball with tracks.** Modify `BouncingBall1` to produce images like the ones shown in the text, which show the track of the ball on a gray background.

**1.5.35 Bouncing ball with gravity.** Modify `BouncingBall1` 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.

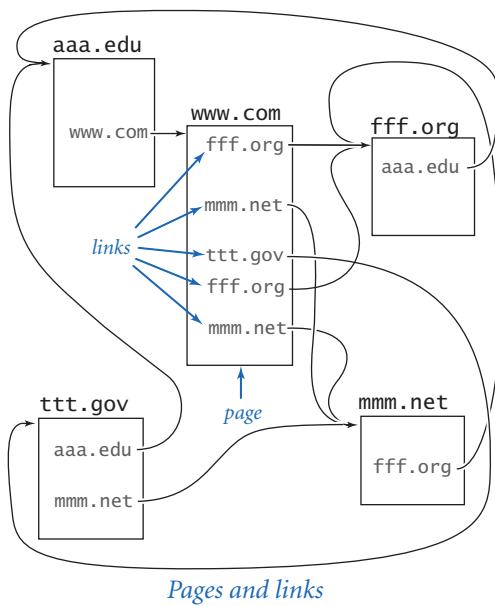
The model is known as the *random surfer* model, and is simple to describe.

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).

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



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.

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:

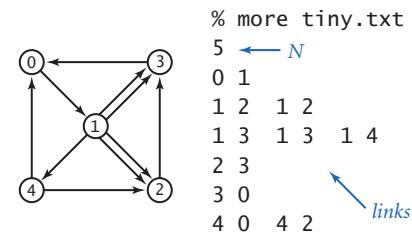
- No one chooses links or pages with equal probability.
- There is no real potential to surf directly to each page on the web.
- The 90-10 (or any fixed) breakdown is just a guess.
- It does not take the back button or bookmarks into account.
- We can only afford to work with a small sample of the web.

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.

**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.



*Random surfer input format*

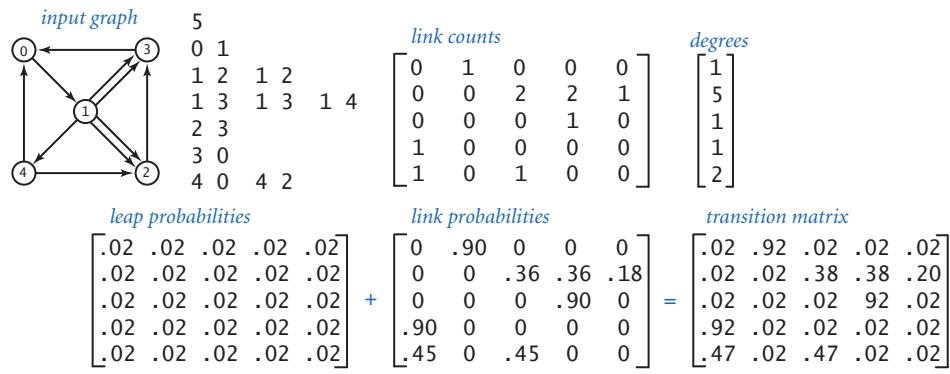
**Transition matrix** We use a two-dimensional matrix, that we refer to as the *transition matrix*, to completely specify the behavior of the random surfer. With  $N$  web pages, we define an  $N$ -by- $N$  matrix such that the entry in row  $i$  and column  $j$  is the probability that the random surfer 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  $.90/\text{degree}[i]$  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.



### 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`,

<code>j</code>	0	1	2	3	4
<code>probabilities p[page][j]</code>	.47	.02	.47	.02	.02
<code>cumulated sum values</code>	.47	.49	.96	.98	1.0

generate .71, return 2

*Generating a random integer from a discrete distribution*

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:

```
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.

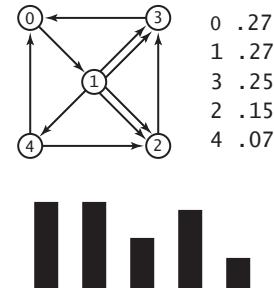
**Markov chains.** The random process that describes the surfer's behavior is known as a *Markov chain*, named after the Russian mathematician Andrey Markov, who developed the concept in the early 20th century. Markov chains are widely applicable, well-studied, and have many remarkable and useful properties. 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.

*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);
```



Page ranks with histogram

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

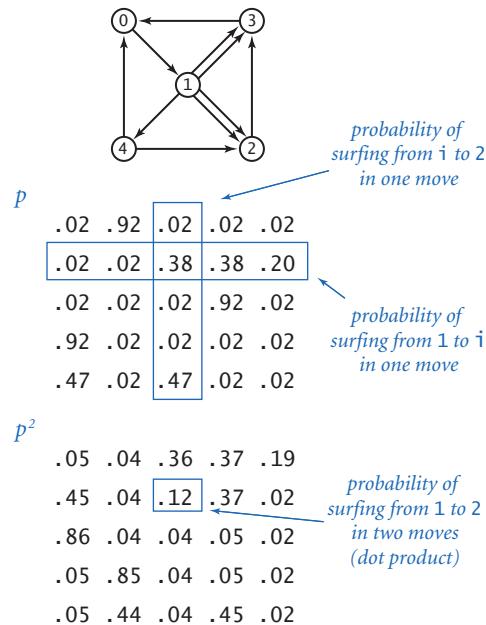
you want to study a new model (perhaps with some minor adjustments to handle larger models).

*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 ( $.02 \times .02$ ), plus the probability of moving from 1 to 1 to 2 ( $.02 \times .38$ ), plus the probability of moving from 1 to 2 to 2 ( $.38 \times .02$ ), plus the probability of moving from 1 to 3 to 2 ( $.38 \times .02$ ), plus the probability of moving from 1 to 4 to 2 ( $.20 \times .47$ ), 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 ( $.02 \times .02$ ), plus the probability of moving from 0 to 1 to 2 ( $.92 \times .38$ ), plus the probability of moving from 0 to 2 to 2 ( $.02 \times .02$ ), plus the probability of moving from 0 to 3 to 2 ( $.02 \times .02$ ), plus the probability of moving from 0 to 4 to 2 ( $.02 \times .47$ ), 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

rank[]	p[][]	newRank[]
<i>first move</i>	$\begin{bmatrix} .02 & .92 & .02 & .02 & .02 \\ .02 & .02 & .38 & .38 & .20 \\ .02 & .02 & .02 & .92 & .02 \\ .92 & .02 & .02 & .02 & .02 \\ .47 & .02 & .47 & .02 & .02 \end{bmatrix}$	$= [ .02 \quad .92 \quad .02 \quad .02 \quad .02 ]$
		<i>probabilities of surfing from 0 to i in one move</i>
<i>second move</i>	$\begin{bmatrix} .02 & .92 & .02 & .02 & .02 \\ .02 & .02 & .38 & .38 & .20 \\ .02 & .02 & .02 & .92 & .02 \\ .92 & .02 & .02 & .02 & .02 \\ .47 & .02 & .47 & .02 & .02 \end{bmatrix}$	$= [ .05 \quad .04 \quad .36 \quad .37 \quad .19 ]$
	<i>probabilities of surfing from i to 2 in one move</i>	<i>probability of surfing from 0 to 2 in two moves (dot product)</i>
<i>third move</i>	$\begin{bmatrix} .02 & .92 & .02 & .02 & .02 \\ .02 & .02 & .38 & .38 & .20 \\ .02 & .02 & .02 & .92 & .02 \\ .92 & .02 & .02 & .02 & .02 \\ .47 & .02 & .47 & .02 & .02 \end{bmatrix}$	$= [ .44 \quad .06 \quad .12 \quad .36 \quad .03 ]$
	<i>probabilities of surfing from 0 to i in two moves</i>	<i>probabilities of surfing from 0 to i in three moves</i>
<i>20th move</i>	$\begin{bmatrix} .02 & .92 & .02 & .02 & .02 \\ .02 & .02 & .38 & .38 & .20 \\ .02 & .02 & .02 & .92 & .02 \\ .92 & .02 & .02 & .02 & .02 \\ .47 & .02 & .47 & .02 & .02 \end{bmatrix}$	$= [ .27 \quad .26 \quad .15 \quad .25 \quad .07 ]$
	<i>probabilities of surfing from 0 to i in 19 moves</i>	<i>probabilities of surfing from 0 to i in 20 moves (steady state)</i>

## 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

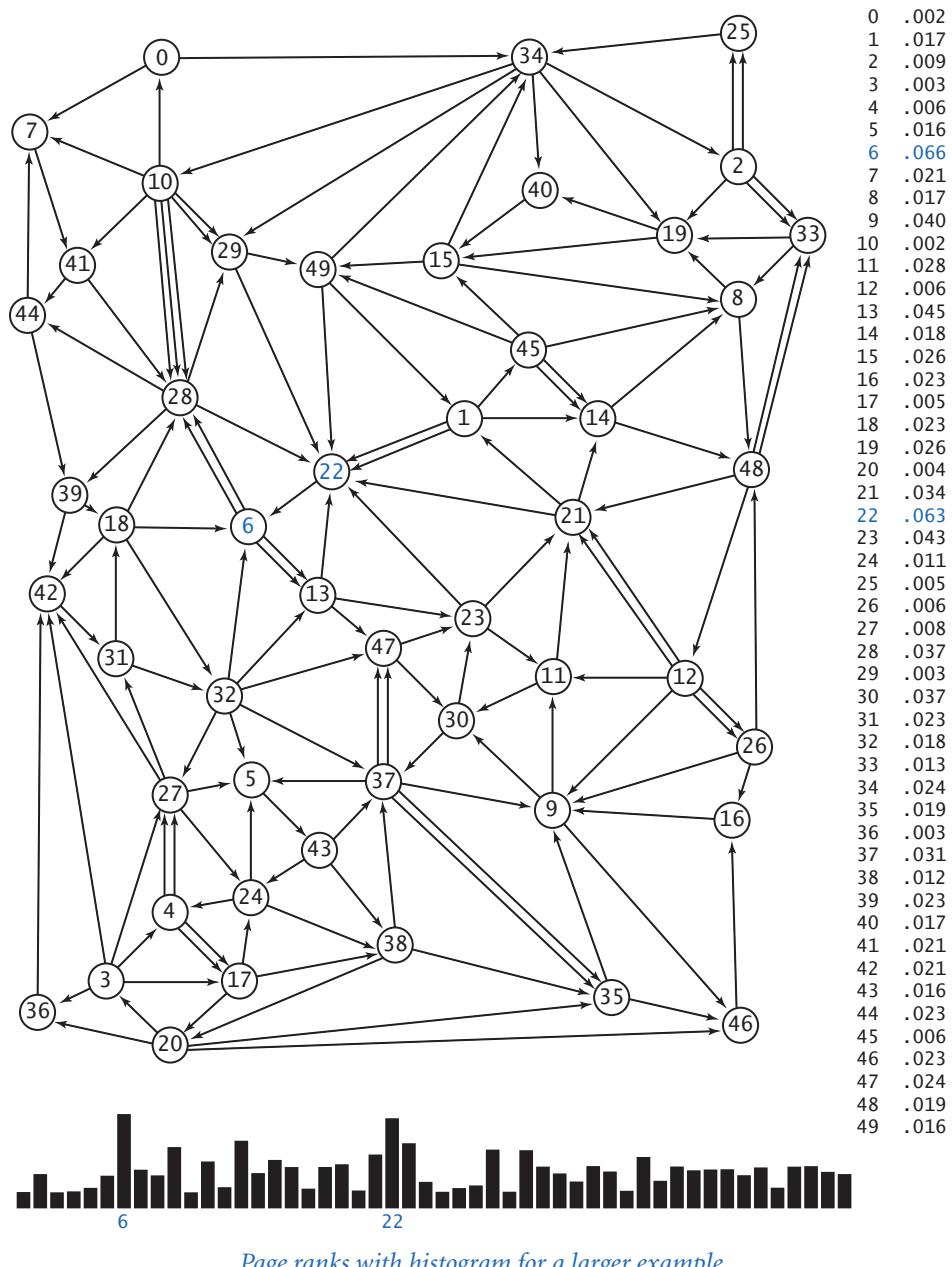
This program reads a transition matrix from standard input and computes the probabilities that a random surfer lands on each page (page ranks) after the number of steps specified as command-line argument.

```

% 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

```



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.

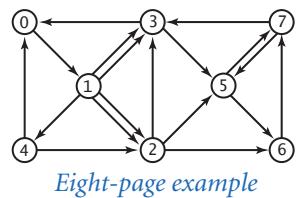
**1.6.9** Suppose that your page is page 23 in `medium.txt`. Is there a link that you could add from your page to some other page that would *raise* the rank of *your* page?

**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.





## 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?

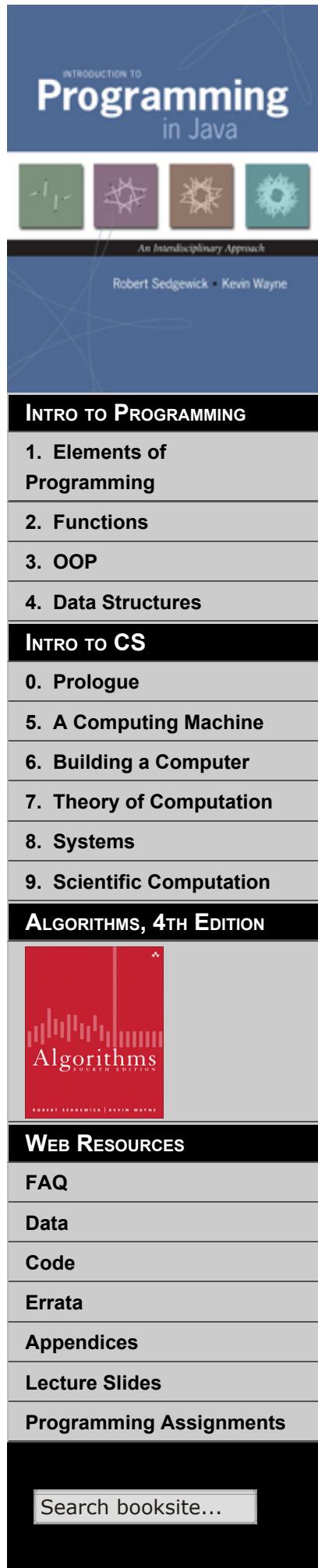
**1.6.16** *Page ranks.* Design an array of pages and links where the highest-ranking page has fewer links pointing to it than some other page.

**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`.





## 2. FUNCTIONS

**Overview.** In this chapter, we consider a concept that has as profound an impact on control flow as do conditionals and loops: the *function*, which allows us to transfer control back and forth between different pieces of code. Functions are important because they allow us to clearly separate tasks within a program and because they provide a general mechanism that enables us to reuse code.

- [2.1 Static Methods](#) introduces the Java mechanism (the *static method*) for implementing functions.
- [2.2 Libraries and Clients](#) describes how to group related static methods into libraries to enable modular programming.
- [2.3 Recursion](#) considers the idea of a function calling *itself*. This possibility is known as recursion.
- [2.4 Percolation](#) presents a case study that uses Monte Carlo simulation to study a natural model known as percolation.

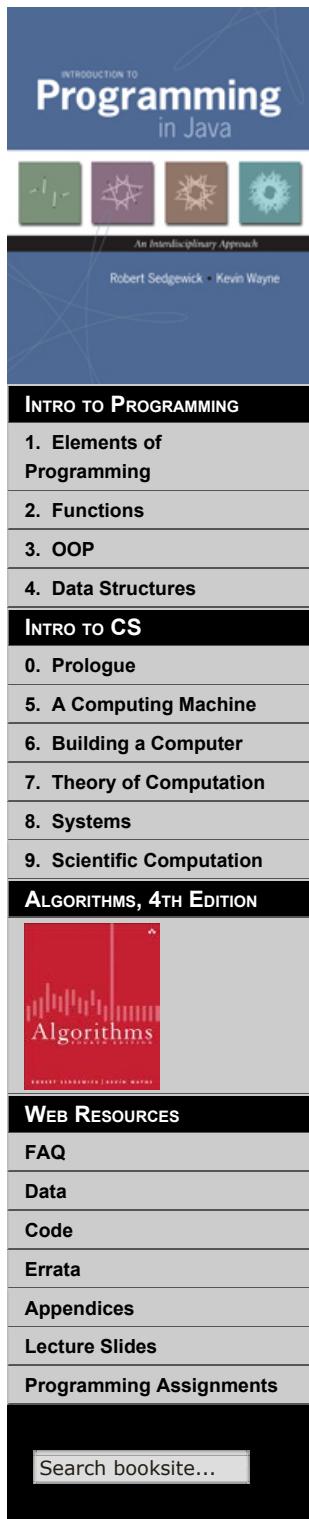
**Java programs in this chapter.** Below is a list of Java programs in this chapter. Click on the program name to access the Java code; click on the reference number for a brief description; read the textbook for a full discussion.

REF	PROGRAM	DESCRIPTION
2.1.1	<a href="#">Newton.java</a>	Newton's method (revisited)
2.1.3	<a href="#">Coupon.java</a>	coupon collector (revisited)
2.2.1	<a href="#">StdRandom.java</a>	random number library
2.2.3	<a href="#">IFS.java</a>	iterated function systems
2.2.5	<a href="#">StdStats.java</a>	data analysis library
2.3.1	<a href="#">Euclid.java</a>	Euclid's algorithm
2.3.3	<a href="#">Beckett.java</a>	Gray code
2.3.5	<a href="#">Brownian.java</a>	Brownian bridge
2.4.2	<a href="#">VerticalPercolation.java</a>	vertical percolation
2.4.4	<a href="#">Estimate.java</a>	percolation probability estimate
2.4.6	<a href="#">PercPlot.java</a>	adaptive plot client
2body.txt 3body.txt 4body.txt 2bodyTiny.txt	<a href="#">4.1.1.java</a>	41analysis

**Exercises.** Include a link.

*Last modified on August 05, 2011.*

Copyright © 2002–2012 [Robert Sedgewick](#) and [Kevin Wayne](#). All rights reserved.

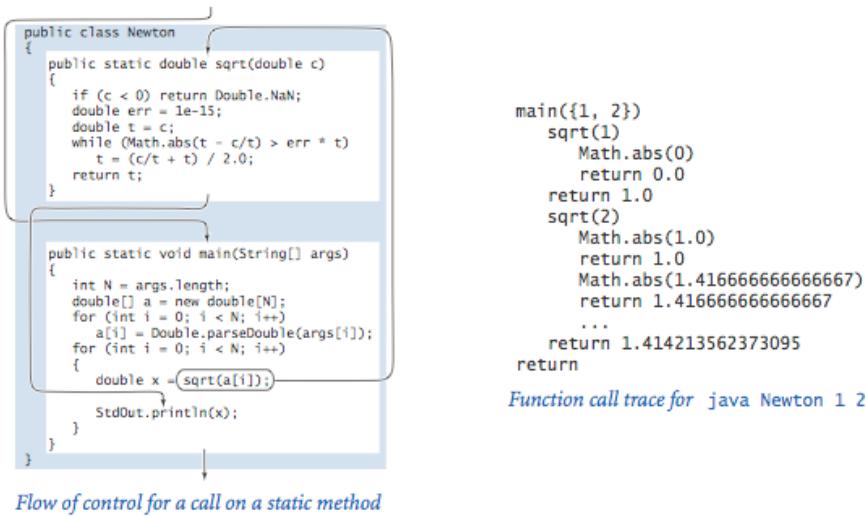


## 2.1 STATIC METHODS

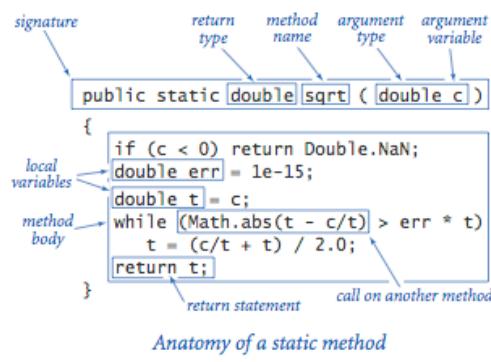
The Java construct for implementing functions is known as the *static method*.

**Using and defining static methods.** The use of static methods is easy to understand. For example, when you write `Math.abs(a-b)` in a program, the effect is as if you were to replace that code by the value that is computed by Java's `Math.abs()` method when presented with the value `a-b`. If you think about what the computer has to do to create this effect, you will realize that it involves changing a program's *flow of control*.

- *Flow-of-control.* `Newton.java` comprises two static methods `sqrt()` and `main()`. Even though `sqrt()` appears first in the code, the first statement executed when the program is executed is, as always, the first statement in `main()`. The next few statements operate as usual, except that the code `sqrt(a[i])`, which is known as a *function call* on the static method `sqrt()`, causes a transfer of control, to the first line of code in `sqrt()`, each time that it is encountered. Moreover, the value of `c` within `sqrt()` is initialized to the value of `a[i]` within `main()` at the time of the call. Then the statements in `sqrt()` are executed in sequence, as usual, until reaching the `return` statement, which transfers control back to the statement in `main()` containing the call on `sqrt()`.



- *Function call trace.* One simple approach to following the flow of control through function calls is to imagine that each function prints its name and argument value when it is called and its return value just before returning, with indentation added on calls and subtracted on returns.
- *Anatomy of a static method.* The square root function maps a nonnegative real number to a nonnegative real number; the `sqrt()` static method in `Newton` maps a `double` to a `double`. For any given initial value of the parameter variable, it will compute a well-defined return value.



Anatomy of a static method

### Properties of static methods.

- *Parameter variables.* You can use parameter variables anywhere in the code in the body of the function in the same way as you use local variables. The only difference between a parameter variable and a local variable is that the parameter variable is initialized with the argument value provided by the calling code.
- *Scope.* The *scope* of a variable name is the set of statements that can refer to that name. The scope of the variables in a static method is limited to that method's body (or a smaller set of statements if declared inside a block within the function body). Therefore, you cannot refer to a variable in one static method that is declared in another.

```

public class Newton
{
    public static double sqrt(double c)
    {
        if (c < 0) return Double.NaN;
        double t = c;
        while (Math.abs(t - c/t) > err * t)
            t = (c/t + t) / 2.0;
        return t;
    }

    public static void main(String[] args)
    {
        int N = args.length;
        double[] a = new double[N];
        for (int i = 0; i < N; i++)
            a[i] = Double.parseDouble(args[i]);
        for (int i = 0; i < N; i++)
        {
            double x = sqrt(a[i]);
            StdOut.println(x);
        }
    }
}

```

### Scope of local and argument variables

- *Multiple methods.* You can define as many static methods as you want in a .java file. These methods are independent, except that they may refer to each other through calls. They can appear in any order in the file.
- *Calling other static methods.* Any static method defined in a .java file can call any other static method in the same file or any static method in a Java library like Math.
- *Multiple arguments.* Like a mathematical function, a Java static method can take on more than one argument, and therefore can have more than one parameter variable.
- *Overloading.* Static methods whose signatures differ are different static methods. Using one name for two static methods whose signatures differ is known as *overloading*.
- *Single return value.* Like a mathematical function, a Java static method can provide only one return value, of the type declared in the method signature.
- *Multiple return statements.* Control goes back to the calling program as soon as the first return statement in a static method is reached. You can put return statements wherever you need them, as in sqrt(). Even though there may be multiple return statements, any static method returns a single value each time it is invoked: the value following the first return statement encountered.
- *Side effects.* A static method may use the keyword void as its return type, to indicate that it has no return value. An explicit return is not necessary in a void static method: control returns to the caller after the last statement. In this book, we use void static methods for two purposes: for output, in StdOut, StdDraw, and StdAudio and to manipulate arrays.

[Functions.java](#) gives a number of examples.

<i>absolute value of an int value</i>	<pre>public static int abs(int x) {     if (x &lt; 0) return -x;     else        return x; }</pre>
<i>absolute value of a double value</i>	<pre>public static double abs(double x) {     if (x &lt; 0.0) return -x;     else        return x; }</pre>
<i>primality test</i>	<pre>public static boolean isPrime(int N) {     if (N &lt; 2) return false;     for (int i = 2; i &lt;= N/i; i++)         if (N % i == 0) return false;     return true; }</pre>
<i>hypotenuse of a right triangle</i>	<pre>public static double hypotenuse(double a, double b) {   return Math.sqrt(a*a + b*b); }</pre>
<i>Harmonic number</i>	<pre>public static double H(int N) {     double sum = 0.0;     for (int i = 1; i &lt;= N; i++)         sum += 1.0 / i;     return sum; }</pre>
<i>uniform random integer in <math>[0, N]</math></i>	<pre>public static int uniform(int N) {   return (int) (Math.random() * N); }</pre>
<i>draw a triangle</i>	<pre>public static void drawTriangle(double x0, double y0,                                 double x1, double y1,                                 double x2, double y2 ) {     StdDraw.line(x0, y0, x1, y1);     StdDraw.line(x1, y1, x2, y2);     StdDraw.line(x2, y2, x0, y0); }</pre>

**Implementing mathematical functions.** We now consider two important functions that play an important role in science, engineering, and finance. The *Gaussian (normal) distribution function* is characterized by the familiar bell-shaped curve and defined by the formula:

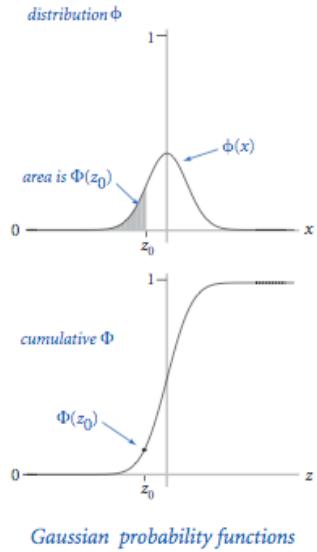
$$\phi(x) = \frac{1}{\sqrt{2\pi}} e^{-x^2/2}$$

and the *cumulative Gaussian distribution function*  $\Phi(z)$  is defined to be the area under the curve defined by  $\phi(x)$  above the  $x$ -axis and to the left of the vertical line  $x = z$ . For example,  $\Phi(1.96) = 0.975$ . This means that 97.5% of the area under the standard Gaussian curve is to the left of  $z = 1.96$ .

- *Closed form.* In the simplest situation, we have a closed-form mathematical equation defining our function in terms of functions that are implemented in the `Math` library. This is the case for  $\phi(x)$ .
- *No closed form.* Otherwise, we may need a more complicated algorithm to compute function values. This situation is the case for  $\Phi(z)$ , for which no closed-form expression exists. For small (respectively large)  $z$ , the value is extremely close to 0 (respectively 1); so the code directly returns 0 (respectively 1); otherwise the following Taylor series approximation is an effective basis for evaluating the function:

$$\begin{aligned}\Phi(z) &= \int_{-\infty}^z \phi(x) dx \\ &= \frac{1}{2} + \phi(z) \left( z + \frac{z^3}{3} + \frac{z^5}{3 \cdot 5} + \frac{z^7}{3 \cdot 5 \cdot 7} + \dots \right)\end{aligned}$$

`Gaussian.java` implements both of these static methods.



*Gaussian probability functions*

**Using static methods to organize code.** With the ability to define functions, we can better organize our programs by defining functions within them when appropriate. For example, `Coupon.java` is a version of `CouponCollector.java` that better separates the individual components of the computation.

- Given  $N$ , compute a random coupon value
- Given  $N$ , do the coupon collection experiment
- Get  $N$  from the command line, then compute and print the result

Whenever you can clearly separate tasks in programs, you should do so.

**Arrays as arguments and return values.** A static method can take an array as an argument or as a return value.

- *Arrays as arguments.* The following static method computes the mean (average) value of an array of double values.

```
public static double mean(double[] a) {
    double sum = 0.0;
    for (int i = 0; i < a.length; i++)
        sum = sum + a[i];
    return sum / a.length;
}
```

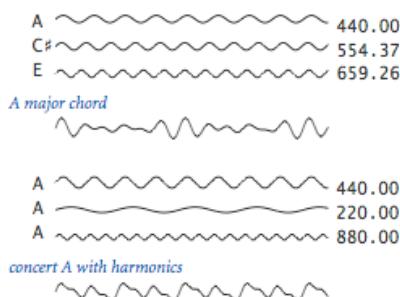
We have been using arrays as arguments from the beginning. The code

```
public static void main(String[] args)
```

defines `main()` as a static method that takes an array of strings as argument and returns nothing. By convention, the Java system collects the strings that you type after the program name in the java command into an array `args[]` and calls `main()`.

- *Side effects with arrays.* It is often the case that the purpose of a static method that takes an array as argument is to produce a side effect (change values in the array). A prototypical example of such a method is one that exchanges the values at two given indices in a given array. A second prototypical example of a static method that takes an array as argument is one that randomly shuffles the values in the array, using this version of the approach that we examined in Section 1.4 (simplified by use of the `exch()` method just defined). Program [Shuffle.java](#) takes this approach.
- *Arrays as return values.* A static method can also provide an array as a return value.

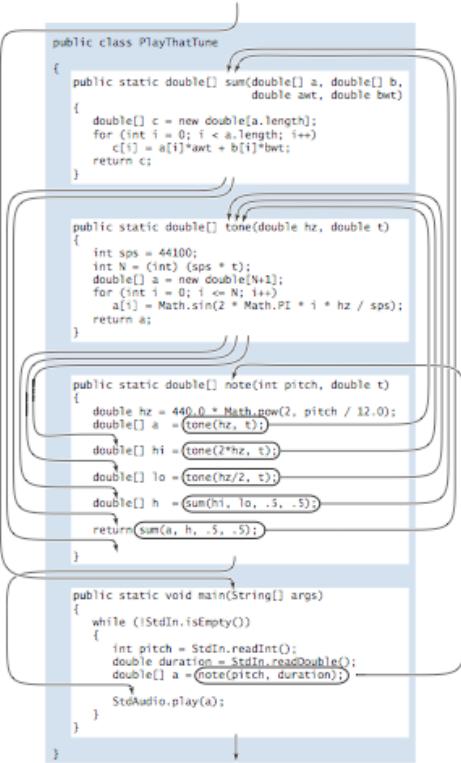
**Superposition of sound waves.** Notes like concert A have a pure sound that is not very musical, because the sounds that you are accustomed to hearing have many other components. Most musical instruments produce harmonics (the same note in different octaves and not as loud), or you might play chords (multiple notes at the same time). To combine multiple sounds, we use superposition: simply add their waves together and rescale to make sure that all values stay between 1 and 1.



*Superposing waves to make composite sounds*

[PlayThatTuneDeluxe.java](#) is a version of [PlayThatTune](#) that encapsulates the sound wave calculation and adds harmonics. Here a few sample data files (created by various students):

- [Ascale.txt](#)
- [elise.txt](#)
- [StairwayToHeaven.txt](#)
- [firstcut.txt](#)
- [looney.txt](#)
- [National\\_Anthem.txt](#)
- [arabesque.txt](#)
- [entertainer.txt](#)
- [freebird.txt](#)
- [tomsdiner.txt](#)



Flow of control among several static methods

## Q + A

**Q.** Why do I need to use the return type `void`? Why not omit the return type?

**A.** Java requires it; we have to include it. Second-guessing a decision made by a programming-language designer is the first step to becoming one.

**Q.** Can I return from a `void` function by using `return`. If so, what return value should I use?

**A.** Yes. use the statement `return;` with no return value.

**Q.** What happens if I leave out the keyword `static`?

**A.** As usual, the best way to answer a question like this is to try it yourself and see what happens. Non-static methods are different from static methods. You will learn about them in [Chapter 3](#).

**Q.** What happens if I write code after a `return` statement?

**A.** Once a `return` statement is reached, control immediately returns to the caller, so any code after a `return` statement is useless. The Java compiler identifies this situation as an error, reporting unreachable code.

**Q.** What happens if I do not include a `return` statement?

**A.** No problem, if the return type is `void`. In this case, control will return to the caller after the last statement. When the return type is not `void`, the compiler will report a missing `return` statement error if there is any path through the code that does not end in a `return`.

**Q.** The issue with side effects and arrays passed as arguments is confusing. Is it really all that important?

**A.** Yes. Properly controlling side effects is one of a programmers most important tasks in large systems. Taking the time to be sure that you understand the difference between passing a value (when arguments are of a primitive type) and passing a reference (when arguments are arrays) will certainly be worthwhile. The very same mechanism is used for all other types of data, as you will learn in Chapter 3.

**Q.** So why not just eliminate the possibility of side effects by making all arguments pass-by-value, including arrays?

**A.** Think of a huge array with, say, millions of elements. Does it make sense to copy all of those values for a static method that is just going to exchange two of them? For this reason, most programming languages support passing an array to a function without creating a copy of the array elements—Matlab is a notable exception.

## Exercises

1. Write a static method `max3()` that takes three `int` values as arguments and returns the value of the largest one. Add an overloaded function that does the same thing with three `double` values.

```

public static int max3(int a, int b, int c) {
    int max = a;
    if (b > max) max = b;
    if (c > max) max = c;
    return max;
}

public static double max3(double a, double b, double c) {
    double max = a;
    if (b > max) max = b;
    if (c > max) max = c;
    return max;
}

```

2. Write a static method `odd()` that takes three `boolean` inputs and returns `true` if an odd number of inputs are `true`, and `false` otherwise.
3. Write a static method `majority()` that takes three `boolean` arguments and returns `true` if at least two of the arguments have the value `true`, and `false` otherwise. Do not use an `if` statement. *Solution:* here are two solutions: the first is concise; the second strictly adheres to the rules.

```

public static boolean majority(boolean a, boolean b, boolean c) {
    return (a && b) || (a && c) || (b && c);
}

public static boolean majority(boolean a, boolean b, boolean c) {
    while (a && b) return true;
    while (a && c) return true;
    while (b && c) return true;
    return false;
}

```

4. Write a static method `eq()` that takes two arrays of integers as arguments and returns `true` if they contain the same number of elements and all corresponding pairs of elements are equal. *Solution.* [ArraysEquals.java](#).
5. Write a static method `areTriangular()` that takes three `double` values as arguments and returns `true` if they could be the sides of a triangle (none of them is greater than or equal to the sum of the other two). See Exercise 1.2.15.
6. Write a static method `sigmoid()` that takes a `double` argument `x` and returns the `double` value obtained from the formula:  $1 / (1 - e^{-x})$ .
7. If the argument of `sqrt()` in [Newton.java](#) has the value `Infinity`, then `Newton.sqrt()` returns the value `Infinity`, as desired. Explain why.
8. Add a method `abs()` to [Newton.java](#), change `sqrt()` to use `abs()` instead of `Math.abs()`, and add print statements to produce a function call trace, as described in the text. Hint : You need to add an argument to each function to give the level of indentation.
9. Give the function call trace for `java Newton 4.0 9.0`.
10. Write a static method `lg()` that takes a `double` value `N` as argument and returns the base 2 logarithm of `N`. You may use Java's `Math` library.
11. Write a static method `lg()` that takes an `int` value `N` as argument and returns the largest `int` not larger than the base-2 logarithm of `N`. Do not use `Math`.
12. Write a static method `signum()` that takes an `int` value `N` as argument and returns `-1` if `N` is less than 0, `0` if `N` is equal to 0, and `+1` if `N` is greater than 0. (See also the static method `Integer.signum()`.)
13. Consider the following static method `duplicate()` below.

```

public static String duplicate(String s) {
    String t = s + s;
    return t;
}

```

What does the following code fragment print out?

```

String s = "Hello";
s = duplicate(s);
String t = "Bye";
t = duplicate(duplicate(duplicate(t)));
StdOut.println(s + t);

```

14. Consider the static method `cube()` below.

```

public static void cube(int i) {
    i = i * i * i;
}

```

How many times is the following for loop iterated?

```

for (int i = 0; i < 1000; i++)
    cube(i);

```

*Answer:* see textbook.

15. The following checksum formula is widely used by banks and credit card companies to validate legal account numbers:

$$d_0 + f(d_1) + d_2 + f(d_3) + d_4 + f(d_5) + d_6 + \dots = 0 \pmod{10}$$

The  $d_i$  are the decimal digits of the account number and  $f(d)$  is the sum of the decimal digits of  $2d$  (for example,  $f(7) = 5$  because  $2 * 7 = 14$  and  $1 + 4 = 5$ ). For example, 17327 is valid because  $1 + 5 + 3 + 4 + 7 = 20$ , which is a multiple of 10. Implement the function  $f$  and write a program to take a 10-digit integer as a command-line argument and print a valid 11-digit number with the given integer as its first 10 digits and the checksum as the last digit.

16. Given two stars with angles of declination and right ascension  $(d_1, a_1)$  and  $(d_2, a_2)$ , the angle they subtend is given by the [Haversine formula](#):

$$2 \arcsin((\sin^2(d/2) + \cos(d) * \cos(d_1) * \sin^2(a/2)^{1/2}),$$

where  $a_1$  and  $a_2$  are angles between -180 and 180 degrees,  $d_1$  and  $d_2$  are angles between -90 and 90 degrees,  $a = a_2 - a_1$ ,  $d = d_2 - d_1$ . Write a program to take declination and right ascension of two stars as command-line arguments and print the angle they subtend. *Hint:* Be careful about converting from degrees to radians.

See also Exercise 1.2.33. Latitude corresponds to declination and longitude to ascension.

17. Write a `readBoolean2D()` method that reads a two-dimensional boolean matrix (with dimensions) into an array. *Answer:* see textbook.
18. Write a method that takes an array of `double` values as argument and rescales the array so that each element is between 0 and 1 (by subtracting the minimum value from each element and then dividing each element by the difference between the minimum and maximum values). Use the `max()` method defined in the table in the text, and write and use a matching `min()` method.
19. Write a method `histogram()` that takes an array `a[]` of `int` values and an integer `M` as argument and returns an array of length `M` whose `i`th entry is the number of times the integer `i` appeared in the argument array. If the values in `a[]` are all between 0 and `M-1`, the sum of the values in the returned array should be equal to `a.length`.
20. Assemble code fragments in this section and in Section 1.4 to develop a program that takes `N` from the command line and prints `N` five-card hands, separated by blank lines, drawn from a randomly shuffled card deck, one card per line using card names like `Ace of Clubs`.
21. Write a method `multiply()` that takes two square matrices of the same dimension as arguments and produces their product (another square matrix of that same dimension). *Extra credit:* Make your program work whenever the number of rows in the first matrix is equal to the number of columns in the second matrix.
22. Write a method `any()` that takes an array of `boolean` values as argument and returns `true` if any of the entries in the array is `true`, and `false` otherwise. Write a method `all()` that takes an array of `boolean` values as argument and returns `true` if all of the entries in the array are `true`, and `false` otherwise.
23. Develop a version of `getCoupon()` that better models the situation when one of the coupons

- is rare: choose one value at random, return that value with probability  $N/1000$ , and return all other values with equal probability. *Extra credit:* How does this change affect the average value of the coupon collector function?
24. Modify [PlayThatTune.java](#) to add harmonics two octaves away from each note, with half the weight of the one-octave harmonics.

## Creative Exercises

25. **Birthday problem.** Develop a class with appropriate static methods for studying the birthday problem (see Exercise 1.4.35).
26. **Euler's totient function.** Euler's totient function is an important function in number theory:  $\phi(n)$  is defined as the number of positive integers less than or equal to  $n$  that are relatively prime with  $n$  (no factors in common with  $n$  other than 1). Write a function that takes an integer argument  $n$  and returns  $\phi(n)$ , and a `main()` that takes an integer from the command line, calls the function, and prints the result.
27. **Harmonic numbers.** Write a program `Harmonic` that contains three static methods `H()`, `Hsmall()`, and `Hlarge()` for computing the Harmonic numbers. The `Hsmall()` method should just compute the sum (as in [Harmonic.java](#)), the `Hlarge()` method should use the approximation  $H(N) = N = \log_e(N) + \gamma + 1/(2N) - 1/(12N^2) + 1/(120N^4)$  (the number  $\gamma = 0.577215664901532\dots$  is known as *Euler's constant*), and the `H()` method should call `Hsmall()` for  $(N < 100)$  and `Hlarge()` otherwise.
28. **Gaussian random values.** Experiment with the following method for generating random variables from the Gaussian distribution, which is based on generating a random point in the unit circle and using a form of the [Box-Muller transform](#). (see Exercise 1.2.27 and the discussion of do-while at the end of Section 1.3).

```
public static double gaussian() {
    double r, x, y;
    do {
        x = uniform(-1.0, 1.0);
        y = uniform(-1.0, 1.0);
        r = x*x + y*y;
    } while (r >= 1 || r == 0);
    return x * Math.sqrt(-2.0 * Math.log(r) / r);
}
```

Take a command-line argument  $N$  and generate  $N$  random numbers, using an array `a[20]` to count the numbers generated that fall between  $i*.05$  and  $(i+1)*.05$  for  $i$  from 0 to 19. Then use `StdDraw` to plot the values and to compare your result with the normal bell curve.

*Remark:* This method is preferred over the one described in Exercise XYZ for both efficiency and accuracy. Although it involves a loop, the do-while loop is executed only  $4/n = 1.273$  times on average. This reduces the overall expected number of calls to transcendental functions.

29. **Binary search.** A general method that we study in detail in [Section 4.2](#) is effective for computing the inverse of a cumulative probability density function like `Phi()`. Such functions are continuous and nondecreasing from  $(0, 0)$  to  $(1, 1)$ . To find the value  $x_0$  for which  $f(x_0) = y_0$ , check the value of  $f(.5)$ . If it is greater than  $y_0$ , then  $x_0$  must be between 0 and .5; otherwise, it must be between .5 and 1. Either way, we halve the length of the interval known to contain  $x_0$ . Iterating, we can compute  $x_0$  to within a given tolerance. Add a method `PhiInverse()` to [Gaussian.java](#) that uses binary search to compute the inverse.

Change `main()` to take a number  $p$  between 0 and 100 as a third command-line argument and print the minimum score that a student would need to be in the top  $p$  percent of students taking the SAT in a year when the mean and standard deviation were the first two command-line arguments.

30. **Black-Scholes option valuation.** The [Black Scholes](#) formula supplies the theoretical value of a European call option on a stock that pays no dividends, given the current stock price  $s$ , the exercise price  $x$ , the continuously compounded riskfree interest rate  $r$ , the standard deviation  $\sigma$  of the stock's return (volatility) and the time (in years) to maturity  $t$ . The value is given by the formula

$$\Phi(a) - sxe^{-rt} \Phi(b)$$

where

$$a = \frac{\ln(s/x) + (r + \sigma^2/2)t}{\sigma\sqrt{t}}, \quad b = a - \sigma\sqrt{t}$$

Write a program [BlackScholes.java](#) that takes  $s$ ,  $x$ ,  $r$ ,  $\sigma$ , and  $t$  from the command line and prints the Black-Scholes value.

Myron Scholes won the 1997 Nobel Prize in Economics for the [Black-Scholes paper](#).

31. **Implied volatility.** Typically the volatility is the unknown value in the Black-Scholes formula. Write a program that reads  $s$ ,  $x$ ,  $r$ ,  $t$ , and the current price of the option from the command line and uses binary search (see Exercise 2.1.29) to compute  $\sigma$ .
  32. **Horner's method.** Write a class `Horner.java` with a method `double eval(double x, double[] p)` that evaluates the polynomial  $p(x)$  whose coefficients are the entries in `p[]`:

$$p_0 + p_1x^1 + p_2x^2 + \dots + p_{N-2}x^{N-2} + p_{N-1}x^{N-1}$$

Use *Horner's method*, an efficient way to perform the computations that is suggested by the following parenthesization:

$$p_0 + x(p_1 + x(p_2 + \dots + x(p_{N-2} + xp_{N-1}) \dots))$$

Write a test client with a static method `exp()` that uses `Horner.eval()` to compute an approximation to  $e^x$ , using the first  $N$  terms of the Taylor series expansion  $e^x = 1 + x + x^2/2! + x^3/3! + \dots$ . Include code to check your answer against that computed by `Math.exp()`.

33. **Benford's law.** The American astronomer Simon Newcomb observed a quirk in a book that compiled logarithm tables: the beginning pages were much grubbier than the ending pages. He suspected that scientists performed more computations with numbers starting with 1 than with 8 or 9, and postulated the first digit law, which says that under general circumstances, the leading digit is much more likely to be 1 (roughly 30%) than the digit 9 (less than 4%). This phenomenon is known as [Benford's law](#) and is now often used as a statistical test. For example, IRS forensic accountants rely on it to discover [tax fraud](#). Write a program [Benford.java](#) that reads in a sequence of integers from standard input and tabulates the number of times each of the digits 1-9 is the leading digit, breaking the computation into a set of appropriate static methods. Use your program to test the law on some tables of information from your computer or from the web. Then, write a program to foil the IRS by generating random amounts from \$1.00 to \$1,000.00 with the same distribution that you observed.

The file [princeton-files.txt](#) is a list of file sizes on a public Unix machine at Princeton.

34. **Binomial distribution.** Write a function `public static double binomial(int k, int N, double p)` to compute the probability of obtaining exactly  $k$  heads in  $N$  biased coin flips (heads with probability  $p$ ) using the formula

$$f(k, N, p) = \frac{N!}{k!(N-k)!} (p^k) ((1-p)^{N-k})$$

*Hint:* To stave off overflow, compute  $x = \ln f(N, k, p)$  and then return  $e^x$ . In `main()`, take  $N$  and  $p$  from the command line and check that the sum over all values of  $k$  between 0 and  $N$  is (approximately) 1. Also, compare every value computed with the normal approximation  $f(N, k, p) = \phi(Np, Np(1-p))$

(see Exercise 2.2.1).

35. **Coupon collecting from a binomial distribution.** Develop a version of `getCoupon()` that uses `binomial()` from the previous exercise to return coupon values according to the binomial distribution with  $p = 1/2$ . Hint : Generate a uniformly distributed random number  $x$  between 0 and 1, then return the smallest value of  $k$  for which the sum of  $f(N, j, p)$  for all  $j < k$  exceeds  $x$ . Extra credit : Develop a hypothesis for describing the behavior of the coupon collector function under this assumption.
  36. **Chords.** Develop a version of `PlayThatTune.java` that can handle songs with chords (including harmonics). Develop an input format that allows you to specify different durations for each chord and different amplitude weights for each note within a chord. Create test files that exercise your program with various chords and harmonics, and create a version of Fur Elise that uses them.
  37. **Postal bar codes.** The [POSTNET barcode](#) used by the U.S. Postal System to route mail is defined as follows: Each decimal digit in the zip code is encoded using a sequence of three half-height and two full-height bars. The barcode starts and ends with a full-height bar (the guard rail) and includes a checksum digit (after the five-digit zip code or ZIP+4), computed by summing up the original digits modulo 10. Implement the following functions

- Draw a half-height or full-height bar on StdDraw.
  - Given a digit, draw its sequence of bars.
  - Compute the checksum digit.

and a test client that reads in a five- (or nine-) digit zip code as the command-line argument and draws the corresponding postal bar code.

Encoding										
----------	--	--	--	--	--	--	--	--	--	--

38. **Calendar.** Write a program [Calendar.java](#) that takes two command-line arguments  $m$  and  $y$  and prints out the monthly calendar for the  $m$ th month of year  $y$ , as in this example:

February 2009						
S	M	Tu	W	Th	F	S
1	2	3	4	5	6	7
8	9	10	11	12	13	14
15	16	17	18	19	20	21
22	23	24	25	26	27	28

*Hint:* See [LeapYear.java](#) and Exercise 1.2.29.

39. **Fourier spikes.** Write a program that takes a command-line argument  $N$  and plots the function  $\text{Plot}(\cos(t) + \cos(2t) + \cos(3t) + \cos(4t) + \dots + \cos(Nt)) / N$  for 500 equally spaced samples of  $t$  from -10 to 10 (in radians). Run your program for  $N = 5$  and  $N = 500$ . *Note:* You will observe that the sum converges to a spike (0 everywhere except a single value). This property is the basis for a proof that any smooth function can be expressed as a sum of sinusoids.

## Web Exercises

1. **Diamond tile.** Write a program [DiamondTile.java](#) that takes a command-line argument  $N$  and creates an  $N$ -by- $N$  tile of diamonds. Include static methods `diamond()` and `filledDiamond()`.
2. **Hexagonal tile.** Write a program [HexTile.java](#) that takes a command-line argument  $N$  and creates an  $N$ -by- $N$  tile of hexagons. Include static methods `hexagon()` and `filledHexagon()`.
3. **Inverse Gaussian cumulative distribution.** Suppose SAT math scores are normally distributed with mean 500 and standard deviation. Estimate how high student must score in order to be among the top 10%. To do this you need to find the value  $z$  for which  $\Phi(z, 500, 100) = 0.9$ . *Hint:* use binary search.
4. **SAT scores.** A prominent northeastern university receives 20,000 student applications. Assume that the SAT scores of these individuals is normally distributed with mean 1200 and standard deviation 100. Suppose the university decides to admit the 5,000 students with the best SAT scores. Estimate the lowest score that will still be admitted.
5. **Voting machines.** Suppose that in a population of 100 million voters, 51% vote for candidate A and 49% vote for candidate B. However, the voting machines are prone to make mistakes, and 5% of the time they produce the wrong answer. Assuming the errors are made independently and at random, is a 5% error rate enough to invalidate the results of a close election? What error rate can be tolerated?
6. **Gambler's histogram.** Write a program [RandomWalk.java](#) that takes one command line parameter  $M$  and simulates a gambler starting with  $M$  who places exactly  $M$  one dollar bets.
  - a. Produce a histogram of the amount of money the gambler ends up with by running this experiment  $N$  times.
  - b. The amount of money the gambler ends up with follows a binomial distribution with mean  $M$  and variance  $N/4$ . The distribution can be approximated by a normal distribution with the same mean and variance. Produce a histogram for the fraction of time you'd expect the gambler to end up with the amount in each histogram bin.
 Organize your program up into several functions.
7. **Statistical sampling.** Write a program [Sampling.java](#) takes a random sample of  $N$  people and asks them a yes/no question. Compute a 95% confidence interval.
8. **Blackjack.** Write a program [Blackjack.java](#) that plays the [basic strategy](#) or write a program [BlackjackCounter.java](#) that implements the [high-low card counting system](#).
9. **Wavelets.** Applications to computer vision, human vision, speech processing, compressing the FBI fingerprint database, filtering noisy data, detecting self-similarity in time series, sound synthesis, computer graphics, medical imaging, analyzing the clumping of galaxies, and analyzing turbulence. The *Haar function* is definite by  $\Phi(x) = 1$  if  $0 \leq x < 1/2$ ,  $\Phi(x) = -1$  if  $1/2 \leq x < 1$ , and  $\Phi(x) = 0$  otherwise. For integer  $m$  and  $n$ , the *Haar basis function*  $\Phi_{m,n}(x) = 2^{-m/2} \Phi(2^{-m}x - n)$ . Write a program [Haar.java](#) that takes two integer input  $M$  and  $N$ , and one real input  $x$  and prints  $\Phi_{m,n}(x)$ . Or maybe plot it?
10. **Baccarat.** [Baccarat](#) is a simple card game which has been romanticized in James Bond movies. When the player is dealt nine, the croupier proclaims "neuf a la banque". Write a program that determines your chance of winning....
11. **Collinear points.** Write a function

```
public boolean areCollinear(int x1, int y1, int x2, int y2, int x3, int y3)
```

that returns `true` if the three points  $(x_1, y_1)$ ,  $(x_2, y_2)$ , and  $(x_3, y_3)$  lie on the same line, and `false` otherwise.

12. **Gauss error function.** The [error function](#) is a function that arises in probability, statistics, and solutions to differential equations. For example,  $\Phi(z) = 1/2 + (1 + \text{erf}(z / \sqrt{2}))$ , where  $\Phi(z)$  is the Gaussian cumulative distribution function defined above.

$$\text{erf}(z) = \frac{2}{\sqrt{\pi}} \int_{t=0}^z e^{-t^2} dt$$

The integral has no closed form solution in terms of [elementary functions](#) so we resort to approximations. When  $z$  is nonnegative, the Chebyshev fitting estimate below is accurate to 7 significant digits:

$$\begin{aligned} \text{erf}(z) \approx & 1 - t \exp(-z^2 - 1.26551223 + 1.00002368t \\ & + 0.37409196t^2 + 0.09678418t^3 \\ & - 0.18628806t^4 + 0.27886807t^5 \\ & - 1.13520398t^6 + 1.48851587t^7 \\ & - 0.82215223t^8 + 0.17087277t^9) \end{aligned}$$

$$\text{where } t = 1/(1 + \frac{1}{2}z)$$

If  $z$  is negative, use the identity  $\text{erf}(z) = -\text{erf}(-z)$ . A particularly efficient way to implement it is via a judicious use of parentheses known as *Horner's method*. Write a function `erf()` in [ErrorFunction.java](#) takes one real input  $z$  and computes the *error function* using the formula above.

13. **Haversine.** Write a function `haversine()` that takes one `double` argument  $\theta$  and returns  $\text{haversine}(\theta) = (\sin(\theta/2))^2$ .
14. **Soil temperature.** (Cleve Moler) Suppose the soil temperature is uniformly  $T_i$  (20 degrees C) at the surface and beneath. A cold front moves in and the surface temperature  $T_s$  (-15 C) remains for the foreseeable future. The temperature  $T(x, t)$  at time  $t$  seconds of the soil at  $x$  meteres beneath the surface is given by  $T(x, t) = T_s + (T_i - T_s) \text{erf}(x / 2 \sqrt{\alpha t})$ , where  $\alpha (0.138 * 10^{-6} \text{ m}^2 / \text{s})$  is the thermal conductivity of the soil. What is the temperature at 10 meters after 30 days of exposure to these conditions? How long until water freezes (0 degrees) at 5 meters beneath the surface? How deep to dig a water main so that it can withstand 60 days of exposure to these conditions without freezing?
15. **Implied volatility of Black-Scholes.** The volatility parameter  $\sigma$  is often unknown. Suppose you know the price of a call option whose underlying asset has current price  $S$ , strike price  $X$ , time to maturity  $T$ , and interest rate  $r$ . Determine the *implied volatility* by solving the Black-Scholes formula for  $\sigma$ . There is no closed form solution, so use binary search. You can use repeated doubling or halving to find appropriate values of `sigma_min` and `sigma_max` to bracket the true value.
16. **Craps.** Calculate the probability of winning a *pass bet* in craps. Here are the rule for a pass bet. Roll two 6-sided dice, and let  $x$  be their sum.
- if  $x$  is 7 or 11, you win instantly
  - if  $x$  is 2, 3, or 12, you lose instantly
  - otherwise, repeatedly roll two dice until their sum is either  $x$  or 7
    - if their sum is  $x$ , you win
    - if their sum is 7, you lose
- Program [Craps.java](#) takes a command line parameter  $N$  and simulates  $N$  pass bets. The program's organization benefits from two helper functions: `sumOfTwoDice` and `winsPassBet`. Both functions have one interesting feature - they do not take any input arguments. The first function simulate the throw of two dice. This returns an integer between 2 and 12, but not all values are equally likely. To simulate the probabilities correctly, we call `StdRandom.random(6)` twice and one to generate a number between 1 and 6. Then, we add the two values. The second function returns a `boolean` value: `true` if we win the pass bet and `false` otherwise. This function has several `return` statements. As soon as the first one is executed, the function terminates with the given return value.
17. **Musical scale.** Using the function `note()`, write a program [Scale.java](#) to play a major scale.
18. **Primality testing.** Create a function `isPrime()` that takes an integer argument  $N$  and returns `true` or `false` depending on whether  $N$  is prime.

```
public static boolean isPrime(long N) {
    if (N < 2) return false;
    for (long i = 2; i*i <= N; i++) {
        if (N % i == 0) return false;
    }
}
```

```
    }  
    return true;  
}
```

19. **Electronic funds transfer routing number check.** Given a 9 digit EFT routing number  $a_1a_2a_3a_4a_5a_6a_7a_8a_9$  the check equation is

$$3a_1 + 7a_2 + a_3 + 3a_4 + 7a_5 + a_6 + 3a_7 + 7a_8 + a_9 \bmod 10 = 0$$

## Check digit reference.

20. Write a static method `nint()` that takes a real number as a parameter and returns the nearest integer. Do not use any Math library function, instead use casting.
  21. Write a static method `int mod(int a, int n)`, where `a` is an integer and `n` is a positive integer, and returns  $a \bmod n$ . This corresponds to `a % n` when `a` is positive, but if `a` is negative, `a % n` returns a nonpositive integer.

## 22. Present value.

- a. Write a method `fv` that computes the amount of money you will have if you invest  $C$  dollars today at the compound interest rate of  $r$  per period, in  $T$  periods. The formula for the *future value* is given by  $C*(1 + r)^T$ .
  - b. Write a method `pv` that computes the amount of money that would have to be invested now, at the compound interest rate  $r$  per period, to obtain a cash flow of  $C$  in  $T$  periods. The formula for the *present value* is given by  $C/(1 + r)^T$ .

23. The ACT is another standardized test. Assume that test scores follow a Gaussian distribution with mean 18 and standard deviation 6. Also assume that the test takers for SAT and ACT are indistinguishable. Which is better, an SAT score of 620 or ACT of 26?

24. Write a program `Factorial.java` that takes one integer command line input  $n$  and prints out  $n! = 1 * 2 * \dots * n$ . Write a function that has the following signature:

```
public static long factorial(int n)
```

What is the largest value of  $n$  that your function can handle without overflow?

25. What is wrong with the following function?

```
static int sum(int n) {
    if (n < 0) return;
    double sum = 0.0;
    for (int i = 0; i < n; i++)
        sum = sum + i;
    return sum;
}
```

**Answer:** The function is declared to return a value of type `int`. The first return statement is wrong since it returns nothing. The second return statement is wrong since it returns a value of type `double`.

- ## 26. What does the following do?

```
public static void negate(int a) {  
    a = -a;  
}  
  
public static int main(String[] args) {  
    int a = 17;  
    System.out.println(a);  
    negate(a);  
    System.out.println(a);  
}
```

*Answer:* It prints 17 twice. A function cannot change the value of a primitive type variable in another function.

27. Write a function that takes three real arguments,  $x$ ,  $y$ , and  $s$ , and plots an equilateral triangle centered on  $(x, y)$  with side length  $s$ . Call the function a number of times in `main` to produce an entertaining pattern.
  28. Which of the following functions returns the minimum of its four inputs? Which is easiest to understand and verify that it is correct?

```

public static int min(int a, int b, int c, int d) {
    // if a is the smallest return it
    if (a <= b && a <= c && a <= d) return a;

    // otherwise, if b is the smallest of b, c, and d, return it
    if (b <= c && b <= d) return b;

    // otherwise, return the smaller of c and d
    if (c <= d) return c;
    return d;
}

public static int min(int a, int b, int c, int d) {
    int min = a;
    if (b < min) min = b;
    if (c < min) min = c;
    if (d < min) min = d;
    return min;
}

public static int min(int a, int b, int c, int d) {
    if (a < b && a < c && a < d) return a;
    if (b < c && b < d) return b;
    if (c < d) return c;
    return d;
}

public static int min(int a, int b, int c, int d) {
    if (a <= b) {
        if (a <= c) {
            if (a <= d) return a;
            else return d;
        }
        if (c <= d) return c;
        else return d;
    }
    if (b <= c) {
        if (b <= d) return b;
        else return d;
    }
    else if (c <= d) return c;
    return d;
}

public static int min(int a, int b) {
    if (a <= b) return a;
    else return b;
}

public static int min(int a, int b, int c, int d) {
    return min(min(a, b), min(c, d));
}

```

29. How would you go about testing whether one of the functions in the previous exercise does what it claims to do?

*Answer:* you can't hope to test it on every conceivable input since there are  $2^{128}$  different possible inputs. Instead, test it on all  $4! = 24$  cases depending on whether  $a < b$ ,  $a < c$ , ...,  $c < d$ . Or all  $4^4$  permutations of 0, 1, 2, and 3.

30. What's wrong with the following method call?

```

double y = 2.0;
double x = sqrt(double y);

```

It re-declare the variable `y` when calling `sqrt()`.

31. **Sawtooth.** Write a program `SawTooth.java` to plot  $2/\pi [\sin(1t)/1 + \sin(2t)/2 + \sin(3t)/3 + \dots]$ . As you plot more and more terms, the wave converges to a [sawtooth wave](#). Then play it using standard audio.
32. **Square wave.** Plot  $4/\pi [\sin(1*2*pi*t)/1 + \sin(3*2*pi*t)/3 + \sin(5*2*pi*t)/5 + \dots]$ . As you plot more and more terms, the wave converges to a [square wave](#). Then play it using standard audio.
33. Write a program to print the lyrics to Old McDonald.

```

public static String sing(String animals, String sound) {
    String s = "";
    s += "Old MacDonald had a farm\n";
    s += "E-I-E-I-O\n";
    s += "And on his farm he had some " + animals + "\n";
    s += "With a " + sound + "-" + sound + " here\n";
    s += "And a " + sound + "-" + sound + " there\n";
    s += "Here a " + sound + ", there a " + sound + "\n";
    s += "Everywhere a + sound + "-" + sound + "\n";
    s += "Old MacDonald had a farm\n";
    s += "E-I-E-I-O\n";
    return s;
}
...
System.out.println(sing("chicks", "cluck"));
System.out.println(sing("cows", "moo"));
System.out.println(sing("pigs", "oink"));
System.out.println(sing("cats", "meow"));
System.out.println(sing("sheep", "baa"));
System.out.println(sing("ducks", "quack"));
System.out.println(sing("horses", "neigh"));

```

34. Write a static method `maxwellBoltzmann()` that returns a pseudo-random value from a *Maxwell-Boltzmann distribution* with parameter  $\sigma$ . To produce such a value, take the sum of the squares of three Gaussian random variables with mean 0 and standard deviation  $\sigma$ , and return the square root. The speeds of molecules in an ideal gas have a Maxwell-Boltzmann distribution, where the parameter  $\sigma$  is related to XYZ.
35. Write a static method `reverse1()` that takes a string array as an argument and creates a new array with the entries in reverse order (and does not change the order of the entries in the argument array). Write a static method `reverse2()` that takes a string array as an argument and reverses its entries. Put your code in a program [Reverse.java](#).
36. Which function gets called by `f(1, 2)` if I have two overloaded functions

```

public static void f(int x, double y) {
    System.out.println("f(int, double)");
}

public static void f(double x, int y) {
    System.out.println("f(double, int)");
}

```

*Solution.* Ordinarily, Java's type promotion rules would promote either `int` argument to `double`. However, in this case, that would result in two matching overloaded signatures. Since Java can't resolve the ambiguity, [Overloaded.java](#) results in a compile-time error.

*Last modified on August 05, 2011.*

Copyright © 2002–2012 [Robert Sedgewick](#) and [Kevin Wayne](#). All rights reserved.

# INTRODUCTION TO Programming in Java



An Interdisciplinary Approach

Robert Sedgewick • Kevin Wayne

## INTRO TO PROGRAMMING

### 1. Elements of Programming

### 2. Functions

### 3. OOP

### 4. Data Structures

## INTRO TO CS

### 0. Prologue

### 5. A Computing Machine

### 6. Building a Computer

### 7. Theory of Computation

### 8. Systems

### 9. Scientific Computation

## ALGORITHMS, 4TH EDITION



## WEB RESOURCES

### FAQ

### Data

### Code

### Errata

### Appendices

### Lecture Slides

### Programming Assignments

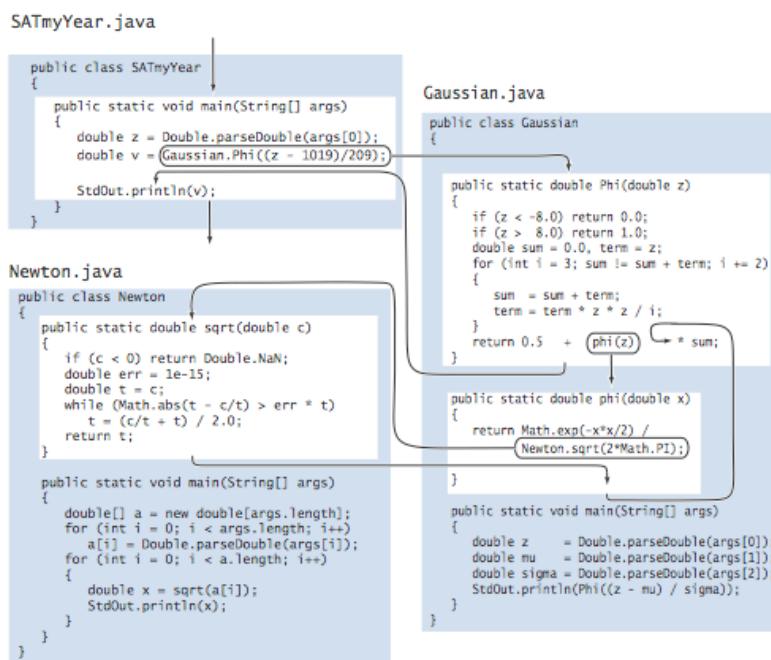
Search booksite...

## 2.2 LIBRARIES AND CLIENTS

Each program that you have written consists of Java code that resides in a single `.java` file. For large programs, keeping all the code in a single file is restrictive and unnecessary. Fortunately, it is very easy in Java to refer to a method in one file that is defined in another. This ability has two important consequences on our style of programming:

- It allows us to *extend the Java language* by developing libraries of static methods for use by any other program, keeping each library in its own file.
- It enables *modular programming*, where we divide a program up into static methods, grouped together in some logical way according to the dictates of the application. Modular programming is important because it allows us to independently develop, debug, and even compile parts of big program one piece at a time, leaving each finished piece in its own file for use without having to worry about its details again.

**Using static methods in other programs.** To refer to a static method in one class that is defined in another: Keep both classes in the same directory in your computer. To call a method, prepend its class name and a period separator. For example, `SATmyYear.java` calls the `Phi()` method in `Gaussian.java`, which calls the `sqrt()` method in `Newton.java`.



We describe several details about the process.

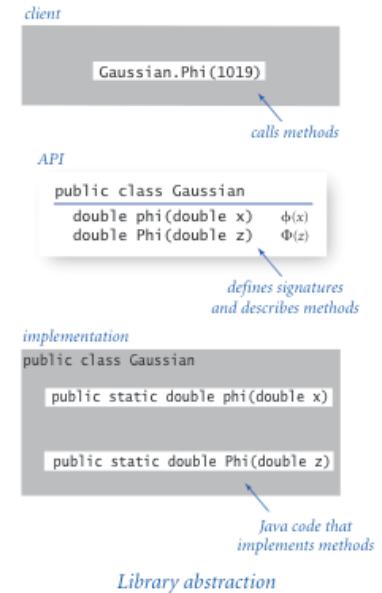
- *The public keyword.* The `public` modifier identifies the method as available for use by any other program with access to the file. You can also identify methods as `private` (and there are a few other categories) but you have no reason to do so at this point.
- *The .class file.* It is natural to use the term *program* to refer to all the code that we keep in a single file, and we use the term *modular program* to refer to a set of files that refer to methods in one another. In Java, by convention, each program is a Java class (kept in a `.java` file with the same name of the class). For now, each class is merely a set of static methods (one of which is `main()`). When you compile the program (by typing `javac` followed by the class name), the Java compiler makes a file with the class name followed by a `.class` extension that has the code of your program in a language more suited to your computer.
- *Compile when necessary.* When you compile a program, the Java compiler will compile everything that needs to be compiled in order to run that program. If you were to be using `Newton.sqrt()` in `Gaussian`, when you type `javac Gaussian.java`, Java would also check whether you modified `Newton.java` since the last time it was compiled (by checking the time it was last changed against the time `Newton.class` was created). If so, it would also compile `Newton`.
- *Multiple main methods.* `SATmyYear`, `Newton`, and `Gaussian` each have `main()` methods. When you type `java` followed by a class name, Java transfers control to the machine code corresponding to the `main()` static method defined in *that* class.

**Libraries.** We refer to a class that comprises methods for use by other programs as a *library*.

- *Clients.* We use the term *client* to refer to the program that calls a given method. When a class contains a method that is a client of a method in another class, we say that the first class is a client of the second class.
- *APIs.* Programmers normally think in terms of a *contract* between the client and the implementation that is a clear specification of what the method is to do.

- *Implementations.* We use the term *implementation* to describe the Java code that implements the methods in an API.

This same mechanism is effective for user-defined libraries. The API allows any client to use the library without having to examine the code in the implementation. The guiding principle in API design is to *provide to clients the methods they need and no others*. For example, [Gaussian.java](#) is an implementation of the following API:



```
public class Gaussian
    double phi(double x) φ(x)
    double phi(double x, double m, double s) φ(x, μ, σ)
    double Phi(double z) Φ(z)
    double Phi(double z, double m, double s) Φ(z, μ, σ)
```

**Random numbers.** [StdRandom.java](#) is a library for generating random numbers from various distributions.

<code>public class StdRandom</code>	<hr/>	
<code>    int uniform(int N)</code>	<i>integer between 0 and N-1</i>	
<code>    double uniform(double lo, double hi)</code>	<i>real between lo and hi</i>	
<code>    boolean bernoulli(double p)</code>	<i>true with probability p</i>	
<code>    double gaussian()</code>	<i>normal, mean 0, standard deviation 1</i>	
<code>    double gaussian(double m, double s)</code>	<i>normal, mean m, standard deviation s</i>	
<code>    int discrete(double[] a)</code>	<i>i with probability a[i]</i>	
<code>    void shuffle(double[] a)</code>	<i>randomly shuffle the array a[]</i>	

- *API design.* Each of the methods in `StdRandom` make certain assumptions about the values of their arguments. For example, we assume that clients will call `uniform(N)` only for positive integers `N` and `bernoulli(p)` only for `p` between 0 and 1. All of these assumptions are part of the contract between the client and the implementation.
- *Unit testing.* Even though we implement `StdRandom` without reference to any particular client, it is good programming practice to include a test method `main()` that, while not used when a client class uses the library, is helpful for use when debugging and testing the methods in the library. Whenever you create a library, you should include a `main()` method for unit testing and debugging. Proper unit testing can be a significant programming challenge in itself, but, at a minimum, you should always include a `main()` method that
  - exercises all the code
  - provides some assurance that the code is working
  - takes a parameter from the command line to allow more testing and then refine that `main()` to do more exhaustive testing as you use the library more extensively.
- *Stress testing.* An extensively used library such as this one should also be subject to *stress testing*, where we make sure that it does not crash when the client breaks some assumption in the contract or makes some assumption that is not explicitly covered. What should `discrete()` do if array entries do not sum to exactly 1? Such cases are sometimes referred to as *corner cases*.

**Input and output for arrays.** It is useful to build a library of static methods for reading arrays of primitive types from standard input and printing them to standard output, as expressed in this API:

```

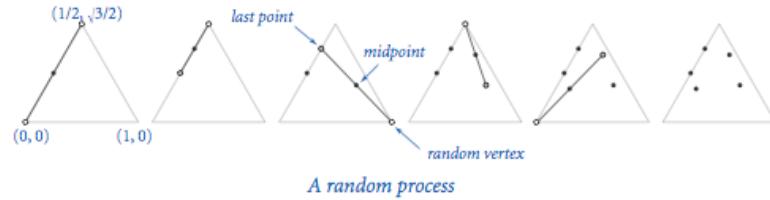
public class StdArrayIO
    double[] readDouble1D()           read a one-dimensional array of double values
    double[][] readDouble2D()          read a two-dimensional array of double values
    void print(double[] a)            print a one-dimensional array of double values
    void print(double[][] a)          print a two-dimensional array of double values

```

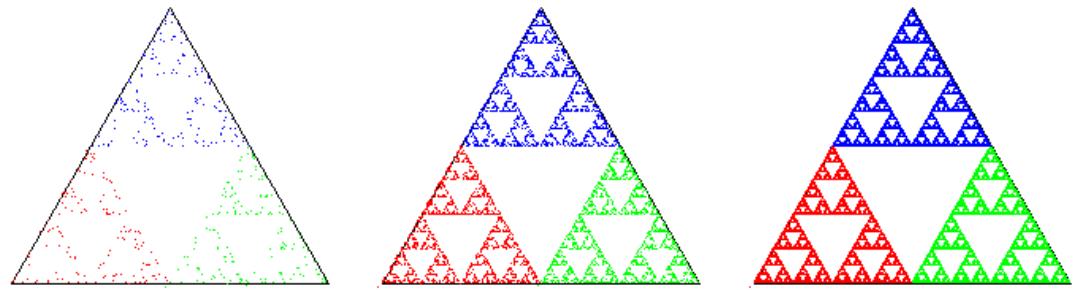
Notes:

1. 1D format is an integer  $N$  followed by  $N$  values.
2. 2D format is two integers  $M$  and  $N$  followed by  $M \times N$  values in row-major order.
3. Methods for `int` and `boolean` are also included.

**Iterated function systems.** An [Iterated function system](#) (IFS) is a general way to produce fractals like the Sierpinski triangle and the Barnsley Fern. As a first example, consider the following simple process: Start by plotting a point at one of the vertices of an equilateral triangle. Then pick one of the three vertices at random and plot a new point halfway between the point just plotted and that vertex. Continue performing the same operation.



[Sierpinski.java](#) simulates this process. Below are snapshots after 1,000, 10,000, and 100,000 steps. You might recognize the figure as the [Sierpinski triangle](#).



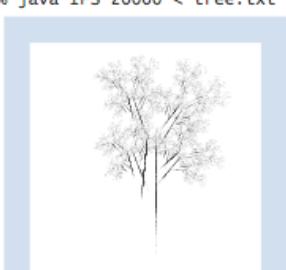
[IFS.java](#) is a data-driven version program that simulates a generalization of this process. See the textbook for details.

You can run it on the inputs [sierpinski.txt](#) or [fern.txt](#) or [tree.txt](#).

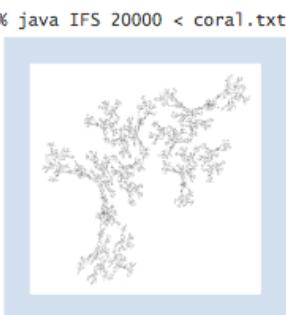
```
% more barnsley.txt          % java IFS 20000 < barnsley.txt
4
.01 .85 .07 .07
4 3
.00 .00 .500
.85 .04 .075
.20 -.26 .400
-.15 .28 .575
4 3
.00 .16 .000
-.04 .85 .180
.23 .22 .045
.26 .24 -.086
```

```
% more tree.txt              % java IFS 20000 < tree.txt
6
.1 .1 .2 .2 .2 .2
6 3
.00 .00 .550
-.05 .00 .525
.46 -.15 .270
.47 -.15 .265
.43 .26 .290
.42 .26 .290
6 3
.00 .60 .000
-.50 .00 .750
.39 .38 .105
.17 .42 .465
-.25 .45 .625
-.35 .31 .525
```

```
% more coral.txt             % java IFS 20000 < coral.txt
3
.40 .15 .45
3 3
.3077 -.5315 .8863
.3077 -.0769 .2166
.0000 .5455 .0106
3 3
-.4615 -.2937 1.0962
.1538 -.4476 .3384
.6923 -.1958 .3808
```



*Examples of iterated function systems*

**Standard statistics.** [StdStats.java](#) is a library for statistical calculations and basic visualizations, as articulated in the following API. See the textbook for details.

---

<code>public class StdStats</code>	
<code>double max(double[] a)</code>	<i>largest value</i>
<code>double min(double[] a)</code>	<i>smallest value</i>
<code>double mean(double[] a)</code>	<i>average</i>
<code>double var(double[] a)</code>	<i>sample variance</i>
<code>double stddev(double[] a)</code>	<i>sample standard deviation</i>
<code>double median(double[] a)</code>	<i>median</i>
<code>void plotPoints(double[] a)</code>	<i>plot points at (i, a[i])</i>
<code>void plotLines(double[] a)</code>	<i>plot lines connecting points at (i, a[i])</i>
<code>void plotBars(double[] a)</code>	<i>plot bars to points at (i, a[i])</i>

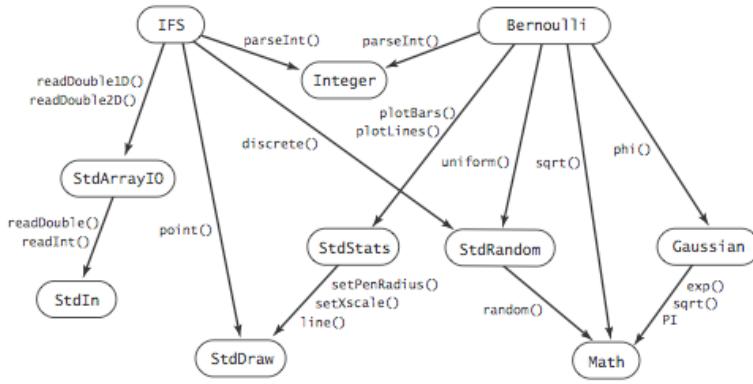
*Note: overloaded implementations are included for all numeric types*

**Bernoulli trials.** Program [Bernoulli.java](#) counts the number of heads found when a fair coin is flipped  $N$  times and compares the result with the predicted Gaussian distribution function. According to the [Central Limit Theorem](#), the resulting histogram is extremely well approximated by the Gaussian distribution with mean  $N/2$  and variance  $N/4$ . Below are sample outputs when  $N = 16, 32$ , and  $64$ .

```
% java Bernoulli 20 100000
```



**Modular programming.** The library implementations that we have developed illustrate a programming style known as *modular programming*. Instead of writing a new program that is self-contained in its own file to address a new problem, we break up each task into smaller, more manageable subtasks, then implement and independently debug code that addresses each subtask. [IFS.java](#) and [Bernoulli.java](#) exemplify modular programming because they are relatively sophisticated computations that are implemented with several relatively small modules.



Dependency graph for the modules in this section

We emphasize modular programming throughout this book because it has many important advantages, including the following:

- *Programs of a reasonable size.* No large task is so complex that it cannot be divided into smaller subtasks.
- *Debugging.* With modular programming and our guiding principle of keeping the scope of variables local to the extent possible, we severely restrict the number of possibilities that we have to consider when debugging. Equally important is the idea of a contract between client and implementation. Once we are satisfied that an implementation is meeting its end of the bargain, we can debug all its clients under that assumption.
- *Code reuse.* Once we have implemented libraries such as `StdStats` and `StdRandom`, we do not have to worry about writing code to compute averages or standard deviations or to generate random numbers again—we can simply reuse the code that we have written.
- *Maintenance.* Like a good piece of writing, a good program can always be improved. Suppose that while developing a new client, you find a bug in some module. With modular programming, fixing that bug amounts to fixing bugs in all of the module's clients.

## Q + A

**Q.** I tried to use `StdRandom`, but get the following error message. What's wrong?

```
Exception in thread "main" java.lang.NoClassDefFoundError: StdRandom.
```

**A.** You need to download `StdRandom.java` into the directory containing your client, or use your operating system's classpath mechanism.

**Q.** Is there a keyword that identifies a class as a library?

**A.** No, any set of public methods will do. There is a bit of a conceptual leap in this viewpoint, because it is one thing to sit down to create a `.java` file that you will compile and run (perhaps run again sometime later with different data), quite another thing to create a `.java` file that you will rely on much later in the future, and still another thing to create a `.java` file for someone else to use in the future. You need to develop some libraries for your own use before engaging in this sort of activity, which is the province of experienced systems programmers.

**Q.** How do I develop a new version of a library that I have been using for a while?

**A.** With care. Any change to the API might break any client program, so it is best to work in a separate directory. But then you are working with a copy of the code. If you are changing a library that has a lot of clients, you can appreciate the problems faced by companies putting out new versions of their software. If you just want to add a few methods to a library, go ahead: that is usually not too dangerous.

**Q.** How do I know that an implementation behaves properly? Why not automatically check that it satisfies the API?

**A.** We use informal specifications because writing a detailed specification is not much different than writing a program. Moreover, a fundamental tenet of theoretical computer science says that doing so does not even solve the basic problem because there is no way in general to check that two different programs perform the same computation.

## Exercises

1. Add to `Gaussian.java` an overloaded method implementation `phi(x, mu, sigma)` that computes the Gaussian distribution with a given mean  $\mu$  and standard deviation  $\sigma$ , based on the formula  $\phi(x, \mu, \sigma) = \varphi((x - \mu)/\sigma)/\sigma$ . Also include an overloaded implementation of the associated cumulative distribution function `Phi(x, mu, sigma)`, based on the formula  $\Phi(x, \mu, \sigma) = \Phi((x - \mu)/\sigma)$ .

2. Write a static method library [Hyperbolic.java](#) that implements the *hyperbolic* trigonometric functions based on the definitions  $\sinh(x) = (e^x - e^{-x})/2$  and  $\cosh(x) = (e^x + e^{-x})/2$ , with  $\tanh(x)$ ,  $\coth(x)$ ,  $\text{sech}(x)$ ,  $\text{csch}(x)$  defined in a manner analogous to standard trigonometric functions.
3. Write a test client for both `StdStats` and `StdRandom` that checks that all of the methods in both libraries operate as expected. Take a command-line parameter  $N$ , generate  $N$  random numbers using each of the methods in `StdRandom`, and print out their statistics. Extra credit : Defend the results that you get by comparing them to those that are to be expected from mathematical analysis.
4. Add to `StdRandom.java` a method `shuffle()` that takes an array of `double` values as argument and rearranges them in random order. Implement a test client that checks that each permutation of the array is produced about the same number of times.
5. Develop a client that does stress testing for `StdRandom`. Pay particular attention to `discrete()`. For example, do the probabilities sum to 1?
6. Develop a full implementation of `StdArrayIO.java` (implement all 12 methods indicated in the API).
7. Write a method that takes double values `ymin` and `ymax` (with `ymin` strictly less than `ymax`) and a double array `a[]` as arguments and uses the `StdStats` library to linearly scale the values in `a[]` so that they are all between `ymin` and `ymax`.
8. Write a `Gaussian.java` and `StdStats.java` client that explores the effects of changing the mean and standard deviation on the Gaussian distribution curve. Create one plot with curves having a fixed mean and various standard deviations and another with curves having a fixed standard deviation and various means.
9. Add to `StdRandom.java` a static method `maxwellBoltzmann()` that returns a random value drawn from a *Maxwell-Boltzmann distribution* with parameter  $\sigma$ . To produce such a value, return the square root of the sum of the squares of three Gaussian random variables with mean 0 and standard deviation  $\sigma$ . The speeds of molecules in an ideal gas have a Maxwell-Boltzmann distribution.
10. Modify `Bernoulli.java` to animate the bar graph, replotting it after each experiment, so that you can watch it converge to the normal distribution. Then add a command-line argument and an overloaded `binomial()` implementation to allow you to specify the probability  $p$  that a biased coin comes up heads, and run experiments to get a feeling for the distribution corresponding to a biased coin. Be sure to try values of  $p$  that are close to 0 and close to 1.
11. Write a library `Matrix.java` that implements the following API.

```

public class Matrix
    double dot(double[] a, double[] b)           vector dot product
    double[][] multiply(double[][] a, double[][] b) matrix-matrix product
    double[] transpose(double[] a)                 transpose
    double[] multiply(double[] a, double[] x)       matrix-vector product
    double[] multiply(double[] x, double[][] a)    vector-matrix product

```

12. Write a `Matrix.java` client `MarkovSquaring.java` that implements the version of `Markov.java` described in Section 1.6 but is based on squaring the matrix, instead of iterating the vector-matrix multiplication.
13. Rewrite `RandomSurfer.java` using the Write a `StdArrayIO.java` and `StdRandom.java` libraries.
14. Add a method `exp()` to `StdRandom.java` that takes an argument  $\lambda$  and returns a random number from the *exponential distribution* with rate  $\lambda$ . Hint: If  $x$  is a random number uniformly distributed between 0 and 1, then  $-\ln x / \lambda$  is a random number from the exponential distribution with rate  $\lambda$ .

## Creative Exercises

15. **Sicherman dice.** Suppose that you have two six-sided dice, one with faces labeled 1, 3, 4, 5, 6, and 8 and the other with faces labeled 1, 2, 2, 3, 3, and 4. Compare the probabilities of occurrence of each of the values of the sum of the dice with those for a standard pair of dice. Use `StdRandom` and `StdStats`.

*Solution:* dice with these properties are called [Sicherman dice](#): they produce sums with the same frequency as regular dice (2 with probability 1/36, 3 with probability 2/36, and so on).

16. **Craps.** Here are the rules for a *pass bet* in the game of *craps*: Roll two 6-sided dice, and let  $x$  be their sum.

- If  $x$  is 7 or 11, you win.
- If  $x$  is 2, 3, or 12, you lose.

Otherwise, repeatedly roll two the dice until their sum is either  $x$  or 7 .

- If their sum is  $x$ , you win.
- If their sum is 7, you lose.

Write a modular program to estimate the probability of winning a pass bet. Modify your program to handle loaded dice, where the probability of a die landing on 1 is taken from the command line, the probability of landing on 6 is 1/6 minus that probability, and 2-5 are assumed equally likely.

*Hint:* Use `StdRandom.discrete()` .

17. **Dynamic 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 `StdStats` 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. Use your program to add code to your solution to exercise 2.2.3 to plot a histogram of the distribution of the numbers produced by each method, taking  $N$  from the command line.

18. **Tukey plot.** A Tukey plot is a data visualization that generalizes a histogram, and is appropriate for use when each integer in a given range is associated with a set of  $y$  values. For each integer in the range, we compute the mean, standard deviation, 10th percentile, and 90th percentile of all the associated  $y$  values; draw a vertical line with  $x$ -coordinate  $i$  running from the 10th percentile  $y$  value to the 90th percentile  $y$  value; and then draw a thin rectangle centered on the line that runs from one standard deviation below the mean to one standard deviation above the mean. Suppose that the standard input stream is a sequence of pairs of numbers where the first number in each pair is an int and the second a double value. Write a `StdStats` and `StdDraw` client that takes an integer  $N$  from the command line and, assuming that all the int values on the input stream are between 0 and  $N-1$ , uses `StdDraw` to make a Tukey plot of the data.
19. **IFS.** Experiment with various inputs to [IFS.java](#) to create patterns of your own design like the Sierpinski triangle, Barnsley fern, or other examples in the table in the text. You might begin by experimenting with minor modifications to the given inputs.
20. **IFS matrix implementation.** Write a version of [IFS.java](#) that uses `Matrix.multiply()` (see Exercise 2.2.11) instead of the equations that compute the new values of  $x_0$  and  $y_0$ .
21. **Stress test.** Develop a client that does stress testing for [StdStats.java](#). Work with a classmate, with one person writing code and the other testing it.
22. **Gamblers ruin.** Develop a [StdRandom.java](#) client to study the gamblers ruin problem (see Program 1.3.8 and Exercises 1.3.21–24). *Note:* defining a static method for the experiment is more difficult than for [Bernoulli.java](#) because you cannot return two values.
23. **Library for properties of integers.** Develop a library based on the functions that we have considered in this book for computing properties of integers. Include functions for determining whether a given integer is prime; whether two integers are relatively prime; computing all the factors of a given integer; the greatest common divisor and least common multiple of two integers; Eulers totient function (Exercise 2.1.26); and any other functions that you think might be useful. Include overloaded implementations for `long` values. Create an API, a client that performs stress testing, and clients that solve several of the exercises earlier in this book.
24. **Voting machines.** Develop a [StdRandom.java](#) client (with appropriate static methods of its own) to study the following problem: Suppose that in a population of 100 million voters, 51% vote for candidate A and 49% vote for candidate B. However, the voting machines are prone to make mistakes, and 5% of the time they produce the wrong answer. Assuming the errors are made independently and at random, is a 5% error rate enough to invalidate the results of a close election? What error rate can be tolerated?
25. **Poker analysis.** Write a [StdRandom.java](#) and [StdStats.java](#) client (with appropriate static methods of its own) to estimate the probabilities of getting one pair, two pair, three of a kind, a full house, and a flush in a five-card poker hand via simulation. Divide your program into appropriate static methods and defend your design decisions. *Extra credit:* Add straight and straight flush to the list of possibilities.
26. **Music library.** Develop a library based on the functions in [PlayThatTuneDeluxe.java](#) that you can use to write client programs to create and manipulate songs.
27. **Animated plots.** Write a program that takes a command-line argument  $M$  and produces a bar graph of the  $M$  most recent double values on standard input. Use the same animation technique that we used for `BouncingBall.java`: erase, redraw, show, and wait briefly. Each time your program reads a new number, it should redraw the whole graph. Since most of the picture does not change as it is redrawn slightly to the left, your program will produce the effect of a fixed-size window dynamically sliding over the input values. Use your program to plot a huge time-variant data file, such as stock prices.
28. **Array plot library.** Develop your own plot methods that improve upon those in [StdStats.java](#). Be creative! Try to make a plotting library that you think that you will use for some application in the future.

## Web Exercises

1. **Sample standard deviation.** The *sample standard deviation* of a sequence of  $N$  observations is defined similar to the standard deviation except that we divide by  $N-1$  instead of  $N$ . Add a method `sampleStddev()` that computes this quantity.
2. **Barnsley fern.** Write a program [Barnsley.java](#) that takes a command line argument  $N$  and plots a sequence of  $N$  points according to the following rules. Set  $(x, y) = (0.5, 0)$ . Then update  $(x, y)$  to one of the following four quantities according to the probabilities given.

PROBABILITY	NEW X	NEW Y
2%	0.5	0.27y
15%	$-0.139x + 0.263y + 0.57$	$0.246x + 0.224y - 0.036$
13%	$0.170x - 0.215y + 0.408$	$0.222x + 0.176y + 0.0893$
70%	$0.781x + 0.034y + 0.1075$	$-0.032x + 0.739y + 0.27$

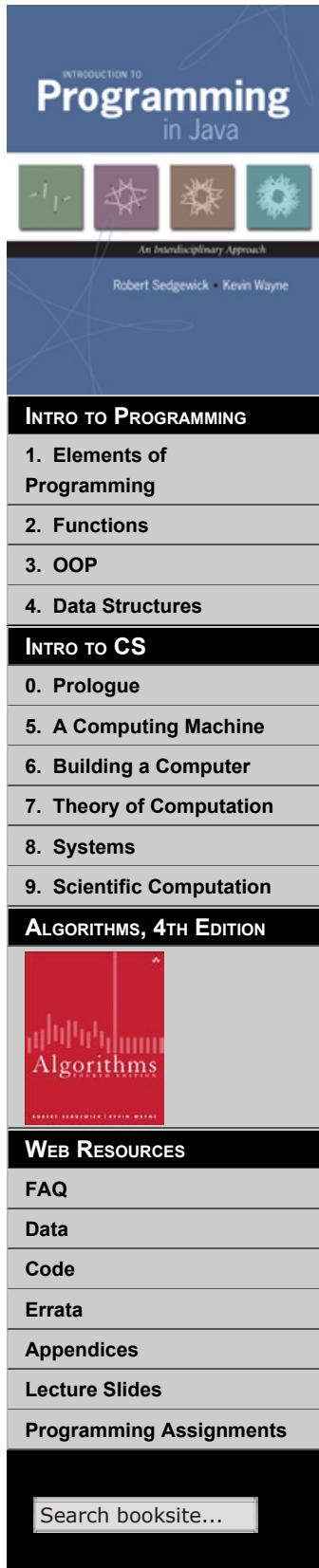
The pictures below show the results after 500, 1000, and 10,000 iterations.



3. **Black-Scholes.** The [Black-Scholes](#) model predicts that the asset price at time  $t$  will be  $S' = S \exp \{ (rt - 0.5\sigma^2 t + \sigma \epsilon \sqrt{t}) \}$ , where  $\epsilon$  is a standard Gaussian random variable. Can use Monte Carlo simulation to estimate. To estimate the value of the option at time  $T$ , compute  $\max(S' - X, 0)$  and take mean over many trials of  $\epsilon$ . The value of the option today is  $e^{-rT} * \text{mean}$ . European put =  $\max(X - S', 0)$ . Reuse function. Name your program [BlackScholes.java](#). See Exercise 2.1.30 for an exact formula for this case.
4. **Simulation.** Application: some kind of simulation which uses `StdRandom` and `StdStats` to flip coins and analyze mean/variance. [Ex: physics, financial based on [Black-Scholes hedge simulation](#). Simulation needed to price options whose payoff depends on the price path, not just the price at the maturity time  $T$ . Ex: Asian average price call =  $\max(0, \bar{S}_T - X)$  where  $\bar{S}_T$  is the average price of the asset from time 0 to  $T$ . [Lookback option](#) =  $\max(0, S(T) - \min_t S_t)$ . Idea: discretize time into  $N$  periods.] [another reference](#) Break up simulation into various pieces encapsulated as functions.
5. **Flaming fractals.** Implement a generalization of IFS to produce [fractal flames](#) like [Water Lilies](#) by [Roger Johnston](#). Flaming fractals differ from classic IFS by using nonlinear update functions (sinusoidal, spherical, swirl, horseshoe), using a log-density display to color pixels according to how many times they result in the process, and incorporating color based on which rule was applied to get to that point.
6. **Random point on a sphere.** Use `StdRandom.gaussian()` to generate a random point on the surface of a sphere or hypersphere using the following method: generate  $N$  random values from the gaussian distribution,  $x[0], \dots, x[N-1]$ . Then  $(x[0]/\text{scale}, \dots, x[N-1]/\text{scale})$  is a random point on the  $N$ -dimensional sphere, where  $\text{scale} = \sqrt{x[0]^2 + \dots + x[N-1]^2}$ .
7. **Coupon collector.** Write a modular program [CouponExperiment.java](#) that runs experiments to estimate the value of the quantity of interest in the coupon collector problem. Compare the experimental results from your program with the mathematical analysis, which says that the expected number of coupons collected before all  $N$  values are found should be about  $N$  times the  $N$ th Harmonic number ( $1 + 1/2 + 1/3 + \dots + 1/N$ ) and the standard deviation should be about  $\sqrt{N}/\sqrt{6}$ .

*Last modified on August 05, 2011.*

Copyright © 2002–2012 [Robert Sedgewick](#) and [Kevin Wayne](#). All rights reserved.



## 2.3 RECURSION

The idea of calling one function from another immediately suggests the possibility of a function calling *itself*. The function-call mechanism in Java supports this possibility, which is known as *recursion*. Recursion is a powerful general-purpose programming technique, and is the key to numerous critically important computational applications, ranging from combinatorial search and sorting methods that provide basic support for information processing (Chapter 4) to the Fast Fourier Transform for signal processing (Chapter 9).

**Your first recursive program.** The `HelloWorld` for recursion is to implement the *factorial* function, which is defined for positive integers  $N$  by the equation

$$N! = N \times (N-1) \times (N-2) \times \dots \times 2 \times 1$$

$N!$  is easy to compute with a `for` loop, but an even easier method in [Factorial.java](#) is to use the following recursive function:

```
public static int factorial(int N) {
    if (N == 1) return 1;
    return N * factorial(N-1);
}
```

You can persuade yourself that it produces the desired result by noting that `factorial()` returns  $1 = 1!$  when  $N$  is 1 and that if it properly computes the value

$$(N-1)! = (N-1) \times (N-2) \times \dots \times 2 \times 1$$

then it properly computes the value

$$N! = N \times (N-1)! = N \times (N-1) \times (N-2) \times \dots \times 2 \times 1$$

We can trace this computation in the same way that we trace any sequence of function calls.

```
factorial(5)
  factorial(4)
    factorial(3)
      factorial(2)
        factorial(1)
          return 1
        return 2*1 = 2
      return 3*2 = 6
    return 4*6 = 24
  return 5*24 = 120
```

Our `factorial()` implementation exhibits the two main components that are required for every recursive function.

- The *base case* returns a value without making any subsequent recursive calls. It does this for one or more special input values for which the function can be evaluated without recursion. For `factorial()`, the base case is  $N = 1$ .
- The *reduction step* is the central part of a recursive function. It relates the function at one (or more) inputs to the function evaluated at one (or more) other inputs. Furthermore, the sequence of parameter values must *converge* to the base case. For `factorial()`, the reduction step is  $N * factorial(N-1)$  and  $N$  decreases by one for each call, so the sequence of parameter values converges to the base case of  $N = 1$ .

**Mathematical induction.** Recursive programming is directly related to *mathematical induction*, a technique for proving facts about discrete functions. Proving that a statement involving an integer  $N$  is true for infinitely many values of  $N$  by mathematical induction involves two steps.

- The *base case* is to prove the statement true for some specific value or values of  $N$

(usually 0 or 1).

- The *induction step* is the central part of the proof. For example, we typically assume that a statement is true for all positive integers less than  $N$ , then use that fact to prove it true for  $N$ .

Such a proof suffices to show that the statement is true for all  $N$ : we can start at the base case, and use our proof to establish that the statement is true for each larger value of  $N$ , one by one.

**Euclid's algorithm.** The *greatest common divisor* (gcd) of two positive integers is the largest integer that divides evenly into both of them. For example, the greatest common divisor of 102 and 68 is 34 since both 102 and 68 are multiples of 34, but no integer larger than 34 divides evenly into 102 and 68.

We can efficiently compute the gcd using the following property, which holds for positive integers  $p$  and  $q$ :

If  $p > q$ , the gcd of  $p$  and  $q$  is the same as the gcd of  $q$  and  $p \% q$ .

The static method `gcd()` in [Euclid.java](#) is a compact recursive function whose reduction step is based on this property.

```
gcd(1440, 408)
  gcd(408, 216)
    gcd(216, 192)
      gcd(192, 24)
        gcd(24, 0)
          return 24
        return 24
      return 24
    return 24
  return 24
return 24
```

**Towers of Hanoi.** No discussion of recursion would be complete without the ancient *towers of Hanoi* problem. We have three poles and  $n$  discs that fit onto the poles. The discs differ in size and are initially arranged on one of the poles, in order from largest (disc  $n$ ) at the bottom to smallest (disc 1) at the top. The task is to move the stack of discs to another pole, while obeying the following rules:

- Move only one disc at a time.
- Never place a disc on a smaller one.

To solve the problem, our goal is to issue a sequence of instructions for moving the discs. We assume that the poles are arranged in a row, and that each instruction to move a disc specifies its number and whether to move it left or right. If a disc is on the left pole, an instruction to move left means to wrap to the right pole; if a disc is on the right pole, an instruction to move right means to wrap to the left pole.

**Exponential time.** One legend says that the world will end when a certain group of monks solve the Towers of Hanoi problem in a temple with 64 golden discs on three diamond needles. We can estimate the amount of time until the end of the world (assuming that the legend is true). If we define the function  $T(n)$  to be the number of move directives issued by [TowersOfHanoi.java](#) to move  $n$  discs from one peg to another, then the recursive code implies that  $T(n)$  must satisfy the following equation:

$$T(n) = 2 T(n - 1) + 1 \text{ for } n > 1, \text{ with } T(1) = 1$$

Such an equation is known in discrete mathematics as a *recurrence relation*. We can often use them to derive a closed-form expression for the quantity of interest. For example,  $T(1) = 1$ ,  $T(2) = 3$ ,  $T(3) = 7$ , and  $T(4) = 15$ . In general,  $T(n) = 2^n - 1$ .

Knowing the value of  $T(n)$ , we can estimate the amount of time required to perform all the moves. If the monks move discs at the rate of one per second, it would take more than one week for them to finish a 20-disc problem, more than 31 years to finish a 30-disc problem, and more than 348 centuries for them to finish a 40-disc problem (assuming that they do not make a mistake). The 64-disc problem would take more than 5.8 billion centuries.

**Gray code.** The playwright Samuel Beckett wrote a play called *Quad* that had the



following property: Starting with an empty stage, characters enter and exit one at a time, but each subset of characters on the stage appears exactly once. The play had four characters and there are  $2^4 = 16$  different subsets of four elements; hence the title. How did Beckett generate the stage directions for this play? How would we do it for 5 actors, or more?

code	subset	move	1-bit code	4-bit code
0 0 0 0	empty		0	0 0 0 0
0 0 0 1	1	enter 1	0 1	0 0 0 1
0 0 1 1	2 1	enter 2	1 1	0 0 1 1
0 0 1 0	2	exit 1	1 0	0 0 1 0
0 1 1 0	3 2	enter 3	0	0 1 1 0
0 1 1 1	3 2 1	enter 1	0	0 1 1 1
0 1 0 1	3 1	exit 2	0	0 1 0 1
0 1 0 0	3	exit 1	0	0 1 0 0
1 1 0 0	4 3	enter 4	0 0 0	1 1 0 0
1 1 0 1	4 3 1	enter 1	0 0 1	1 1 0 1
1 1 1 1	4 3 2 1	enter 2	0 1 1	1 1 1 1
1 1 1 0	4 3 2	exit 1	0 1 0	1 1 1 0
1 0 1 0	4 2	exit 3	1 1 0	1 0 1 0
1 0 1 1	4 2 1	enter 1	1 1 1	1 0 1 1
1 0 0 1	4 1	exit 2	1 0 1	1 0 0 1
1 0 0 0	4	exit 1	1 0 0	1 0 0 0

*Gray code representations*

2-, 3-, and 4-bit Gray codes

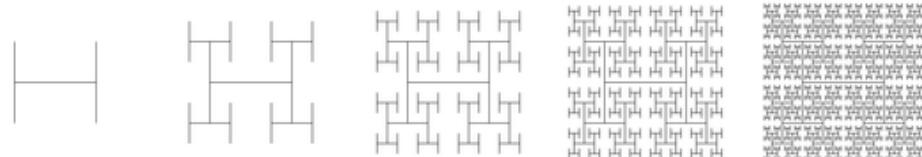
An  $n$ -bit Gray code is a list of the  $2^n$  different  $n$ -bit binary numbers such that each entry in the list differs in precisely one bit from its predecessor. Gray codes directly apply to Beckett's problem because we can interpret each bit as denoting whether the integer corresponding to its bit position is in the subset. Changing the value of a bit from 0 to 1 corresponds to an integer entering the subset; changing a bit from 1 to 0 corresponds to an integer exiting the subset.

How do we generate a Gray code? A recursive plan that is very similar to the one that we used for the towers of Hanoi problem is effective. The  $n$  bit binary reflected Gray code is defined recursively as follows:

- the  $n-1$  bit code, with 0 prepended to each word, followed by
- the  $n-1$  bit code in reverse order, with 1 prepended to each word.

The 0-bit code is defined to be null, so the 1-bit code is 0 followed by 1. The recursive definition leads, after some careful thought, to the implementation in [Beckett.java](#) for printing out Beckett's stage directions.

**Recursive graphics.** Simple recursive drawing schemes can lead to pictures that are remarkably intricate. For example, an *H-tree* of order  $n$  is defined as follows: The base case is null for  $n = 0$ . The reduction step is to draw, within the unit square three lines in the shape of the letter H four H-trees of order  $n-1$ , one connected to each tip of the H with the additional provisos that the H-trees of order  $n-1$  are centered in the four quadrants of the square, halved in size. Program [Htree.java](#) takes a command-line argument  $n$ , and plots an order  $n$  H-tree using standard draw.



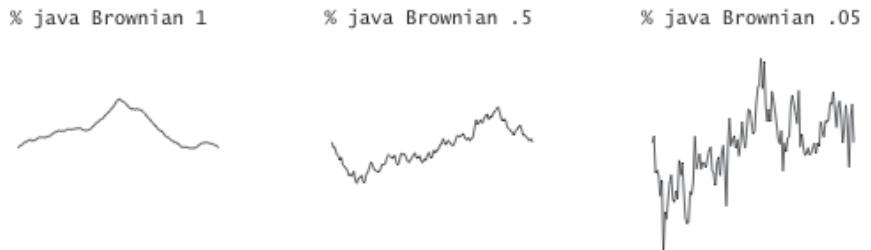
**Brownian bridge.** An H-tree is a simple example of a *fractal*: a geometric shape that can be divided into parts, each of which is (approximately) a reduced size copy of the original. The study of fractals plays an important and lasting role in artistic expression, economic analysis, and scientific discovery. Artists and scientists use them to build compact models of complex shapes that arise in nature and resist description using conventional geometry, such as clouds, plants, mountains, riverbeds, human skin, and many others. Economists also use fractals to model function graphs of economic indicators.

Program [Brownian.java](#) produces a function graph that approximates a simple example known as a *Brownian bridge* and closely related functions. You can think of this graph as a random walk that connects two points, from  $(x_0, y_0)$  to  $(x_1, y_1)$ , controlled by a few parameters. The implementation is based on the *midpoint displacement method*, which is a recursive plan for drawing the plot within an interval  $[x_0, x_1]$ . The base case (when the size of the interval is smaller than a given tolerance) is to draw a straight line connecting the two endpoints. The reduction case is to divide the interval into two halves,

proceeding as follows:

- Compute the midpoint  $(xm, ym)$  of the interval.
- Add to the  $y$ -coordinate of the midpoint a random value  $\delta$ , chosen from the Gaussian distribution with mean 0 and given variance.
- Recur on the subintervals, dividing the variance by a given scaling factor  $s$ .

The shape of the curve is controlled by two parameters: the *volatility* (initial value of the variance) controls the distance the graph strays from the straight line connecting the points, and the *Hurst exponent* controls the smoothness of the curve. We denote the Hurst exponent by  $H$  and divide the variance by  $2^{(2H)}$  at each recursive level. When  $H$  is  $1/2$  (divide by 2 at each level) the standard deviation is constant throughout the curve: in this situation, the curve is a Brownian bridge.



**Pitfalls of recursion.** With recursion, you can write compact and elegant programs that fail spectacularly at runtime.

- *Missing base case.* The recursive function in [NoBaseCase.java](#) is supposed to compute Harmonic numbers, but is missing a base case:

```
public static double H(int N) {  
    return H(N-1) + 1.0/N;  
}
```

If you call this function, it will repeatedly call itself and never return.

- *No guarantee of convergence.* Another common problem is to include within a recursive function a recursive call to solve a subproblem that is not smaller. The recursive function in [NoConvergence.java](#) will go into an infinite recursive loop if it is invoked with an argument  $N$  having any value other than 1:

```
public static double H(int N) {  
    if (N == 1) return 1.0;  
    return H(N) + 1.0/N;  
}
```

- *Excessive space requirements.* Java needs to keep track of each recursive call to implement the function abstraction as expected. If a function calls itself recursively an excessive number of times before returning, the space required by Java for this task may be prohibitive. The recursive function in [ExcessiveSpace.java](#) correctly computes the  $N$ th harmonic number. However, we cannot use it for large  $N$  because the recursive depth is proportional to  $N$ , and this creates a `StackOverflowError` for  $N = 5,000$ .

```
public static double H(int N) {  
    if (N == 0) return 0.0;  
    return H(N-1) + 1.0/N;  
}
```

- *Excessive recomputation.* The temptation to write a simple recursive program to solve a problem must always be tempered by the understanding that a simple program might require exponential time (unnecessarily), due to excessive recomputation. For example, the Fibonacci sequence

0 1 1 2 3 5 8 13 21 34 55 89  
144 233 377 ...

is defined by the formula  $F_n = F_{n-1} + F_{n-2}$  for  $n \geq 2$  with  $F_0 = 0$  and  $F_1 = 1$ . A novice programmer might implement this recursive function to compute numbers in the Fibonacci sequence, as in [Fibonacci.java](#):

```
F(7)
F(6)
F(5)
F(4)
F(3)
F(2)
F(1)
    return 1
F(0)
    return 0
    return 1
F(1)
    return 1
    return 2
F(2)
F(1)
    return 1
F(0)
    return 0
    return 1
    return 3
F(3)
F(2)
F(1)
    return 1
F(0)
    return 0
    return 1
F(1)
    return 1
    return 2
    return 5
F(4)
F(3)
F(2)
    .
    .
    .

```

*Wrong way to compute Fibonacci numbers*

```
public static long F(int n) {
    if (n == 0) return 0;
    if (n == 1) return 1;
    return F(n-1) + F(n-2);
}
```

However, this program is spectacularly inefficient! To see why it is futile to do so, consider what the function does to compute  $F(8) = 21$ . It first computes  $F(7) = 13$  and  $F(6) = 8$ . To compute  $F(7)$ , it recursively computes  $F(6) = 8$  *again* and  $F(5) = 5$ . Things rapidly get worse because both times it computes  $F(6)$  it ignores the fact that it already computed  $F(5)$ , and so forth. The the number of times this program computes  $F(1)$  when computing  $F(n)$  is precisely  $F_n$ . The mistake of recomputation is compounded, exponentially. No imaginable computer will ever be able to do this many calculations.

## Q + A

**Q.** Are there situations when iteration is the only option available to address a problem?

**A.** No, any loop can be replaced by a recursive function, though the recursive version might require excessive memory.

**Q.** Are there situations when recursion is the only option available to address a problem?

**A.** No, any recursive function can be replaced by an iterative counterpart. In Section 4.3, we will see how compilers produce code for function calls by using a data structure called a *stack*.

**Q.** Which should I prefer, recursion or iteration?

**A.** Whichever leads to the simpler, more easily understood, or more efficient code.

**Q.** I get the concern about excessive space and excessive recomputation in recursive code. Anything else to be concerned about?

**A.** Be extremely wary of creating arrays in recursive code. The amount of space used can pile up very quickly, as can the amount of time required for memory management.

## Exercises

1. What happens if you run `factorial()` with negative value of `n`? With a large value, say 35?
2. Write a recursive program that computes the value of  $\ln(N!)$ .
3. Give the sequence of integers printed by a call to `ex233(6)`:

```
public static void ex233(int n) {
    if (n <= 0) return;
    StdOut.println(n);
    ex233(n-2);
    ex233(n-3);
    StdOut.println(n);
}
```

4. Give the value of `ex234(6)`:

```
public static String ex234(int n) {
    if (n <= 0) return "";
    return ex234(n-3) + n + ex234(n-2) + n;
}
```

5. Criticize the following recursive function:

```
public static String ex235(int n) {
    String s = ex233(n-3) + n + ex235(n-2) + n;
    if (n <= 0) return "";
    return s;
}
```

6. Given four positive integers `a`, `b`, `c`, and `d`, explain what value is computed by  $\text{gcd}(\text{gcd}(a, b), \text{gcd}(c, d))$ .
7. Explain in terms of integers and divisors the effect of the following Euclid-like function.

```
public static boolean gcdlike(int p, int q) {
    if (q == 0) return (p == 1);
    return gcdlike(q, p % q);
}
```

*Solution.* Returns whether `p` and `q` are relatively prime.

8. Consider the following recursive function.

```
public static int mystery(int a, int b) {
    if (b == 0) return 0;
    if (b % 2 == 0) return mystery(a+a, b/2);
    return mystery(a+a, b/2) + a;
}
```

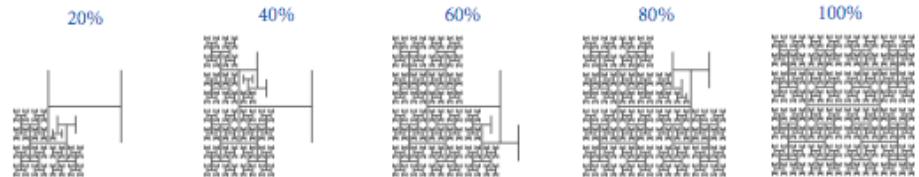
What are the values of `mystery(2, 25)` and `mystery(3, 11)`? Given positive integers `a` and `b`, describe what value `mystery(a, b)` computes. Answer the same question, but replace `+` with `*` and replace `return 0` with `return 1`.

*Solution.* 50 and 33. It computes  $a^*b$ . If you replace `+` with `*`, it computes  $a^b$ .

9. Write a recursive program `Ruler.java` to plot the subdivisions of a ruler using `StdDraw` as in Program 1.2.1.
10. Solve the following recurrence relations, all with  $T(1) = 1$ . Assume  $N$  is a power of

two.

- $T(N) = T(N/2) + 1$
  - $T(N) = 2T(N/2) + 1$
  - $T(N) = 2T(N/2) + N$
  - $T(N) = 4T(N/2) + 3$
11. Prove by induction that the minimum possible number of moves needed to solve the towers of Hanoi satisfies the same recurrence as the number of moves used by our recursive solution.
  12. Prove by induction that the recursive program given in the text makes exactly  $F(n)$  recursive calls to  $F(1)$  when computing  $F(N)$ .
  13. Prove that the second argument to `gcd()` decreases by at least a factor of two for every second recursive call, then prove that  $\text{gcd}(p, q)$  uses at most  $\log_2 N$  recursive calls, where  $N$  is the larger of  $p$  and  $q$ .
  14. Write a program [AnimatedHtree.java](#) that animates the drawing of the H-tree.



Next, rearrange the order of the recursive calls (and the base case), view the resulting animation, and explain each outcome.

## Creative Exercises

15. **Binary representation.** Write a program [IntegerToBinary.java](#) that takes a positive integer  $N$  (in decimal) from the command line and prints out its binary representation. Recall, in program 1.3.7, we used the method of subtracting out powers of 2. Instead, use the following simpler method: repeatedly divide 2 into  $N$  and read the remainders backwards. First, write a `while` loop to carry out this computation and print the bits in the wrong order. Then, use recursion to print the bits in the correct order.
16. **A4 paper.** The width-to-height ratio of paper in the [ISO format](#) is the square root of 2 to 1. Format A0 has an area of 1 square meter. Format A1 is A0 cut with a vertical line into two equal halves, A2 is A1 cut with a horizontal line into two halves, and so on. Write a program that takes a command-line argument  $n$  and uses `StdDraw` to show how to cut a sheet of A0 paper into  $2^n$  pieces. Here's a nice [illustration of A size formats](#).
17. **Permutations.** Write a program [Permutations.java](#) that takes a command-line argument  $n$  and prints out all  $n!$  permutations of the  $n$  letters starting at `a` (assume that  $n$  is no greater than 26). A *permutation* of  $n$  elements is one of the  $n!$  possible orderings of the elements. As an example, when  $n = 3$  you should get the following output. Do not worry about the order in which you enumerate them.

```
bca cba cab acb bac abc
```

18. **Permutations of size  $k$ .** Modify [Permutations.java](#) to [PermutationsK.java](#) so that it takes two command-line arguments  $n$  and  $k$ , and prints out all  $P(n, k) = n! / (n-k)!$  permutations that contain exactly  $k$  of the  $n$  elements. Below is the desired output when  $k = 2$  and  $n = 4$ . You need not print them out in any particular order.

```
ab ac ad ba bc bd ca cb cd da db dc
```

19. **Combinations.** Write a program [Combinations.java](#) that takes one command-line argument  $n$  and prints out all  $2^n$  *combinations* of any size. A combination is a subset of the  $n$  elements, independent of order. As an example, when  $n = 3$  you should get the following output.

```
a ab abc ac b bc c
```

Note that the first element printed is the empty string (subset of size 0).

20. **Combinations of size k.** Modify [Combinations.java](#) to [CombinationsK.java](#) so that it takes two command-line arguments  $n$  and  $k$ , and prints out all  $C(n, k) = n! / (k! * (n-k)!) combinations$  of size  $k$ . For example, when  $n = 5$  and  $k = 3$  you should get the following output.

```
abc abd abe acd ace ade bcd bce bde cde
```

Alternate solution using arrays instead of strings: [Comb2.java](#).

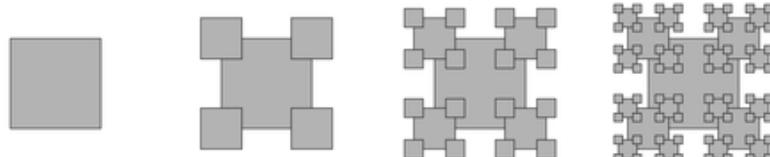
21. **Hamming distance.** The Hamming distance between two bit strings of length  $n$  is equal to the number of bits in which the two strings differ. Write a program that reads in an integer  $k$  and a bit string  $s$  from the command line, and prints out all bit strings that have Hamming distance at most  $k$  from  $s$ . For example if  $k$  is 2 and  $s$  is 0000 then your program should print out

```
0011 0101 0110 1001 1010 1100
```

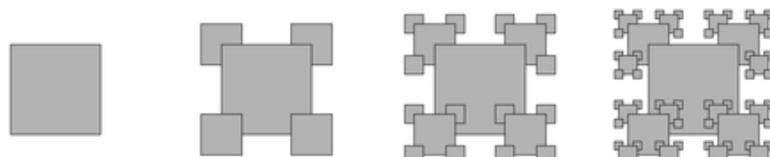
*Hint:* choose  $k$  of the  $N$  bits in  $s$  to flip.

22. **Recursive squares.** Write a program to produce each of the following recursive patterns. The ratio of the sizes of the squares is 2.2. To draw a shaded square, draw a filled gray square, then an unfilled black square.

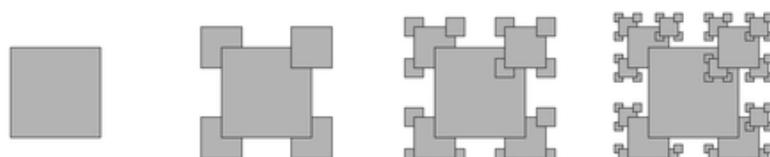
a.



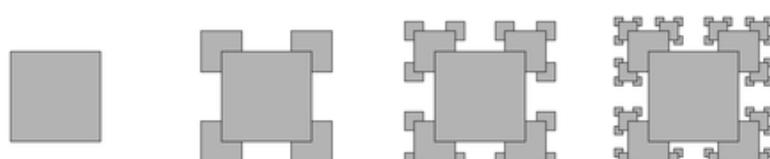
b.



c.



d.



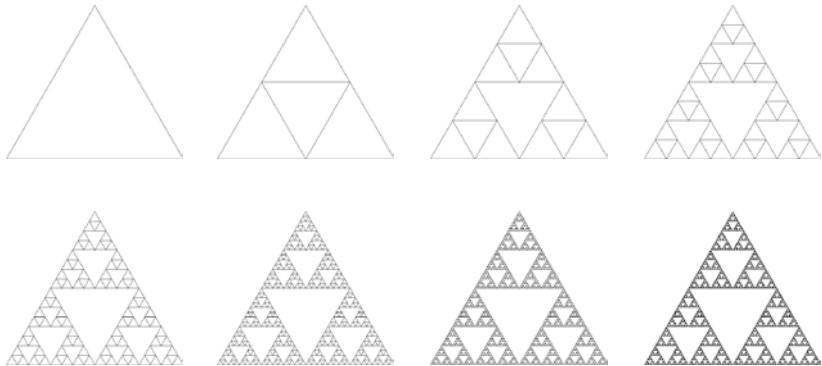
[RecursiveSquares.java](#) gives a solution to part a.

23. **Pancake flipping.** You have a stack of  $N$  pancakes of varying sizes on a griddle. Your goal is to re-arrange the stack in descending order so that the largest pancake is on the bottom and the smallest one is on top. You are only permitted to flip the top  $k$  pancakes, thereby reversing their order. Devise a scheme to arrange the pancakes in the proper order by using at most  $2N - 3$  flips. *Hint:* you can [try out strategies](#) here.
24. **Gray code.** Modify [Beckett.java](#) to print out the Gray code (not just the sequence of bit positions that change).

*Solution.* [GrayCode.java](#) uses Java's string data type; [GrayCodeArray.java](#) uses a

boolean array.

25. **Towers of Hanoi variant.** Consider the following variant of the towers of Hanoi problem. There are  $2N$  discs of increasing size stored on three poles. Initially all of the discs with odd size (1, 3, ...,  $2N-1$ ) are piled on the left pole from top to bottom in increasing order of size; all of the discs with even size (2, 4, ...,  $2N$ ) are piled on the right pole. Write a program to provide instructions for moving the odd discs to the right pole and the even discs to the left pole, obeying the same rules as for towers of Hanoi.
26. **Animated towers of Hanoi animation.** Write a program [AnimatedHanoi.java](#) that uses `StdDraw` to animate a solution to the towers of Hanoi problem, moving the discs at a rate of approximately 1 per second.
27. **Sierpinski triangles.** Write a recursive program to draw the *Sierpinski gasket*. As with `Htree`, use a command-line argument to control the depth of recursion.



28. **Binomial distribution.** Estimate the number of recursive calls that would be used by the code

```
public static double binomial(int N, int k) {  
    if ((N == 0) || (k < 0)) return 1.0;  
    return (binomial(N-1, k) + binomial(N-1, k-1)) / 2.0;  
}
```

to compute `binomial(100, 50)`. Develop a better implementation that is based on memoization. Hint: See Exercise 1.4.26.

29. **Collatz function.** Consider the following recursive function in [Collatz.java](#), which is related to a famous unsolved problem in number theory, known as the [Collatz problem](#) or the *3n + 1 problem*.

```
public static void collatz(int n) {  
    StdOut.print(n + " ");  
    if (n == 1) return;  
    if (n % 2 == 0) collatz(n / 2);  
    else  
        collatz(3*n + 1);  
}
```

For example, a call to `collatz(7)` prints the sequence

7 22 11 34 17 52 26 13 40 20 10 5 16 8 4 2 1

as a consequence of 17 recursive calls. Write a program that takes a command-line argument  $N$  and returns the value of  $n < N$  for which the number of recursive calls for `collatz(n)` is maximized. Hint: use memoization. The unsolved problem is that no one knows whether the function terminates for all positive values of  $n$  (mathematical induction is no help because one of the recursive calls is for a larger value of the argument).

30. **Brownian island.** Benoit Mandelbrot asked the famous question *How long is the coast of Britain?* Modify [Brownian.java](#) to get a program [BrownianIsland.java](#) that plots [Brownian islands](#), whose coastlines resemble that of Great Britain. The modifications are simple: first, change `curve()` to add a gaussian to the  $x$  coordinate as well as to the  $y$  coordinate; second, change `main()` to draw a curve from the point at the center of the canvas back to itself. Experiment with various values of the arguments to get your program to produce islands with a realistic

look.



*Brownian islands with Hurst exponent of .76*

31. **Plasma clouds.** Write a recursive program [Program PlasmaCloud.java](#) to draw plasma clouds, using the method suggested in the text. *Remark:* this technique can be used to produce synthetic terrain, by interpreting the color value as the altitude. Variants of this scheme are widely used in the entertainment industry to generate artificial background scenery for movies and the "Genesis effect" in Star Trek II and the "outline of the "Death Star" in Return of the Jedi.
32. **A strange function.** Consider [McCarthy's 91 function](#):

```
public static int McCarthy(int n) {  
    if (n > 100) return n - 10;  
    else return McCarthy(McCarthy(n+11));  
}
```

Determine the value of `McCarthy(50)` without using a computer. Give the number of recursive calls used by `McCarthy()` to compute this result. Prove that the base case is reached for all positive integers `n` or give a value of `n` for which this function goes into a recursive loop.

33. **Recursive tree.** Write a program [Tree.java](#) that takes a command-line argument `N` and produces the following recursive patterns for `N` equal to 1, 2, 3, 4, and 8.



## Web Exercises

1. Does [Euclid.java](#) still work if the inputs can be negative? If not, fix it. *Hint:* Recall that `%` can return a negative integer if the first input is negative. When calling the function, take the absolute value of both inputs.
2. Write a recursive program [GoldenRatio.java](#) that takes an integer input `N` and computes an approximation to the [golden ratio](#) using the following recursive formula:

$$f(N) = \begin{cases} 1 & \text{if } N = 0 \\ 1 + 1 / f(N-1) & \text{if } N > 0 \end{cases}$$

Redo, but do not use recursion.

3. Discover a connection between the [golden ratio](#) and Fibonacci numbers. *Hint:* consider the ratio of successive Fibonacci numbers:  $2/1, 3/2, 8/5, 13/8, 21/13, 34/21, 55/34, 89/55, 144/89, \dots$
4. Consider the following recursive function. What is `mystery(1, 7)`?

```
public static int mystery(int a, int b) {  
    if (0 == b) return 0;  
    else return a + mystery(a, b-1);  
}
```

Will the function in the previous exercise terminate for every pair of integers a and b between between 0 and 100? Give a high level description of what `mystery(a, b)` returns, given integers a and b between 0 and 100.

*Answer:* `mystery(1, 7) = 1 + mystery(1, 6) = 1 + (1 + mystery(1, 5)) = ... 7 + mystery(1, 0) = 7.`

*Answer:* Yes, the base case is `b = 0`. Successive recursive calls reduce `b` by 1, driving it toward the base case. The function `mystery(a, b)` returns `a * b`. Mathematically inclined students can prove this fact via induction using the identity  $ab = a + a(b-1)$ .

5. Consider the following function. What does `mystery(0, 8)` do?

```
public static void mystery(int a, int b) {  
    if (a != b) {  
        int m = (a + b) / 2;  
        mystery(a, m);  
        StdOut.println(m);  
        mystery(m, b);  
    }  
}
```

*Answer:* infinite loop.

6. Consider the following function. What does `mystery(0, 8)` do?

```
public static void mystery(int a, int b) {  
    if (a != b) {  
        int m = (a + b) / 2;  
        mystery(a, m - 1);  
        StdOut.println(m);  
        mystery(m + 1, b);  
    }  
}
```

*Answer:* stack overflow.

7. Repeat the previous exercise, but replace `if (a != b)` with `if (a <= b)`.
8. What does `mystery(0, 8)` do?

```
public static int mystery(int a, int b) {  
    if (a == b) StdOut.println(a);  
    else {  
        int m1 = (a + b) / 2;  
        int m2 = (a + b + 1) / 2;  
        mystery(a, m1);  
        mystery(m2, b);  
    }  
}
```

9. What does the following function compute?

```
public static int f(int n) {  
    if (n == 0) return 0;  
    if (n == 1) return 1;  
    if (n == 2) return 1;  
    return 2*f(n-2) + f(n-3);  
}
```

10. Write a program [Fibonacci2.java](#) that takes a command-line argument N and prints out the first N Fibonacci numbers using the following alternate definition:

$\begin{aligned} F(n) &= 1 \\ &= F((n+1)/2)^2 + F((n-1)/2)^2 \\ &= F(n/2 + 1)^2 - F(n/2 - 1)^2 \end{aligned}$	$\begin{aligned} &\text{if } n = 1 \text{ or } n = 2 \\ &\text{if } n \text{ is odd} \\ &\text{if } n \text{ is even} \end{aligned}$
---	--

What is the biggest Fibonacci number you can compute in under a minute using this definition? Compare this to [Fibonacci.java](#).

11. Write a program that takes a command-line argument  $N$  and prints out the first  $N$  Fibonacci numbers using the [following method](#) proposed by Dijkstra:

```

F(0) = 0
F(1) = 1
F(2n-1) = F(n-1)^2 + F(n)^2
F(2n) = (2F(n-1) + F(n)) * F(n)

```

12. Prove by mathematical induction that the alternate definitions of the Fibonacci function given in the previous two exercises are equivalent to the original definition.
13. Write a program [Pell.java](#) that takes a command-line argument  $N$  and prints out the first  $N$  *Pell numbers*:  $p_0 = 0$ ,  $p_1 = 1$ , and for  $n \geq 2$ ,  $p_n = 2p_{n-1} + p_{n-2}$ . Print out the ratio of successive terms and compare to  $1 + \sqrt{2}$ .
14. Consider the following function from program [Recursion.java](#):

```

public static void mystery(int n) {
    if (n == 0 || n == 1) return;
    mystery(n-2);
    StdOut.println(n);
    mystery(n-1);
}

```

What does `mystery(6)` print out? Hint: first figure out what `mystery(2)`, `mystery(3)`, and so forth print out.

15. What would happen in the previous exercise if the base case was replaced with the following statement?

```

if (n == 0) return;

```

16. Consider the following recursive functions.

```

public static int square(int n) {
    if (n == 0) return 0;
    return square(n-1) + 2*n - 1;
}

public static int cube(int n) {
    if (n == 0) return 0;
    return cube(n-1) + 3*(square(n)) - 3*n + 1;
}

```

What is the value of `square(5)`? `cube(5)`? `cube(123)`?

17. Consider the following pair of mutually recursive functions. What does `g(g(2))` evaluate to?

```

public static int f(int n) {           public static int g(int n) {
    if (n == 0) return 0;               if (n == 0) return 0;
    return f(n-1) + g(n-1);           return g(n-1) + f(n);
}

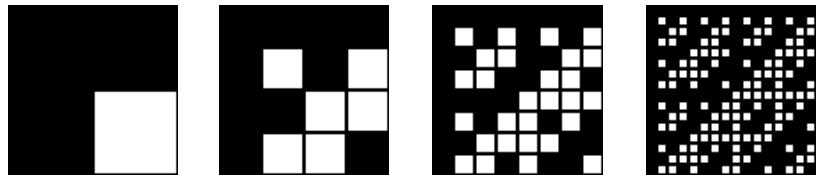
```

18. Write program to verify that (for small values of  $n$ ) the sum of the cubes of the first  $n$  Fibonacci numbers  $F(0)^3 + F(1)^3 + \dots + F(n)^3$  equals  $(F(3n+4) + (-1)^n * 6 * f(n-1)) / 10$ , where  $F(0) = 1$ ,  $F(1) = 1$ ,  $F(2) = 2$ , and so forth.
19. **Transformations by increment and unfolding.** Given two integers  $a \leq b$ , write a program [Sequence.java](#) that transforms  $a$  into  $b$  by a minimum sequence of increment (add 1) and unfolding (multiply by 2) operations. For example,

```
% java Sequence 5 23
23 = ((5 * 2 + 1) * 2 + 1)

% java Sequence 11 113
113 = (((11 + 1) + 1) + 1) * 2 * 2 * 2 + 1)
```

20. **Hadamard matrix.** Write a recursive program [Hadamard.java](#) that takes a command-line argument  $n$  and plots an  $N$ -by- $N$  Hadamard pattern where  $N = 2^n$ . Do *not* use an array. A 1-by-1 Hadamard pattern is a single black square. In general a  $2N$ -by- $2N$  Hadamard pattern is obtained by aligning 4 copies of the  $N$ -by- $N$  pattern in the form of a 2-by-2 grid, and then inverting the colors of all the squares in the lower right  $N$ -by- $N$  copy. The  $N$ -by- $N$  Hadamard  $H(N)$  matrix is a boolean matrix with the remarkable property that any two rows differ in exactly  $N/2$  bits. This property makes it useful for designing *error-correcting codes*. Here are the first few Hadamard matrices.



21. **8 queens problem.** In this exercise, you will solve the classic [8-queens problem](#): place 8 queens on an 8x8 chess board so that no two queens are in the same row, column, or diagonal. There are  $8! = 40,320$  ways in which no two queens are placed in the same row or column. Any permutation  $p[]$  of the integers 0 to 7 gives such a placement: put queen  $i$  in row  $i$ , column  $p[i]$ . Your program [Queens.java](#) should take one command-line argument  $N$  and enumerate all solutions to the  $N$ -queens problem by drawing the location of the queens in ASCII like the two solutions below.

```
* * * * Q * * * *
* Q * * * * * *
* * * * * * Q *
* * * Q * * * * *
* * * * * * Q *
* * * * * * * Q
* * * * * Q * *
Q * * * * * * *
```

*Hint:* to determine whether setting  $q[n] = i$  conflicts with  $q[0]$  through  $q[n-1]$

- if  $q[i]$  equals  $q[n]$ : two queens are placed in the same column
  - if  $q[i] - q[n]$  equals  $n - i$ : two queens are on same major diagonal
  - if  $q[n] - q[i]$  equals  $n - i$ : two queens are on same minor diagonal
22. **Another 8 queens solver.** Program [Queens2.java](#) solves the 8 queens problem by implicitly enumerating all  $N!$  permutations (instead of the  $N^N$  placements). It is based on program [Permutations.java](#).
  23. **Euclid's algorithm and  $\pi$ .** The probability that two numbers chosen from a large random set of numbers have no common factors (other than 1) is  $6 / \pi^2$ . Use this idea to estimate  $\pi$ . Robert Matthews use the same idea to estimate  $\pi$  by taken the set of numbers to be a function of the positions of stars in the sky.
  24. **Towers of Hanoi variant II.** (Knuth-Graham and Pathashnik) Solve the original Towers of Hanoi problem, but with the extra restriction that you are not allowed to directly transfer a disk from A to C. How many moves does it take to solve a

problem with  $N$  disks? *Hint:* move  $N-1$  smallest disks from A to C recursively (without any direct A to C moves), move disk  $N$  from A to B, move  $N-1$  smallest disks from C to A (without any direct A to C moves), move disk  $N$  from B to C, and move  $N-1$  smallest disks from A to C recursively (without any direct A to C moves).

25. **Towers of Hanoi variant III.** Repeat the previous question but disallow both A to C and C to A moves. That is, each move must involve pole B.
26. **Towers of Hanoi with 4 pegs.** Suppose that you have a fourth peg. What is the least number of moves needed to transfer a stack of 8 disks from the leftmost peg to the rightmost peg? [Answer](#). Finding the shortest such solution in general has remained an open problem for over a hundred years and is known as *Reve's puzzle*.
27. **Another tricky recursive function.** Consider the following recursive function. What is  $f(0)$ ?

```
public static int f(int x) {
    if (x > 1000) return x - 4;
    else return f(f(x+5));
}
```

28. **Checking if  $N$  is a Fibonacci number.** Write a function to check if  $N$  is a Fibonacci number. *Hint:* a positive integer is a Fibonacci number if and only if either  $(5^N \cdot N + 4)$  or  $(5^N \cdot N - 4)$  is a perfect square.
29. **Random infix expression generator.** Run [Expr.java](#) with different command-line argument  $p$  between 0 and 1. What do you observe?

```
public static String expr(double p) {
    double r = Math.random();
    if (r <= 1*p) return "(" + expr(p) + " + " + expr(p) + ")";
    if (r <= 2*p) return "(" + expr(p) + " * " + expr(p) + ")";
    return "" + (int) (100 * Math.random());
}
```

30. **A tricky recurrence.** Define  $F(n)$  so that  $F(0) = 0$  and  $F(n) = n - F(F(n-1))$ . What is  $F(100000000)$ ?  
*Solution:* The [answer](#) is related to the Fibonacci sequence and the [Zeckendorf representation](#) of a number.
31. **von Neumann ordinal.** The *von Neumann integer*  $i$  is defined as follows: for  $i = 0$ , it is the empty set; for  $i > 0$ , it is the set containing the von Neumann integers 0 to  $i-1$ .

$0 = \{\}$	$= \{\}$
$1 = \{0\}$	$= \{\{\}\}$
$2 = \{0, 1\}$	$= \{\{\}, \{\{\}\}\}$
$3 = \{0, 1, 2\}$	$= \{\{\}, \{\{\}\}, \{\{\}, \{\{\}\}\}\}$

Write a program [Ordinal.java](#) with a recursive function `vonNeumann()` that takes a nonnegative integer  $N$  and returns a string representation of the von Neumann integer  $N$ . This is a method for defining ordinals in set theory.

32. **Subsequences of a string.** Write a program [Subsequence.java](#) that takes a string command-line argument  $s$  and an integer command-line argument  $k$  and prints out all subsequences of  $s$  of length  $k$ .

```
% java Subsequence abcd 3
abc abd acd bcd
```

33. **Interleaving two strings.** Given two strings  $s$  and  $t$  of distinct characters, print out all  $(M+N)! / (M! N!)$  interleavings, where  $M$  and  $N$  are the number of characters in the two strings. For example, if

```

s = "ab"  t = "CD"
abCD    CabD
aCbD    CaDb
aCDb    CDab

```

34. **Binary GCD.** Write a program [BinaryGCD.java](#) that finds the greatest common divisor of two positive integers using the [binary gcd algorithm](#):  $\text{gcd}(p, q) =$

- $p$  if  $q = 0$
- $q$  if  $p = 0$
- $2 * \text{gcd}(p/2, q/2)$  if  $p$  and  $q$  are even
- $\text{gcd}(p/2, q)$  if  $p$  is even and  $q$  is odd
- $\text{gcd}(p, q/2)$  if  $p$  is odd and  $q$  is even
- $\text{gcd}((p-q)/2, q)$  if  $p$  and  $q$  are odd and  $p \geq q$
- $\text{gcd}(p, (q-p)/2)$  if  $p$  and  $q$  are odd and  $p < q$

35. **Integer partitions.** Write a program [Partition.java](#) that takes a positive integer  $N$  as a command-line argument and prints out all partitions of  $N$ . A [partition](#) of  $N$  is a way to write  $N$  as a sum of positive integers. Two sums are considered the same if they only differ in the order of their constituent summands. Partitions arise in symmetric polynomials and group representation theory in mathematics and physics.

```

% java Partition 4          % java Partition 6
4
3 1
2 2
2 1 1
1 1 1 1
                                6
                                5 1
                                4 2
                                4 1 1
                                3 3
                                3 2 1
                                3 1 1 1
                                2 2 2
                                2 2 1 1
                                2 1 1 1 1
                                1 1 1 1 1 1

```

36. **Johnson-Trotter permutations.** Write a program [JohnsonTrotter.java](#) that take a command-line argument  $n$  and prints out all  $n!$  permutations of the integer 0 through  $n-1$  in such a way that consecutive permutations differ in only one adjacent transposition (similar to way Gray code iterates over combinations in such a way that consecutive combinations differ in only one bit).

```

% java JohnsonTrotter 3
012    (2 1)
021    (1 0)
201    (2 1)
210    (0 1)
120    (1 2)
102    (0 1)

```

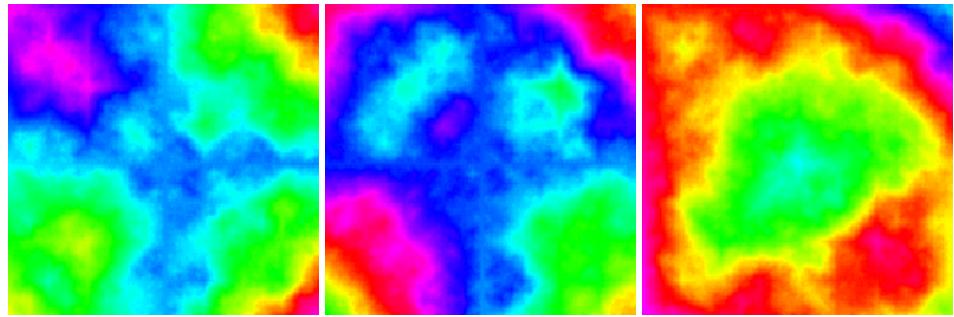
37. **Permutations in lexicographic order.** Write a program [PermutationsLex.java](#) that take a command-line argument  $N$  and prints out all  $N!$  permutations of the integer 0 through  $N-1$  in lexicographic order.

```

% java PermutationsLex 3
012
021
102
120
201
210

```

38. **Derangements.** A [derangement](#) is a permutation  $p[]$  of the integers from 0 to  $N-1$  such that  $p[i]$  doesn't equal  $i$  for any  $i$ . For example there are 9 derangements when  $N = 4$ : 1032, 1230, 1302, 2031, 2301, 2310, 3012, 3201, 3210. Write a program to count the number of derangements of size  $N$  using the following recurrence:  $d[N] = (N-1) (d[N-1] + d[N-2])$ , where  $d[1] = 0$ ,  $d[2] = 1$ . The first few terms are 0, 1, 2, 9, 44, 265, 1854, 14833, 133496, and 1334961.
39. **Tribonacci numbers.** The [tribonacci numbers](#) are similar to the Fibonacci numbers, except that each term is the sum of the three previous terms in the sequence. The first few terms are 0, 0, 1, 1, 2, 4, 7, 13, 24, 44, 81. Write a program to compute tribonacci numbers. What is the ratio successive terms? *Answer.* Root of  $x^3 - x^2 - x - 1$ , which is approximately 1.83929.
40. **Sum of first n Fibonacci numbers.** Prove by induction that the sum of the first  $n$  Fibonacci numbers  $F(1) + F(2) + \dots + F(N)$  is  $F(N+2) - 1$ .
41. **Combinational Gray Code.** Print out all combination of  $k$  of  $n$  items in such a way that consecutive combinations differ in exactly one element, e.g., if  $k = 3$  and  $n = 5$ , 123, 134, 234, 124, 145, 245, 345, 135, 235, 125. *Hint:* use the Gray code, but only print out those integers with exactly  $k$  1's in their binary representation.
42. **Maze generation.** [Create a maze](#) using divide-and-conquer: Begin with a rectangular region with no walls. Choose a random gridpoint in the rectangle and construct two perpendicular walls, dividing the square into 4 subregions. Choose 3 of the four regions at random and open a one cell hole at a random point in each of the 3. Recur until each subregion has width or height 1.
43. **Plasma clouds.** Program [PlasmaCloud.java](#) takes a command-line argument  $N$  and produces a random  $N$ -by- $N$  plasma fractal using the midpoint displacement method.

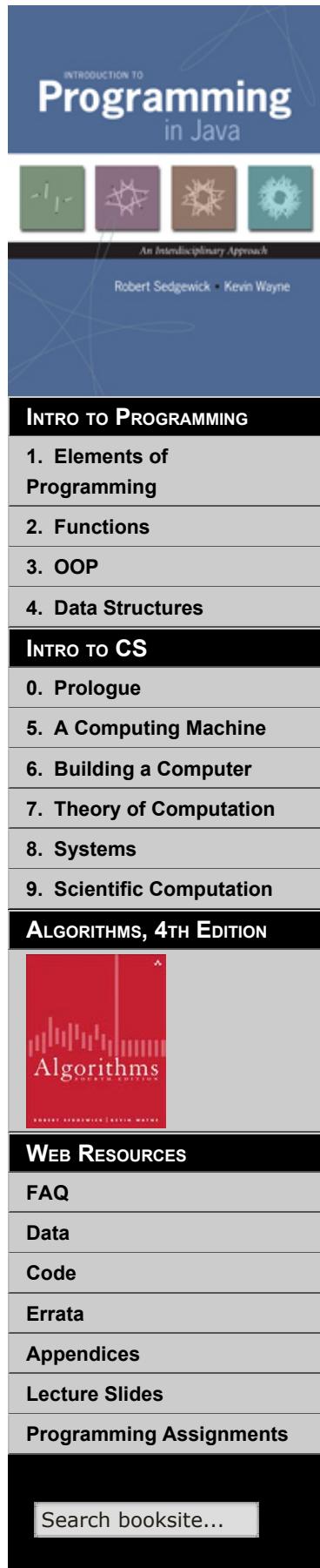


Here's an [800-by-800 example](#). Here's a [reference](#), including a simple 1D version. Note: some visual artifacts are noticeable parallel to the x and y axes. Doesn't have all of the statistical properties of 2D fractional Brownian motion.

44. **Fern fractal.** Write a recursive program to draw a fern or tree, as in this [fern fractal demo](#).
45. **Integer set partition.** Use memoization to develop a program that solves the set partition problem for positive integer values. You may use an array whose size is the sum of the input values.
46. **Voting power.** John F. Banzhaf III proposed a ranking system for each coalition in a block voting system. Suppose party  $i$  control  $w[i]$  votes. A strict majority of the votes is needed to accept or reject a proposal. The [voting power](#) of party  $i$  is the number of minority coalitions it can join and turn it into a winning majority coalition. Write a program [VotingPower.java](#) that takes in a list of coalition weights as command-line argument and prints out the voting power of each coalition. *Hint:* use [Schedule.java](#) as a starting point.
47. **Scheduling on two parallel machines.** Program [Schedule.java](#) takes a command-line argument  $N$ , reads in  $N$  real number of standard input, and partitions them into two groups so that their difference is minimized.

*Last modified on March 02, 2012.*

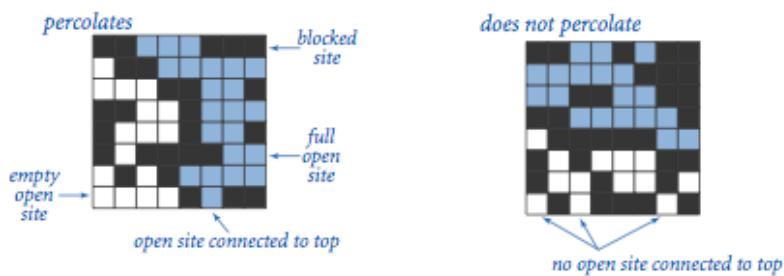
Copyright © 2002–2012 [Robert Sedgewick](#) and [Kevin Wayne](#). All rights reserved.



## 2.4 A CASE STUDY: PERCOLATION

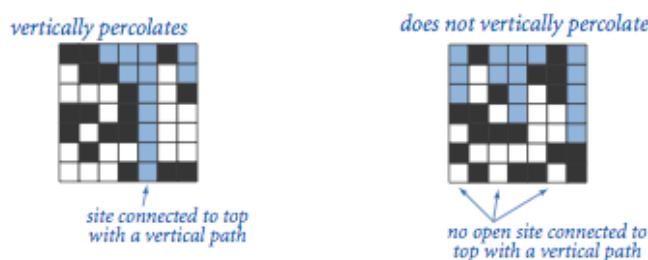
We conclude our study of functions and modules by considering a case study of developing a program to solve an interesting scientific problem, as *Monte Carlo* simulation to study a natural model known as *percolation*.

**Percolation.** We model the system as an  $N$ -by- $N$  grid of sites. Each site is either *blocked* or *open*; open sites are initially *empty*. A *full* site is an open site that can be connected to an open site in the top row via a chain of neighboring (left, right, up, down) open sites. If there is a full site in the bottom row, then we say that the system *percolates*. If sites are independently set to be open with *vacancy probability*  $p$ , what is the probability that the system percolates? No mathematical solution to this problem has yet been derived. Our task is to write computer programs to help study the problem.

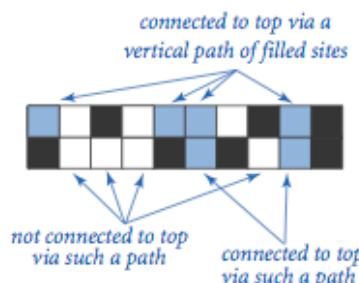


**Scaffolding.** Our first step is to pick a representation of the data. We use one boolean matrix to represent which sites are open and another boolean matrix to represent which sites are full. We will design a `flow()` method that takes as an argument a two-dimensional boolean array `open[][]` that specifies which sites are open and returns another two-dimensional boolean array `full[][]` that specifies which sites are full. We will also include a `percolates()` method that checks whether the array returned by `flow()` has any full sites on the bottom.

**Vertical percolation.** Given a boolean matrix that represents the open sites, how do we figure out whether it represents a system that percolates? For the moment, we will consider a much simpler version of the problem that we call *vertical percolation*. The simplification is to restrict attention to vertical connection paths.

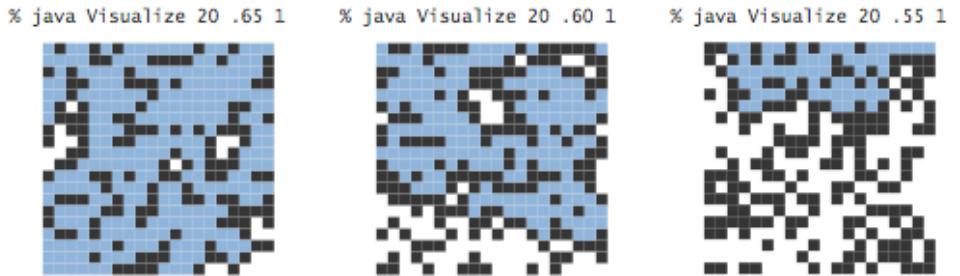


Determining the sites that are filled by some path that is connected vertically to the top is a simple calculation.



Program `VerticalPercolation.java`

**Data visualization.** Program [Visualize.java](#) is a test client that generates random boolean matrices and plots them using standard draw.

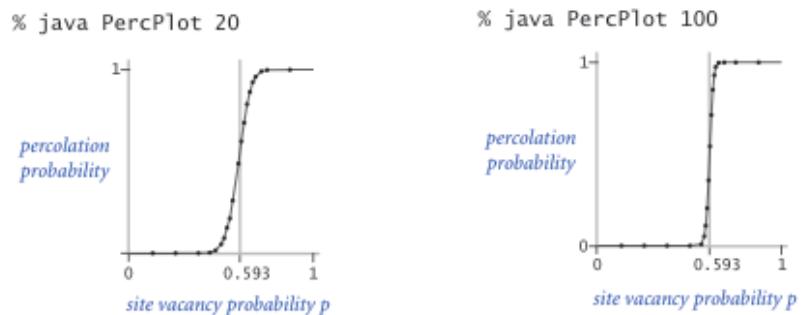


*Percolation is less probable as the site vacancy probability decreases*

**Estimating probabilities.** The next step in our program development process is to write code to estimate the probability that a random system (of size  $N$  with site vacancy probability  $p$ ) percolates. We refer to this quantity as the *percolation probability*. To estimate its value, we simply run a number of experiments. Program [Estimate.java](#) encapsulates this computation in a method `eval()`.

**Recursive solution for percolation.** How do we test whether a system percolates in the general case when any path starting at the top and ending at the bottom (not just a vertical one) will do the job? Remarkably, we can solve this problem with a compact program, based on a classic recursive scheme known as *depth-first search*. Program [Percolation.java](#) contains an implementation of `flow()` that computes the flow array. See the textbook for details.

**Adaptive plot.** To gain more insight into percolation, the next step in program development is to write a program that plots the percolation probability as a function of the site vacancy probability  $p$  for a given value of  $N$ . Immediately, we are faced with numerous decisions. For how many values of  $p$  should we compute an estimate of the percolation probability? Which values of  $p$  should we choose? Program [PercPlot.java](#) implements a recursive approach that produces a good-looking curve at relatively low cost. See the textbook for details.

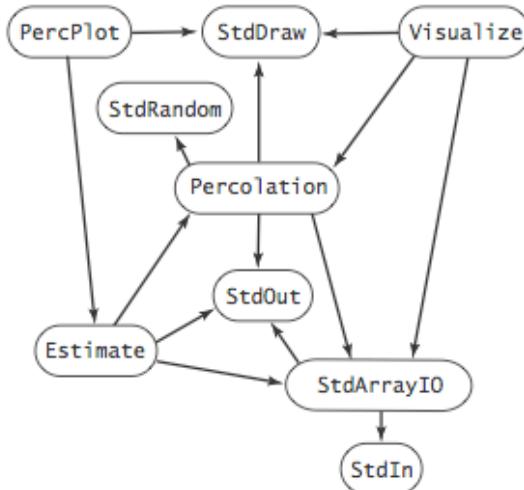


The curves support the hypothesis that there is a *threshold* value (about .593): if  $p$  is greater than the threshold, then the system almost certainly percolates; if  $p$  is less than the threshold, then the system almost certainly does not percolate. As  $N$  increases, the curve approaches a step function that changes value from 0 to 1 at the threshold. This phenomenon, known as a *phase transition*, is found in many physical systems.

### Lessons.

- *Expect bugs.* Every interesting piece of code that you write is going to have at least one or two bugs, if not many more. By running small pieces of code on small test cases that you understand, you can more easily isolate any bugs and then more easily fix them when you find them.
- *Keep modules small.* You can focus attention on at most a few dozen lines of code at a time, so you may as well break your code into small modules as you write it.

- *Limit interactions.* In a well-designed modular program, most modules should depend on just a few others. In particular, a module that calls a large number of other modules needs to be divided into smaller pieces. Modules that are called by a large number of other modules (you should have only a few) need special attention, because if you do need to make changes in a module's API, you have to reflect those changes in all its clients.



*Case study dependencies (not including system calls)*

- *Develop code incrementally.* You should run and debug each small module as you implement it. That way, you are never working with more than a few dozen lines of unreliable code at any given time.
- *Solve an easier problem.* It is typical to begin by putting together the simplest code that you can craft that solves a given problem, as we did with vertical-only percolation.
- *Consider a recursive solution.* Recursion is an indispensable tool in modern programming that you should learn to trust. If you are not already convinced of this fact by the simplicity and elegance of the `flow()` method in [Percolation.java](#), you might wish to try to develop a nonrecursive version.
- *Build tools when appropriate.* Our visualization method `show()` and random boolean matrix generation method `random()` are certainly useful for many other applications, as is the adaptive plotting method of [PercPlot.java](#). Incorporating these methods into appropriate libraries would be simple.
- *Reuse software when possible.* Our `StdIn`, `StdRandom`, and `StdDraw` libraries all simplified the process of developing the code in this section.

## Q + A

**Q.** [Visualize.java](#) and [Estimate.java](#) to rename [Percolation.java](#) to [PercolationEZ.java](#) or whatever method we want to study seems to be a bother. Is there a way to avoid doing so?

**A.** Yes, this is a key issue to be revisited in Chapter 3. In the meantime, you can keep the implementations in separate subdirectories and use the classpath, but that can get confusing. Advanced Java language mechanisms are also helpful, but they also have their own problems.

**Q.** That recursive `flow()` method makes me nervous. How can I better understand what it's doing?

**A.** Run it for small examples of your own making, instrumented with instructions to print a function call trace. After a few runs, you will gain confidence that it always fills the sites connected to the start point.

**Q.** Is there a simple nonrecursive approach?

**A.** There are several methods that perform the same basic computation. We

will revisit the problem in Section 4.5. In the meantime, working on developing a nonrecursive implementation of `flow()` is certain to be an instructive exercise, if you are interested.

**Q.** [PercPlot.java](#) seems to involve a huge amount of calculation to get a simple function graph. Is there some better way?

**A.** Well, the best would be a mathematical proof of the threshold value, but that derivation has eluded scientists. As  $N$  gets large, there is a

**Q.** What is the percolation threshold for different lattices?

**A.** [percolation threshold](#) probability  $p^*$  such that if  $p < p^*$  then no spanning cluster exists, and if  $p \geq p^*$  then one spanning cluster exists.

**Q.** Any relevant papers?

**A.** Here's a recent [physics paper](#) that uses this technique.

## Exercises

1. Write a program that takes  $N$  from the command line and creates an  $N$ -by- $N$  matrix with the entry in row  $i$  and column  $j$  set to true if  $i$  and  $j$  are relatively prime, then shows the matrix on the standard drawing (see Exercise 1.4.13). Then, write a similar program to draw the Hadamard matrix of order  $N$  (see Exercise 1.4.25) and the matrix such with the entry in row  $N$  and column  $j$  set to true if the coefficient of  $x^k$  in  $(1+x)^N$  (binomial coefficient) is odd (see Exercise 1.4.322). You may be surprised at the pattern formed by the latter.
2. Implement a `print()` method for [Percolation.java](#) that prints 1 for blocked sites, 0 for open sites, and \* for full sites.
3. Give the recursive calls for [Percolation.java](#) given the following input:

```
33
1 0 1
0 0 0
1 1 0
```

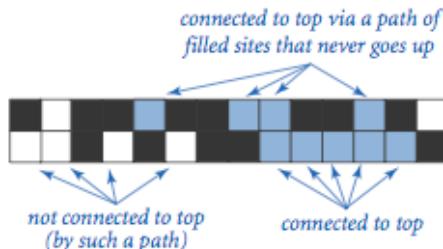
4. Write a client of [Percolation.java](#) like [Visualize.java](#) that does a series of experiments for a value of  $N$  taken from the command line where the site vacancy probability  $p$  increases from 0 to 1 by a given increment (also taken from the command line).
5. Create a program [PercolationDirected.java](#) that test for *directed* percolation (by leaving off the last recursive call in the recursive `show()` method in [Percolation.java](#), as described in the text), then use [PercPlot.java](#) to draw a plot of the directed percolation probability as a function of the site vacancy probability.
6. Write a client of [Percolation.java](#) and [PercolationDirected.java](#) that takes a site vacancy probability  $p$  from the command line and prints an estimate of the probability that a system percolates but does not percolate down. Use enough experiments to get an estimate that is accurate to three decimal places.
7. Describe the order in which the sites are marked when [Percolation](#) is used on a system with no blocked sites. Which is the last site marked? What is the depth of the recursion?
7. Experiment with using [PercPlot.java](#) to plot various mathematical functions (just by replacing the call on `Estimate.eval()` with an expression that evaluates the function). Try the function  $\sin(x) + \cos(10*x)$  to see how the plot adapts to an oscillating curve, and come up with interesting plots for three or four functions of your own choosing.
8. Modify [Percolation.java](#) to animate the flow computation, showing the

sites filling one by one. Check your answer to the previous exercise.

9. Modify [Percolation.java](#) to compute that maximum depth of the recursion used in the flow calculation. Plot the expected value of that quantity as a function of the site vacancy probability  $p$ . How does your answer change if the order of the recursive calls is reversed?
10. Modify [Estimate.java](#) to produce output like that produced by [Bernoulli.java](#). *Extra credit:* Use your program to validate the hypothesis that the data obeys the Gaussian (normal) distribution.

## Creative Exercises

11. *Vertical percolation.* Show that a system with site vacancy probability  $p$  vertically percolates with probability  $1 - (1 - p^N)^N$ , and use [Estimate.java](#) to validate your analysis for various values of  $N$ .
12. *Rectangular percolation systems.* Modify the code in this section to allow you to study percolation in rectangular systems. Compare the percolation probability plots of systems whose ratio of width to height is 2 to 1 with those whose ratio is 1 to 2.
13. *Adaptive plotting.* Modify [PercPlot.java](#) to take its control parameters (gap tolerance, error tolerance, and number of trials) from the command line. Experiment with various values of the parameters to learn their effect on the quality of the curve and the cost of computing it. Briefly describe your findings.
14. *Percolation threshold.* Write a [Percolation.java](#) client that uses binary search to estimate the threshold value (see Exercise 2.1.29).
15. *Nonrecursive directed percolation.* Write a nonrecursive program [PercolationDirectedNonrecursive.java](#) that tests for directed percolation by moving from top to bottom as in our vertical percolation code. Base your solution on the following computation: if any site in a contiguous subrow of open sites in the current row is connected to some full site on the previous row, then all of the sites in the subrow become full.



16. *Fast percolation test.* Modify the recursive `flow()` method in [Percolation.java](#) so that it returns as soon as it finds a site on the bottom row (and fills no more sites). Hint: Use an argument done that is true if the bottom has been hit, false otherwise. Give a rough estimate of the performance improvement factor for this change when running [PercPlot.java](#). Use values of  $N$  for which the programs run at least a few seconds but not more than a few minutes. Note that the improvement is ineffective unless the first recursive call in `flow()` is for the site below the current site.
17. *Bond percolation.* Write a modular program for studying percolation under the assumption that the edges of the grid provide connectivity. That is, an edge can be either empty or full, and a system percolates if there is a path consisting of full edges that goes from top to bottom. Note: This problem has been solved analytically, so your simulations should validate the hypothesis that the percolation threshold approaches  $1/2$  as  $N$  gets large.
18. *Percolation in three dimensions.* Implement a class `Percolation3D` and a class `BooleanMatrix3D` (for I/O and random generation) to study percolation in three-dimensional cubes, generalizing the two-dimensional case studied in this chapter. A percolation system is an  $N$ -by- $N$ -by- $N$  cube of sites that are unit cubes, each open with probability

$p$  and blocked with probability  $1-p$ . Paths can connect an open cube with any open cube that shares a common face (one of six neighbors, except on the boundary). The system percolates if there exists a path connecting any open site on the bottom plane to any open site on the top plane. Use a recursive version of `flow()` like Program 2.4.5, but with eight recursive calls instead of four. Plot percolation probability versus site vacancy probability for as large a value of  $N$  as you can. Be sure to develop your solution incrementally, as emphasized throughout this section.

19. *Bond percolation on a triangular grid.* Write a modular program for studying bond percolation on a triangular grid, where the system is composed of  $2N^2$  equilateral triangles packed together in an  $N$ -by- $N$  grid of rhombus shapes. Each interior point has six bonds; each point on the edge has four; and each corner point has two.
20. *Game of life.* Implement a class `Life` that simulates *Conway's game of life*. Consider a boolean matrix corresponding to a system of cells that we refer to as being either live or dead. The game consists of checking and perhaps updating the value of each cell, depending on the values of its neighbors (the adjacent cells in every direction, including diagonals). Live cells remain live and dead cells remain dead, with the following exceptions:
  - A dead cell with exactly three live neighbors becomes live.
  - A live cell with exactly one live neighbor becomes dead.
  - A live cell with more than three live neighbors becomes dead.

Initialize with a random matrix, or use one of the starting patterns. This game has been heavily studied, and relates to foundations of computer science.

## Web Exercises

1. **2-by-2 percolation.** Verify that the probability that a 2-by-2 system percolates is  $p^2(2 - p^2)$ , where  $p$  is the probability that a site is open.
2. **Cubic curves.** Use recursive subdivision algorithm to plot [cubic curves](#).
3. **Random walk.** Term coined by Hungarian mathematician George Polya in a 1921 paper. Start at 0, go left with probability  $1/2$ , go right with probability  $1/2$ . Reflecting barrier at 0 - if particle hits 0, it must switch direction at next step and return to 1. Absorbing barrier at  $N$  - particle stops when it hits state  $N$ . Estimate number of steps as a function of  $N$  until the particle is absorbed.

*Analytical solution:*  $N^2$ .

4. **3d random walk.** Take a random walk on a 3d lattice, starting from  $(0, 0, 0)$ . Write a program [Polya.java](#) to estimate the chance that you return to the origin after at most some number of steps, say 1,000. (In 1d and 2d, you will surely return; in 3d, there is a less than 50% chance.) Repeat the exercise on a 4d lattice. *Solution:* the actual probability (without the artificial restriction on the number of steps) is known as [Polya's random walk constant](#). It is slightly more than  $1/3$  for 3D and slightly less than  $1/5$  for 4D.
5. **Self-avoiding random walk.** Simulate random walk on lattice until it intersects. Among all self-avoiding walks (SAW) of length  $N$ , what is average distance from origin? To save time in the simulation, exclude SAWs that ever take a step backwards. How long until at least one SAW of length  $N = 40$ ,  $N = 80$ ? What is half-life of a SAW? To save more time in the simulation, allow a SAW only to take a step into an unoccupied cell (and repeat until it gets trapped). Practically nothing is known rigorously about these questions, so simulation is the best recourse. Here's an [article](#) from the New Scientist.

Famous 1949 problem of Nobel prize winning chemist, Flory.

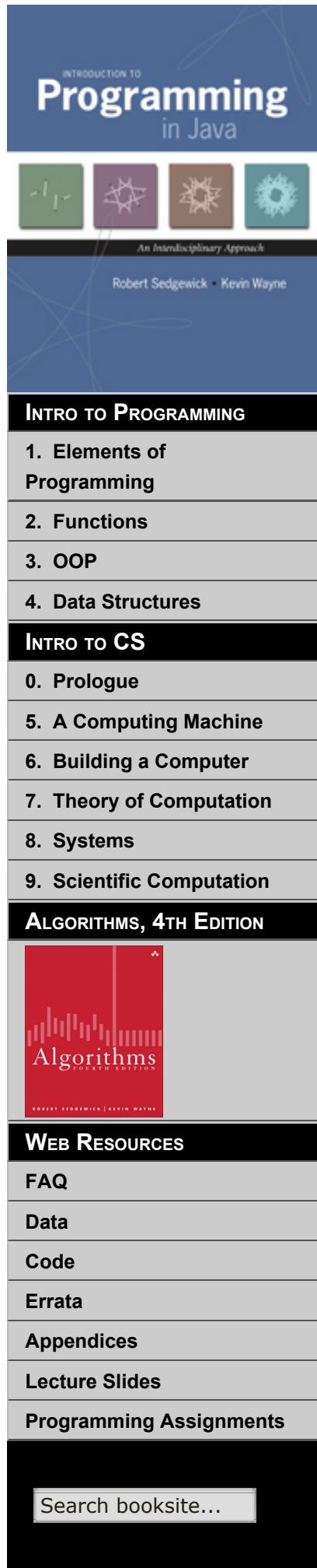
Conjecture: exponent of root mean square displacement is 3/4 in 2D and 3/5 in 3D.

6. **Self-avoiding random walk.** Write a program [SelfAvoidingWalk.java](#) to simulate and animate a 2D self-avoiding random walk. Self-avoiding random walks arise in modeling physical processes like the folding of polymer molecules. Such walks are difficult to model using classical mathematics. so they are best studied by direct numerical simulation. See what fraction of such random walks end up more than  $R^2$  (say 30) from the starting point.

Or keep a trail of length  $N$ .

*Last modified on August 30, 2011.*

Copyright © 2002–2012 [Robert Sedgewick](#) and [Kevin Wayne](#). All rights reserved.



### 3. OBJECT-ORIENTED PROGRAMMING

This section under construction.

**Overview.** In object-oriented programming, we write Java code to create new data types, specifying the values and operations to manipulate those values. The idea originates from modeling (in software) real-world entities such as electrons, people, buildings, or solar systems and extends readily to modeling abstract entities such as bits, numbers, programs or operating systems.

- [3.1 Data Types](#) describes how to use existing reference data types, for text processing image processing.
- [3.2 Creating Data Types](#) describes how to create user-defined data types using Java's class mechanism.
- [3.3 Designing Data Types](#) considers important techniques for designing data types, emphasizing APIs, encapsulation, immutability, and design-by-contract.
- [3.4 Case Study: N-Body Simulation](#) presents a case study that simulates the motion of N particles, subject to Newton's laws of gravity.

**Java programs in this chapter.** Below is a list of Java programs in this chapter. Click on the program name to access the Java code; click on the reference number for a brief description; read the textbook for a full discussion.

REF	PROGRAM	DESCRIPTION
3.1.2	<a href="#">AlbersSquares.java</a>	Albers squares
3.1.4	<a href="#">Grayscale.java</a>	converting color to grayscale
3.1.6	<a href="#">Fade.java</a>	fade effect
3.1.8	<a href="#">GeneFind.java</a>	finding genes in a genome
3.1.10	<a href="#">StockQuote.java</a>	screen scraping for stock quotes
3.2.1	<a href="#">Charge.java</a>	charged-particle data type
3.2.3	<a href="#">Histogram.java</a>	histogram data type
3.2.5	<a href="#">Spiral.java</a>	Spira mirabilis
3.2.7	<a href="#">Mandelbrot.java</a>	Mandelbrot set
3.3.1	<a href="#">Complex.java</a>	Complex numbers (revisited)
3.3.3	<a href="#">Vector.java</a>	spatial vector data type
3.3.5	<a href="#">CompareAll.java</a>	Similarity detection
3.4.2	<a href="#">Universe.java</a>	N-body simulation

**Exercises.** Include a link.

**Old stuff.**

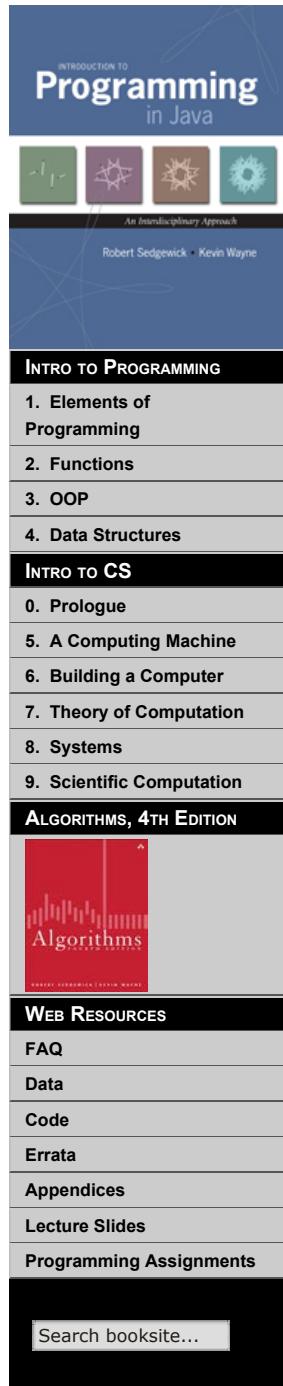
- [3.5 Inheritance](#) introduces a technique to define the

relationships among objects known as inheritance. We cover interfaces and subtyping.

- [3.6 Purple America](#) presents a case study to visualization geographical and election data for US presidential elections.

*Last modified on August 05, 2011.*

Copyright © 2002–2012 [Robert Sedgewick](#) and [Kevin Wayne](#). All rights reserved.



## 3.1 DATA TYPES

This section is under construction.

Organizing the data for processing is an essential step in the development of a computer program. Programming in Java is largely based on doing so with data types known as *reference types* that are designed to support object-oriented programming, a style of programming that facilitates organizing and processing data. The eight primitive data types (boolean, byte, char, double, float, int, long, and short) that you have been using are supplemented in Java by extensive libraries of reference types that are tailored for a large variety of applications. As examples, you will learn in this section how to use several reference types, for image processing and string processing applications. Some of them are built into Java (String and Color) and some were developed for this book (In, Out, and Picture).

**Basic definitions.** A *data type* is a set of values and a set of operations defined on those values. For example, the values of the primitive data type int are integers between  $-2^{31}$  and  $2^{31} - 1$ ; the operations of int are the basic arithmetic and logical operations that we have been using since Chapter 1. You have already been using a data type that is not primitive - the String data type. The values of String are sequences of characters; we have used only a few of the available operations on those values (such as concatenating two strings).

**API.** We specify the behavior of a data type for clients by listing its methods, in an API (*applications program interface*). For example, the table below specifies a class Charge for writing programs that operate on particles that carry electrical charges.

```
public class Charge (PROGRAM 3.2.1)
    Charge(double x0, double y0, double q0)
    double potentialAt(double x, double y)  potential at (x, y) due to charge
    String toString()                      string representation
```

The first method, with the same name as the class and no return type, is a special method known as a *constructor*. The other methods can take arguments and return values in the same manner as the static methods that we have been using, but they are *not* static methods: they implement operations for the data type. Charge has two methods: The first is potentialAt(), which computes and returns the electric potential at the given point that is due to the charge. Coulomb's law tells us that the potential is given by  $V = kq/r$ , where q is the charge value, r is the distance from the point to the charge, and  $k = 8.99 \times 10^9 \text{ Nm}^2/\text{C}^2$  is the electrostatic constant. For consistency, we use SI units. The second is toString(), which returns a string that represents the point charge. While the API does not give details of the implementation, it provides the information we need to write programs that use the data type.

**Color.** Color is a sensation in the eye from electromagnetic radiation. Since people want to view and manipulate color images on their computers, color is a widely used abstraction in computer graphics and Java provides a Color data type. Java has thousands of data types in its libraries, so we need to explicitly list which Java libraries we are using in our program to avoid naming conflicts. Specifically, we include the statement

```
import java.awt.Color;
```

at the beginning of any program that uses Color.

To represent color values, Color uses the RGB system where a color is defined by three integers (each between 0 and 255) that represent the intensity of the red, green, and blue (respectively) components of the color. Other color values are obtained by mixing the red, blue and green components. That is, the data type values of Color are three 8-bit integers. With this convention, Java is using 24 bits to represent each color and can represent  $256^3 = 2^{24} \sim 16.7$  million possible colors. Scientists estimate that the human eye can distinguish only about 10 million distinct colors.

Color has a constructor that takes three integer arguments, so that you can write, for example

```
Color red    = new Color(255, 0, 0);
Color white  = new Color(255, 255, 255);
Color sienna = new Color(160, 82, 45);
```

red	green	blue	
255	0	0	red
0	255	0	green
0	0	255	blue
0	0	0	black
100	100	100	dark gray
255	255	255	white
255	255	0	yellow
255	0	255	magenta
160	82	45	sienna

*Some color values*

to create objects whose values represent the colors red, white, and sienna, respectively. We have been using colors in StdDraw since Section 1.5, but have been limited to a set of predefined colors such as StdDraw.BLACK, StdDraw.RED, StdDraw.BLUE, and StdDraw.GRAY. Now we have millions of colors from which to choose. [Spectrum.java](#) is a demonstration that shows you all of the colors, using StdDraw.

The API for Color contains several constructors and over twenty methods; we briefly summarize the ones that we will use:

---

```
public class java.awt.Color (Java color data type)
    Color(int r, int g, int b)
    int getRed()          red intensity
    int getGreen()        green intensity
    int getBlue()         blue intensity
    Color brighter()     brighter version of this color
    Color darker()       darker version of this color
    String toString()    string representation of this color
    boolean equals(Color c) is this color's value the same as c's?
```

---

*Albers squares.* [AlbersSquares.java](#) displays the two colors entered in RGB representation on the command line in the familiar format developed in the 1960s by the color theorist Josef Albers that revolutionized the way that people think about color.

- *Luminance.* The quality of the images on modern displays such as LCD monitors, plasma TVs, and cell-phone screens depends on an understanding of a color property known as *monochrome luminance*, or effective brightness. A standard formula for luminance is derived from the eye's sensitivity to red, green, and blue. It is a linear combination of the three intensities: if a color's red, green and blue values are  $r$ ,  $g$ , and  $b$ , respectively then its luminance is defined by the equation

$$Y = 0.299 r + 0.587g + 0.114b.$$

- *Grayscale.* The RGB system has the property that when all three color intensities are the same, the resulting color is on a grayscale that ranges from black (all 0s) to white (all 255s). To print a color photograph in a black-and-white newspaper (or a book) we need a static method to convert from color to grayscale. Luminance provides a simple solution: to convert a color to grayscale, simply replace the color by a new color whose red, green, and blue values equal its luminance!
- *Color compatibility.* The luminance value is also crucial in determining whether two colors are compatible, in the sense that printing text in one of the colors on a background in the other color will be readable. A widely used rule of thumb is that the difference between the luminance of the foreground and background colors should be at least 128. For example, black text on white background has a luminance difference of 255, but black text on a blue background has a luminance difference of only 29. This rule is important in the design of advertising, road signs, websites, and many other applications.

[Luminance.java](#) is a static method library that we can use to convert to grayscale and test whether two colors are compatible. The methods in [Luminance](#) illustrate the utility of using data types to organize information. Using the [Color](#) reference type and passing objects as arguments and return values of functions makes these implementations substantially simpler than the alternative of having to pass around the three intensity values. Returning multiple values from a function is particularly problematic.

**Image processing.** You are familiar with the concept of a photograph. The history of photography is a history of technological development. During the last century, photography was based on chemical processes, but its future is now based in computation.

*Digital images.* We have been using standard draw to plot geometric objects (points, lines, circles, squares) in a window on the computer screen. The basic abstraction for computer displays is the same one that is used for digital photographs and is very simple: A *digital image* is a rectangular grid of pixels (picture elements), where the color of each pixel is individually defined. Digital images are sometimes referred to as *raster* or *bitmapped* images.

Our class [Picture.java](#) is a data type for digital images. The set of values is a two-dimensional array of [Color](#) values, and the operations are what you might expect: create an image (either a blank one with a given width and height or one initialized from a picture file), set the value of a pixel to a given color, return the color of a given pixel, return the width or the height, show the image in a window on your computer screen and save the image to a file, as specified in the following API summary:

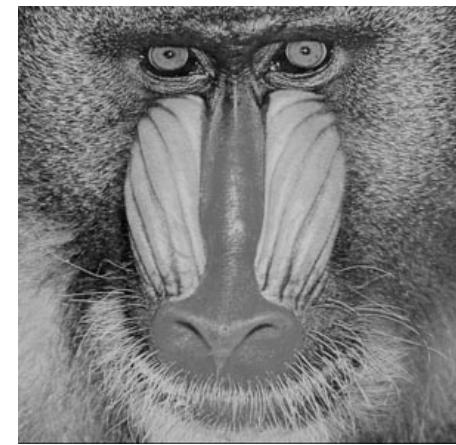
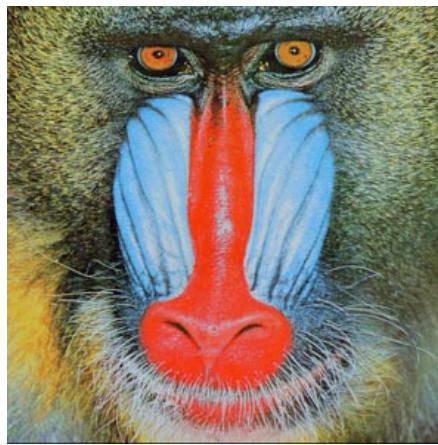
---

```
public class Picture (our data type for image processing)
    Picture(String s)          create a picture from a file
    Picture(int w, int h)      create a blank w-by-h picture
    int width()                return the width of the picture
    int height()               return the height of the picture
    Color get(int i, int j)    return the color of pixel (i, j)
    void set(int i, int j, Color c) set the color of pixel (i, j) to c
    void show()                display the image in a window
    void save(String s)        save the image to a file
```

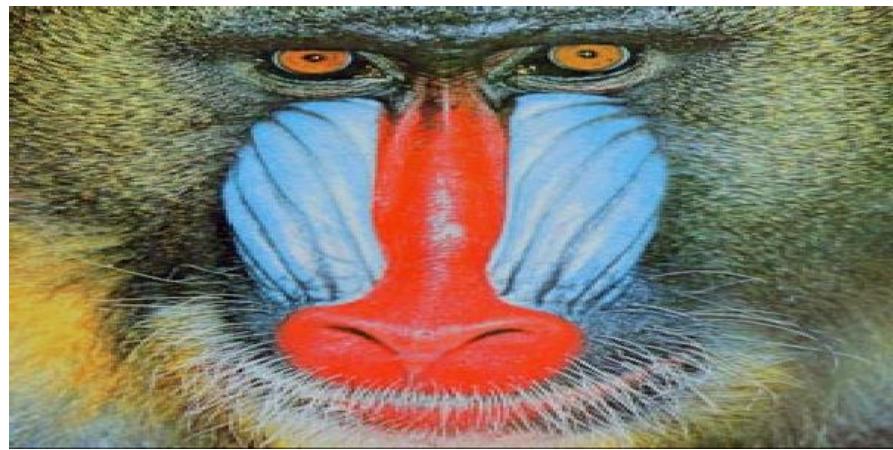
---

These methods, coupled with Java's [Color](#) data type, open the door to image processing for you. By convention,  $(0, 0)$  is the upper leftmost pixel, so the image is laid out in the customary manner for arrays (by contrast, the convention for [StdDraw](#) is to have the point  $(0,0)$  at the lower left corner, so that drawings are oriented in customary manner for Cartesian coordinates).

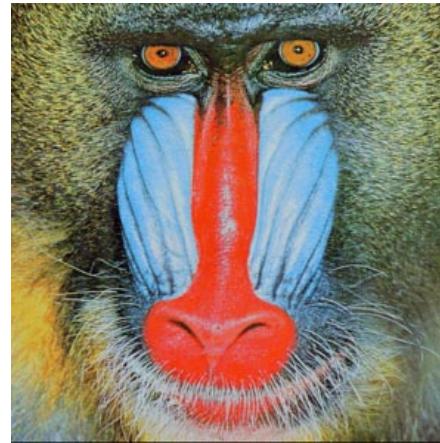
- *Grayscale.* [Grayscale.java](#) converts an image from color to grayscale. This task is a prototypical image-processing task: for each pixel in the source image, we have a pixel in the target image, with a different color.



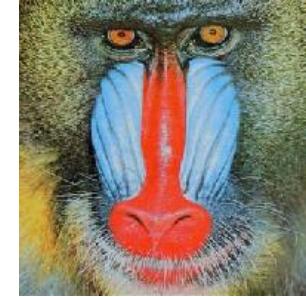
- *Scale*. One of the most common image-processing tasks is to make an image smaller or larger. [Scale.java](#) takes a filename and the width and height of the target image as command-line arguments, and rescale the image to the specified size.



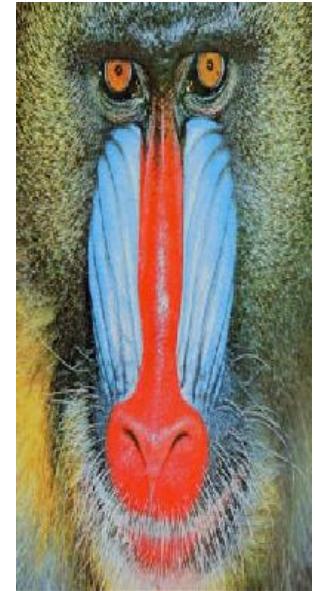
600-by-300



298-by-298

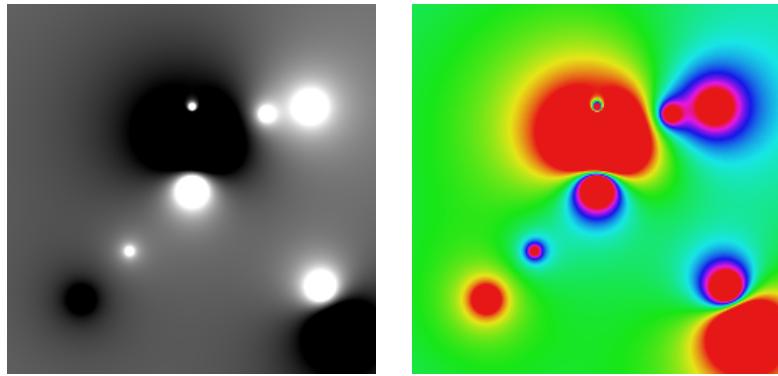


200-by-200



200-by-400

- *Fade effect.* Program [Fade.java](#) takes three command line inputs. The first specifies the number of frames. The next two inputs are the names of two image files (of the same dimensions). The program displays a sequence of images, fading from the first picture to the second one by taking convex combinations of the color values. A *convex combination* of two RGB colors  $(r_1, g_1, b_1)$  and  $(r_2, g_2, b_2)$  is  $(\beta r_1 + (1-\beta)r_2, \beta g_1 + (1-\beta)g_2, \beta b_1 + (1-\beta)b_2)$  where  $\beta$  is a real number between 0 and 1.
- *Electric potential visualization.* Image processing is also helpful in scientific visualization. Program [Potential.java](#) visualizes the potential values created by a set of charges. It relies on the data file [charges.txt](#).



**Strings.** We've been using strings and string concatenation since our very first Java program. Now we will explore many additional operations built in to Java's `String` data type that open up the world of text processing. Before using them, we must know their calling conventions. The *Application Programming Interface* (API) describes the set of operations associated with a data type and how to invoke them. You can find formal descriptions in Sun's [online documentation of the String class](#). The table below summarizes several useful string processing methods and gives brief examples to illustrate their usage. As with arrays, the characters of a string are indexed starting at 0.

Operation	Description	Invoking string <code>s</code>	Return value
<code>s.length()</code>	return length of <code>s</code>	Hello	5
<code>s.charAt(1)</code>	return character of <code>s</code> with index 1	Hello	e
<code>s.substring(1, 4)</code>	return substring from 1 (inclusive) to 4 (exclusive)	Hello	ell
<code>s.substring(1)</code>	return substring starting at index 1	Hello	ello
<code>s.toUpperCase()</code>	return upper case version of <code>s</code>	Hello	HELLO
<code>s.toLowerCase()</code>	return lower case version of <code>s</code>	Hello	hello
<code>s.startsWith("http: ")</code>	does <code>s</code> start with <code>http: ?</code>	<code>http://www.cnn.com</code>	true
<code>s.endsWith(".com")</code>	does <code>s</code> end with <code>.com?</code>	<code>http://www.cnn.com</code>	true
<code>s.indexOf(".java")</code>	return index of first occurrence of <code>.java</code> in <code>s</code> (-1 if no occurrence)	<code>Hello.java.html</code>	5
<code>s.indexOf(".java", 6)</code>	return index of first occurrence of <code>.java</code> in <code>s</code> , starting at index 6	<code>Hello.java.html</code>	-1
<code>s.lastIndexOf(".")</code>	return index of last occurrence of <code>.</code> in <code>s</code>	<code>Hello.java.html</code>	10
<code>s.trim()</code>	return <code>s</code> with leading and trailing whitespace removed	<code>" Hello there "</code>	<code>"Hello there"</code>
<code>s.replace(",", ".")</code>	return <code>s</code> with all occurrences of <code>,</code> replace by <code>.</code>	13,125,555	13.125.555
<code>s.compareTo("abc")</code>	compare <code>s</code> to <code>abc</code> lexicographically	<code>"abc"</code>	0

**Genomics.** Program [GeneFind.java](#) identifies possible genes in a genome.

**Input.** In Section XYZ we learned how to read numerical and text input from the terminal using `StdIn.java`. However, this supported only one input stream (standard input). Sometimes our programs need to read data from several input sources (standard input, files, web sites). Program [In.java](#) is a convenient class to do exactly this.

```
public class In (our data type for input streams)
    In()           create an input stream from standard input
    In(String name)  create an input stream from a file or website
    boolean isEmpty() true if no more input, false otherwise
    int readInt()    read a value of type int
    double readDouble() read a value of type double
    .
    .
    .
    All operations supported by StdIn are also supported for In objects
```

**Screen scraping.** Now we illustrate a nice combination of using the built-in `String` library with our `In` library.

The goal is to query a web page, extract some information, and report back the results. This process is known as *screen scraping*. Program [StockQuote.java](#) takes the symbol of New York Stock Exchange stock and prints out its current trading price. To report the stock price of Google (NYSE symbol = goog), it reads the Web page <http://finance.yahoo.com/q?s=goog>. Then, it identifies the relevant information using `indexOf` and `substring`. The relevant information is enclosed between the tags `<b>` and `</b>` immediately following the text Last Trade.

```
public static void main(String[] args) {
    String name = "http://finance.yahoo.com/q?s=" + args[0];
    In in = new In(name);
    String input = in.readAll();
    int p      = input.indexOf("Last Trade:", 0);
    int from   = input.indexOf("<b>", p);
    int to     = input.indexOf("</b>", from);
    String price = input.substring(from + 3, to);
    System.out.println(price);
}
```

The program heavily depends on the web page format of Yahoo. If Yahoo changes their web page format, we would need to change our program. Nevertheless, this is likely more convenient than maintaining the data ourselves.

**Output.** We are also interested in writing output to files or the network instead of just standard output. Program [Out.java](#) provides a mechanism for writing data to various output streams. Using `Out`, writing to a file is almost as easy as writing to standard output.

```
public class Out (our data type for output streams)
    Out()           create an output stream to standard output
    Out(String name) create an output stream to a file
    void print(String s) append s to the output stream
    void println(String s) append s and a newline to the output stream
    :
All operations supported by StdOut are also supported for Out objects
```

Program [Cat.java](#) takes a sequence of strings as command line inputs (names of text files), and concatenates them, printing the results to standard output.

Program [Split.java](#) This program uses multiple output streams to split a CSV file into separate files, one for each comma-delimited field.

**Primitive types vs. reference types.** Java has two different categories of data types: *primitive types* (also known as *value types*) and *reference types*. We're already familiar with many of the eight primitive types in Java: `boolean`, `char`, `byte`, `short`, `int`, `long`, `float`, and `double`. Value types store the integral, floating pointing or `boolean` value, e.g., `17`, `3.14`, or `true`. The primitive types and the associated operations are typically implemented directly in hardware, so they are especially efficient. When we declare a variable of a primitive type, the system allocates enough memory to store a value of that type (e.g., 4 bytes for an `int` and 8 bytes for a `double`). We can initialize or modify the value using the assignment operator. However, when we pass a variable to a function, the system passes a copy of the integral, floating point or `boolean` value itself. This is called *pass-by-value*. One consequence is that if we pass an integer variable `a` to a function, the function cannot change the value that is stored in `a` since we have only passed a copy of the value stored in `a`.

Reference types have very different characteristics. A reference stores the *memory address* of an object. It captures the difference between a thing and its name.

Thing	Name
Web page	<a href="http://www.princeton.edu">www.princeton.edu</a>
Email inbox	<a href="mailto:nobody@princeton.edu">nobody@princeton.edu</a>
Bank account	45-234-23310076
US citizen	166-34-9114
Word of TOY memory	1C
Byte of computer memory	FFBEFB24
Cell phone	(609) 876-5309
House	35 Olden Street

We say that the reference *points* to the object and often draw an arrow from the reference to what it points to. A reference of a given type always points to an object of the correct type or to the special value `null` which indicates that the reference points to nothing. Each reference points to one object, but two or more references can point to the same object. We initialize a reference by using an assignment statement. We can either set it equal to another reference (of the appropriate type) or we can also use the keyword `new` to make it point to a newly created object. Java manipulates objects by reference, but passes them to methods and functions by value. This means that if we pass a variable `p` of type `Picture` to a function, the function *can* change the state of the object referenced by `p`, e.g., change the colors of some of the pixels. It cannot, however, change what object `p` references.

Analogy. Object = house. Reference = paper with street address of house written in pencil. Each piece of paper can have at most one address. We can give piece of paper to a house painter and tell them to paint the house red. Can have multiple pieces of paper with the same address. When the house is painted, both pieces of paper have

the street address of the same red house. It's possible to erase what's on the paper, and write down a new street address. But if you change what's written on your piece of paper, it doesn't change what's written on my piece of paper.

Cash (value) vs. credit card (pointer to value).

In Program xyz, it is important to compare the strings `s1 s2` with `s1.equals(s2)` instead of `(s1 == s2)`. The former is a built-in method that tests whether the two strings are composed of exactly the same sequence of characters. The latter checks whether the two objects reference the same location in memory.

The distinction between primitive and reference types is a tradeoff between efficiency and elegance. There is substantial memory overhead (around 16 bytes) associated with creating each new object. OOP purists would argue that a language should not have any primitive types, only objects and reference types.

**Arrays are objects.** Arrays are treated as objects in Java, except that there is special syntax for indexing into an array using square brackets. Otherwise, an array is just another example of a reference types. When we pass an array to a function, the system passes a copy of the reference, not a copy of the array. This means that the function is free to modify the contents of the array. If the array is huge, there is substantial savings in passing a reference to the array. Give example of pass-by-value....

**Automatic memory management.** One of the most significant features of the Java programming language is its ability to automatically manage memory. Memory management is straightforward with primitive types: allocate a fixed chunk of memory (e.g., 4 bytes for an `int`) when we declare a variable, and release it when the variable goes out of scope. It is important to free the memory whenever possible since your computer only has a finite amount of memory, and it may eventually consume it all. Managing memory for reference types is substantially more challenging. Each time we create an object with the keyword `new`, the system finds a free chunk of memory of the right size, and reserves it for the object. When the object is no longer accessible (e.g., the last reference to it goes out of scope), we want the system to free up the memory and recycle it for use the next time you create an object with `new`. In many languages (including C and C++) the programmer is responsible for marking those objects that it no longer needs. This process is tedious and notoriously error-prone. If the programmer is not diligent, the system may slowly *leak* memory and eventually run out. Many modern languages (including Java) include a *garbage collector* to transfer the burden of memory management from the programmer to the system. The garbage collector periodically identifies chunks of memory that are not in use, and notifies the system to reclaim them. If a chunk of memory has no references to it, then it can be safely garbage-collected since the programmer would have no way of accessing it.

**Particle system physics engine.** Jeff Traer-Bernstein has a [particle system physics engine](#).

## Q + A

Q. What's the difference between `=`, `==`, and `equals`?

A. Variant of same question of 2.2 and 2.3.

Q. Why doesn't `(s == "")` work reliably for checking whether `s` is the empty string?

A. It compares whether the two strings refer to the same memory location. The two strings can refer to different memory locations which each represent the string `"`.

Q. How can I check whether a string `s` is the empty string?

A. Use `(s.equals(""))` or `(s.length() == 0)`.

Q. Is there a difference between the empty string and `null`?

A. Yes. The empty string is a string consisting of 0 characters. You can invoke all of the usual string methods, e.g., `length`. On the other hand, `null` is not a string object. You will get a `NullPointerException` if you try to invoke any method on a variable storing `null`.

Q. What's the substring trap?

A. The `String` method call `s.substring(i, j)` returns the substring of `s` starting at index `i` and ending at `j-1` (not at `j` as you might suspect).

Q. Can I apply several string operations at once?

A. Yes, the statement `s.trim().toLowerCase().equals("saturday")` works as expected (e.g., returns `true` if `s` represents the string `" Saturday "`). The reason it works is because (i) each of the methods returns its result as a string and (ii) methods in Java are invoked from left to right.

Q. What special capabilities do strings have over other objects?

A. String concatenation with `+` and assignment using quoted sequences of characters.

Q. How can I change the value of a string object?

A. You can't since strings are immutable in Java. However, you can make a string reference point to a different string by reassigning it.

```
String s = "aaaa";           // s refers to the value "aaaa"
s.replaceAll("a", "b");       // s still refers to the value "aaaa"
String t = s;                // t refers to the value "aaaa"
s = s.replaceAll("a", "b");   // s now refers to the value "bbbb"
                            // t still refers to the value "aaaa"
```

Q. Why use the RGB format for representing colors?

A. RGB is commonly used in television screens, computer monitors and digital cameras. The screens (CRT or LCD) are comprised of thousands or millions of tiny red, green, and blue dots. The device can light each pixel up in various degrees of brightness. It is an *additive model* (the device emits a combination of the colors) as opposed to a *subtractive model* (where the medium reflects the colors, e.g., with dyes or pigments).

Q. When using a linear filter, each pixel becomes a weighted average of its 8 neighbors. What do I do when the pixel has less than 8 neighbors because it is near the border?

A. You could assume the image is toroidal (periodic boundary conditions) and make the left boundary wrap around to the right boundary.

Q. How can I convert from an uppercase letter to an integer between 0 and 25?

A. Recall that characters are 16-bit integers and that the uppercase letters are stored consecutively in ascending order. The expression `c - 'A'` does the job. You can also use `c - '0'` to convert from one of the characters '`0`' through '`9`' to the corresponding integer.

Q. Where can I download some test files for image processing?

A. [USC SIPI](#) contains standard test images (including [Peppers](#) and [Baboon](#)).

Q. What special capabilities do arrays have over other objects?

A. Indexing into the array using square brackets. Declaring an array involves specifying its type with square brackets. Initializing an array involves using either `new` with square braces or list its constituent values within curly braces.

Q. How can I pass an array to a function in such a way that the function cannot change the array?

A. You can't since arrays are mutable. However, you can achieve the same effect by building a wrapper data type and passing that instead. Stay tuned.

Q. I've heard that Java has no "pointers." Is this true?

A. It's true that Java doesn't have an explicit pointer type, but you should view Java references as "safe pointers." Java's implementation of references is opaque so you cannot do pointer arithmetic on them or cast them to numeric types. Java also automatically dereferences pointers as needed.

Q. Is it correct to say that Java passes primitive types *by value* and objects *by reference*?

A. Not quite. See [this explanation](#). For C++ programmers, it's better to think of a Java reference as a "safe pointer." Java references behave like C++ references, except that assignment and `==` work like pointers.

Q. When I declare an object, when do I need to use `new`?

A. Every time you call `new`, you get a new instance of the data type, with its own instance variables. If you don't need a new instance, but rather just want a new variable of the given type to reference an existing instance, don't use `new`. Methods can also return new instances by invoking `new`, e.g., the `plus` method with the `Complex` data type.

## Exercises

1. Write a program [FourChargeClient.java](#) that takes a `double` command-line argument `r`, creates four `Charge` objects that are each distance `r` from the center of the screen (.5, .5), and prints the potential at location (.25, .5) due to the combined four charges. All four charges should have the same unit charge.

2. Write a function that takes as input a string and returns the number of occurrences of the letter `e`.

3. Give a one line Java code fragment to replace all periods in a string with commas. *Answer:* `s = s.replace(".", ",")`.

Don't use `s = s.replaceAll(".", ",")`. The `replaceAll()` method uses regular expressions (See Section 7.2) and `."` string has a special meaning.

4. Replace all tabs with four spaces. *Answer:* `s = s.replace("\t", " ")`.

5. Write a program that takes a command line input string `s`, reads strings from standard input, and prints out the number of times `s` appears. *Hint:* use `don't forget to use equals instead of == with references.`

6. Write a program that reads in the name of a month (3 letter abbreviation) as a command line parameter and prints the number of days in that month in a non leap year.

```
public static void main(String[] args) {
    String[] months = {
        "Jan", "Feb", "Mar", "Apr", "May", "Jun",
        "Jul", "Aug", "Sep", "Oct", "Nov", "Dec"
    };
    int[] days = {
        31, 28, 31, 30, 31, 30,
        31, 31, 30, 31, 30, 31
    };
    String name = args[0];
    for (int i = 0; i < 12; i++)
        if (name.equalsIgnoreCase(months[i]))
            System.out.println(name + " has " + days[i] + " days");
}
```

7. Write a program [Squeeze.java](#) that takes as input a string and removes adjacent spaces, leaving at most one space in-a-row.

8. What does the following code fragment do?

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

```

String s1 = args[0];
String s2 = args[1];
int length1 = s1.length();
int length2 = s2.length();
if (length1 > length2) System.out.println(length1);
else System.out.println(length2);
}

```

Which one or more of the following converts all of the strings in the array `a` to upper case?

```

for (int i = 0; i < a.length; i++) {
    String s = a[i];
    s = s.toUpperCase();
}

for (int i = 0; i < a.length; i++) {
    a[i].toUpperCase();
}

for (int i = 0; i < a.length; i++) {
    a[i] = a[i].toUpperCase();
}

```

*Answer:* only the last one.

- Write a function that takes as input a string and returns the string in reverse order.
- What does the following recursive function return, given an input string `s`?

```

public static String mystery(String s) {
    int N = s.length();
    if (N <= 1) return s;
    String a = s.substring(0, N/2);
    String b = s.substring(N/2, N);
    return mystery(b) + mystery(a);
}

```

- Describe the string that the following function returns, given a positive integer `N`?

```

public static String mystery(int N) {
    String s = "";
    while(N > 0) {
        if (N % 2 == 1) s = s + s + "x";
        else s = s + s;
        N = N / 2;
    }
    return s;
}

```

- Write a function that takes a string `s` and an integer `n` and returns a new string `t` of length exactly `n` that consists of `s` (truncated if its length is greater than `n`) followed by a sequence of '-' characters (if the length of `s` is less than `n`).
- Write a function that takes as input a string and returns `true` if the string is a palindrome, and `false` otherwise. A *palindrome* is a string that reads the same forwards or backwards.
- Write a function that takes as input a string and returns `true` if the string is a Watson-Crick complemented palindrome, and `false` otherwise. A *Watson-Crick complemented palindrome* is a DNA string that is equal to the complement (A-T, C-G) of its reverse.
- Write a function that takes as input a DNA string of A, C, G, and T characters and returns the string in reverse order with all of characters replaced by their complements. For example, if the input is ACGGAT, then return ATCCGT.
- What does the following recursive function return, given two strings `s` and `t` of the same length?

```

public static String mystery(String s, String t) {
    int N = s.length();
    if (N <= 1) return s + t;
    String a = mystery(s.substring(0, N/2), t.substring(0, N/2));
    String b = mystery(s.substring(N/2, N), t.substring(N/2, N));
    return a + b;
}

```

- Write a program that reads in a string and prints out the first character that appears exactly once in the string. Ex: ABCDBADDAB -> C.
- Given a string, create a new string with all the consecutive duplicates removed. Ex: ABBCCCCCBBAB -> ABCBAB.

19. Write a function that takes two string arguments *s* and *t*, and returns the index of the first character in *s* that appears in *ts* (or -1 if no character in *s* appears in *t*).
20. Given a string *s*, determine whether it represents the name of a web page. Assume that all web page names start with `http`:
 

*Solution:* The easiest way is using the `startsWith` method in Java's string library, e.g., if `(s.startsWith("http:"))`.
21. Given a string *s* that represents the name of a web page, break it up into pieces, where each piece is separated by a period, e.g., `http://www.cs.princeton.edu` should be broken up into `www`, `cs`, `princeton`, and `edu`, with the `http://` part removed. Use either the `split` or `indexOf` methods.
22. Given a string *s* that represents the name of a file, write a code fragment to determine its file extension. The *file extension* is the substring following the last period. For example, the file type of `monalisa.jpg` is `jpg`, and the file type of `mona.lisa.png` is `png`.

*Library solution:* this solution is used in `Picture.java` to save an image to the file of the appropriate type.

```
String extension = s.substring(s.lastIndexOf('.') + 1);
```

23. Given a string *s* that represents the name of a file, write a code fragment to determine its directory portion. This is the prefix that ends with the last `/` character (the directory delimiter); if there is no such `/`, then it is the empty string. For example, the directory portion of `/Users/wayne/monalisa.jpg` is `/Users/wayne/`.
24. Given a string *s* that represents the name of a file, write a code fragment to determine its base name (filename minus any directories). For `/Users/wayne/monalisa.jpg`, it is `monalisa.jpg`.
25. What does the following code fragment print out?

```
String string1 = "hello";
String string2 = string1;
string1 = "world";
System.out.println(string2);
```

26. Write a program that reads in text from standard input and prints it back out, removing any lines that consist of only whitespace.
27. Write a program that reads in text from standard input and prints it back out, replacing all single quotation marks with double quotation marks.
28. Write a program `Paste.java` that takes an arbitrary number of command line inputs and concatenates the corresponding lines of each file, and writes the results to standard output. (Typically each line in given file has the same length.) Counterpart of the program `Cat.java`.
29. What does the program `LatinSquare.java` print when *N* = 5?

```
String alphabet = "ABCDEFGHIJKLMNOPQRSTUVWXYZ";
for (int i = 0; i < N; i++) {
    for (int j = 0; j < N; j++) {
        char c = alphabet.charAt((i + j) % N);
        System.out.print(c + " ");
    }
    System.out.println();
}
```

A Latin square of order *N* is an *N*-by-*N* array consisting of *N* different symbols, such that each symbol appears exactly once in each row and column. Latin squares are useful in statistical design and cryptography.

30. What does the following code fragment print?

```
String s = "Hello World";
s.toUpperCase();
s.substring(6, 11);
System.out.println(s);
```

*Answer:* Hello World. The methods `toUpperCase` and `substring` return the resulting strings, but the program ignores these so *s* is never changed. To get it to print `WORLD`, use `s = s.toUpperCase()` and `s = s.substring(6, 11)`.

31. What happens when you execute the following code fragment?

```
String s = null;
int length = s.length();
```

*Answer:* you get a `NullPointerException` since *s* is `null` and you are attempting to dereference it.

32. What are the values of *x* and *y* after the two assignment statements below?

```
int x = '-'-'';
int y = '/' '/'';
```

33. Suppose that `a` and `b` are each integer arrays consisting of 100 million integers. What does the following code do. How long does it take?

```
int[] t = a;
a = b;
b = t;
```

*Answer* It swaps them, but it does so without copying millions of elements.

34. What does the following statement do where `c` is of type `char`?

```
System.out.println((c >= 'a' && c <= 'z') ||
(c >= 'A' && c <= 'Z'));
```

*Answer*: prints `true` if `c` is an uppercase or lowercase letter, and `false` otherwise.

35. Write an expression that tests whether or not a character represents one of the digits '0' through '9' without using any library functions.

```
boolean isDigit = ('0' <= c && c <= '9');
```

36. Write a program `FlipY.java` that reads in an image and flips it vertically.

37. Write a program `WidthChecker.java` that takes a command line parameter `N`, reads text from standard input, and prints to standard output all lines that are longer than `N` characters (including spaces).

38. Write a program `Hex2Decimal.java` that converts from a hexadecimal string (using A-F for the digits 11-15) to decimal.

*Answer*: the following solution uses several string library methods and Horner's method.

```
public static int hex2decimal(String s) {
    String digits = "0123456789ABCDEF";
    s = s.toUpperCase();
    int val = 0;
    for (int i = 0; i < s.length(); i++) {
        char c = s.charAt(i);
        int d = digits.indexOf(c);
        val = 16*val + d;
    }
    return val;
}
```

Alternate solution: `Integer.parseInt(String s, int radix)`. More robust, and works with negative integers.

## Creative Exercises

- Picture dimensions.** Write a program `Dimension.java` that takes the name of an image file as a command line input and prints out its dimension (width-by-height).
- Bounding box.** Write a program `BoundingBox.java` that reads in an image file and output the smallest *bounding box* (rectangle parallel to the x and y axes) that contains all of the non-white pixels. Useful for automatic cropping.
- Anti-aliasing.** *Anti-aliasing* is a method of removing artifacts from representing a smooth curve with a discrete number of pixels. A very crude way of doing this (which also blurs the image) is to convert an  $N$ -by- $N$  grid of pixels into an  $(N-1)$ -by- $(N-1)$  by making each pixel be the average of four cells in the original image as below. Write a program `AntiAlias` that reads in an integer `N`, then an  $N$ -by- $N$  array of integers, and prints out the antialiased version. [Reference](#).
- Thresholding.** Write a program `Threshold.java` that reads in a grayscale version of a black-and-white picture, creates and plots a histogram of 256 grayscale intensities, and determines the threshold value for which pixels are black, and which are white.
- Mirror image.** Read in a  $W$ -by- $H$  picture and produce a  $2W$ -by- $H$  picture which concatenates the original  $W$ -by- $H$  picture with the mirror image of the  $W$ -by- $H$  picture. Repeat by mirror around the y-axis. Or create a  $W$ -by- $H$  picture, but mirror around the center, deleting half the picture.
- Linear filters.** A *box filter* or *mean filter* replaces the color of pixel  $(x, y)$  by the average of its 9 neighboring pixels (including itself). The matrix  $[1 \ 1 \ 1; \ 1 \ 1 \ 1; \ 1 \ 1 \ 1] / 9$  is called the *convolution kernel*. The kernel is the set of pixels to be averaged together. Program `MeanFilter.java` implements a mean filter using the `Picture` data type.
- Blur filter.** Use low-pass 3-by-3 uniform filter  $[1/13 \ 1/13 \ 1/13; \ 1/13 \ 5/13 \ 1/13; \ 1/13, \ 1/13, \ 1/13]$ .

8. **Emboss filter.** Use prewitt masks  $[-1\ 0\ 1; -1\ 1\ 1; -1\ 0\ 1]$  (east) or  $[1\ 0\ -1; 2\ 0\ -2; 1\ 0\ -1]$ ,  $[-1\ -1\ 0; -1\ 1\ 1; 0\ 1\ 1]$  (south-east),
9. **Sharpen filter.** Psychophysical experiments suggest that a photograph with crisper edges is more aesthetically pleasing than exact photographic reproduction. Use a high-pass 3-by-3 filter. Light pixels near dark pixels are made lighter; dark pixels near light pixels are made darker. Laplace kernel. Attempts to capture region where second derivative is zero.  $[-1\ -1\ -1; -1\ 8\ -1; -1\ -1\ -1]$
10. **Oil painting filter.** Set pixel  $(i, j)$  to the color of the most frequent value among pixels with Manhattan distance  $W$  of  $(i, j)$  in the original image.
11. **Reverse string.** Write a recursive function to reverse a string. Do not use any loops. *Hint:* use the String method `substring`.

```
static String reverse(String s) {
    int N = s.length();
    if (N == 0) return "";
    else return reverse(s.substring(1, N)) + s.charAt(0);
}
```

12. **Frequency analysis of English text.** Write a program `LetterFrequency.java` that reads in text from standard input (e.g., [Moby Dick](#)) and calculate the fraction of times each of the 26 lowercase letters appears. Ignore uppercase letters, punctuation, whitespace, etc. in your analysis. Use `CharStdIn.java` from Section 2.4 to read process the text file.
13. **Complemented DNA string.** Write a program to read in a DNA string (A, C, T, G) and print out its complement (substitute A for T, T for A, C for G, and G for C). Hint: use `replace` several times, but be careful.
14. **Print longest word.** Read a list of words from standard input, and print out the longest word. Use the `length` method.
15. **Print longest word(s).** Repeat the previous exercise, but print out all of the longest words if there is a tie, say up to a maximum of 10 words. Use an array of strings to store the current longest words.
16. **Test if two files are equal.** Write a program that takes the name of two text files as command line inputs and checks if their contents are identical.
17. **Parsing command line options.** Unix command line programs typically support *flags* which configure the behavior of a program to produce different output, e.g., "wc -c". Write a program that takes any number of flags from the command line and runs whichever options the user specifies. To check options, use something like `if (s.equals("-v"))`.
18. **Capitalization.** Write a program `Capitalizer.java` that reads in text strings from standard input and modifies each one so that the first letter in each word is uppercase and all other letters are lowercase.
19. **Reverse domain.** Write a program to read in a domain name as a command line input and print the *reverse domain*. For example, the reverse domain of `cs.princeton.edu` is `edu.princeton.cs`. This is useful for web log analysis.
20. **Railfence transposition cipher.** Write a program `RailFenceEncoder.java` that reads in text from standard input and prints out the characters in the odd positions, followed by the even positions. For example, if the original message is "Attack at Dawn", then you should print out "Atc tDwtaka an". This is a crude form of cryptography.
21. **Railfence transposition cipher.** Write a program `RailFenceDecoder.java` that reads in a message encoded using the railfence transposition cipher and prints out the original message by reversing the encryption process.
22. **Scytale cipher.** The *scytale cipher* is one of the first cryptographic devices used for military purposes. (See The Code Book, p. 8 for a nice picture.) It was used by the Spartans in the fifth century BCE. To scramble the text, you print out every  $k$ th character starting at the beginning, then every  $k$ th character starting at the second character, and so forth. Write a pair of programs `ScytaleEncoder.java` and `ScytaleDecoder.java` that implement this encryption scheme.
23. **Kama-sutra cipher.** The *Kama-sutra*, written in the fourth century BCE by Vatsyayana, outlines 64 arts that women should study. Number 45 outlines a method to help women conceal their secret affairs. Each letter in the alphabet is paired up with another one, as in the table below:

A	B	C	E	F	G	H	K	L	M	N	P	R
Q	D	Z	U	J	I	X	Y	W	S	O	V	T

Then a message is encoded by replacing each letter with its pair. For example, the message "MEET AT ELEVEN" is encoded as "SUUR QR UWUPUO". This is one of the earliest known substitution ciphers. Write a program `KamaSutra.java` that scrambles a message using this scheme. Observe that you can unscramble a message by applying the same scheme again.

24. **Password checker.** Write a program that reads in a string from the command line and checks whether it is a "good" password. Here, assume "good" means that it (i) is at least 8 characters long, (ii) contains at least one digit 0-9, (iii) contains at least one upper case letter, (iv) contains at least one lower case letter, and (v) contains at least one non-alphanumeric character.
25. **Subsequence.** Given two strings  $s$  and  $t$ , write a program `Subsequence.java` that determines whether  $s$  is a subsequence of  $t$ . That is, the letters of  $s$  should appear in the same order in  $t$ , but not necessarily contiguously. For example `accag` is a subsequence of `taagcccaaccgg`.
26. **Bible codes.** Some religious zealots believe that the Torah contains hidden phrases that appear by reading every  $k$ th letter, and that such pattern can be used to find the Ark of the Covenant, cure cancer, and predict the future. Results not based on scientific method and results have been [debunked](#) by mathematicians and attributed to illicit data manipulation. Using the same methodology one can find statistically similar patterns in a Hebrew translation of War and Peace.

27. **Word chain checker.** Write a program that reads in a list of words from the command line and prints `true` if they form a *word chain* and `false` otherwise. In a word chain, adjacent words must differ in exactly one letter, e.g., HEAL, HEAD, DEAD, DEED, BEER.
28. **Haiku detector.** Write a program that reads in text from standard input and checks whether it forms a haiku. A haiku consists of three lines containing the correct number of syllables (5, 7, and 5, respectively). For the purpose of this problem, define a syllable to be any contiguous sequence of consecutive vowels (a, e, i, o, u, or y). According to this rule, *haiku* has two syllables and *purpose* has three syllables. Of course, the second example is wrong since the e in *purpose* is silent.
29. **ISBN numbers.** Write a program to check whether an ISBN number is valid. Recall check digit. An ISBN number can also have hyphens inserted at arbitrary places.
30. **Longest common prefix.** Write a function that takes two input string *s* and *t*, and returns the longest common prefix of both strings. For example, if *s* = ACCTGAACCCCCC and *t* = ACCTAGGACCCCC, then the longest common prefix is ACCT. Be careful if *s* and *t* start with different letters, or if one is a prefix of the other.
31. **Complemented palindrome detector.** In DNA sequence analysis, a *complemented palindrome* is a string equal to its reverse complement. Adenine (A) and Thymine (T) are complements, as are Cytosine (C) and Guanine (G). For example, ACGGT is a complement palindrome. Such sequences act as transcription-binding sites and are associated with gene amplification and genetic instability. Given a text input of *N* characters, find the longest complemented palindrome that is a substring of the text. For example, if the text is GACACGGTTTA then the longest complemented palindrome is ACGGT. *Hint:* consider each letter as the center of a possible palindrome of odd length, then consider each pair of letters as the center of a possible palindrome of even length.
32. **DNA validation.** Write a function that takes as input a string and returns `true` if it consists entirely of A, C, G, and T's, and `false` otherwise.
33. **Highest density C+G region.** Given a DNA string *s* of A, C, T, G and a parameter *L*, find a substring of *s* that contains the highest ratio of C + G characters among all substrings that have at least *L* characters.
34. **DNA to RNA.** Write a function that takes a DNA string (A, C, G, T) and returns the corresponding RNA string (A, C, G, U).
35. **DNA complement.** Write a function that takes as input a DNA string (A, C, G, T) and returns the complementary base pairs (T, G, C, A). DNA is typically found in a *double helix* structure. The two complementary DNA strands are joined in a spiral structure.
36. **Circular shifts.** Application: computational biology. A string *s* is a *circular shift* of a string *t* if its characters can be circularly shifted to the right by some number of positions, e.g., actgacg is a circular shift of tgacgac, and vice versa. Write a program that checks whether one string *s* is a circular shift of another *t*. *Hint:* it's a one liner with `indexOf` and string concatenation.
37. **Substring of a circular shifts.** Write a function that takes two strings *s* and *t*, and returns `true` if *s* is a substring of a circular string *t*, and `false` otherwise. For example *gactt* is a substring of the circular string *tgacgact*.
38. **DNA to Protein.** A protein is a large molecule (polymer) consisting of a sequence of amino acids (monomers). Some examples of proteins are: hemoglobin, hormones, antibodies, and ferritin. There are 20 different amino acids that occur in nature. Each amino acid is specified by three DNA base pairs (A, C, G, or T). Write a program to read in a protein (specified by its base pairs) and converts it into a sequence of amino acids. Use the following table. For example, the amino acid Isoleucine (I) is encode by ATA, ATC, or ATT.

Rosetta stone of life.

TTT Phe	TCT Ser	TAT Tyr	TGT Cys
TTC Phe	TCC Ser	TAC Tyr	TGC Cys
TTA Leu	TCA Ser	TAA ter	TGA ter
TTG Leu	TCG Ser	TAG ter	TGG Trp
CTT Leu	CCT Pro	CAT His	CGT Arg
CTC Leu	CCC Pro	CAC His	CGC Arg
CTA Leu	CCA Pro	CAA Gln	CGA Arg
CTG Leu	CCG Pro	CAG Gln	CGG Arg
ATT Ile	ACT Thr	AAT Asn	AGT Ser
ATC Ile	ACC Thr	AAC Asn	AGC Ser
ATA Ile	ACA Thr	AAA Lys	AGA Arg
ATG Met	ACG Thr	AAG Lys	AGG Arg
GTT Val	GCT Ala	GAT Asp	GGT Gly
GTC Val	GCC Ala	GAC Asp	GGC Gly
GTA Val	GCA Ala	GAA Glu	GGA Gly
GTG Val	GCG Ala	GAG Glu	GGG Gly

Amino acid	Abbrev	Abbrev	Amino acid	Abbrev	Abbrev
Alanine	ala	A	Lleucine	leu	L
Arginine	arg	R	Lysine	lys	K
Asparagine	asn	N	Methionine	met	M
Aspartic Acid	asp	D	Phenylalanine	phe	F
Cysteine	cys	C	Proline	pro	P
Glutamic Acid	glu	E	Serine	ser	S

Glutamine	gln	Q	Threonine	thr	T
Glycine	gly	G	Tryptophan	trp	W
Histidine	his	H	Tyrosine	tyr	Y
Isoleucine	ile	I	Valine	val	V

39. **Counter.** Write a program that reads in a decimal string from the command line (e.g., 56789) and starts counting from that number (e.g., 56790, 56791, 56792). Do not assume that the input is a 32 or 64 bit integer, but rather an arbitrary precision integer. Implement the integer using a `String` (not an array).
40. **Arbitrary precision integer arithmetic.** Write a program that takes two decimal strings as inputs, and prints out their sum. Use a string to represent the integer.
41. **Boggle.** The game of Boggle is played on a 4-by-4 grid of characters. There are 16 dice, each with 6 letters on the them. Create a 4-by-4 grid, where each die appears in one of the cells at random, and each die displays one of the 6 characters at random.

```
FORIXB MOQABJ GURILW SETUPL CMPDAE ACITAO SLCRAE ROMASH
NODESW HEFIYE ONUDTK TEVIGN ANEDVZ PINESH ABILYT GKYLEU
```

42. **Generating cryptograms.** A *cryptogram* is obtained by scrambling English text by replacing each letter with another letter. Write a program to generate a random permutation of the 26 letters and use this to map letters. Give example: Don't scramble punctuation or whitespace.
43. **Scrabble.** Write a program to determine the longest legal Scrabble word that can be played? To be legal, the word must be in [The Official Tournament and Club Wordlist](#) (TWL98), which consists of all 168,083 words between 2 and 15 letters in TWL98. The number of tiles representing each letter are given in the table below. In addition, there are two *blanks* which can be used to represent any letter.

```
a b c d e f g h i j k l m n o p q r s t u v w x y z -
9 2 2 4 12 2 3 2 9 1 1 4 2 6 8 2 1 6 4 6 4 2 2 1 2 1 2
```

44. **Soundex.** The [soundex algorithm](#) is a method of encoding last names based on the way it sounds rather than the way it is spelled. Names that sound the same (e.g., SMITH and SMYTH) would have the same soundex encoding. The soundex algorithm was originally invented to simplify census taking. It is also used by genealogists to cope with names with alternate spellings and by airline receptionists to avoid embarrassment when later trying to pronounce a customer's name.

Write a program `Soundex.java` that reads in two lowercase strings as parameters, computes their soundex, and determines if they are equivalent. The algorithm works as follows:

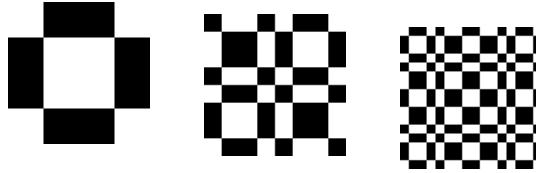
- Keep the first letter of the string, but remove all vowels and the letters 'h', 'w', and 'y'.
- Assign digits to the remaining letter using the following rules:

```
1: B, F, P, V
2: C, G, J, K, Q, S, X, Z
3: D, T
4: L
5: M, N
6: R
```

- If two or more consecutive digits are the same, delete all of the duplicates.
  - Convert the string to four characters: the first character is the first letter of the original string, the remaining three characters are the first three digits in the string. Pad the string with trailing 0's if there are not enough digits; truncate it if there are too many digits.
45. **Longest word.** Given a dictionary of words and a starting word *s*, find the longest word that can be formed, starting at *s*, and inserting one letter at a time such that each intermediate word is also in the dictionary. For example, if the starting word is *cal*, then the following is a sequence of valid words *coal*, *coral*, *choral*, *chorale*. [Reference](#).
46. **Phone words.** Write a program `PhoneWords.java` that takes a 7 digit string of digits as a command line input, reads in a list of words from standard input (e.g., the dictionary), and prints out all 7-letter words (or 3-letter words followed by 4-letter words) in the dictionary that can be formed using the standard phone rules, e.g., 266-7883 corresponds to `compute`.

```
0: No corresponding letters
1: No corresponding letters
2: A B C
3: D E F
4: G H I
5: J K L
6: M N O
7: P Q R S
8: T U V
9: W X Y Z
```

47. **Rot13**. Rot13 is a very simple encryption scheme used on some Internet newsgroups to conceal potentially offensive postings. It works by cyclically shifting each lowercase or uppercase letter 13 positions. So, the letter 'a' is replaced by 'n' and the letter 'n' is replaced by 'a'. For example, the string "Encryption" is encoded as "Rapelcgvba." Write a program [ROT13.java](#) that reads in a String as a command line parameter and encodes it using Rot13.
48. **Longest Rot13 word**. Write a program that reads in a dictionary of words into an array and determines the longest pair of words such that each is the Rot13 of the other, e.g., bumpily and unfiber.
49. **Thue-Morse weave**. Recall the [Thue-Morse sequence](#) from Exercises in Section 2.3. Write a program [ThueMorse.java](#) that reads in a command line input  $N$  and plots the  $N$ -by- $N$  Thue-Morse weave in turtle graphics. Plot cell  $(i, j)$  black if the  $i$ th and  $j$ th bits in the Thue-Morse string are different. Below are the Thue-Morse patterns for  $N = 4, 8$ , and  $16$ .



Because of the mesmerizing non-regularity, for large  $N$ , your eyes may have a hard time staying focused.

50. **Repetition words**. Write a program [Repetition.java](#) to read in a list of dictionary words and print out all words for which each letter appears exactly twice, e.g., intestines, antiperspirantes, appeases, arraigning, hotshots, arraigning, teammate, and so forth.
51. **Text twist**. Write a program [TextTwist.java](#) that reads in a word from the command line and a dictionary of words from standard input, and prints out all words of at least four letters that can be formed by rearranging a subset of the letters in the input word. This forms the core of the Yahoo game [Text Twist](#). *Hint:* create a profile of the input word by counting the number of times each of the 26 letters appears. Then, for each dictionary word, create a similar profile and check if each letter appears at least as many times in the input word as in the dictionary word.
52. **Word frequencies**. Write a program (or several programs and use piping) that reads in a text file and prints out a list of the words in decreasing order of frequency. Consider breaking it up into 5 pieces and use piping: read in text and print the words one per line in lowercase, sort to bring identical words together, remove duplicates and print count, sort by count.
53. **VIN numbers**. A [VIN number](#) is a 17-character string that uniquely identifies a motor vehicle. It also encodes the manufacturer and attributes of the vehicle. To guard against accidentally entering an incorrect VIN number, the VIN number incorporates a check digit (the 9th character). Each letter and number is assigned a value between 0 and 9. The check digit is chosen so to be the weighted sum of the values mod 11, using the symbol X if the remainder is 10.

A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z
1	2	3	4	5	6	7	8	-	1	2	3	4	5	-	7	-	9	2	3	4	5	6	7	8	9
1st	2nd	3rd	4th	5th	6th	7th	8th	9th	10	11	12	13	14	15	16	17									
8	7	6	5	4	3	2	10	0	9	8	7	6	5	4	3	2									

For example the check digit of the partial VIN number 1FA-CP45E-?-LF192944 is X because the weighted sum is 373 and 373 mod 11 is 10.

1	F	A	C	P	4	5	E	X	L	F	1	9	2	9	4	4
1	6	1	3	7	4	5	5	-	3	6	1	9	2	9	4	4
8	7	6	5	4	3	2	10	-	9	8	7	6	5	4	3	2
-----																
8	42	6	15	28	12	10	50	-	27	48	7	54	10	36	12	8

Write a program [VIN.java](#) that takes a command line string and determines whether or not it is a valid VIN number. Allow the input to be entered with upper or lower case, and allow dashes to be inserted. Do thorough error checking, e.g., that the string is the right length, that no illegal characters are used (I, O, Q), etc.

54. **Music CDs**. Screen-scrape [MusicBrainz](#) to identify information about music CDs.
55. **Pig Latin**. Pig Latin is a fun secret language for young children. To convert a word to Pig Latin:
- If it begins with a vowel, append "hay" to the end. At the beginning of a word, treat y as a vowel unless it is followed by a vowel.
  - If it begins with a sequence of consonants, move the consonants to the end, then append "ay". Treat a u following a q as a consonant.

For example, "input" becomes "input-hay", "standard" becomes "andard-stay", "quit" becomes "it-quay". Write a program [PigLatinCoder.java](#) that reads in a sequence of words from standard input and prints them to standard output in Pig Latin. Write a program [PigLatinDecoder.java](#) that reads in a sequence of words encoded in Pig Latin from standard input and prints the original words out in.

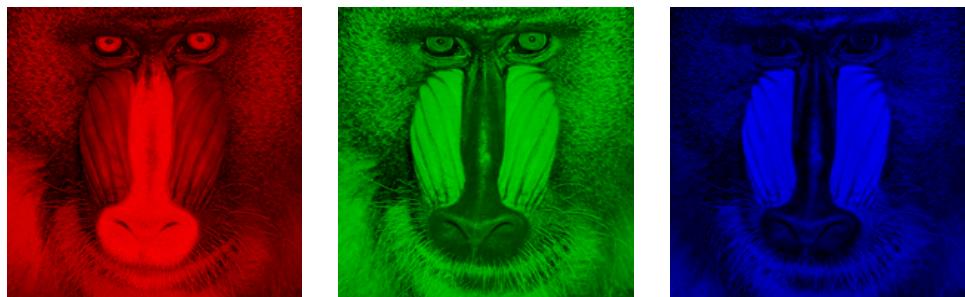
56. **Rotating drum problem**. Applications to pseudo-random number generators, computational biology, coding theory. Consider a rotating drum (draw picture of circle divided into 16 segments, each of one of two types - 0 and 1). We want that any sequence of 4 consecutive segments to uniquely identify the quadrant of the drum. That is, every 4 consecutive segments should represent one of the 16 binary numbers from 0000 to 1111. Is this possible? A *de Bruijn* sequence of order  $n$  is a shortest (circular) string such that every

sequence of  $n$  bits appears as a substring at least once. For example, 0000111101100101 is a de Bruijn sequence of order 4, and all  $2^4$  possible 4-bit sequence (0000, 0001, ..., 1111) occur exactly once. Write a program [DeBruijn.java](#) that reads in a command line parameter  $n$  and prints out an order  $n$  de Bruijn sequence. Algorithm: start with  $n$  0's. Append a 1 if the  $n$ -tuple that would be formed has not already appeared in the sequence; append a 0 otherwise. Hint: use the methods `String.indexOf` and `String.substring`.

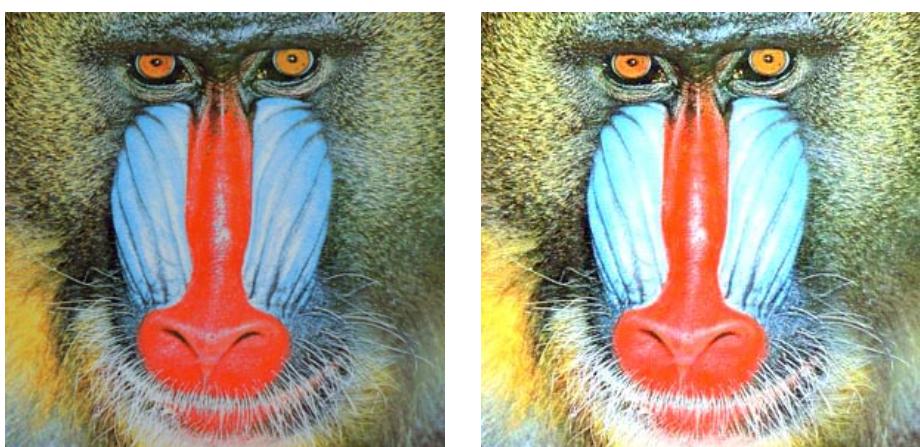
57. **Ehrenfecucht-Mycielski sequence.** The *Ehrenfecucht-Mycielski sequence* in a binary sequence that starts with "010". Given the first  $n$  bits  $b_0, b_1, \dots, b_{n-1}, b_n$  is determined by finding the longest suffix  $b_j, b_{j+1}, \dots, b_{n-1}$  that occurs previously in the sequence (if it occurs multiple times, take the last such occurrence). Then,  $b_n$  is the opposite of the bit that followed the match.  
 $0100110101110001000011110110010100100111010000101101111100$ . Use `substring` and `lastIndexOf`.
58. **Luminance and chrominance.** Decompose a picture into its monochrome luminance  $Y = .299 r + .587 g + .114 b$ , chrominance  $I = .596 r - .274 g - .322 b$ , and chrominance  $Q = .211 r - .523 g + .312 b$ . Plot all 3 images.
59. **Flip horizontally.** Write a program [FlipX.java](#) that takes a command line argument which is the name of a JPG or PNG file, displays it in a window, flips the image horizontally, and displays the resulting image in the window. We illustrate using standard computer graphics test images - [baboon.jpg](#) and [peppers.jpg](#).



60. **Color separation.** Write a program [ColorSeparation.java](#) that takes the name of an image file as a command line input, and creates three images, one that contains only the red components, one for green, and one for blue.



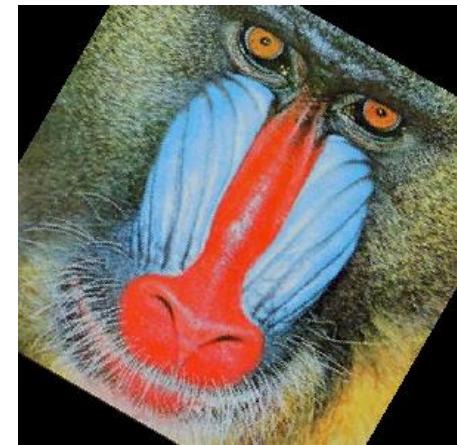
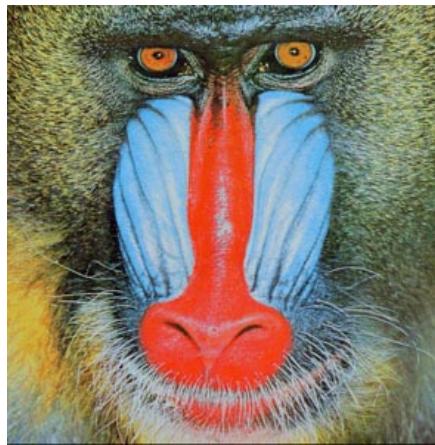
61. **Saturation. saturation.**
62. **Brighten.** Write a program [Brighter.java](#) that takes a command line argument which is the name of a JPG or PNG file, displays it in a window, and display a second version which is a brighter copy. Use the `Color` method `brighter()`, which return a brighter version of the invoking color.



63. **Rotate.** Write a program [Rotation.java](#) that rotates the image 30 degrees counterclockwise. [reference](#). We check if the pre-image is out-of-bounds.

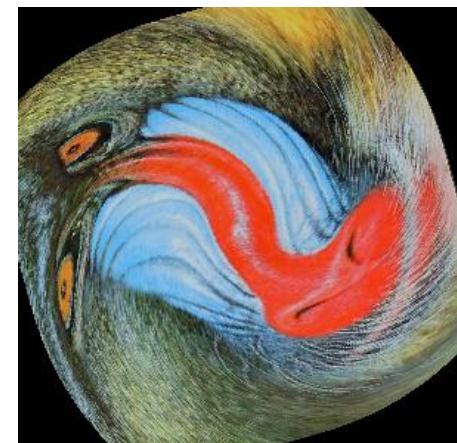
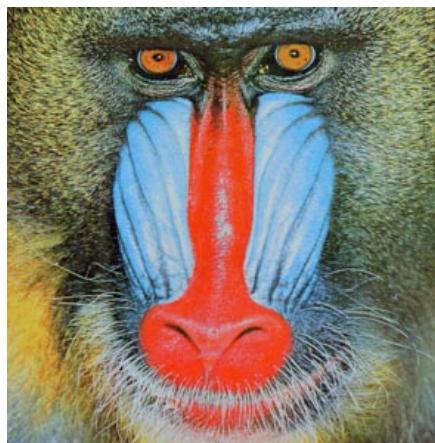
```
theta = pi / 6 radians
ii = -(i - i0)*cos(theta) + (j - j0)*sin(theta) + i0
jj = (i - i0)*sin(theta) + (j - j0)*cos(theta) + j0
```

Here  $(i_0, j_0)$  is the center of the image and  $\theta$  is the angle of rotation (counterclockwise). Important idea = *sampling*.



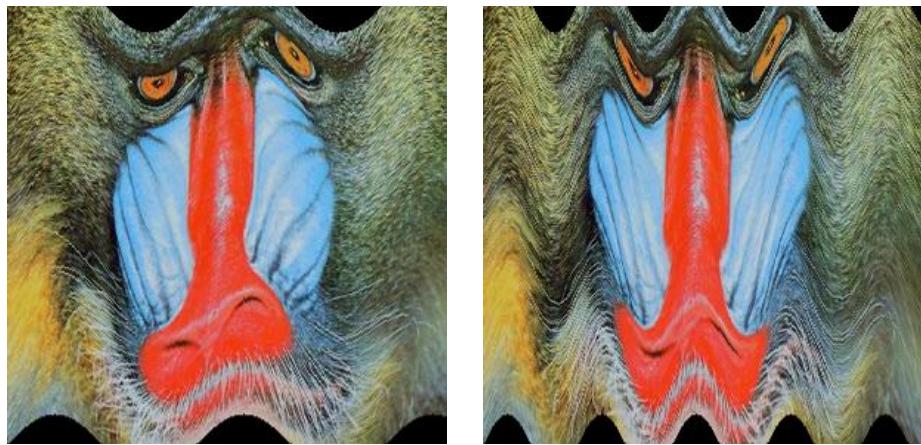
64. **Swirl.** Write a program [Swirl.java](#) that applies a swirl filter. Similar to rotation filter, except that angle changes as a function of distance to the center.

```
r = sqrt((i - i0)^2 + (j - j0)^2)
theta = r * pi / 256
ii = -(i - i0)*cos(theta) + (j - j0)*sin(theta) + i0
jj = (i - i0)*sin(theta) + (j - j0)*cos(theta) + j0
```

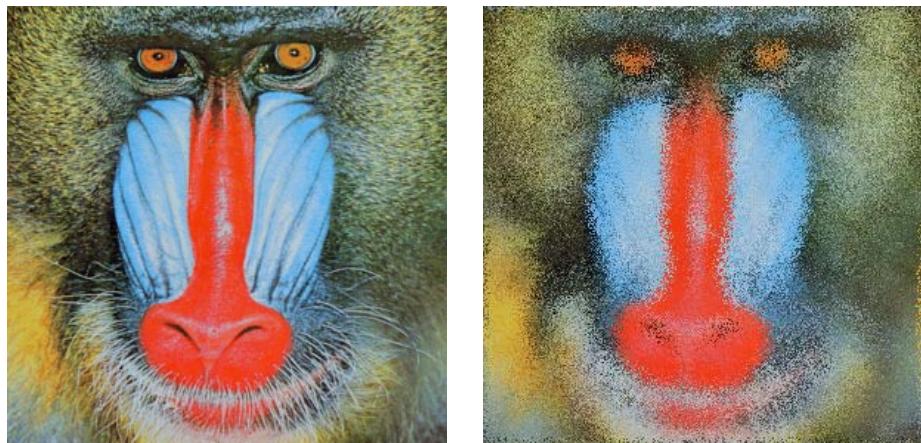


65. **Wave.** Write a program [Wave.java](#) that produces a wave effect.

```
ii = i
jj = j + 20*sin(2*pi*j/128)
```



66. **Glass filter.** Write a program [Glass.java](#) that takes a command line argument which is the name of a JPG or PNG file, displays it in a window, applies a "glass filter", and displays the resulting image in the window. This effect is achieved by setting pixel  $(i, j)$  to be the color of a random neighboring pixel (within Manhattan distance 5). This has the effect of blurring the images as well.



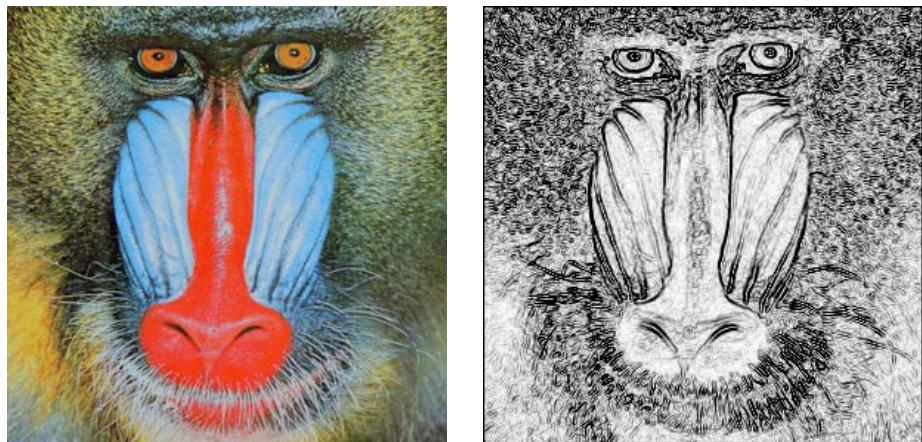
67. **Slide show.** Write a program to display a sequence of images in a slide show (one every two seconds), using a fade effect (or some other transition effect) between pictures.
68. **Tile.** Program [Tile.java](#) takes a command line argument which is the name of a JPG or PNG file, two command line integers  $M$  and  $N$ , and creates an  $M$ -by- $N$  tiling of the picture. Below is a 2-by-3 tiling of the baboon picture.



69. **Edge detection.** Goal: form mathematical model of some feature of the image. To accomplish this, we want to detect edges or lines. An *edge* is an area of a picture with a strong contrast in intensity from one pixel to the next. Edge detection is a fundamental problem in image processing and computer vision. The *Sobel method* is a popular edge detection technique. We assume that the image is grayscale. (If not, we can convert by taking the average of the red, green, and blue intensities.) For each pixel  $(i, j)$  we calculate the *edge strength* by computing two 3-by-3 convolution masks. This involves taking the grayscale values of the nine pixels in the 3-by-3 neighborhood centered on  $(i, j)$ , multiplying them by the corresponding weight in the 3-by-3 mask, and summing up the products.

-1 0 +1	+1 +2 +1
-2 0 +2	0 0 0
-1 0 +1	-1 -2 -1

This produces two values  $G_x$  and  $G_y$ . In the output picture, we color the pixel  $(i, j)$  according to the grayscale value  $255 - \text{Sqrt}(G_x^*G_x + G_y^*G_y)$ . There are various ways to handle the boundary. For simplicity, we ignore this special case and color the boundary pixels black. Program [EdgeDetector.java](#) takes the name of an image as a command line input and applies the Sobel edge detection algorithm to that image.



## Web exercises

1. **Painter's and printer's color triangles.** Create the following two images. The primary hues of the painter's triangle are red, green, and blue; the primary hues of the printer's triangle are magenta, cyan, and yellow.



2. **Entropy.** The *Shannon entropy* measures the information content of an input string and plays a cornerstone role in information theory and data compression. It was proposed by Claude Shannon in 1948, borrowing upon the concept in statistical thermodynamics. Assuming each character  $i$  appears with probability  $p_i$ , the entropy is defined to be  $H = - \sum p_i \log_2 p_i$ , where the contribution is 0 if  $p_i = 0$ . Compute entropy of DNA sequence.

- a. Write a program to read in a ASCII text string from standard input, count the number of times each ASCII character occurs, and compute the entropy, assuming each character appears with the given probabilities.
  - b. Repeat part (a) but use Unicode.
3. **wget.** Write a program [Wget.java](#) that takes the name of a URL as a command-line argument and saves the referenced file using the same filename.
  4. **Capitalize.** Write a program [Capitalize.java](#) that reads in text from standard input and capitalizes each word (make first letter uppercase and make the remaining letters lowercase).
  5. **Shannon's entropy experiment.** Recreate Shannon's experiment on the entropy of the English language by listing a number of letters in a sentence and prompting the user for the next symbol. Shannon concluded that there is approximately 1.1 bits of info per letter in the alphabet.
  6. **Scrambled text.** Some cognitive psychologists believe that people recognize words based on their shape.

to a rscheearch at an Elingsh uinervtisy, it deosn't mtaer in waht oredr the ltteers in a wrod are, the olny iprmoehtn tihng is taht frist and lsat ltteer is at the rghit pclae. The rset can be a toatl mses and you can stil raed it wouthit porbelm. Tihis is bcuseae we do not raed ervey lteter by itslef but the wrod as a wlohe.

Write a program that reads in text from standard input and prints the text back out, but shuffles the internal letters in each word. Write and use a function `scramble()` that takes as input a string and returns another string with the internal letters in random order. Use the shuffling algorithm in [Shuffle.java](#) for the shuffling part.

7. **Date format conversion.** Write a program to read in a date of the form 2003-05-25 and convert it to 5/25/03.
8. [interesting English words](#)
9. **Two-stroke apparent motion.** Create the optical illusion of [two-stroke apparent motion](#) or [four-stroke](#)
10. **De Valois' checkerboard.** Create the optical illusion of [De Valois' checkerboard](#) or one of the other optical illusions from the [Shapiro Perception Lab](#).

**More stuff.** Here's a FAQ for [manipulating pixels in Java](#). Here's a list of more [possible transformations for monochrome images \[pdf\]](#).

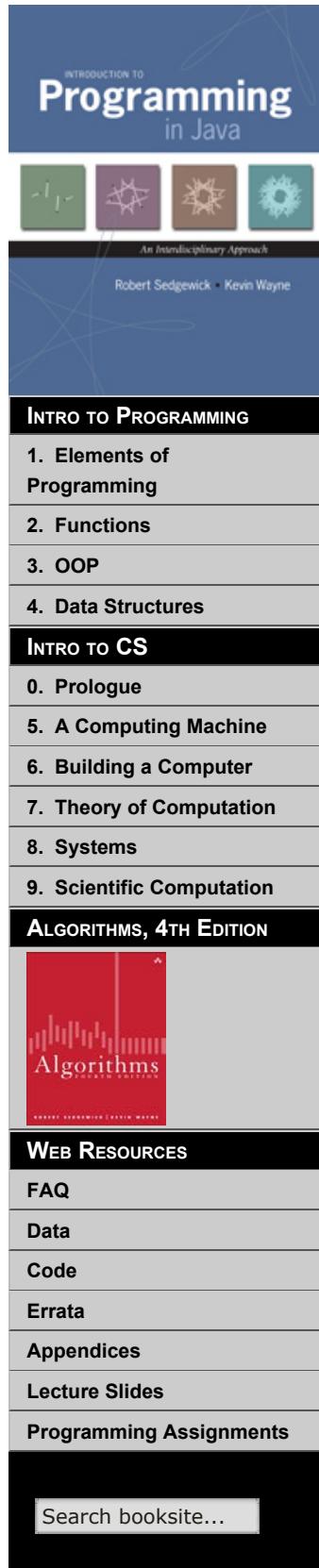
- brighten: brighten image by multiplying by a constant.

- invert: create a negative by inverting each pixel.
- rotate
- addGreen: add more green
- removeRed: remove red-eye?
- Floyd-Steinberg dithering
- cross-dissolving (blend images via convex combinations of pixels)
- contrast: compute mean luminance  $L = 0.30r + 0.59g + 0.11b$  for each pixel, and scale deviation from  $L$  for each pixel component.
- color quantization
- swirl (iterate over destination pixels and compute inverse map to determine color, possibly by taking nearest point or bilinear)
- scan-line flood fill
- draw line, curve, ellipse using Bresenham / midpoint method
- see here for [more ideas](#)

[Miasma animation using MemoryImageSource](#).

*Last modified on August 05, 2011.*

Copyright © 2002–2012 [Robert Sedgewick](#) and [Kevin Wayne](#). All rights reserved.



## 3.2 CREATING DATA TYPES

This section under major construction.

In principle, we could write all of our programs using only the eight built-in primitive types, but, it is much more convenient to write programs at a higher level of abstraction. Accordingly, a variety of data types are built into the Java language and libraries. Still, we certainly cannot expect Java to contain every conceivable data type that we might ever wish to use, so we need to be able to define our own.

**Basic elements of a data type.** To illustrate the process of implementing a data type in a Java class, we discuss each component for the `Charge.java` data type of Section 3.1. Every data-type implementation that you will develop has the same basic ingredients as this simple example.

- **API.** The applications program interface is the contract with all clients and therefore the starting point for any implementation:

```
public class Charge (PROGRAM 3.2.1)
    Charge(double x0, double y0, double q0)
    double potentialAt(double x, double y)  potential at (x, y) due to charge
    String toString()                      string representation
```

To implement `Charge`, we need to define the data type values, then implement the constructor that creates objects having specified values, and then implement two methods that can manipulate those values.

- **Class.** The data-type implementation is a Java `class`. As usual, we put the code for a data type in a file with the same name as the class, followed by the `.java` extension. We have been implementing Java classes, but the classes that we have been implementing do not have the key features of data types: *instance variables*, *constructors*, and *instance methods*. Instance variables are similar to the variables that we have been using in our program; constructors and methods are similar to functions, but their effect is quite different. Each of these building blocks is also qualified by an access modifier. We next consider these four concepts, with examples.
- **Access modifiers.** We designate every instance variable and method within a class (and the class itself) as either `public` (this entity is available to clients) or `private` (this entity is not visible to clients). Our convention is to use `public` for the constructors and methods in the API (since we are promising to provide them to clients) and `private` for everything else.
- **Instance variables.** To write code for the methods that manipulate data type values, the first thing that we need is to declare variables that we can use to refer to the values in code. These variables can be any type of data. We declare the types and names of these instance variables in the same way as we declare local variables: for `Charge`, we use three `double` values, two to describe the charge's position in the plane and one to describe the amount of charge. There is a critical distinction between instance variables and the local variables within a static method or a block that you are accustomed to using: there is just *one* value corresponding to each local variable name, but there are *numerous* values corresponding to each instance variable (one for each object that is an instance of the data type).

```
public class Charge()
{
    instance variable declarations → private double rx, ry;
    → private double q;
    :
}
```

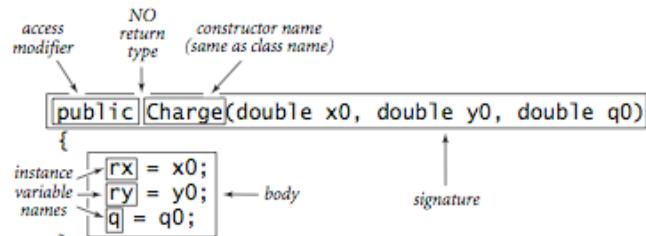
*Instance variables*

- **Constructors.** A constructor creates an object and provides a reference to that object. Java automatically invokes a constructor when a client program uses the keyword `new`. Java does most of the work: our code only needs to initialize the instance variables to meaningful values. Constructors always share the same

name as the class. To the client, the combination of `new` followed by a constructor name (with argument values enclosed within parentheses) is the same as a function call that returns a value of the corresponding type. A constructor signature has no return type because constructors always return a reference to an object of its data type. Each time that a client invokes a constructor Java automatically

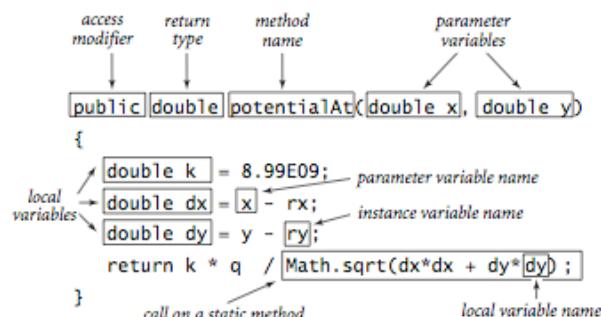
- allocates memory space for the object
- invokes the constructor code to initialize the data type values
- returns a reference to the object

The constructor in `Charge` is typical: it initializes the data type values with the values provided by the client as arguments.



Anatomy of a constructor

- *Instance methods.* To implement instance methods, we write code that is precisely like the code that we learned in Chapter 2 to implement static methods (functions). Each method has a signature (which specifies its return type and the types and names of its parameter variables) and a body (which consists of a sequence of statements, including a return statement that provides a value of the return type back to the client). When a client invokes a method, the parameter values are initialized with client values, the lines of code are executed until a return value is computed, and the value is returned to the client, with the same effect as if the method invocation in the client were replaced with that value. All of this action is the same as for static methods, but there is one critical distinction for instance methods: *they can perform operations on instance values.*



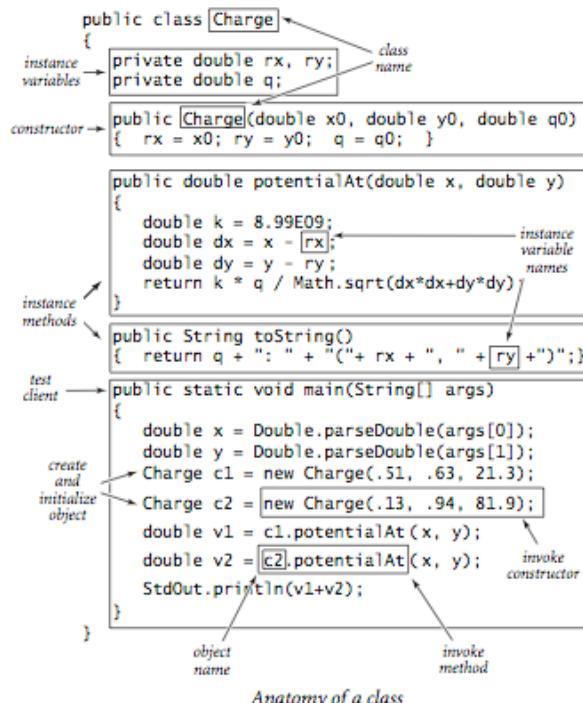
Anatomy of a data-type method

- *Variable names in methods.* Accordingly, code in instance methods uses three kinds of variables:
  - parameter variables
  - local variables
  - instance variables

The first two of these are familiar: parameter variables are specified in the method signature and initialized with client values when the method is called, and local variables are declared and initialized within the method body, just as for static methods. The scope of parameter variables is the entire method; the scope of local variables is the block where they are defined. Instance variables are completely different: they hold data-type values for objects in a class, and their scope is the entire class. How do we specify which object's values we want to use? If you think for a moment about this question, you will recall the answer. Each object in the class has a value: the code in a class method refers to the value for *the object that was used to invoke the method*. When we write `c1.potentialAt(x, y)`, the code in `potentialAt()` is referring to the instance variables for `c1`. The code in `potentialAt()` uses all three kinds of variable names:

variable	purpose	example	scope
instance	data-type value	rx	class
parameter	pass value from client to method	x	method
local	temporary use within method	dx	block

**Summary.** Here is a diagram illustrating all of the basic components that you need to understand to be able to build data types in Java.



Anatomy of a class

**Stopwatch.** [Stopwatch.java](#) implements the following API:

It is a stripped-down version of an old-fashioned stopwatch. When you create one, it starts running, and you can ask it how long it has been running by invoking the method `elapsedTime()`.

**Histogram.** Data-type instance variables can be arrays. [Histogram.java](#) maintains an array of the frequency of occurrence of integer values in a given interval  $[0, N]$  and uses [StdStats.java](#) to display a histogram of the values, controlled by this API:

**Turtle graphics.** [Turtle.java](#) is a mutable type for turtle graphics.

- *Ngon.* Program [Ngon.java](#) takes a command-line argument  $N$  and draws a regular  $N$ -gon.

By taking  $N$  to a sufficiently large value, we obtain a good approximation to a circle. In Cartesian geometry, a circle is the locus of points at distance  $r$  from a fixed center point  $(a, b)$ . *Declarative* definition describes points on a circle.

$$(x - a)^2 + (y - b)^2 = r^2$$

In Differential geometry, a circle is shape of constant curvature (how much we turn for each step we move). *Imperative* definition tells us *how* to draw it.

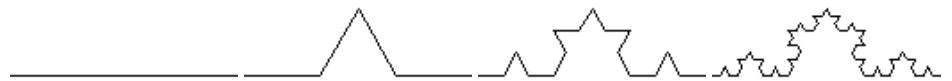
```
REPEAT
  GO FORWARD 1 UNIT
  TURN LEFT 1 UNIT
```

- *Koch curves.* The Koch curve of order 0 is a line segment. To draw a Koch curve of

order n in turtle graphics

- Draw a Koch curve of order n-1
- Rotate 60° counterclockwise
- Draw a Koch curve of order n-1
- Rotate 120° clockwise
- Draw a Koch curve of order n-1
- Rotate 60° counterclockwise
- Draw a Koch curve of order n-1

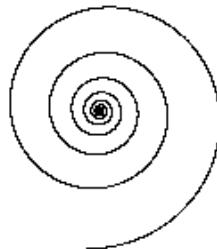
Below are the Koch curves of order 0, 1, 2, and 3.



Program [Koch.java](#) reads in a command line parameter N and plots an order N Koch snowflake using our graphics library. It uses some high school geometry to calculate what size to draw the line segments and where to start drawing.

Historical context: In 1872 Karl Weierstrass made the shocking discovery a function that is everywhere continuous but nowhere differentiable. Later this would become one of the characteristic properties of fractals. Helge von Koch wanted to find a less abstract example. In 1904, von Koch described the geometric construction, which we now refer to as the Koch curve. Von Koch proved that, in the limit, the Koch snowflake is a curve of infinite length, but it does not have a tangent at any point. From this, he proved that there exist continuous functions  $f(t)$  and  $g(t)$  such that the Koch snowflake is  $(f(t), g(t))$  for  $0 \leq t \leq 1$ , but  $f(t)$  and  $g(t)$  are nowhere differentiable.

- *Spira mirabilis*. [Spiral.java](#) creates a logarithmic spiral (aka equiangular spiral or growth spiral). Biological principle: as size increases, shape is unaltered. Radius grows exponentially with the angle.



The [logarithmic spiral](#) was first described by Descartes in 1638. Bernoulli was so amazed by its many properties that he named it *Spira mirabilis* (miraculous or marvelous spiral) and asked to have it engraved on his tombstone (but an Archimedean spiral was put there instead). Shape arises in nature: chambers of a nautilus shell, arms of tropical hurricanes, arms of spiral galaxy, approach of a hawk to its prey, a charged particle moving in a uniform magnetic field, perpendicular to that field.





- *Brownian motion.* Imagine a disoriented turtle (again following its standard alternating turn and step regimen) turns in a random direction before each step. [DrunkenTurtle.java](#) plots the path followed by such a turtle. In 1827, the botanist Robert Brown observed through a microscope that pollen grains immersed in water seemed to move about in just such a random fashion, which later became known as Brownian motion and led to Albert Einstein's insights into the atomic nature of matter. [DrunkenTurtles.java](#) plots many such turtles, all of whom wander around.

Turtle graphics was originally developed by Seymour Papert at MIT in the 1960s as part of an educational programming language, Logo. But turtle graphics is no toy, as we have just seen in numerous scientific examples. Turtle graphics also has numerous commercial applications. For example, it is the basis for PostScript, a programming language for creating printed pages that is used for most newspapers, magazines, and books.

**Complex numbers.** A *complex* number is a number of the form  $x + iy$ , where  $x$  and  $y$  are real numbers and  $i$  is the square root of  $-1$ . The number  $x$  is known as the real part of the complex number, and  $iy$  is known as the imaginary part. Complex numbers are a quintessential mathematical abstraction: whether or not one believes that it makes sense physically to take the square root of  $-1$ , complex numbers help us understand the natural world. They are extensively used in applied mathematics and play an essential role in many branches of science and engineering. They are used to model physical systems of all sorts, from circuits to sound waves to electromagnetic fields. These models typically require extensive computations involving manipulating complex numbers according to well-defined arithmetic operations, so we want to write computer programs to do the computations. In short, we need a new data type.

Developing a data type for complex numbers is a prototypical example of the value of object-oriented programming. No programming language can provide implementations of every mathematical abstraction that we might need, but the ability to implement data types give us not just the ability to write programs to easily manipulate abstractions like complex numbers, polynomials, vectors, matrices, and many other basic tools that have been developed by mathematicians over the past several centuries, but also the freedom to think in terms of new abstractions.

The basic operations on complex numbers are to add and multiply them by applying the commutative, associative, and distributive laws of algebra (along with the identity  $i^2 = -1$ ); and to compute the magnitude, as follows:

- addition:  $(x+iy) + (v+iw) = (x+v) + i(y+w)$ .
- multiplication:  $(x + iy) * (v + iw) = (xv - yw) + i(yv + xw)$ .
- magnitude:  $|x + iy| = \sqrt{x^2 + y^2}$
- real and imaginary parts:  $\text{re}(x + iy) = x$ , and  $\text{im}(x + iy) = y$

For example, if  $a = 3 + 4i$ , and  $b = -2 + 3i$ , then  $a + b = 1 + 7i$ ,  $a * b = -18 + i$ , and  $|a| = 5$ .

With these basic definitions, the path to implementing a data type for complex numbers is clear. We start with an API that specifies the data-type operations:

---

<code>public class Complex (PROGRAM 3.2.2)</code>	
<code>Complex(double real, double imag)</code>	
<code>Complex plus(Complex b)</code>	<i>sum of this number and b</i>
<code>Complex times(Complex b)</code>	<i>product of this number and b</i>
<code>double abs()</code>	<i>magnitude</i>
<code>double Re()</code>	<i>real part</i>
<code>double Im()</code>	<i>imaginary part</i>
<code>String toString()</code>	<i>string representation</i>

Program `Complex.java` is an immutable class that implements this API. It has all of the same components as did `Charge` (and every Java data type implementation): instance variables (`re` and `im`), a constructor, instance methods, and a test client.

The code that implements the arithmetic methods makes use of a new mechanism for accessing object values:

- *Accessing values in objects of this type.* Both `plus()` and `times()` need to access values in two objects: the object passed as a parameter and the object used to invoke the method. If the method is called with `a.plus(b)`, we can access the values of `a` using the instance variable names `re` and `im`, as usual, but to access the values of `b` we use the code `b.re` and `b.im`. However, since we keep the instance variables `private`, *you cannot access directly instance variables in another class* (but you can access any object's instance variables within the same class). This policy reduces flexibility but improves design when developing modular programs.
- *Chaining.* Observe the manner in which `main()` chains two method calls into one compact expression: the expression `z.times(z).plus(z0)` evaluates to  $z^2 + z_0$ . This usage is convenient because we do not have to invent a variable name for the intermediate value. If you study the expression, you can see that there is no ambiguity: moving from left to right, each method returns a reference to a `Complex` object, which is used to invoke the next method. If desired, we can use parentheses to override the precedence order (for example, `z.times(z.plus(z0))` evaluates to  $z(z + z_0)$ ).
- *Creating and returning new objects.* Observe the manner in which `plus()` and `times()` provide return values to clients: they need to return a `Complex` value, so they each compute the requisite real and imaginary parts, use them to create a new object and then return a reference to that object. This arrangement allows clients to manipulate complex numbers by manipulating local variables of type `Complex`.

**Mandelbrot set.** The uses complex numbers to plot a gray scale version of the The **Mandelbrot set** is a specific set of complex numbers with many fascinating properties. It is a fractal pattern that is related to the Barnsley fern, the Sierpinski triangle, the Brownian bridge, and other recursive (self-similar) patterns and programs that we have seen in this book. Patterns of this kind are found in natural phenomena of all sorts, and these models and programs are very important in modern science. The set of points in the Mandelbrot set cannot be described by a single mathematical equation. Instead, it is defined by an *algorithm*, and therefore a perfect candidate for a `Complex` client: we study the set by writing programs to plot it.

The rule for determining whether or not a complex number  $z_0$  is in the Mandelbrot set is simple: Consider the sequence of complex numbers  $z_0, z_1, z_2, \dots, z_i, \dots$ , where  $z_{i+1} = (z_i)^2 + z_0$ . For example, the following table shows the first few entries in the sequence corresponding to  $z_0 = 1 + i$ :

$t$	$z_t$	$(z_t)^2$	$(z_t)^2 + z_0$
0	$1 + i$	$1 + 2i + i^2 = 2i$	$2i + (1 + i) = 1 + 3i$
1	$1 + 3i$	$1 + 6i + 9i^2 = -8 + 6i$	$-8 + 6i + (1 + i) = -7 + 7i$
2	$-7 + 7i$	$49 - 98i + 49i^2 = -98i$	$-98i + (1 + i) = 1 - 97i$

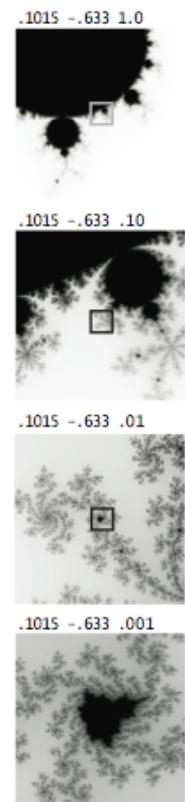
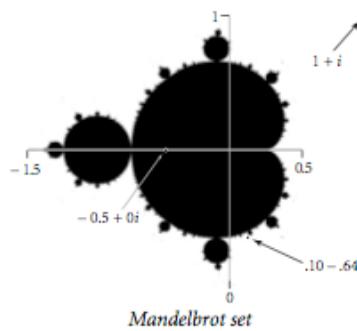
Now, if the sequence  $|z_i|$  diverges to infinity, then  $z_0$  is *not* in the Mandelbrot set; if the sequence is bounded, then  $z_0$  is in the Mandelbrot set. For many points, the test is simple; for many other points, the test requires more computation, as indicated by the examples in the following table:

$t$	$0 + 0i$	$2 + 0i$	$1 + i$	$-.05 + 0i$	$.10 - .64i$
0	$0 + 0i$	$2 + 0i$	$1 + i$	$-.05 + 0i$	$.10 - .64i$
1	$0 + 0i$	$6 + 0i$	$1 + 3i$	$-.25 + 0i$	$-.30 - .77i$
2	$0 + 0i$	$36 + 0i$	$-7 + 7i$	$-.44 + 0i$	$-.40 - .18i$
3	$0 + 0i$	$1446 + 0i$	$1 - 97i$	$-.30 + 0i$	$-.09 - .87i$
4	$0 + 0i$	$2090918 + 0i$	$-9407 - 193i$	$-.40 + 0i$	$-.64 - .78i$

*Is a complex number in the set?* In some cases, we can prove whether or not numbers are in the set: for example,  $0 + 0i$  is certainly in the set (since the magnitude of all the

numbers in its sequence is 0) and  $2 + 0i$  is certainly not in the set (since its sequence dominates the powers of 2, which diverges). Given a complex point, we can compute the terms at the beginning of its sequence, but may not be able to know for sure that the sequence remains bounded. Remarkably, there is a test that tells us for sure that a point is *not* in the set: if the magnitude of any number in the sequence ever gets to be greater than 2 (like  $3 + 0i$ ), then the sequence will surely diverge.

*Plotting the Mandelbrot set.* A visual representation of the Mandelbrot set is easy to define. Just as we plot a real function by sampling points in an interval, we plot the Mandelbrot set by sampling complex points. Each complex number  $x + iy$  corresponds to a point  $(x, y)$  in the plane so we can plot the results as follows: for a specified resolution  $N$ , we define a regularly spaced  $N$ -by- $N$  pixel grid within a specified square and draw a black pixel if the corresponding point is in the Mandelbrot set and a white pixel if it is not. This plot is a strange and wondrous pattern, with all the black dots connected and falling roughly within the 2-by-2 square centered on the point  $-1/2 + 0i$ . Large values of  $N$  will produce higher resolution images, at the cost of more computation. Looking closer reveals self-similarities throughout the plot. For example, the same bulbous pattern with self-similar appendages appears all around the contour of the main black cardioid region. When we zoom in near the edge of the cardioid, tiny self-similar cardioids appear!



Zooming in on the set

Program [Mandelbrot.java](#) uses this test to plot a visual representation of the Mandelbrot set. Since our knowledge of the set is not quite black-and-white, we use grayscale in our visual representation. It is based on the following function (a `Complex` client), which computes the sequence starting at a given number and returns the number of iterations for which the magnitude stays less than 2, up to a given maximum:

```
public static int mand(Complex z0, int d) {
    Complex z = z0;
    for (int t = 0; t < d; t++) {
        if (z.abs() >= 2.0) return t;
        z = z.times(z).plus(z0);
    }
    return d;
}
```

For each pixel, the method `main()` in `Mandelbrot` computes the point `z0` corresponding to the pixel and computes  $255 - \text{mand}(z0, 255)$  to represent the grayscale value of the pixel. Any pixel that is not black corresponds to a point that we know to be not in the Mandelbrot set because the magnitude of the numbers in its sequence grew past 2 (and therefore will go to infinity); the black pixels (grayscale value 0) correspond to points that we assume to be in the set because the magnitude stayed less than 2 for 255 iterations, but we do not necessarily know for sure. The complexity of the images that this simple program produces is remarkable, even when we zoom in on a tiny portion of the plane. For even more dramatic pictures, we can use use color (see exercise 3.2.33).

**Q.** Do instance variables have initial values that we can depend upon?

**A.** Yes. They are automatically set to 0 for numeric types, `false` for the `boolean` type, and the special value `null` for all reference types.

**Q.** What is `null`?

**A.** It is a literal value that refers to no object. Invoking a method using the `null` reference is meaningless and results in a `NullPointerException`.

**Q.** Can we initialize instance variables to other values when declaring them?

**A.** Yes, you can initialize instance variables using the same conventions as you have been using for initializing local variables. Each time an object is created by a client with `new`, its instance variables are initialized with those same values, then the constructor is called.

**Q.** Must every class have a constructor?

**A.** Yes, but if you do not specify a constructor, Java provides a default (no-argument) constructor automatically. When the client invokes that constructor with `new`, the instance variables are initialized as usual. If you do specify a constructor, the default no-argument constructor disappears.

**Q.** Suppose I do not include a `toString()` method. What happens if I try to print an object of that type with `StdOut.println()`?

**A.** The printed output is an integer: the object's hash code. Most objects that do not have an explicit `toString()` implementation also do not have an explicit `hashCode()` implementation. The default value is "typically implemented by converting the internal address of the object into an integer."

**Q.** Can I have a static method in a class that implements a data type?

**A.** Of course. All of our classes have `main()`. But it is easy to get confused when static methods and instance methods are mixed up in the same code. For example, it is natural to consider using static methods for operations that involve multiple objects where none of them naturally suggests itself as the one that should invoke the method. For example, we say `z.abs()` to get  $|z|$ , but saying `a.plus(b)` to get the sum is perhaps not so natural. Why not `b.plus(a)`? An alternative is to define a static method within `Complex`, like

```
public static Complex plus(Complex a, Complex b) {
    return new Complex(a.re + b.re, a.im + b.im);
}
```

but we avoid such usage so that we can write

```
z = z.times(z).plus(z0).
```

instead of

```
z = Complex.plus(Complex.times(z, z), z0)
```

**Q.** These computations with `plus()` and `times()` seem rather clumsy. Is there some way to use symbols like `*` and `+` in expressions involving objects where they make sense, like `Complex` and `Vector`, so that we could write expressions like `z = z * z + z0` instead?

**A.** Some languages (notably C++) support this feature, which is known as operator overloading, but Java does not do so (except that there is language support for overloading `+` with string concatenation). As usual, this is a decision of the language designers that we just live with, but many Java programmers do not consider this to be much of a loss. Operator overloading makes sense only for types that represent numeric or algebraic abstractions, a small fraction of the total, and many programs are easier to understand when operations have descriptive names such as `plus` and `times`.

**Q.** Are there other kinds of variables besides parameter, local, and instance variables in

a class?

**A.** If you include the keyword `static` in a class declaration (outside of any type) it creates a completely different type of variable, known as a *static variable*. Like instance variables, static variables are accessible to every method in the class; however, they are not associated with any object. In older programming languages, such variables are known as *global variables*, because of their global scope. In modern programming, we focus on limiting scope and therefore rarely use such variables.

**Q.** Is there a relationship between the `Vector` in this section and the `Vector` class in the Java library?

**A.** No. We use the name because the term `vector` properly belongs to physics and linear algebra.

**Q.** Mandelbrot creates hundreds of millions of `Complex` objects. Doesn't all that object-creation overhead slow things down?

**A.** Yes, but not so much that we cannot generate our plots. Our goal is to make our programs readable and easy to maintain - limiting scope via the complex number abstraction helps us achieve that goal. You certainly could speed up Mandelbrot by bypassing the complex number abstraction or by using a different implementation of `Complex`.

**Q.** What does the error message "can't make a static reference to a non-static variable" mean?

**A.**

```
public class Test {  
    private static int M; // class variable  
    private int N; // instance variable  
    public static int getM() { return M; } // OK  
    public static int getN() { return N; } // ERROR  
}
```

## Exercises

1. Why does program [Bug1.java](#) create a `java.lang.NullPointerException` when executed?

```
public class Bug1 {  
    private String s;  
    public void Bug1() { s = "hello"; }  
    public String toString() { return s; }  
    public static void main(String[] args) {  
        Bug1 x = new Bug1();  
        StdOut.println(x);  
    }  
}
```

*Answer:* the programmer probably intended to make the no argument constructor set the string to `hello`. However, it has a return type (`void`) so it is an ordinary instance method instead of a constructor. It just happens to have the same name as the class.

2. Why does program [Bug2.java](#) create a `java.lang.NullPointerException` when executed?
3. Implement a data type `Die` for rolling a fair die, say with 6 sides. Include a mutator method `roll()` and an accessor method `value`.
4. Implement a mutable data type `LFSR` for a linear feedback shift register.
5. Implement a mutable data type `Counter`.
6. Implement a mutable data type `Odometer`.
7. Implement a mutable data type `StopWatch`.

8. Implement a data type `VotingMachine` for tabulating votes. Include mutator methods `voteRepublican()`, `voteDemocrat()`, and `voteIndependent()`. Include an accessor method `getCount()` to retrieve the total number of votes.
9. What happens when you try to compile and execute the following code fragment?

```
Student x;
StdOut.println(x);
```

*Answer:* it complains that `x` may not be initialized, and does not compile.

10. Suppose you want to add a constructor to `Complex` that takes one real argument and initializes a complex object with that value. You write the following code

```
public void Complex(double real) {
    re = real;
    im = 0.0;
}
```

Why does the statement `Complex c = new Complex(1.0)` given a compiler error? *Answer:* constructors do not have return types, not even `void`. So the code above is actually a method named `Complex` not a constructor. Remove the keyword `void` and it will work. This is a common gotcha.

11. What happens when you try to compile and execute the following code fragment?

```
Student[] students = new Student[10];
StdOut.println(students[5]);
```

*Answer:* it compiles and prints out `null`.

12. What is wrong with the following code fragment?

```
int N = 17;
Dog[] dogs = new Dog[N];
for (int i = 0; i < N; i++) {
    dog.bark();
    dog.eat();
}
```

*Answer:* it produces a `NullPointerException` because we forgot use `new` to create each individual `Dog` object. To correct, add the following loop after the array initialization statement.

```
for (int i = 0; i < N; i++)
    dogs[i] = new Dog();
```

13. What does the following code fragment print?

```
Complex c = new Complex(2.0, 0.0);
StdOut.println(c);
StdOut.println(c.mul(c).mul(c).mul(c));
StdOut.println(c);
```

14. Modify the `toString` method in `Complex.java` so that it prints  $3 - i$  as  $3 - i$  instead of  $3 + -i$ , and prints 3 as 3 instead of  $3 + 0i$ .
15. What's wrong with the following code fragment that swaps the `Student` objects x

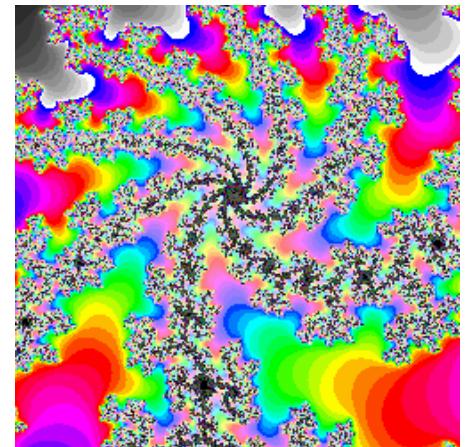
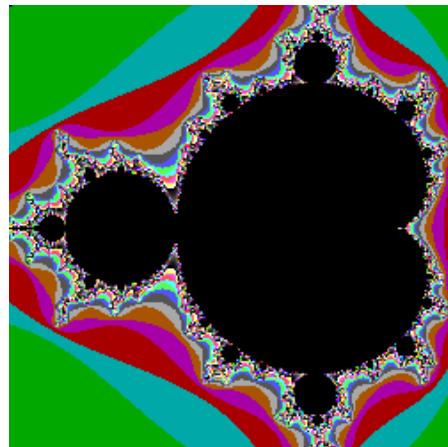
and y?

```
Student swap = new Student();
swap = x;
x = y;
y = swap;
```

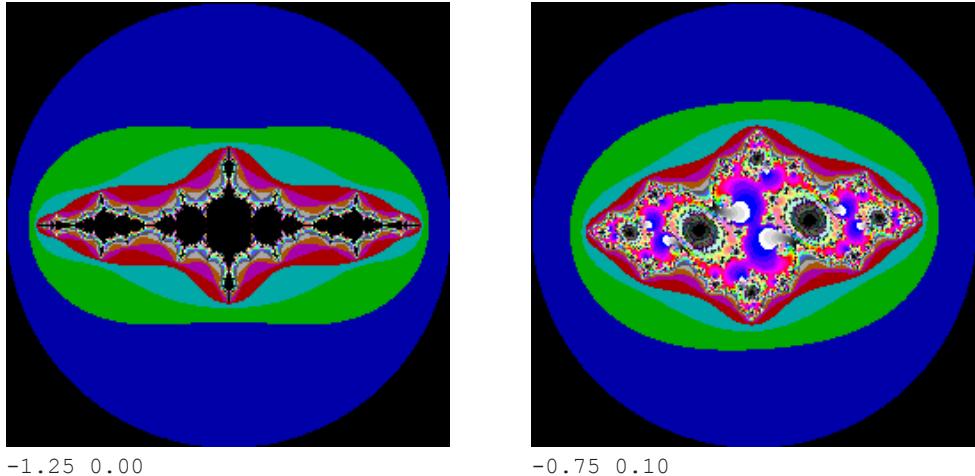
*Answer:* First, the data type `Student` does not have a no-argument constructor. If it did, then it would technically be correct, but the `new Student()` line is unnecessary and wasteful. It allocates memory for a new student object, sets `swap` equal to that memory address, then immediately sets `swap` equal to the memory address of `x`. The allocated memory is no longer accessible. The following version is correct.

```
Student swap = x;
x = y;
y = swap;
```

16. Add a method `minus(Complex b)` to `Complex` that takes a complex argument `b` and returns the difference between the invoking object and `b`.
17. Add a method `divides(Complex b)` to `Complex` that takes a complex argument `b` and returns the quotient of the invoking object divided by `b`.
18. Add a method `conjugate()` to `Complex` that returns the complex conjugate of the invoking object.
19. Write a program [RootsOfUnity.java](#) that takes a command line argument `N` and uses `Complex` to compute and print out the `N` Nth roots of unity.
20. Write a program to read in three real numbers `a`, `b`, and `c` and print out all roots of  $ax^2 + bx + c$ , including complex ones.
21. Find inputs to the Mandelbrot update formula that make it converge ( $z_0 = 1/2 + 0i$ ), cycle with a period of 1 ( $z_0 = -2 + 0i$ ), cycle with a period of 2 ( $z_0 = -1 + 0i$ ), or stay bounded without converging ( $z_0 = -3/2 + 0i$ ).
22. Write a program [ColorMandelbrot.java](#) that plots a color version of the Mandelbrot set. Read in a 256-by-3 array of color values from standard input into an array, and then use the `i`th color if the Mandelbrot function takes `i` iterations. Use the data file [mandel.txt](#) as an example.



23. **Julia sets.** The *Julia set* for a given complex number  $c$  is a set of points related to the Mandelbrot function. Instead of fixing  $z$  and varying  $c$ , we fix  $c$  and vary  $z$ . Those points  $z$  for which the modified Mandelbrot function stays bounded are in the Julia set; those for which the sequence diverges to infinity are not in the set. All points  $z$  of interest lie in the the 4-by-4 box centered at the origin. The Julia set for  $c$  is connected if and only if  $c$  is in the Mandelbrot set! Write a program [ColorJulia.java](#) that takes two command line parameters  $a$  and  $b$ , and plots a color version of the Julia set for  $c = a + ib$ . Read in a 256-by-3 array of color values from standar input into an array, and then use the  $i$ th color if the Julia function takes  $i$  iterations. Use the data file [mandel.txt](#) as an example.



24. **Point3D.** Create a data type for points in 3 dimensional space. Include a constructor that takes three real coordinates  $x$ ,  $y$ , and  $z$ . Include methods `distance`, `distanceSquared`, and `distanceL1` for the Euclidean distance, Euclidean distance squared, and the L1-distance.
25. **Intervals.** Create a data type [Interval.java](#) for intervals on the x-axis. An interval is the set of points in the range  $[left, right]$ . Include methods for a constructor (including a check that the left endpoint is no greater than the right endpoint) and a method `intersects` so that `a.intersects(b)` returns true if the intervals `a` and `b` intersect, and false otherwise.
26. Create a data type [PhoneNumber.java](#) that represents a US phone number. The constructor should take three string arguments, the area code (3 decimal digits), the exchange (3 decimal digits) and the extension (4 decimal digits). Include a `toString` method that prints out phone numbers of the form (800) 867-5309. Include a method so that `p.equals(q)` returns `true` if the phone numbers `p` and `q` are the same, and `false` otherwise.
27. Redo [PhoneNumber.java](#) but implement it using three integer fields. Make the constructor take three integer arguments. Comment on the advantages and disadvantages of this approach over the string representation.
- Answer:* more efficient with time and memory. More hassle to handle leading 0s correct in constructor and `toString` method.
28. Write a program to draw the field lines for a *uniform field*. Arrange  $N$  evenly-spaced particles with charge  $e$  in a vertical column, and arrange  $N$  particles with charge  $-e$  in a vertical column so that each charge on one side is lined up with a corresponding charge on the other side. This simulates the field inside a plane capacitor. What can you say about the resulting electric field? A. Almost uniform.

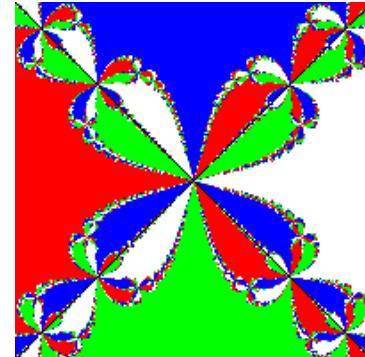
## Creative Exercises

1. **IP addresses.** Create a data type for IPv4 (Internet Protocol, version 4) addresses. An IP address is a 32-bit integer.
2. **Dates.** Create a data type `Date` that represents a date. You should be able to create a new `Date` by specifying the month, day, and year. It should support methods to compute the number of days between two dates, return the day of the week that a day falls on, etc.
3. **Time bombs.** UNIX represents the date with a signed integer measuring the number of seconds since January 1, 1970. Write a client program to calculate when this date will occur. Add a static method `add(Date d, int days)` to your date data type that returns a new date which is the specified number of days after the date `d`. Note that there are 86,400 seconds in a day.

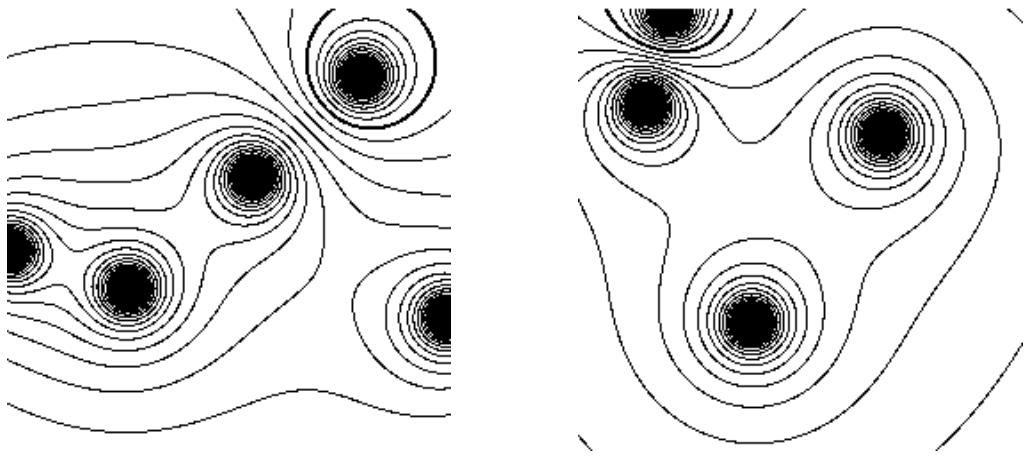
4. **Qubits.** In quantum computing, a *qubit* plays the role of a bit. It is a complex number  $a + bi$  such that  $|a + bi| = 1$ . Once we measure a qubit, it "decides" to be a 1 with probability  $a^2$  and a 0 with probability  $b^2$ . Any subsequent observations will always yield the same value. Implement a data type `Qubit` that has a constructor `Qubit(a, b)` and a boolean method `observe` that returns `true` or `false` with the proscribed probabilities.
  5. **Biorhythms.** A *biorhythm* is a pseudo-scientific profile of the three natural cycles of your body: physical (23 days), emotional (28 days), and intellectual (31 days). Write a program that takes six command line inputs  $M$ ,  $D$ ,  $Y$ ,  $m$ ,  $d$ , and  $y$  where  $(M, D, Y)$  is the month (1-12), day (1-31), and year (1900-2100) of your birthday and  $(m, d, y)$  is today's month, day, and year. It should then print out your biorhythm on a scale of -1.0 to 1.0 according to the formula:  $\sin(2\pi \text{age} / \text{cycle length})$ . Use the date data type created in the previous exercise.
  6. **Particle.** Create a data type for elementary or composite particles (electron, proton, quark, photon, atom, molecule). Each particle should have an instance variable to store its name, its mass, its charge, and its spin (multiple of  $1/2$ ).
  7. **Quark.** Quarks are the smallest known building blocks of matter. Create a data type for quarks. Include a field for its type (up, down, charm, strange, top, or bottom) and its color (red, green, or blue). The charges are  $+2/3$ ,  $-1/3$ ,  $+2/3$ ,  $-1/3$ ,  $+2/3$ ,  $-1/3$ , respectively. All have spin  $1/2$ .
- Diffusion of particles in a fluid.** Simulate diffusion of particles in a fluid. See [BrownianParticle.java](#) in Section 9.8.
8. **Biorhythms.** Plot your biorhythm in turtle graphics over a 6 week interval. Identify *critical days* when your rhythm goes from positive to negative - according to biorhythm theory, this is when you are most prone to accident, instability, and error.
  9. **Vector3.** Include normal vector operations for 3-vectors, including *cross product*. The cross product of two vectors is another vector.  $a \text{ cross } b = ||a|| ||b|| \sin(\theta) n$ , where  $\theta$  is angle between  $a$  and  $b$ , and  $n$  is unit normal vector perpendicular to both  $a$  and  $b$ .  $(a_1, a_2, a_3) \text{ cross } (b_1, b_2, b_3) = (a_2 b_3 - a_3 b_2, a_3 b_1 - a_1 b_3, a_1 b_2 - a_2 b_1)$ . Note that  $|a \text{ cross } b| = \text{area of the parallelogram with sides } a \text{ and } b$ . Cross product arises in definition of torque, angular momentum and vector operator *curl*.
  10. **Four-vector.** Create a data type for *four-vectors*. A four-vector is a four-dimensional vector  $(t, x, y, z)$  subject to Lorentz transformations. Useful in special relativity.
  11. **Euclidean points.** Create a data type `EuclideanPoint.java` that represents a  $d$ -dimensional point. Include a method so that `p.distanceTo(q)` returns the Euclidean distance between points  $p$  and  $q$ .
  12. **Vector field.** A *vector field* associates a vector with every point in a Euclidean space. Widely used in physics to model speed and direction of a moving object or or strength and direction of a Newtonian force.
  13. **Soda machine.** Create a data type `SodaMachine` that has methods `insertCoin()`, `getChange()`, `buy()`, etc.
  14. **Months.** Write a data type `Month` that represents one of the twelve months of the year. It should have fields for the name of the month, the number of days in the month, and the birthstone.

MONTH	DAYS	BIRTHSTONE
January	31	Garnet
February	28	Amethyst
March	31	Aquamarine
April	30	Diamond
May	31	Emerald
June	30	Alexandrite
July	31	Ruby
August	31	Peridot
September	30	Sapphires
October	31	Opal

15. **Chaos with Newton's method.** The polynomial  $f(z) = z^4 - 1$  has 4 roots at 1, -1,  $i$ , and  $-i$ . We can find the roots using Newton's method over the complex plane:  $z_{k+1} = z_k - f(z_k)/f'(z_k)$ . Here  $f(z) = z^4 - 1$  and  $f'(z) = 4z^3$ . The method converges to one of the 4 roots depending on the starting point  $z_0$ . Choose each point in the complex plane with coordinates between -1 and 1 and determine which of the four roots it converges to, and plot the point either white, red, green, or blue accordingly. If Newton's method doesn't converge after 100 iterations, color the point black. Name your program [NewtonChaos.java](#).



16. **Gauss multiplication.** Implement complex multiplication using only 3 floating point multiplications (instead of 4). You may use as many as 5 floating point additions. Answer: Gauss gave the following method to multiply  $(a + bi)(c + di)$ . Set  $x_1 = (a + b)(c + d)$ ,  $x_2 = ac$ ,  $x_3 = bd$ . Then the product is given by  $x + yi$  where  $x = x_2 - x_3$ ,  $y = x_1 - x_2 - x_3$ .
17. **Rational numbers.** Create a data type [Rational.java](#) and [BigRational.java](#) for positive rational numbers.
18. **Rational numbers.** Modify [Rational.java](#) to provide support for negative rationals and zero.
19. **Quaternions.** In 1843, Sir William Hamilton discovered an extension to complex numbers called [quaternions](#). [Quaternions](#) extend the concept of rotation in three dimensions to four dimensions. They are used in computer graphics, control theory, signal processing, and orbital mechanics, e.g., command for spacecraft attitude control systems. are related to Pauli spin matrices in physics. Create a data type [Quaternion.java](#) to represent quaternions. Include operations for adding, multiplying, inverting, conjugating, and taking the norm of quaternions.
- A quaternion can be represented by a 4-tuple of real numbers  $(a_0, a_1, a_2, a_3)$ , which represents  $a_0 + a_1 i + a_2 j + a_3 k$ . The fundamental identity is  $i^2 = j^2 = k^2 = ijk = -1$ . The quaternion conjugate is  $(a_0, -a_1, -a_2, -a_3)$ . The quaternion norm is  $\sqrt{a_0^2 + a_1^2 + a_2^2 + a_3^2}$ . The inverse of a quaternion is  $(a_0/d, -a_1/d, -a_2/d, -a_3/d)$  where  $d$  is the square of the quaternion norm. The sum of two quaternions  $(a_0, a_1, a_2, a_3)$  and  $(b_0, b_1, b_2, b_3)$  is  $(a_0+b_0, a_1+b_1, a_2+b_2, a_3+b_3)$ , the product is  $(a_0b_0 - a_1b_1 - a_2b_2 - a_3b_3, a_0b_1 + a_1b_0 + a_2b_3 - a_3b_2, a_0b_2 - a_1b_3 + a_2b_0 + a_3b_1, a_0b_3 + a_1b_2 - a_2b_1 + a_3b_0)$ , and the quotient is the product of the inverse of  $(a_0, a_1, a_2, a_3)$  and  $(b_0, b_1, b_2, b_3)$ . Note that  $ab$  doesn't equal  $ba$  in general.
20. **Equipotential surfaces.** An *equipotential surface* is the set of all points that have the same electric potential  $V$ . Given a group of  $N$  point charges, it is useful to visualize the electric potential by plotting equipotential surfaces (aka contour plots). Program [Equipotential.java](#) draws a line every  $5V$  by computing the potential at each gridpoint and checking whether the potential is within 1 pixel of a multiple of  $5V$ . Since the electric field  $E$  measures how much the potential changes,  $E * \text{eps}$  is the range that the potential changes in a distance of 1 pixel.



It is also interesting to plot the field lines and the equipotential lines simultaneously. The field lines are always perpendicular to the equipotential lines.

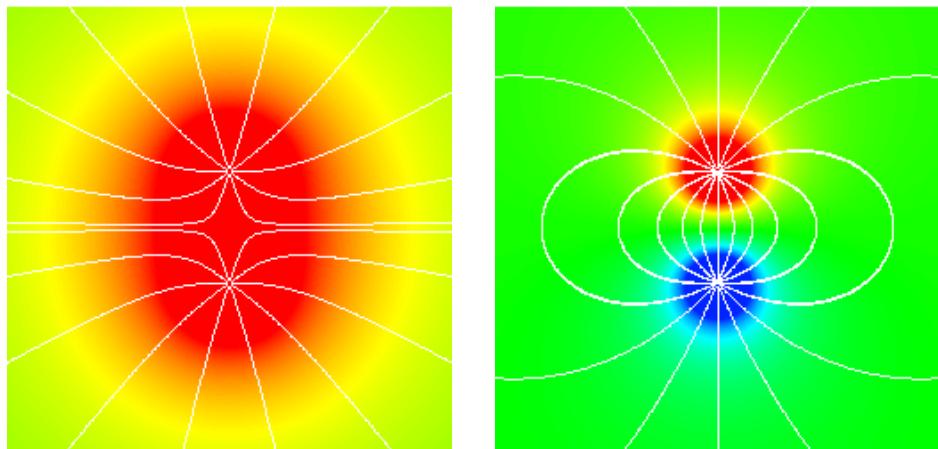
21. **Tensors.** Create a data type for [tensors](#).
22. **UN Countries.** Create a data type `Country` for [UN countries](#). Include fields for 3 digit UN Code, 3 letter ISO abbreviation, country name, and capital. Write a program `Country.java` that reads in a list of countries and stores them in an array of type `Country`. Use the method `String.split` to help parse the input file.
23. **Area codes.** Create a data type for [telephone area codes in North America](#). Include fields for the area code, the city, and state, and the two letter state abbreviation. Or for [international phone codes](#). Include a field for zone, code, and country.
24. **Congressional districts.** Create a data type for places, counties, and congressional districts. Include fields for place name, county name, county code, zip code, congressional district, etc. Use the data sets from the [1998 FIPS55-DC3 Index](#): Pennsylvania ([2MB](#)) or all 50 states plus DC and 9 outlying areas ([30MB](#)).
25. **Latitudes and longitudes.** For USA latitudes and longitudes, use the [TIGER database](#) or [www.bcca.org](#) or [gazetteer](#). For the rest of the world, use [earth-info](#).
26. **Astronomy.** Data for [asteroids](#), [meteors](#), and [comets](#).
27. **Fortune 1000 companies.** Create a data type for [the Fortune 1000](#). Include fields for company name and sales revenue in millions of dollars. Data taken from April 15, 2002 issue of Fortune. Note: currently need to parse data.
28. **Elements.** Create a data type `Element.java` for the [Periodic Table of Elements](#). Include fields for element, atomic number, symbol, and atomic weight. (Ignore fields for boiling point, melting point, density (kg/m<sup>3</sup>), Heat vapour (kJ/mol), heat fusion (kJ/mol), thermal conductivity (W/m/K), and specific heat capacity (J/kg/K) since it's not known for all elements). The file is in CSV format (fields separated by commas).
29. **Molecular weight.** Write a program so that the user enters a molecule H<sub>2</sub>O and the program calculates its molecular weight.
30. Some potentially useful datafiles: [aroma therapies](#), [nutritional information](#), [meteorological glossary](#), [psychiatric disorders](#), [words translated in 15 languages](#), [dictionary of emoticons](#), [meanings of common names](#), [World Almanac facts about countries](#).
31. **Student records.** Create a data type `Student.java` to represent students in an introductory computer science course. Each student record object is comprised of four instance variables representing the first name, last name, email address, and section number. Include a `toString()` method. Also include a method `less()` to compare students by section.
32. **Impedance.** Impedance is the generalization of resistance from DC circuits to AC circuits. In an AC circuit, the *impedance* of a component measures its opposition to the flow of electrons at a given frequency  $\omega$ . The impedance has two components: the resistance and the reactance. The *resistance*  $R$  of a circuit component measures its opposition to the movement of electrons (friction against motion of electrons) when a given voltage is applied. The *reactance*  $X$  of a circuit component measures its ability to store and release energy as the current and voltage fluctuate (inertia against motion of electrons).

In circuits with resistors only, the current is directly proportional to the voltage. However, with capacitors and inductors, there is a  $+- 90$  degree "phase shift" between the current and voltage. This means that when the voltage wave is at its maximum, the current is 0, and when the current is at its maximum the voltage is 0. To unify the treatment of resistors (R), inductors (L), and capacitors (C) it is convenient to treat the impedance as the complex quantity  $Z = R + iX$ . the impedance of an inductor is  $iwL$  and the impedance of a capacitor is  $1/iwC$ . To determine the impedance of a sequence of circuit elements in series, we simply add up their individual impedances. Two important quantities in electrical engineering are the *magnitude of the impedance* and the *phase angle*. The magnitude is the ratio of the RMS voltage to the RMS current - it equals the magnitude of the complex impedance. The *phase angle* is the amount by which the voltage leads or lags the current - it is the phase of the complex impedance. Program [CircuitRLC.java](#) does a computation involving complex numbers and impedance of circuits with resistors, inductors, and capacitors in series.

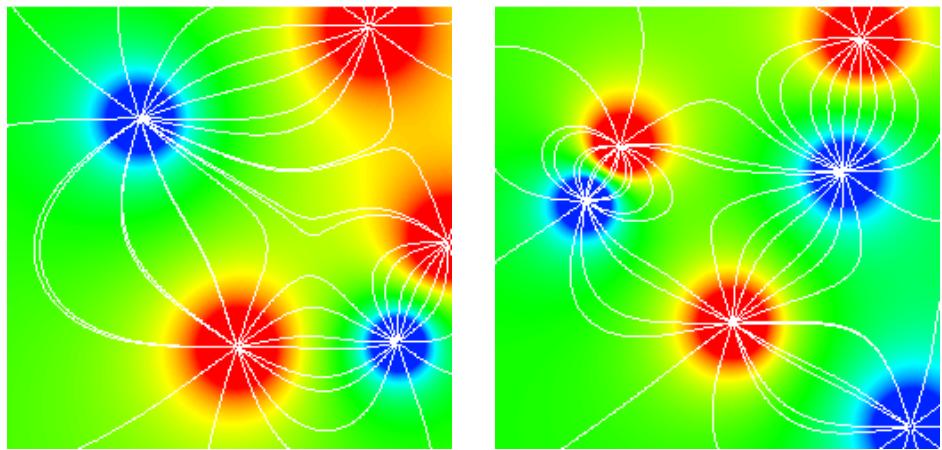
Exercise: RLC circuit in parallel.  $1/Z = 1/Z_1 + 1/Z_2 + \dots + 1/Z_n$ .

Exercise (for objects): repeat series-parallel network for RLC circuits with impedances instead of resistance

33. **Diffusion of particles in a fluid.** Simulate diffusion of particles in a fluid. See [BrownianParticle.java](#) in Section 9.8.
34. **Electric field lines.** Michael Faraday introduced an abstraction called *electric field lines* to visualize the electric field. By Coulombs law, the electric field at a point induced by a point charge  $q_i$  is given by  $E_i = k q_i / r^2$ , and the direction points to  $q_i$  if  $q_i$  is negative and away from  $q_i$  it is negative. If there are a group of  $n$  point charges, the electric field at a point is the *vector sum* of the electric fields induced by the  $n$  individual point charges. We can compute it by summing up the components in the  $x$ - and  $y$ - directions. The figure below illustrates the field lines for two equal positive point charges (left) and two point charges of opposite signs (right). The second configuration is called an *electric dipole*: the charges cancel each other out, and the electric field weakens very quickly as you move away from the charges. Examples of dipoles can be found in molecules where charge is not evenly distributed. Oscillating dipoles can be used to produce electromagnetic waves to transmit radio and television signals.

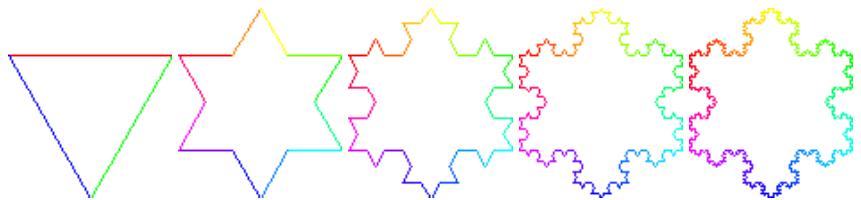


Program [FieldLines.java](#) draws 10 electric field lines coming out of each charge. (We take some liberties since traditionally the number of field lines per unit area should be proportional to the magnitude of the field strength.) Each line starts on a 1-pixel circle around the charge, at twelve equally spaced angles. The electric field at a point  $(x, y)$  from a point charge  $q_i$  is given by  $E_i = k q_i / r^2$ , where  $q_i$  is the magnitude of the charge  $i$  and  $r$  is the radial distance from it. The field due to several charges is the vector sum of the field due to each, and can be found by adding the  $x$ - and  $y$ -components. After calculating the electric field strength, we move in the direction of the vector field and draws a spot. We repeat this process until we reach the boundary of the region or another point charge. The figures below illustrate the electric potential and field lines for several random charges of equal magnitude.

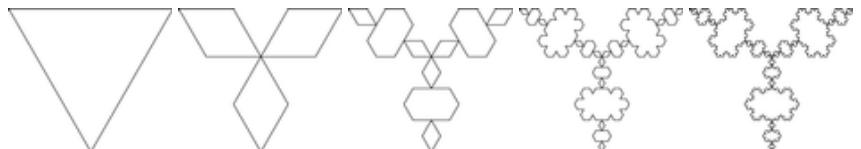


35. **Koch snowflake with rainbow of colors.**

The *Koch snowflake* of order  $n$  consists of three copies of the Koch curve of order  $n$ . We draw the three Koch curves one after the other, but rotate  $120^\circ$  clockwise in between. Below are the Koch snowflakes of order 0, 1, 2, and 3. Write a program [KochRainbow.java](#) that plots the Koch snowflake in a continuous spectrum of colors from red, to orange, yellow, green, blue, and indigo, and violet.



36. **Anti-Koch snowflakes.** The *anti-Koch snowflake* is generated exactly like the Koch snowflake, except that clockwise and counterclockwise are interchanged. Write a program [AntiKoch.java](#) that takes a command line parameter  $N$  and plots the anti-Koch snowflake of order  $N$  using Turtle graphics.



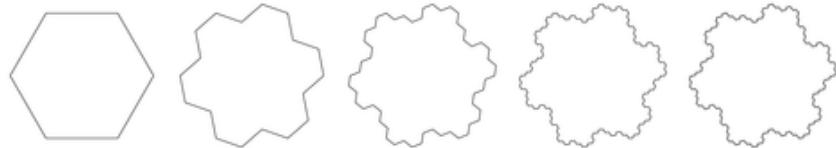
37. **Randomized Koch snowflakes.** A *randomized Koch snowflake* is generated exactly like the Koch snowflake, except that we flip a coin to generate the clockwise and counterclockwise direction at each step.

38. **Turtle graphics.**

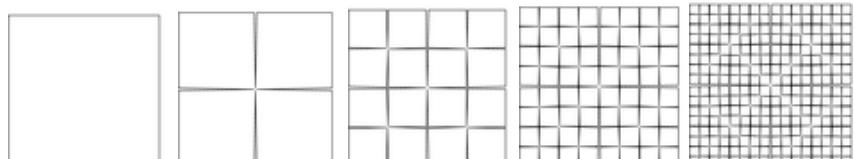
- a. *Minkowski sausage*. ([Sausage.java](#))



- b. *Gosper island*.

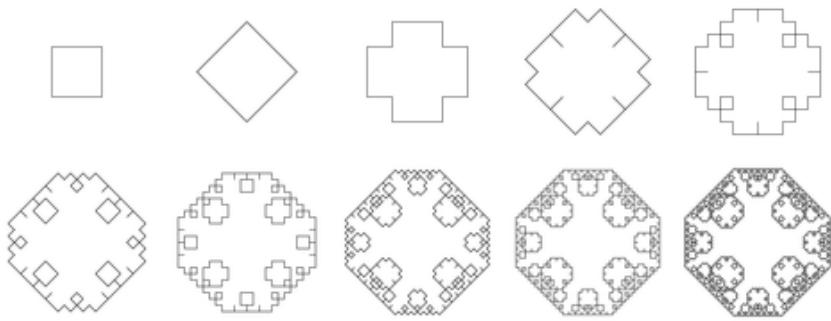


- c. *Cesaro broken square*.

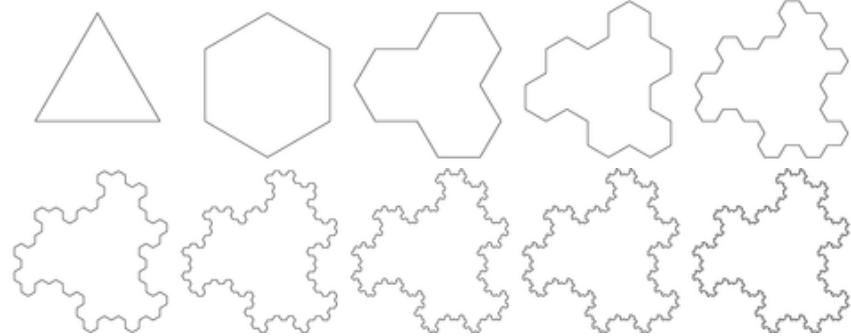


39. **More turtle graphics.** Write a program to produce each of the following recursive patterns.

a. *Levy tapestry.* ([Levy.java](#))

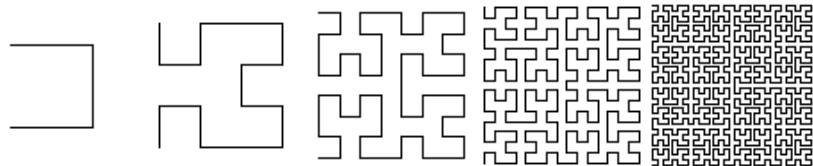


b. *Fudgeflake.*



40. **Turtle graphics (hard).** Write a program to produce each of the following recursive patterns without lifting the pen or tracing over the same line segment more than once.

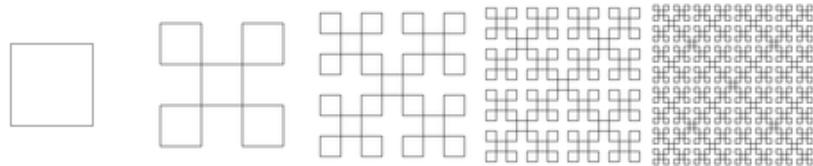
a. *Hilbert space-filling curve.* ([Hilbert.java](#) or [SingleHilbert.java](#)) Informally, a **space-filling curve** is a continuous curve in the unit square that passes through every point. In 1890, Giuseppe Peano discovered the first such space-filling curve. In 1891, David Hilbert discovered a simpler version, which came to be known as the Hilbert curve.



b. *Sierpinski arrowhead.*



c. *Sierpinski curve.*



41. **Dragon curve.** Write a program [Dragon.java](#) that reads in a command-line parameter  $N$  and plots the order  $N$  dragon curve using turtle graphics. The dragon curve was first discovered by three NASA physicists (John E. Heighway, Bruce A. Banks, and William G. Harter) and later popularized by Martin Gardner in *Scientific American* (March and April 1967) and Michael Crichton in *Jurassic Park*.





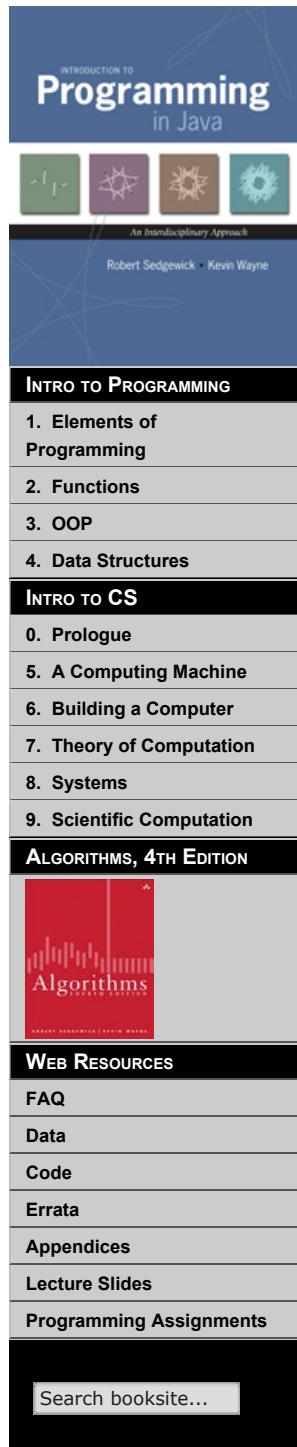
This is a sophisticated program that uses two mutually recursive functions.

Program [SequentialDragon.java](#) is an iterative version of the dragon curve. It is a hacker's paradise.

42. **Mandelbrot trajectory.** Write an interactive program [Trajectory.java](#) that plots the sequence of points in the Mandelbrot iteration in the complex plane. If the user clicks on  $(x, y)$ , plot the sequence of iterates for  $z = x + iy$ .
43. **Faster Mandelbrot.** Speed up Mandelbrot by performing the computation directly instead of using `Complex`. Compare. Incorporate periodicity checking or [boundary tracing](#) for further improvements. Use divide-and-conquer: choose 4 corners of a rectangle and a few random points inside; if they're all the same color, color the whole rectangle that color; otherwise divide into 4 rectangles and recur.
44. **More Complex methods.** Add methods to `Complex.java` to support trigonometric and exponential functions on complex numbers.
  - $\exp(a + ib) = e^a \cos(b) + i e^a \sin(b)$
  - $\sin(a + ib) = \sin(a) \cosh(b) + i \cos(a) \sinh(b)$
  - $\cos(a + ib) = \cos(a) \cosh(b) - i \sin(a) \sinh(b)$
  - $\tan(a + ib) = \sin(a + ib) / \cos(a + ib)$
45. Add a method `power` to `Complex` so that `a.power(b)` returns the result of taking the complex number `a` and taking it to the complex power `b`.
46. What is the principle value of  $i^i$ ?  
*Answer:  $e^{-\pi/2} = 0.207879576\dots$*
47. **Random walker.** Write a data type `RandomWalker` that simulates the motion of a random walker in the plane that starts at the origin and makes a random step (left, right, up, or down) at each step. Include a method `step()` that moves the random walker one step and a method `distance()` that returns the distance the random walker is from the origin. Use this data type to formulate a hypothesis as to how far (as a function of  $N$ ) the random walker is from the origin after  $N$  steps. (See also Exercise 1.x.)
48. *Turtle graphics.* Extend Turtle in various ways. Make `DeluxeTurtle` that adds color, etc. Add a version that supports error checking. For example, throw a `TurtleOutOfBoundsException` exception if turtle goes outside designated boundary.

*Last modified on October 23, 2011.*

Copyright © 2002–2012 Robert Sedgewick and Kevin Wayne. All rights reserved.



### 3.3 DESIGNING DATA TYPES

This section under major construction.

#### Designing data types

- Designing APIs
- Encapsulation (use `private` for information hiding)
- Immutability (use `final` to help enforce)
- Design-by-contract (use `assert` to check pre-conditions and post-conditions)

Theme = limit what you, the programmer, can do. Emphasize encapsulation and immutability.

**Designing APIs.** Often the most important and most challenging step in building software is designing the APIs. In many ways, designing good programs is more challenging than writing the code itself. Takes practice, careful deliberation, and many iterations.

- *Specification problem.* Document the API in English. Clearly articulate behavior for all possible inputs, including side effects. "Write to specification." Difficult problem. Many bugs introduced because programmer didn't correctly understand description of API. See booksite for information on automatic documentation using `Javadoc`.
- *Wide interfaces.* "API should do one thing and do it well." "APIs should be as small as possible, but no smaller." "When in doubt, leave it out." (It's easy to add methods to an existing API, but you can never remove them without breaking existing clients.) APIs with lots of bloat are known as wide interfaces. Supply all necessary operations, but no more. Try to make methods orthogonal in functionality. No need for a method in `Complex` that adds three complex numbers since there is a method that adds two. The `Math` library includes methods for `sin()`, `cos()`, and `tan()`, but not `sec()`.

Java libraries tend to have wide interfaces (some designed by pros, some by committee). Sometimes this seems to be the right thing, e.g., `String`. Although, sometimes you end up with poorly designed APIs that you have to live with forever.

- *Deprecated methods.* Sometimes you end up with *deprecated methods* that are no longer fully supported, but you still need to keep them or break *backward compatibility*. Once Java included a method `Character.isSpace()`, programmers wrote programs that relied on its behavior. Later, they wanted to change the method to support additional Unicode whitespace characters. Can't change the behavior of `isSpace()` or that would break many programs. Instead, add a new method `Character.isWhiteSpace()` and "deprecate" the old method. The API is now more confusing than needed.

Almost all methods in `java.util.Date` are deprecated in favor of `java.util.GregorianCalendar`.

*Backward compatibility.* The need for backward compatibility shapes much of the way things are done today (from operating systems to programming languages to ...). [Insert a story.]

*Standards.* It is easy to understand why writing to an API is so important by considering other domains. Fax machines, radio, MPEG-4, MP3 files, PDF files, HTML, etc. Simpler to use a common standard. Lack of incompatibilities enables business opportunities that would otherwise be impossible. One of the challenges of writing software is making it portable so that it works on a variety of operating systems including Windows, OS X, and Linux. Java Virtual Machine enables portability of Java across platforms.

**Encapsulation.** Designing reusable data types requires great care. The API is a contract between the client and the implementation that specifies the set of permissible operations and *what* is their required behavior. *Encapsulation* is the process of separating and compartmentalizing the client from the implementation by hiding *how* the data type is implemented. This enables you to change the "how" as the program evolves. "A nut and bolt are compatible if they have the same diameter and threads per inch." This is the interface. However, "they may be made of carbon steel, steel, bronze, nylon, titanium, whatever."

Encapsulation enables us to:

- Keep instance variables in a consistent state, e.g., nonnegative integer.
- Do error-checking on instance variables to monitor when they change their value. This can help debugging since you can check when a value unexpectedly changes state since it can only happen by calling a function within that class.
- Reduce dependencies between different modules. This enables us to debug and test one module independently from others.
- Improve resiliency of software systems by limiting and localizing the effects of changing the internal representation of a data type as the program evolves.
- Substitute different implementations of the same data type to improve performance, accuracy, or memory footprint.

For example, although we've been using the `String` data type since our first program, we have not specified *how* Java implements it. This is perfectly fine because we only need to know *what* operations it supports in the API, not the underlying representation.

#### Some mottos.

- *Abstraction and information hiding are nice for small programs and absolutely necessary for big ones.*  
Brian Kernighan.

- *Limit scope.* The tighter the scope, the easier a program is to maintain.
- *Principle of least privilege.* Each part of the program should know as much as it needs, but no more.
- *Ask don't touch.* Client shouldn't directly manipulate the instance variables. Instead it should ask the data type to do the work.

**Zip codes and time bombs.** There are many examples of programmers making assumptions about the internal representation of a data type, and later having to pay a big price to fix their mistakes.

- *ZIP codes.* In 1963, The United States Postal Service (USPS) began using a 5 digit [ZIP code](#) (Zoning Improvement Plan) to improve the sorting and delivery of mail. Programmers wrote lots of software that assumed zip codes would remain 5 digits forever, and represented them in their programs using a single 32-bit integer. In 1983, the USPS introduced an expanded ZIP code called ZIP+4 which consists of the original 5 digit ZIP code plus 4 extra digits to assist in delivery. This broke hundreds of fragile programs and required millions of dollars to fix.
- *Y2K.* The much publicized [Y2K bug](#) resulted from many programmers encoding years using two digits (years since 1900) to save memory (back in the 1960s when saving two characters per date field was more significant). As the year 2000 approached, programmers had to scavenge through millions of lines of code to find all those places that relied on a two digit year. Some of these programs controlled nuclear power plants, financial markets, commercial air traffic, and military warships. The cost for fixing all of these problems was estimated to be around \$300 billion.
- *IPv4 vs. IPv6.* The [Internet protocol](#) is a standard used by electronic devices to exchange data over the Internet. Each device is assigned a unique integer or address. IPv4 uses 32-bit addresses and supports about 4.3 billion addresses. IPv6 uses 128-bit addresses and supports  $2^{128}$  addresses (over 340 decillion!). This will increase network traffic since each network packet contains the source and destination address. The total cost of the transition in the US might reach \$500 billion!
- *Vehicle identification numbers.* In 1981, The Society of Automotive Engineers established a unique 17-character naming scheme for vehicles known as the [Vehicle Identification Number](#) (or VIN for short). The VIN describes the make, model, year, and other attributes of cars, trucks, buses, and other vehicles in the United States. Automakers expect to run out by 2010. One solution is to increase the length of the VIN. Another is to reuse existing VINs. Both would require major logistical changes to supporting software systems, estimates in the billions of dollars. Moral: whenever a unique number is needed, don't skimp!

One of the lessons we should learn from this is to use encapsulated data types. If we must change the data representation, the effects are localized to one ADT and we don't need to search through millions of lines of code to find all of the places where we assumed one particular data representation.

**Encapsulation in Java.** Java provides language support for information hiding. When we declare an instance variable (or method) as `private`, this means that the client (code written in another module) cannot directly access that instance variable (or method). The client can only access the API through the `public` methods and constructors. Programmer can modify the implementation of `private` methods (or use different instance variables) with the comfort that no client will be directly affected.

Program [Counter.java](#) implements a counter, e.g., for an electronic voting machine. It encapsulates a single integer to ensure that it can only get incremented by one at a time and to ensure that it never goes negative. The goal of data abstraction is to *restrict* which operations you can perform. Can ensure that data type value always remains in a consistent state. Can add logging capability to `hit()`, e.g., to print timestamp of each vote. In the 2000 presidential election, Al Gore received negative 16,022 votes on an electronic voting machine in Volusia County, Florida. The counter variable was not properly encapsulated in the voting machine software!

<pre>non-encapsulated data type ----- public class Counter {     public final String name;     public int count;     public Counter(String name) {         count = 0;         this.name = name;     }     public void hit() { count++; }     public int get() { return count; } } client ----- Counter c = new Counter("Volusia"); c.count = -16022; // legal</pre>	<pre>encapsulated data type ----- public class Counter {     private final String name;     private int count;     public Counter(String name) {         count = 0;         this.name = name;     }     public void hit() { count++; }     public int get() { return count; } } client ----- Counter c = new Counter("Volusia"); c.count = -16022; // compile-time error</pre>
---	--

- *Access control.* Java provides a mechanism for *access control* to prevent the use of some variable or method in one part of a program from direct access in another. We have been careful to define all of our instance variables with the `private` access modifier. This means that they cannot be directly accessed from another class, thereby encapsulating the data type. For this reason, we *always* use `private` as the access modifier for our instance variables and recommend that you do the same. If you use `public` then you will greatly limit any opportunity to modify the class over time. Client programs may rely on your `public` variable in thousands of places, and you will not be able to remove it without breaking dependent code.
- *Getters and setters.* A data type should not have `public` instance variables. You should obey this rule not just in letter, but also in spirit. Novice programmers are often tempted to include `get()` and

`set()` methods for each instance variable, to read and write its value.

```
Complex a = new Complex(1.0, 2.0);
Complex b = new Complex(3.0, 4.0);

// violates spirit of encapsulation
Complex c = new Complex(0.0, 0.0);
c.setRe(a.re() + b.re());
c.setIm(a.im() + b.im());

// better design
Complex a = new Complex(1.0, 2.0);
Complex b = new Complex(3.0, 4.0);
Complex c = a.plus(b);
```

The purpose of encapsulation is not just to hide the data, but to *hide design decisions* which are subject to change. In other words, the client should tell an object *what* to do, rather than asking an object about its state (`get()`), making a decision, and then telling it *how* to do it (`set()`). Usually it's better design to not have the `get()` and `set()` methods. When a `get()` method is warranted, try to avoid including a `set()` method.

**Data representation changes in scientific applications.** Simple example: represent a point using Cartesian or polar coordinates. Polynomials (coefficients vs. point-value), matrices (sparse vs. dense).

**Immutability.** An *immutable* data type is a data type such that the value of an object never changes once constructed. Examples: `Complex` and `String`. When you pass a `String` to a method, you don't have to worry about that method changing the sequence of characters in the `String`. On the other hand, when you pass an array to a method, the method is free to change the elements of the array.

Immutable data types have numerous advantages. they are easier to use, harder to misuse, easier to debug code that uses immutable types, easier to guarantee that the class variables remain in a consistent state (since they never change after construction), no need for copy constructor, are thread-safe, work well as keys in symbol table, don't need to be defensively copied when used as an instance variable in another class. Disadvantage: separate object for each value.

Josh Block, a Java API architect, advises that "Classes should be immutable unless there's a very good reason to make them mutable....If a class cannot be made immutable, you should still limit its mutability as much as possible."

Give example where function changes value of some `Complex` object, which leaves the invoking function with a variable whose value it cannot rely upon.

mutable	immutable
Counter	Complex
MovingCharge	Charge
Draw	String
array	Vector
java.util.Date	primitive types
Picture	wrapper types

- *Final.* Java provides language support to enforce immutability. When you declare a variable to be `final`, you are promising to assign it a value only once, either in an initializer or in the constructor. It is a compile-time error to modify the value of a `final` variable.

```
public class Complex {
    private final double re;
    private final double im;

    public Complex(double real, double imag) {
        re = real;
        im = imag;
    }

    // compile-time error
    public void plus(Complex b) {
        re = this.re + b.re;      // oops, overwrites invoking object's value
        im = this.im + b.re;      // compile-time error since re and im are final
        return new Complex(re, im);
    }
}
```

It is good style to use the modifier `final` with instance variables whose values never change.

- Serves as documentation that the value does not change.
- Prevents accidental changes.

- Makes programs easier to debug, since it's easier to keep track of the state: initialized at construction time and never changes.
- *Mutable instance variables.* If the value of a `final` instance variable is mutable, the value of that instance variable (the reference to an object) will never change - it will always refer to the same object. However, the value of the object itself can change. For example, in Java, arrays are mutable objects: if you have an `final` instance variable that is an array, you can't change the array (e.g., to change its length), but you can change the individual array elements.

This creates a potential mutable hole in an otherwise immutable data type. For example, the following implementation of a `Vector` is mutable.

```
public final class Vector {
    private final int N;
    private final double[] coords;

    public Vector(double[] a) {
        N = a.length;
        coords = a;
    }

    ...
}
```

A client program can create a `Vector` by specifying the entries in an array, and then *change* the elements of the `Vector` from (3, 4) to (0, 4) after construction (thereby bypassing the public API).

```
double[] a = { 3.0, 4.0 };
Vector vector = new Vector(a);
StdOut.println(vector.magnitude()); // 5.0
a[0] = 0.0; // bypassing the public API
StdOut.println(vector.magnitude()); // 4.0
```

- *Defensive copy.* To guarantee immutability of a data type that includes an instance variable of a mutable type, we perform a *defensive copy*. By creating a local copy of the array, we ensure that any change the client makes to the original array has no effect on the object.

```
public final class Vector {
    private final int N;
    private final double[] coords;

    public Vector(double[] a) {
        N = a.length;

        // defensive copy
        coords = new double[N];
        for (int i = 0; i < N; i++) {
            coords[i] = a[i];
        }
    }

    ...
}
```

Program `Vector.java` encapsulates an immutable array.

- *Global constants.* The `final` modifier is also widely used to specify local or global constants. For example, the following appears in Java's `Math` library.

```
public static final double E = 2.7182818284590452354;
public static final double PI = 3.14159265358979323846;
```

If the variables were declared `public` and not `final`, a client could wreak havoc by re-assigning `Math.PI = 1.0`; Since `Math.PI` is declared to be `final`, such an attempt would be flagged as a compile-time error.

**Design-by-contract.** Language mechanism that enables you to verify assumptions about your program as it is running. For example, if you have a data type that represents a particle, you might assert that its mass is positive and its speed is less than the speed of light. Or if you have a method to add two vectors of the same length, you might assert that the length of the resulting vector also has the same length. Design-by-contract model of programming.

- *Precondition.* A condition that the client promises to satisfy when calling a method.

- *Postcondition*. A condition that the implementation promises to achieve when returning from a method.
- *Invariant*. A condition that should always be true when a certain statement is executing.
- *Side effects*. Always document the side effects of a method, including I/O and exceptions.

Transition to exceptions and assertions.

- *Exceptions*. An exception is a disruptive event that occurs while a program is running, often to signal an error. Action is known as *throwing an exception*. We have seen several exceptions: StackOverflowException, DivideByZeroException, NullPointerException and ArrayOutOfBoundsException. You can also create your own. The simplest kind is a RuntimeException that terminates execution of the program and print out an error message.

```
throw new RuntimeException("Error message here");
```

- *Assertions*. An *assertion* is a boolean expression that you are affirming is true at that point in the program. If it is not, the program will terminate and report an error message. Assertions are widely used by programmers to detect bugs and gain confidence in the correctness of programs. They also serve to document the programmer's intent.

```
// check that a given boolean condition is true
assert distance >= 0.0;

// with optional detail message
assert distance >= 0.0 : "distance can't be negative";
```

By default, assertions are disabled. To enable, execute with the `-enableassertions` flag (`-ea` for short).

```
% java -ea Date
```

The result will be a message like

```
Exception in thread "main" java.lang.AssertionError at Date.next(Date.java:42)
```

Do *not* use assertions to check arguments in a `public` method: your program should not rely on assertions for normal operation since they may be disabled.

**Date.** A culminating example that illustrates use of `final`, `assert`, exceptions, immutability, and `private` helper methods. Would make a nice anatomy diagram. Also could be a set-up for Comparable in the next section.

Application: commercial transaction system. Want to know 3 (business days) from today.

Exercise: add a method `daysUntil()` and use this to implement `compareTo()` in next section. Add a method `day()` that returns a `String` representing the day of the week (e.g., Sunday, Monday).

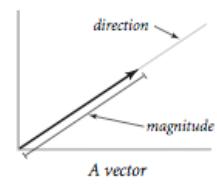
Program `Date.java` is an immutable data type that represents a date. Internally, it uses three integers for the month, day, and year. It includes a constructor, `toString()`, `next()`, `isAfter()`.

By accessing the data only through the interface, our data type can perform *range checking* (e.g., no such thing as January 0 or February 30). For simplicity, we ignore leap years. (See Exercise XYZ.) By declaring all instance variables as `final`, we ensure that they can only be initialized in the constructor. This makes it easy to check the invariant that the values are always in a consistent state. The time invested in performing routine error-checking more than pays off as we start writing more complex programs. Using encapsulation enables us to isolate a few places where an object can change state.

**Vectors.** As second example of a mathematical abstraction, we now consider the *vector* data type. Like Complex, the basic definition of the vector abstraction is familiar because it has played a central role in applied mathematics for over 100 years. The field of mathematics known as *linear algebra* is concerned with properties of vectors. Linear algebra is a rich and successful theory with numerous applications that plays an important role in all fields of social and natural science.

A *spatial vector* is an abstract entity that has a *magnitude* and a *direction*. Vectors provide a natural way to describe properties of the physical world, such as force, velocity, momentum, or acceleration. One standard way to specify a vector is as an arrow from the origin to a point in a Cartesian coordinate system: the direction is the ray from the origin to the point and the magnitude is the length of the arrow (distance from the origin to the point). To specify the vector it suffices to specify the point.

The concept extends to any number of dimensions: an ordered list of  $N$  real numbers (the coordinates of an  $N$ -dimensional point) suffices to specify a vector in  $N$ -dimensional space. By convention, we use a boldface letter to refer to a vector and numbers or indexed variable names (the same letter in italics) separated by



commas within square brackets to denote its value. For example, we might use `x` to denote  $(x[0], x[1], \dots, x[N-1])$  and `y` to denote  $(y[0], y[1], \dots, y[N-1])$ . In Java, this decision implies that we can represent each `Vectors` data type value as an array of `double` values.

The basic operations on vectors are to add two vectors, multiply a vector by a scalar, compute the dot product of two vectors, and to compute the magnitude and direction, as follows:

- addition:  $x + y = (x[0] + y[0], x[1] + y[1], \dots, x[N-1] + y[N-1])$
- scalar multiplication:  $t x = (t x[0], t x[1], \dots, t x[N-1])$
- dot product:  $x \cdot y = x[0]y[0] + x[1]y[1] + \dots + x[N-1]y[N-1]$
- magnitude:  $|x| = (\sum_{i=0}^{N-1} x[i]^2)^{1/2}$
- direction:  $x / |x| = (x[0] / |x|, x[1] / |x|, \dots, x[N-1] / |x|)$

The result of addition and scalar multiplication and the direction are vectors, but the magnitude and the dot product are scalar quantities (real numbers). For example, if  $x = (0, 3, 4, 0)$ , and  $y = (0, -3, 1, -4)$ , then  $x + y = (0, 0, 5, -4)$ ,  $3x = (0, 9, 12, 0)$ ,  $x \cdot y = -5$ ,  $|x| = 5$ , and  $x / |x| = (0, .6, .8, 0)$ . The direction vector is a *unit vector*: its magnitude is 1. These basic mathematical definitions lead immediately to an API:

<code>public class Vector (PROGRAM 3.2.4)</code>	
<code>Vector(double[] a)</code>	
<code>Vector plus(Vector b)</code>	<i>sum of this vector and b</i>
<code>Vector minus(Vector b)</code>	<i>difference of this vector and b</i>
<code>Vector times(double t)</code>	<i>scalar product of this vector and t</i>
<code>double dot(Vector b)</code>	<i>dot product of this vector and b</i>
<code>double magnitude()</code>	<i>magnitude of this vector</i>
<code>Vector direction()</code>	<i>unit vector with same direction as this vector</i>
<code>double cartesian(int i)</code>	<i>ith cartesian coordinate of this vector</i>
<code>String toString()</code>	<i>string representation</i>

Program `Vector.java` is an immutable class that implements this API. Note that the instance variable is not a primitive type, but rather an array.

*The this reference.* The `magnitude()` and `direction()` methods in `Vector` make use of the name `this`. Java provides the `this` keyword to give us a way to refer to an instance method to the object whose name was used to invoke this method. You can use `this` in code in the same way you use any other name. For example, some Java programmers would prefer to code `plus()` in `Complex` as follows:

```
public Complex plus(Complex b) {
    Complex a = this;
    return new Complex(a.re + b.re, a.im + b.im);
}
```

to emphasize that `a` and `b` play an equal role in the operation. Some Java programmers always use `this` to refer to instance variables. Their scope is so large (the whole class) that this policy is easy to defend. However, it does result in a surfeit of `this` keywords, so we take the opposite tack and use `this` sparingly in our code. If a method has no parameters, or parameters whose names do not conflict with instance variable names, we normally refer directly to instance variables instead of using `this`.

**Similarity search.** Reference: [Gauging Similarity via N-Grams](#). `CompareAll.java` performs similarity search. It relies on the data type `Document.java` that uses `hashCode()`.

## Q + A

**Q.** What happens if I try to access a private instance variable or method from another file?

**A.** A compile-time error that says the given instance variable or method has `private` access in the given class.

**Q.** Does Java have other access modifiers besides `public` and `private`? Yes, although we restrict attention in this book to `public` and `private`. Here's a table.

Access modifier	Class	Package	Subclass	Any Class
<code>private</code>	Y	N	N	N
[no specifier]	Y	Y	N	N
<code>protected</code>	Y	Y	Y	N
<code>public</code>	Y	Y	Y	Y

**Q.** The instance variable `x` in type `Complex` is declared with access modifier `private`. How come in the `plus` method I can access both `this.x` and `b.x`?

**A.** The granularity of `private` access is at the class level, not the instance level. Declaring an instance

variable as `private` means that it is not directly accessible from any other class. Methods within the `Complex` class can access (read or write) the instance variables of any instance in that class. It might be nice to have a more restrictive access modifier, say `superprivate`, that would make the granularity at the instance level so that only the invoking object can access its instance variables.

Q. How would I add a constructor to the `Complex` data type that takes `r` and `theta` as arguments?

A. It's a problem since you already have one constructor that takes two real arguments. Instead, you should probably create two methods `createRect(x, y)` and `createPolar(r, theta)` that create and return new objects. This style will also help remind the client to pass in arguments of the correct form.

Q. Is it possible to defeat Java's type safety features?

A. If there are no bugs in the Java implementation, an untrusted program should not have access to the private data of a trusted program. This guarantee assumes that "the computer faithfully executes its specified instruction set." If a cosmic ray strikes the computer's memory, it can flip a bit, thereby invalidating this axiom. [Appel and Govindavajhala](#) showed how to craft a program so that when executed, any single bit error would yield a 70% chance of the program gaining unfettered access to the JVM. They also demonstrated that such errors could be induced by shining a 50 watt spotlight bulb at the memory for one minute. Smartcards are particularly susceptible since the attacker has physical access to the computer.

Also reflection can defeat the type safety features, but this requires that the user permit such actions.

## Exercises

1. Modify `Date` to handle leap years. *Hint:* change 28 to 29, add a helper method `isLeapYear()`, and add one line to `next()`.
2. Add a method `daysUntil()` to `Date` that takes a `Date` `b` as an argument and returns the number of days between `b` and the invoking date.
3. Create an implementation `Date2.java` that represents a date as a single integer that counts the number of days since January 1, 1970. Compare to `Date.java`.
4. Represent a point in time by using an `int` to store the number of seconds since January 1, 1970. When will programs that use this representation face a time bomb? How should you proceed when that happens?
5. Create a data type `GeographicCoordinate` that represents a geographic coordinate either in (degrees, minutes, seconds, sign) or in floating point.
6. Create a data type `Location` for dealing with locations on Earth using spherical coordinates (latitude/longitude). Include methods to generate a random location on the surface of the Earth, parse a location "25.344 N, 63.5532 W" and compute the great circle distance between two locations.
7. Create a `Rectangle` ADT that represents a rectangle. Represent a rectangle by two points. Include a constructor, a `toString` method, a method for computing the area, and a method for drawing using our graphics library.
8. Repeat the previous exercise, but this time represent a `Rectangle` as the lower left endpoint and the width and height.
9. Repeat the previous exercise, but this time represent a `Rectangle` as the center and the width and height.
10. **Particle.** Create a data type for a 3D particle with position (`rx, ry, rz`), mass (`m`), and velocity (`vx, vy, vz`). Include a method to return its kinetic energy =  $1/2 m (vx^2 + vy^2 + vz^2)$ .  
Alternate representation: position (`rx, ry, rz`), mass (`m`), and momentum (`px, py, pz`). In this case  $vx = px / mass$ ,  $vy = py / mass$ ,  $vz = pz / mass$ . kinetic energy =  $1/2 (px^2 + py^2 + pz^2) / mass$ .
11. **Generating random numbers.** Different methods to generate a random number from the standard Gaussian distribution. Here, encapsulation enables us to replace one version with another that is more accurate or efficient. Trigonometric method is simple, but may be slow due to calling several transcendental functions. More importantly, it suffers from numerical stability problems when `x1` is close to 0. Better method is alternate form of Box-Muller method. [reference](#). Both methods require two values from a uniform distribution and produce two values from the Gaussian distribution with mean 0 and standard deviation 1. Can save work by remembering the second value for the next call. (This is how it is implemented in `java.util.Random`.) Their implementation is the polar method of Box-Muller, saving the second random number for a subsequent call. (See Knuth, ACP, Section 3.4.1 Algorithm C.)
12. **LAX airport shutdown.** On September 14, 2004 Los Angeles airport was [shut down](#) due to software breakdown of a radio system used by air traffic controllers to communicate with pilots. The program used a Windows API function call `GetTickCount()` which returns the number of milliseconds since the system was last rebooted. The value is returned as a 32 bit integer, so after approximately 49.7 days it "wraps around." The software developers were aware of the bug, and instituted a policy that a technician would reboot the machine every month so that it would never exceed 31 days of uptime. Oops. LA Times blamed the technician, but the developers are more to blame for shoddy design.

## Creative Exercises

1. International bank account (`depositDollars`, `depositEuros`, `withdrawDollars`, `withdrawEuros`). How to represent money (cents, `BigDecimal`, precision, etc.)
2. **Turtle graphics.** Turtle location starting at origin (forward, rotate, draw) Use polar and Cartesian coordinates.
3. **Polar representation of points.** `Point.java` and `PointPolar.java` implement the following point

interface using rectangular and polar coordinates, respectively.

```
Point()
Point(double, double)
double x()
double y()
double r()
double theta()
double distance(Point)
public String toString()
```

4. **Spherical coordinates.** Represent a point in 3D space using Cartesian coordinates  $(x, y, z)$  or spherical coordinates  $(r, \theta, \phi)$ . To convert from one to the other, use
$$\begin{aligned} r &= \sqrt{x^2 + y^2 + z^2} & x &= r \cos \theta \sin \phi \\ \theta &= \tan^{-1}(y/x) & y &= r \sin \theta \sin \phi \\ \phi &= \cos^{-1}(z/r) & z &= r \cos \phi \end{aligned}$$
5. **Colors.** Can represent in RGB, CMYK, or HSV formats. Natural to have different implementations of same interface.
6. **Time.** Implement an ADT that represents the time of day, say with seconds, minutes, and hours. Representation 1: three fields for hours, minutes, and seconds. Representation 2: number of seconds since 12 midnight. Include `compareTo`, plus.
7. **ZIP codes.** Implement an ADT that represents a USPS ZIP code. Support both the original 5 digit format and the newer (but optional) ZIP+4 format.
8. **VIN number.** Read in a [VIN number](#) and report back all relevant information.
9. **Vector fields.** Write a data type for force vectors in two dimensions. You should be able to add vectors using the usual rules from physics using  $x \sin(\theta)$  and  $x \cos(\theta)$ ....
10. **Complex numbers.** In Section 3.2, we considered program [Complex.java](#), which is an ADT for complex numbers. Internally, it stores a complex number  $c = x + iy$  as its real and imaginary parts  $(x, y)$ . This is called the *rectangular representation*. Write a new implementation that represents a complex number as  $c = r(\cos \theta + i \sin \theta)$ . This is known as the *polar representation*, and we refer to  $r$  as the *modulus* and  $\theta$  as the *angle*. This representation is especially useful if we are multiplying or dividing complex numbers. Program [Complex.java](#) implements the same complex number data type as [Complex.java](#), but stores the polar representation  $(r, \theta)$  instead of the the rectangular representation  $(x, y)$ . Since it has exactly the same public methods and public constructor as the version with rectangular coordinates, we can use it interchangeably with the other version of [Complex.java](#). The main difference to the client is in different performance properties. For example, implementing an exponentiation method becomes much easier with the polar representation, especially if the power is a negative integer or if the power itself is complex! Other operations become more cumbersome to implement and expensive to compute, e.g., addition and subtraction. There is one minor caveat that makes these two implementations not completely interchangeable: floating point precision may influence the two implementations in different ways. For example, the rectangular implementation represents  $3 + 4i$  exactly, whereas the polar implementation must store an approximation to the angle  $\theta$  since it is an irrational number. Such difference should not effect a well-behaved client, but we should be aware of any places where an encapsulated data type leaks information.
11. **Statistics.** Encapsulation enables us to easily replace one implementation with another. This is desirable when we anticipate creating an improved implementation with better performance or accuracy properties. We consider a simple, but representative example below. Our goal is to store statistics for a set of real values, say the sample mean  $\bar{x}$  and the sample variance  $s^2$ .

$$\begin{aligned} \bar{x} &= \frac{1}{n} \sum_i x_i \\ s^2 &= \frac{\sum_i (x_i - \mu)^2}{n-1} = \frac{n \sum_i x_i^2 - (\sum_i x_i)^2}{n(n-1)} \end{aligned}$$

We want our data type to support the following methods: `add`, `size`, `mean`, `stddev`, `variance`. [OnePass.java](#) stores  $n$ , the sum, and the sum of squares. [TwoPass.java](#) stores  $n$ , the sum, and all of the data values in an array. The one pass algorithm is faster and uses substantially less space, but is susceptible to roundoff error. [StableOnePass.java](#) is a well-engineered alternative.

$$\begin{aligned} m_0 &= 0 \\ s_0 &= 0 \\ m_n &= m_{n-1} + \frac{1}{n} (x_n - m_{n-1}) \\ s_n &= s_{n-1} + \frac{n-1}{n} (x_n - m_{n-1})^2 \\ \bar{x} &= m_n \\ s^2 &= \frac{1}{n-1} s_n \end{aligned}$$

It is numerically stable and does not require an array to store the elements.

```

public void add(double x) {
    n++;
    s = s + 1.0 * (n-1) / n * (x - m) * (x - m);
    m = m + (x - m) / n;
}

public double mean()      { return m;      }
public double variance() { return s / (n - 1); }

```

12. **Encapsulation.** Why does the following break encapsulation, even though all instance variables are declared `private`.

```

public class Appointment {
    private Date date;
    private String customer;
    public Appointment(Date date) {
        // check that date is in some legal range
        this.date = date;
    }
    public Date getDate() { return date; }

```

**Answer:** The reason is that the class `Date` is mutable. The method `setDate(seconds)` changes the value of the invoking date to the number of milliseconds since January 1, 1970, 00:00:00 GMT. This has the unfortunate consequence that when the function `d = getDate()` returns the date, the client program can invoke `d.setDate()` and change the date in an `Appointment` object type, perhaps setting it to an illegal value for a member of `Appointment`. Must not let references to mutable objects escape since caller can then modify its state. One solution is to create a *defensive copy* of the `Date` before returning it using `new Date(date.getTime())`; also need to do a defensive copy when storing it via `this.date = new Date(date.getTime())`. Many programmers regard the mutability of `Date` as a design flaw. (`GregorianCalendar` is a more modern Java library for storing dates; but it is mutable too.)

13. **Sparse vector.** Create a data type for sparse vectors. Represent a sparse vector by an array of indices (of nonzeros) and a parallel array of the corresponding nonzero values. Assume the indices are in ascending order. Implement the dot product operation.
14. **Genome.** Implement a data type to store the genome of an organism. Biologists often abstract away the genome to a sequence of nucleotides (A, C, G, or T). The data type should support the method `addNucleotide`, `nucleotideAt(int i)`, and `doSomeComputation`. Perhaps change to `addCodon`. Advantages of encapsulation: can check that only legal nucleotides are added, can change to more time or memory efficient implementation without affecting client.
- `StringGenome.java` has one instance variable of type `String`. It implements `addNucleotide` with string concatenation. Each method call takes time proportional to the size of the current genome. Not practical spacewise either for large genomes since nucleotide is stored as a 16-bit `char`.
  - `Genome.java` implements a genome as an array of characters. The size of the array is doubled when the array fills up. The method `addNucleotide` is now constant time. Space consumption is still 16 bits per nucleotide.
  - `CompactGenome.java` implements a genome as boolean array. We need to use two bits per nucleotide since there are 4 different nucleotides. As in the previous implementation, we use a dynamic array with repeated doubling. Now, each nucleotide consumes 2 bits of storage (instead of 16).
15. Given a `Genome` Include a method to return the reverse-complemented genome. Include a method for testing for equality.
16. Add methods to `Date` to check which season (Spring, Summer, Fall, Winter) or astrological sign (Pisces, Libra, ...) a given date lies. Be careful about events that span December to January.
17. **Copy constructor.** Only needed if data type is mutable. Otherwise, assignment statement works as desired.

```

public Counter(Counter x) {
    count = x.count;
}

Counter counter1 = new Counter();
counter1.hit();
counter1.hit();
counter1.hit();
Counter counter2 = new Counter(counter1);
counter2.hit();
counter2.hit();
StdOut.println(counter1.get() + " " + counter2.get());    // 3 5

```

practical problem. The design is limited by resource constraints including time, money, usability, flexibility for future modification, maintainability, safety, esthetics, marketing, and ethics. The analysis models how well the design will perform and whether it will be the project constraints. Prototyping, testing, simulation are all components of a proper analysis. Designing quality software is similar in many ways to designing bridges. "A good scientist is a person with original ideas. A good engineer is a person who makes a design that works with as few original ideas as possible." -- Freeman Dyson. When programming, use the principle of least surprise.

**Software engineering.** Programming is a form of engineering: apply computer science to the benefit of humankind. Unlike other forms of engineering (which have been around for hundreds of years), software engineering is very young (~30 years) and no widespread consensus on best practice and engineering standards. Nevertheless, some key ideas have emerged (modular programming, data abstraction, encapsulation). OOP is one of the main tools for building complex programs. We will also consider other software engineering aspects that complement OOP including specification, design, coding, and testing. a computer program to solve a problem. Testing, unit tests (testing smallest compilable element in isolation), integration tests (testing several interacting modules), functional tests tests, regression testing... Test characteristics: functional (does it do what it should do, and not do what it should not), performance, usability, security, etc.) Test detail: unit test (one xxx), integration (several interacting modules), system (the whole thing). Test access: black box (treat module as a black box), white box (use internal structure to determine test cases). Each test should return true or false. Specification is often the biggest source of errors. Compile-time, run-time, and logic errors. Compiler error messages can be cryptic, but usually can be resolved after a few minutes. On the other hand logic errors are the bane of programming. Can take hours to days to discover the error, even if it was just a simple typo. We should be thankful of all the errors that the compiler discovers, and it is a blessing to use a language and compiler that facilitate discovering errors at compile time.

In practice, companies hire employees to solely perform testing. Ratio of testers to programmers can range from 10:1 to 1:10.

Fred Brooks hypothesizes that the number of lines of code a programmer writes per day is about the same, regardless of which language they use (Java, C, machine language). One of main goals of using a higher level programming language (e.g., Java) and OOP is so that we can write 10 lines of Java code that replaces 1,000 lines of machine language code. Another goal is to write programs that are easier to debug than machine language programs. OOP helps address both goals.

**Software life-cycle.** Analysis and specification, design, construction, testing, operation, maintenance. Waterfall model: do all of the tasks in sequence, reaching decisive milestones along the way. (Inflexible, especially if design is wrong.) Prototype model: build quick prototype, evaluate, and redesign if needed. (harder to manage and throw away previous designs)

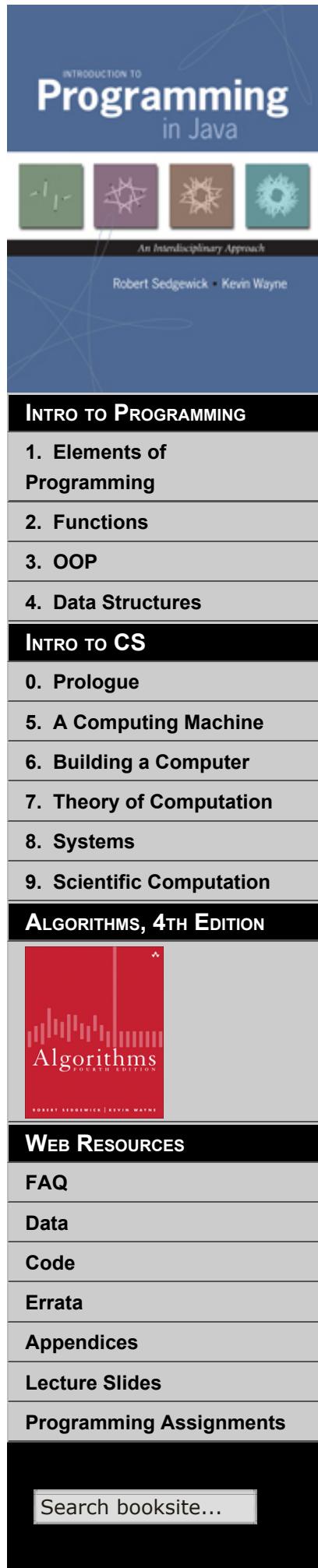
**Testing.** Design good test cases (especially at boundaries), test coverage (use computer to generate test inputs), unit tests, regression testing. Program should always either complete successfully or output a useful error message. Bugs in computer-aided design (CAD) program nuclear reactor control system, guided missile, spacecraft environmental control can be disastrous. Validation = checking that the software does what the customer specified (producing the right product). Verification = checking that the software has been written according to design spec (producing the product right).

Often, it's convenient to write the tests *before* the code. If you can't write the tests, you might need to change the interface or specs. If you still can't, you need to spend more time thinking before you begin to write code.

**Bugs.** CMU study of 30,000 programs: 5-6 defects per 1,000 lines of code in production software. New programs: 100 defects per 1,000 lines of code. \$60 million bug (misused break statement in C program) in [ATT long distance system](#) - 114 switching nodes crashed on January 15, 1990 [collection of software bugs](#).

*Last modified on August 30, 2011.*

Copyright © 2002–2012 [Robert Sedgewick](#) and [Kevin Wayne](#). All rights reserved.



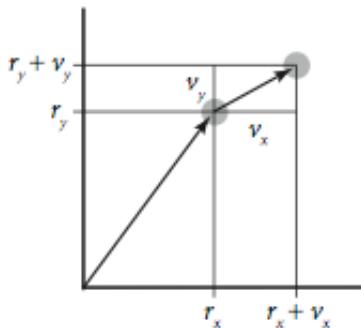
## 3.4 CASE STUDY: N-BODY SIMULATION

This section under major construction.

**Bouncing balls.** The data type [Ball.java](#) represents a ball that moves with a fixed velocity in the box with coordinates between  $-1$  and  $+1$ . When it collides with the boundary, it bounces according to the law of elastic collision. Each ball has a position  $(rx, ry)$  and a velocity  $(vx, vy)$ . The client program [BouncingBalls.java](#) takes a command-line argument  $N$  and creates  $N$  bouncing balls. ([ColoredBall.java](#) and [BouncingColoredBalls.java](#) are variants that associate a color with each ball.)

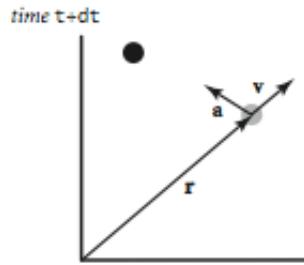
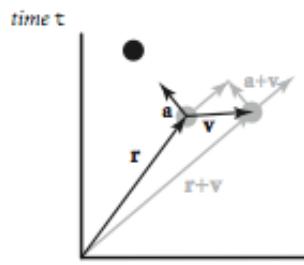
**N-body simulation.** The bouncing ball simulation is based on *Newton's first law of motion*: a body in motion remains in motion at the same velocity unless acted on by an outside force. Embellishing that example to include *Newton's second law of motion* (which explains how outside forces affect velocity) leads us to a basic problem that has fascinated scientists for ages. Given a system of  $N$  bodies, mutually affected by gravitational forces, the problem is to describe their motion.

The data type [Body.java](#) represents a body with a given position  $(rx, ry)$ , velocity  $(vx, vy)$ , and mass  $(m)$ : it uses the [Vector.java](#) data type to represent displacement, velocity, and force as vector quantities.



Adding vectors to move a ball

Search booksite...



### *Motion near a stationary body*

The client program [Universe.java](#) takes a command-line parameter `dt`, reads in a universe from standard input, and simulates the universe using time quantum `dt`. The data file [body4.txt](#) represents a 4-body system.

<code>4</code> ← <i>N</i>	<code>5.000e10</code> ← <i>radius</i>	<i>velocity</i>	<i>mass</i>
		<code>-3.500e10 0.000e00 0.000e00</code> ↓	<code>1.400e03 3.000e28</code>
		<code>-1.000e10 0.000e00 0.000e00</code> ↓	<code>1.400e04 3.000e28</code>
		<code>1.000e10 0.000e00 0.000e00</code> ↓	<code>-1.400e04 3.000e28</code>
		<code>3.500e10 0.000e00 0.000e00</code> ↓	<code>0.000e00 -1.400e03 3.000e28</code>
		↑ <i>position</i>	

*Data file for a 4-body universe*

```
java Universe 25000 < twins.txt
```

100 steps



500 steps



1000 steps



3000 steps



*Simulating a 4-body universe*

## Q + A

### Exercises

1. Write an object-oriented version [Ball.java](#) off [BouncingBall.java](#). Include a constructor that starts each ball moving a random direction at a random velocity (within reasonable limits) and a test client [BouncingBalls.java](#) that takes an integer  $N$  from the command line and simulates the motion of  $N$  bouncing balls.
2. What happens in a universe where Newton's second law does not apply? This situation would correspond to `forceTo()` in `Body` always returning the zero vector.
3. Write a client `Universe3D` and develop a data file to simulate the motion of the planets in our solar system around the sun.

4. Modify `Universe` so that its constructor takes an `In` object and a `Draw` object as arguments. Write a test client that simulates the motion of two different universes (defined by two different files and appearing in two different `Draw` windows). You also need to modify the `draw()` method in `Body`.
5. Implement a class `RandomBody` that initializes its instance variables with (carefully chosen) random values instead of using a constructor and a client `RandomUniverse` that takes a single parameter `N` from the command line and simulates motion in a random universe with `N` bodies.

## Creative Exercises

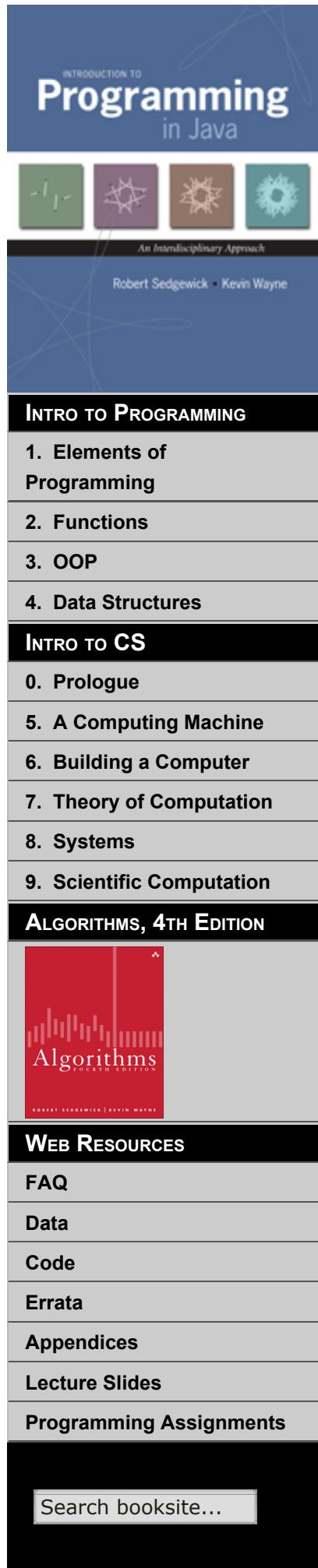
1. **New universe.** Design a new universe with interesting properties and simulate its motion with `Universe`. This exercise is truly an opportunity to be creative!

## Web Exercises

1. **Generative music based on a gravity simulator.** Generate music based on an N-body simulation where bodies makes notes when they collide. Simran Gleason's [website](#) describes the process and includes example videos.

*Last modified on August 05, 2011.*

Copyright © 2002–2012 [Robert Sedgewick](#) and [Kevin Wayne](#). All rights reserved.



## 4. ALGORITHMS AND DATA STRUCTURES

**Overview.** In this chapter we describe and implement some of the most important algorithms and data structures in use on computers today. (For a more in-depth treatment, we recommend the companion textbook [Algorithms, 4th Edition](#).) We begin by considering a powerful framework for measuring and analyzing the efficiency of our programs. This enables us to compare algorithms and accurately predict performance. Next, we consider several novel algorithms for the classic problem of sorting. Then, we build the most important higher level data structures, including stacks, queues, and symbol tables.

- [4.1 Performance](#) outlines a scientific method and powerful theory for understanding the performance and resource consumption of the program that we write.
- [4.2 Sorting and Searching](#) describes two classical algorithms - mergesort and binary search - along with several applications where their efficiency plays a critical role.
- [4.3 Stacks and Queues](#) introduces two closely related data structures for manipulating arbitrary large collections of data.
- [4.4 Symbol Tables](#) considers a quintessential data structure known as the symbol table for storing information, and an efficient implementation known as the binary search tree.
- [4.5 Small World Phenomenon](#) presents a case study to investigate the small world phenomenon - the principle that we are all linked by short chains of acquaintances.

**Java programs in this chapter.** Below is a list of Java programs in this chapter. Click on the program name to access the Java code; click on the reference number for a brief description; read the textbook for a full discussion.

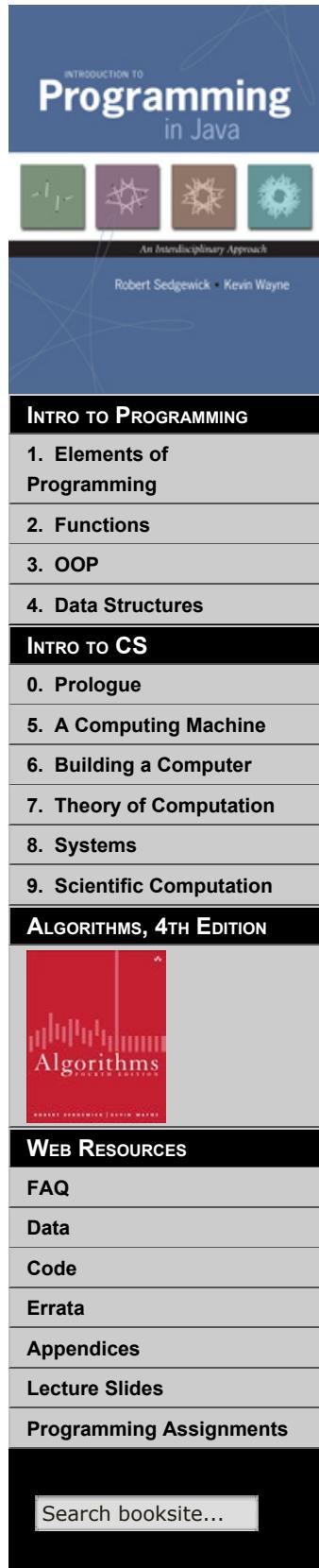
REF	PROGRAM	DESCRIPTION
4.1.2	<a href="#">DoublingTest.java</a>	validating a doubling hypothesis
4.2.2	<a href="#">Gaussian.java</a>	bisection search (function inversion)
4.2.4	<a href="#">Insertion.java</a>	insertion sort
4.2.6	<a href="#">Merge.java</a>	mergesort
4.2.8	<a href="#">LRS.java</a>	longest repeated substring
4.3.2	<a href="#">LinkedStackOfStrings.java</a>	stack of strings (linked list)
4.3.4	<a href="#">Stack.java</a>	stack data type
4.3.6	<a href="#">Queue.java</a>	queue data type
4.3.8	<a href="#">LoadBalance.java</a>	load balancing simulation
4.4.2	<a href="#">Index.java</a>	indexing
4.4.4	<a href="#">DeDup.java</a>	dedup filter

4.5.2	<a href="#">IndexGraph.java</a>	using a graph to invert an index
4.5.4	<a href="#">PathFinder.java</a>	shortest-paths implementation

**Exercises.** Include a link.

*Last modified on August 05, 2011.*

Copyright © 2002–2012 [Robert Sedgewick](#) and [Kevin Wayne](#). All rights reserved.



## 4.1 ANALYSIS OF ALGORITHMS

This section under construction.

In this section, you will learn to respect a principle whenever you program: *Pay attention to the cost*. To study the cost of running them, we study our programs themselves via the *scientific method*, the commonly accepted body of techniques universally used by scientists to develop knowledge about the natural world. We also apply mathematical analysis to derive concise models of the cost.

**Scientific method.** Our approach is the *scientific method*, and it involves the following 5 step approach.

- *Observe* some feature of the natural world.
- *Hypothesize* a model that is consistent with the observations.
- *Predict* events using the hypothesis.
- *Verify* the predictions by making further observations.
- *Validate* by repeating until the hypothesis and observations agree.

One of the key tenets of the scientific method is that the experiments we design must be *reproducible*, so that others can convince themselves of the validity of the hypothesis. In addition, the hypotheses we formulate must be *falsifiable*, so that we can know for sure when a hypothesis is wrong (and thus needs revision).

**Observations.** Our first challenge is to make quantitative measurements of the running time of our programs. Although measuring the exact running time of our program is difficult, usually we are happy with approximate estimates. There are a number of tools available to help us make quantitative measurements of the running time of our programs. Perhaps the simplest is a physical stopwatch or the [Stopwatch.java](#) data type (from Section 3.2). We can simply run a program on various inputs, measuring the amount of time to process each input.

Our first qualitative observation about most programs is that there is a *problem size* that characterizes the difficulty of the computational task. Normally, the problem size is either the size of the input or the value of a command-line argument. Intuitively, the running time should increase with the problem size, but the question of *how much* it increases naturally arises every time we develop and run a program.

**A concrete example.** To illustrate the approach, we start with [ThreeSum.java](#) which counts the number of triples in a set of  $N$  numbers that sums to 0. What is the relationship between the problem size  $N$  and running time for `ThreeSum`?

- *Doubling hypothesis.* For a great many programs, we can quickly formulate a hypothesis for the following question: *What is the effect on the running time of doubling the size of the input?*
- *Empirical analysis.* One simple way to develop a doubling hypothesis is to double the size of the input and observe the effect on the running time. [DoublingTest.java](#) generates a sequence of random input arrays for `ThreeSum`, doubling the array size at each step, and prints the ratio of running times of `ThreeSum.count()` for each input over the previous (which was one-half the size). If you run this program, you will find that the elapsed time increases by about a factor of eight to print each line. This leads immediately to the hypothesis that the running time increases by a factor of eight when the input size doubles. We might also plot the running times, either on a standard plot (left), which clearly shows that the rate of increase of the running time increases with input size, or on a log-log plot. The log-log plot is a straight line with slope 3, clearly suggesting the hypothesis that the running time satisfies a power law of the form  $cN^3$ .
- *Mathematical analysis.* The total running time is determined by two primary factors:
  - The cost of executing each statement.
  - The frequency of execution of each statement.

The former is a property of the system, and the latter is a property of the algorithm. If we know both for all instructions in the program, we can multiply them together and sum for all instructions in the program to get the running time. The primary challenge is to determine the frequency of execution of the statements. Some statements are easy to analyze: for example, the statement that sets `cnt` to 0 in `ThreeSum.count()` is executed only once. Others require

higher-level reasoning: for example, the `if` statement in `ThreeSum.count()` is executed precisely  $N(N-1)(N-2)/6$  times (See Exercise 4.1.4).

**Tilde notation.** We use *tilde notation* to develop simpler approximate expressions. First, we work with the *leading term* of mathematical expressions by using a mathematical device known as the tilde notation. We write  $\sim f(N)$  to represent any quantity that, when divided by  $f(N)$ , approaches 1 as  $N$  grows. We also write  $g(N) \sim f(N)$  to indicate that  $g(N) / f(N)$  approaches 1 as  $N$  grows. With this notation, we can ignore complicated parts of an expression that represent small values. For example, the `if` statement in `ThreeSum.count()` is executed  $\sim N^3 / 6$  times because  $N(N-1)(N-2) / 6 = N^3/6 - N^2/2 + N/3$ , which, when divided by  $N^3/6$ , approaches 1 as  $N$  grows.

We focus on the instructions that are executed most frequently, sometimes referred to as the *inner loop* of the program. In this program it is reasonable to assume that the time devoted to the instructions outside the inner loop is relatively insignificant.

**Order of growth.** The key point in analyzing the running time of a program is this: for a great many programs, the running time satisfies the relationship  $T(N) \sim c f(N)$  where  $c$  is a constant and  $f(N)$  a function known as the *order of growth* of the running time. For typical programs,  $f(N)$  is a function such as  $\log N$ ,  $N$ ,  $N \log N$ ,  $N^2$ , or  $N^3$ .

The order of growth of the running time of `ThreeSum` is  $N^3$ . The value of the constant  $c$  depends both on the cost of executing instructions and on details of the frequency analysis, but we normally do not need to work out the value. Knowing the order of growth typically leads immediately to a doubling hypothesis. In the case of `ThreeSum`, knowing that the order of growth is  $N^3$  tells us to expect the running time to increase by a factor of eight when we double the size of the problem because

$$T(2N) / T(N) \rightarrow c (2N)^3 / c (N)^3 = 8$$

**Order of growth classifications.** We use just a few structural primitives (statements, conditionals, loops, and method calls) to build Java programs, so very often the order of growth of our programs is one of just a few functions of the problem size, summarized in the table below.

Complexity	Description	Examples
1	<i>Constant</i> algorithm does not depend on the input size. Execute one instruction a fixed number of times	Arithmetic operations (+, -, *, /, %) Comparison operators (<, >, ==, !=) Variable declaration Assignment statement Invoking a method or function
$\log N$	<i>Logarithmic</i> algorithm gets slightly slower as $N$ grows. Whenever $N$ doubles, the running time increases by a constant.	Bits in binary representation of $N$ Binary search Insert, delete into heap or BST
$N$	<i>Linear</i> algorithm is optimal if you need to process $N$ inputs. Whenever $N$ doubles, then so does the running time.	Iterate over $N$ elements Allocate array of size $N$ Concatenate two string of length $N$
$N \log N$	<i>Linearithmic</i> algorithm scales to huge problems. Whenever $N$ doubles, the running time more (but not much more) than doubles.	Quicksort Mergesort FFT
$N^2$	<i>Quadratic</i> algorithm practical for use only on relatively small problems. Whenever $N$ doubles, the running time increases fourfold.	All pairs of $N$ elements Allocate $N$ -by- $N$ array
$N^3$	<i>Cubic</i> algorithm is practical for use on only small problems. Whenever $N$ doubles, the running time increases eightfold.	All triples of $N$ elements $N$ -by- $N$ matrix multiplication
$2^N$	<i>Exponential</i> algorithm is not usually appropriate for practical use. Whenever $N$ doubles, the running time squares!	Number of $N$ -bit integers All subsets of $N$ elements Discs moved in Towers of Hanoi

N!	<i>Factorial</i> algorithm is worse than exponential. Whenever N increases by 1, the running time increases by a factor of N	All permutations of N elements
----	--	--------------------------------

**Estimating memory usage.** To pay attention to the cost, you need to be aware of memory usage. You probably are aware of limits on memory usage on your computer (even more so than for time) because you probably have paid extra money to get more memory. Memory usage is well-defined for Java on your computer (every value will require precisely the same amount of memory each time that you run your program), but Java is implemented on a very wide range of computational devices, and memory consumption is implementation-dependent. For primitive types, it is not difficult to estimate memory usage: We can count up the number of variables and weight them by the number of bytes according to their type.

type	bytes	type	bytes	type	bytes
boolean	1	byte[]	16 + N	object reference	4
byte	1	boolean[]	16 + N	String	40 + 2N
char	2	char[]	16 + 2N	Charge	32
int	4	int[]	16 + 4N	Charge[]	36N + 16
float	4	double[]	16 + 8N	Complex	24
long	8	int[][]	4N^2 + 20N + 16	Color	12
double	8	double[][]	8N^2 + 20N + 16		

- *Primitive types.* For example, since the Java `int` data type is the set of integer values between -2,147,483,648 and 2,147,483,647, a grand total of  $2^{32}$  different values, it is reasonable to expect implementations to use 32 bits to represent `int` values.
- *Objects.* To determine the memory consumption of an object, we add the amount of memory used by each instance variable to the overhead associated with each object, typically 8 bytes. For example, a `Charge.java` object uses 32 bytes (8 bytes of overhead, plus 8 bytes for each of its three `double` instance variables). A reference to an object typically uses 4 bytes of memory. When a data type contains a reference to an object, we have to account separately for the 4 bytes for the reference and the 8 bytes overhead for each object, *plus* the memory needed for the object's instance variables.
- *Arrays.* Arrays in Java are implemented as objects, typically with two instance variables (a pointer to the memory location of the first array element and the length). For primitive types, an array of N elements uses 16 bytes of header information, plus N times the number of bytes needed to store an element.
- *Two-dimensional arrays.* A two-dimensional array in Java is an array of arrays. For example, the two-dimensional array in `Markov.java` uses 16 bytes (overhead for the array of arrays) plus 4N bytes (references to the row arrays) plus N times 16 bytes (overhead from the row arrays) plus N times N times 8 bytes (for the N double values in each of the N rows) for a grand total of  $8N^2 + 20N + 16 \sim 8N^2$  bytes.
- *Strings.* A `String` uses a total of  $40 + 2N$  bytes: object overhead (8 bytes), a reference to a character array (4 bytes), three `int` values (4 bytes each), plus a character array of size N ( $16 + 2N$  bytes). Note that when working with substrings, two strings may share the same underlying character array.

Typically the JVM allocates memory in 8 byte blocks so a string of size 1, 2, 3, or 4 would consume the same amount of memory (48 bytes). `Float` and `Double` each use 16 bytes, as would a user-defined data type just containing a single `double` instance variable.

**Perspective.** Good performance is important. An impossibly slow program is almost as useless as an incorrect one. In particular, it is always wise to have some idea of which code constitutes the inner loop of your programs. Perhaps the most common mistake made in programming is to pay too much attention to performance characteristics. Your first priority is to make your code clear and correct. Modifying a program for the sole purpose of speeding it up is best left for experts. Indeed, doing so is often counterproductive, as it tends to create code that is complicated and difficult to understand. C. A. R. Hoare (a leading proponent of writing clear and correct code) once summarized this idea by saying that "premature optimization is the root of all evil," to

which D. Knuth added the qualifier "(or at least most of it) in programming."

Perhaps the second most common mistake made in developing an algorithm is to ignore performance characteristics. Users of a surprising number of computer systems lose substantial time waiting for simple quadratic algorithms to finish solving a problem, even though linear or linearithmic algorithms are available that are only slightly more complicated and could therefore solve the problem in a fraction of the time. When we are dealing with huge problem sizes, we often have no choice but to seek better algorithms.

Improving a program to make it clearer, more efficient, and elegant should be your goal every time that you work on it. If you pay attention to the cost all the way through the development of a program, you will reap the benefits every time you use it.

## Q + A

**Q.** How do I find out how long it takes to add or multiply two `double` values on my computer?

**A.** Run some experiments! The program [TimePrimitives.java](#) tests the execution time of various arithmetic operations on primitive types. On our system division is slower than addition and multiplication and trigonometric operations are substantially slower than arithmetic operations. This technique measures the actual elapsed time as would be observed on a wall clock. If your system is not running many other applications, this can produce accurate results. Also, the JIT compiler needs to get warmed up, so we disregard the first bunch of output.

**Q.** How much time do functions such as `Math.sqrt()`, `Math.log()`, and `Math.sin()` take?

**A.** Run some experiments! [Stopwatch.java](#) makes it easy to write programs to answer questions of this sort for yourself, and you will be able to use your computer much more effectively if you get in the habit of doing so.

**Q.** How much time do string operations take?

**A.** Run some experiments! (Have you gotten the message yet?) The standard implementation is written to allow the methods `length()`, `charAt()`, and `substring()` to run in constant time. Methods such as `toLowerCase()` and `replace()` are linear ear in the string size. The methods `compareTo()`, and `startsWith()` take time proportional to the number of characters needed to resolve the answer (constant in the best case and linear in the worst case), but `indexOf()` can be slow. String concatenation takes time proportional to the total number of characters in the result.

**Q.** Why does allocating an array of size N take time proportional to N?

**A.** In Java, all array elements are automatically initialized to default values (0, false, or null). In principle, this could be a constant time operation if the system would defer initialization of each element until just before the program accesses that element for the first time, but most Java implementations go through the whole array to initialize each value.

**Q.** How do I find out how much memory is available for my Java programs?

**A.** Since Java will tell you when it runs out of memory, it is not difficult to run some experiments. For example, if you use [PrimeSieve.java](#) by typing

```
% java PrimeSieve 1000000000
5761455

% java PrimeSieve 10000000000
Exception in thread "main"
java.lang.OutOfMemoryError: Java heap space
```

then you can figure that you have enough room for an array of 100 million boolean values but not for an array of 1 billion boolean values. You can increase the amount of memory allotted to Java with command-line flags.

```
% java PrimeSieve -Xmx1100m 1000000000
50847534
```

The 1100MB is the amount of memory you are requesting from the system.

**Q.** What does it mean when someone says that the running time of an algorithm is  $O(N \log N)$ ?

**A.** That is an example of a notation known as *big-Oh* notation. We write  $f(N)$  is  $O(g(N))$  if there exists a constant  $c$  such that  $f(N) \leq cg(N)$  for all  $N$ . We also say that the running time of an algorithm is  $O(g(N))$  if the running time is  $O(g(N))$  for all possible inputs. This notation is widely used by theoretical computer scientists to prove theorems about algorithms, so you are sure to see it if you take a course in algorithms and data structures. It provides a worst-case performance guarantee.

**Q.** So can I use the fact that the running time of an algorithm is  $O(N \log N)$  or  $O(N^2)$  to predict performance?

**A.** No, because the actual running time might be much less. Perhaps there is some input for which the running time is proportional to the given function, but perhaps the input is not found among those expected in practice. Mathematically, big-Oh notation is less precise than the tilde notation we use: if  $f(N) \sim g(N)$ , then  $f(N)$  is  $O(g(N))$ , but not necessarily vice versa. Consequently, big-Oh notation cannot be used to predict performance. For example, knowledge that the running time of one algorithm is  $O(N \log N)$  and the running time of another algorithm is  $O(N^2)$  does not tell you which will be faster when you run implementations of them. Generally, hypotheses that use big-Oh notation are not useful in the context of the scientific method because they are not falsifiable.

**Q.** Is the loop `for (int i = N-1; i >= 0; i--)` more efficient than `for (int i = 0; i < N; i++)`?

**A.** Some programmers think so (because it simplifies the loop continuation expression), but in many cases it is actually less efficient. Don't do it unless you have a good reason for doing so.

**Q.** Any automated tools for profiling a program?

**A.** If you execute with the `-Xprof` option, you will obtain all kinds of information.

```
% java -Xprof TwoSum < input5000.txt
Flat profile of 3.18 secs (163 total ticks): main

Interpreted + native      Method
0.6%    0 +    1  sun.misc.URLClassPath$JarLoader.getJarFile
0.6%    0 +    1  sun.nio.cs.StreamEncoder$CharsetSE.writeBytes
0.6%    0 +    1  sun.misc.Resource.getBytes
0.6%    0 +    1  java.util.jar.JarFile.initializeVerifier
0.6%    0 +    1  sun.nio.cs.UTF_8.newDecoder
0.6%    1 +    0  java.lang.String.toLowerCase
3.7%    1 +    5  Total interpreted

Compiled + native      Method
88.3%   144 +    0  TwoSum.main
1.2%     2 +    0  StdIn.readString
0.6%     1 +    0  java.lang.String.charAt
0.6%     1 +    0  java.io.BufferedReader.read
0.6%     1 +    0  java.lang.StringBuffer.length
0.6%     1 +    0  java.lang.Integer.parseInt
92.0%   150 +    0  Total compiled
```

For our purposes, the most important piece of information is the number of seconds listed in the "flat profile." In this case, the profiler says our program took 3.18 seconds. Running it a second time may yield an answer of 3.28 or 3.16 since the measurement is not perfectly accurate.

**Q.** Any performance tips?

**Q.** Here is a huge list of [Java performance tips](#).

## Exercises

1. Implement the method `printAll()` for [ThreeSum.java](#), which prints all of the

triples that sum to zero.

2. Modify [ThreeSum.java](#) to take a command-line argument  $x$  and find a triple of numbers on standard input whose sum is closest to  $x$ .
3. Write a program [FourSum.java](#) that takes an integer  $N$  from standard input, then reads  $N$  long values from standard input, and counts the number of 4-tuples that sum to zero. Use a quadruple loop. What is the order of growth of the running time of your program? Estimate the largest  $N$  that your program can handle in an hour. Then, run your program to validate your hypothesis.
4. Prove by induction that the number of distinct pairs of integers between 0 and  $N-1$  is  $N(N-1)/2$ , and then prove by induction that the number of distinct triples of integers between 0 and  $N-1$  is  $N(N-1)(N-2)/6$ .
5. Show by approximating with integrals that the number of distinct triples of integers between 0 and  $N$  is about  $N^3/6$ .
6. What is the value of  $x$  after running the following code fragment?

```
int x = 0;
for (int i = 0; i < N; i++)
    for (int j = i + 1; j < N; j++)
        for (int k = j + 1; k < N; k++)
            x++;
```

Answer:  $N \choose 3 = N(N-1)(N-2)/6$ .

7. Use tilde notation to simplify each of the following formulas, and give the order of growth of each:
  - a.  $N(N-1)(N-2)(N-3)/24$
  - b.  $(N-2)(\lg N - 2)(\lg N + 2)$
  - c.  $N(N+1) - N^2$
  - d.  $N(N+1)/2 + N\lg N$
  - e.  $\ln((N-1)(N-2)(N-3))^2$
8. Determine the order of growth of the running time of the input loop of [ThreeSum](#):

```
int N = Integer.parseInt(args[0]);
int[] a = new int[N];
for (int i = 0; i < N; i++) {
    a[i] = StdIn.readInt();
}
String s = sb.toString();
```

Answer: Linear. The bottlenecks are the array initialization and the input loop. Depending on your system and the implementation, the `readInt()` statement might lead to inconsistent timings for small values of  $N$ . The cost of an input loop like this might dominate in a linearithmic or even a quadratic program with  $N$  that is not too large.

9. Determine whether the following code fragment is linear, quadratic, or cubic (as a function of  $N$ ).

```
for (int i = 0; i < N; i++) {
    for (int j = 0; j < N; j++) {
        if (i == j) c[i][j] = 1.0;
        else c[i][j] = 0.0;
    }
}
String s = sb.toString();
```

10. Suppose the running time of an algorithm on inputs of size one thousand, two thousand, three thousand, and four thousand is 5 seconds, 20 seconds, 45 seconds, and 80 seconds, respectively. Estimate how long it will take to solve a problem of size 5,000. Is the algorithm linear, linearithmic, quadratic, cubic, or exponential?

11. Which would you prefer: a quadratic, linearithmic, or linear algorithm?

*Answer:* While it is tempting to make a quick decision based on the order of growth, it is very easy to be misled by doing so. You need to have some idea of the problem size and of the relative value of the leading coefficients of the running time. For example, suppose that the running times are  $N^2$  seconds,  $100N \lg N$  seconds, and  $10000N$  seconds. The quadratic algorithm will be fastest for  $N$  up to about 1000, and the linear algorithm will never be faster than the linearithmic one ( $N$  would have to be greater than 2100, far too large to bother considering).

12. Apply the scientific method to develop and validate a hypothesis about order of growth of the running time of the following code fragment, as a function of the input argument  $n$ .

```
public static int f(int n) {
    if (n == 0) return 1;
    return f(n-1) + f(n-1);
}
```

13. Apply the scientific method to develop and validate a hypothesis about order of growth of the running time of the [Coupon.collect\(\)](#) as a function of the input argument  $N$ . Note: Doubling is not effective for distinguishing between the linear and linearithmic hypotheses - you might try squaring the size of the input.
14. Apply the scientific method to develop and validate a hypothesis about order of growth of the running time of [Markov.java](#) as a function of the input parameters  $T$  and  $N$ .
15. Apply the scientific method to develop and validate a hypothesis about order of growth of the running time of each of the following two code fragments as a function of  $N$ .

```
String s = "";
for (int i = 0; i < N; i++) {
    if (StdRandom.bernoulli(0.5)) s += "0";
    else s += "1";
}

StringBuilder sb = new StringBuilder();
for (int i = 0; i < N; i++) {
    if (StdRandom.bernoulli(0.5)) sb.append("0");
    else sb.append("1");
}
String s = sb.toString();
```

*Answer:* The first is quadratic; the second is linear.

16. Each of the four Java functions below returns a string of length  $N$  whose characters are all  $x$ . Determine the order of growth of the running time of each function. Recall that concatenating two strings in Java takes time proportional to the sum of their lengths.

```
public static String method1(int N) {
    if (N == 0) return "";
    String temp = method1(N / 2);
    if (N % 2 == 0) return temp + temp;
    else return temp + temp + "x";
}

public static String method2(int N) {
    String s = "";
    for (int i = 0; i < N; i++)
        s = s + "x";
    return s;
}

public static String method3(int N) {
    if (N == 0) return "";
    if (N == 1) return "x";
    if (N == 2) return "xx";
    if (N == 3) return "xxx";
    if (N == 4) return "xxxx";
    if (N == 5) return "xxxxx";
    if (N == 6) return "xxxxxx";
    if (N == 7) return "xxxxxxx";
    if (N == 8) return "xxxxxxxx";
    if (N == 9) return "xxxxxxxxx";
    if (N == 10) return "xxxxxxxxxx";
```

```

        return method3(N/2) + method3(N - N/2);
    }

    public static String method4(int N) {
        char[] temp = new char[N];
        for (int i = 0; i < N; i++)
            temp[i] = 'x';
        return new String(temp);
    }
}

```

Program [Repeat.java](#) contains the four functions.

17. The following code fragment (adapted from a Java programming book) creates a random permutation of the integers from 0 to N-1. Determine the order of growth of its running time as a function of N. Compare its order of growth with [Shuffle.java](#) from Section 1.4.

```

int[] a = new int[N];
boolean[] taken = new boolean[N];
int count = 0;
while (count < N)
{
    int r = StdRandom.uniform(N);
    if (!taken[r])
    {
        a[r] = count;
        taken[r] = true;
        count++;
    }
}

```

18. What is order of growth of the running time of the following function, which reverses a string s of length N?

```

public static String reverse(String s) {
    int N = s.length();
    String reverse = "";
    for (int i = 0; i < N; i++)
        reverse = s.charAt(i) + reverse;
    return reverse;
}

```

19. What is order of growth of the running time of the following function, which reverses a string s of length N?

```

public static String reverse(String s) {
    int N = s.length();
    if (N <= 1) return s;
    String left = s.substring(0, N/2);
    String right = s.substring(N/2, N);
    return reverse(right) + reverse(left);
}

```

20. Give a linear algorithm for reversing a string.

*Answer:*

```

public static String reverse(String s) {
    int N = s.length();
    char[] a = new char[N];
    for (int i = 0; i < N; i++)
        a[i] = s.charAt(N-i-1);
    String reverse = new String(a);
}

```

```

        return reverse;
    }
}

```

21. Write a program `Moore'sLaw` that takes a command-line argument `N` and outputs the increase in processor speed over a decade if microprocessors double every `N` months. How much will processor speed increase over the next decade if speeds double every `N = 15` months? 24 months?
22. Using the model in the text, give the memory requirements for each object of the following data types from Chapter 3:
  - a. `Stopwatch`
  - b. `Turtle`
  - c. `Vector`
  - d. `Body`
  - e. `Universe`
23. Estimate, as a function of the grid size `N`, the amount of space used by [Visualize.java](#) with the vertical percolation detection (Program 2.4.2).

*Extra credit:* Answer the same question for the case where the recursive percolation detection method in Program 2.4.5 is used.
24. Estimate the size of the biggest two-dimensional array of `int` values that your computer can hold, and then try to allocate such an array.
25. Estimate, as a function of the number of documents `N` and the dimension `d`, the amount of space used by [CompareAll.java](#).
26. Write a version of [PrimeSieve.java](#) that uses a `byte` array instead of a `boolean` array and uses all the bits in each `byte`, to raise the largest value of `N` that it can handle by a factor of 8.
27. The following table gives running times for various programs for various values of `N`. Fill in the blanks with estimates that you think are reasonable on the basis of the information given.

program	1,000	10,000	100,000	1,000,000
A	.001 seconds	.012 seconds	.16 seconds	? seconds
B	1 minute	10 minutes	1.7 hours	? hours
C	1 second	1.7 minutes	2.8 hours	? days

Give hypotheses for the order of growth of the running time of each program.

## Creative Exercises

1. **Closest pair.** Design a quadratic algorithm that finds the pair of integers that are closest to each other. (In the next section you will be asked to find a linearithmic algorithm.)
2. **Sum furthest from zero.** Design an algorithm that finds the pair of integers whose sum is furthest from zero. Can you discover an algorithm that linear running time?
3. **The "beck" exploit.** In the [Apache 1.2 web server](#), there is a function called `no2slash` whose purpose is to collapse multiple '/'s. For example `/d1///d2///d3/test.html` becomes `/d1/d2/d3/test.html`. The [original algorithm](#) was to repeatedly search for a '/' and copy the remainder of the string over.

```

void no2slash(char *name) {
    int x, y;
    for(x = 0; name[x]; )
        if(x && (name[x-1] == '/') && (name[x] == '/'))
            for(y = x+1; name[y-1]; y++)
                name[y-1] = name[y];
            else x++;
}

```

Unfortunately, it's running time is quadratic in the number of '/'s in the input. By sending multiple simultaneous requests with large numbers of '/'s, you can deluge a server and starve other processes for CPU time, thereby creating a denial of service attack. Fix the version of `no2slash` so that it runs in linear time and does not allow for the above attack.

```

offset = 0
for i = 2 to n do
    if a[i-1] = '/' and a[i] = '/' then
        offset = offset + 1
    else
        a[i-offset] = a[i]
end for

```

4. **Young tableaux.** Suppose you have in memory an N-by-N grid of integers `a` such that  $a[i][j] < a[i+1][j]$  and  $a[i][j] < a[i][j+1]$  for all  $i$  and  $j$  like the table below.

5	23	54	67	89
6	69	73	74	90
10	71	83	84	91
60	73	84	86	92
99	91	92	93	94

Devise an  $O(N)$  time algorithm to determine whether or not a given integer  $x$  is in a Young tableaux.

*Answer:* Start at the upper right corner. If element =  $x$ , done. If element >  $x$ , go left. Otherwise go down. If you reach bottom left corner, then it's not in table.  $O(N)$  since can go left at most  $N$  times and down at most  $N$  times.

5. **3-D searching.** Repeat the previous question, but assume the grid is N-by-N-by-N and  $a[i][j][k] < a[i+1][j][k]$ ,  $a[i][j][k] < a[i][j+1][k]$ , and  $a[i][j][k] < a[i][j][k+1]$  for all  $i, j$ , and  $k$ . Devise an algorithm that can determine whether or not a given integer  $x$  is in the 3-d table in time better than  $N^2 \log N$ . *Hint:* treat the problem as  $N$  independent Young table queries.
6. **Moore's Law.** This problem investigates the ramifications of exponential growth. Moore's Law (named after Intel co-founder Gordon Moore) states that microprocessor power doubles every 18 months. See [this article](#) which argues against this conventional wisdom. Write a program `MooreLaw.java` that takes an integer parameter  $N$  and outputs the increase in processor speed over a decade if microprocessors double every  $N$  months.
- How much will processor speed increase over the next decade in speeds double every  $N = 18$  months?
  - The true value may be closer to speeds doubling every two years. Repeat (a) with  $N = 24$ .
- Subset sum.** Write a program [Exponential.java](#) that takes a command line integer  $N$ , reads in  $N$  long integer from standard input, and finds the *subset* whose sum is closest to 0. Give the order of growth of your algorithm.
7. **String reversal.** Given an array of  $N$  elements, give a linear time algorithm to reverse its elements. Use at most a constant amount of extra space (array indices and array values).
8. **String rotation.** Given an array of  $N$  elements, give a linear time algorithm to rotate the string  $k$  positions. That is, if the array contains  $a_1, a_2, \dots, a_N$ , the rotated array is  $a_k, a_{k+1}, \dots, a_N, a_1, \dots, a_{k-1}$ . Use at most a constant amount of extra space (array indices and array values). This operation is a primitive in some programming languages like APL. Also, arises in the implementation of a text editor (Kernighan and Plauger). *Hint:* reverse three sub-arrays as in the previous exercise.
9. **Finding a duplicated integer.** Given an array of  $n$  integers from 1 to  $n$  with one integer repeated twice and one missing. Find the missing integer using  $O(n)$  time and  $O(1)$  extra space. No overflow allowed.
10. **Finding a duplicated integer.** Given a read-only array of  $n$  integers, where each integer from 1 to  $n-1$  occurs once and one occurs twice, design an  $O(n)$  time

algorithm to find the duplicated integer. Use only O(1) space.

11. **Finding a duplicated integer.** Given a read-only array of  $n$  integers between 1 and  $n-1$ , design an  $O(n)$  time algorithm to find a duplicated integer. Use only  $O(1)$  space. *Hint:* equivalent to finding a loop in a singly linked structure.
12. **Finding the missing integer.** (Bentley's Programming Pearls.) Given a list of 1 million 20-bit integers, given an efficient algorithm for finding a 20-bit integer that is not on the list
  - a. given as much random access memory as you like
  - b. given as many tapes as you like (but no random access) and a few dozen words of memory
  - c. (Blum) given only a few dozen words of memory

*Solutions.* (a) allocate a boolean array of  $2^{20}$  bits. (b) copy all numbers starting with 0 to one tape and 1 to the other. Choose leading bit of missing number to be whichever tape has fewer entries (breaking ties arbitrarily). Recur on the smaller half. (c) Pick twenty 20-bit integers at random and search list sequentially to see if they're missing. There are 48,576 missing integers, so you'll have at least a  $(1 - (1 - 48576/2^{20})^{20}) > 0.5$  chance of getting one. If you get unlucky, repeat.

13. **Pattern matching.** Given an  $N$ -by- $N$  array of black (1) and white (0) pixels, find the largest contiguous subarray that consists of entirely black pixels. In the example below there is a 6-by-2 subarray.

1	0	1	1	1	0	0	0
0	0	0	1	0	1	0	0
0	0	1	1	1	0	0	0
0	0	1	1	1	0	1	0
0	0	1	1	1	1	1	1
0	1	0	1	1	1	1	0
0	1	0	1	1	1	1	0
0	0	0	1	1	1	1	0

14. **Factorial.** Compute  $1000000!$  as fast as you can. You may use the `BigInteger` class.
15. **Longest increasing subsequence.** Given a sequence of  $N$  64-bit integers, find the longest strictly increasing subsequence. For example, if the input is 56, 23, 33, 22, 34, 78, 11, 35, 44, then the longest increasing subsequence is 23, 33, 34, 35, 44. Your algorithm should run in  $N \log N$  time.
16. **Maximum sum.** Given a sequence of  $N$  64-bit integers, find a sequence of at most  $U$  consecutive integers that has the highest sum.
17. **Maximum average.** Given a sequence of  $N$  64-bit integers, find a sequence of at least  $L$  consecutive integers that has the highest average.  $O(N^2)$  not too hard;  $O(N)$  possible but much harder.  $O(NL)$  follows since there must exist an optimal interval between  $L$  and  $2L$ . *Hint:* can compute average from  $i$  to  $j$  in  $O(1)$  time with  $O(N)$  preprocessing by precomputing  $\text{prefix}[i] = a[0] + \dots + a[i]$  for each  $i$ . Then the average of the interval from  $i$  to  $j$  is  $(\text{prefix}[j] - \text{prefix}[i]) / (j - i + 1)$ .
18. **Knuth's parking problem.**  $N$  cars arrive at an initially empty lot with  $N$  parking space, arranged in a one-way circle. Car  $i$  uniformly chooses a spot at random, independent of previous cars. If spot  $j$  is taken, try  $j+1, j+2$ , and so on. Write a program to estimate how many tries are made (total displacement) as a function of  $N$ .

*Answer.*  $N^{3/2}$ .

19. **Amdahl's law.** Limits how much you can speed up a computation by improving one of its two constituent parts.
20. **Sieve of Eratosthenes.** Estimate running time of sieve of Eratosthenes for finding all primes less than or equal to  $N$  as a function of  $N$ .

*Answer:* in theory, the answer is proportional to  $N \log \log N$ . Follows from Mertens' theorem in number theory.

## Web Exercises

1. Suppose the running time of an algorithm on inputs of size 1,000, 2,000, 3,000, and 4,000 is 5 seconds, 20 seconds, 45 seconds, 80 seconds, and 125 seconds, respectively. Estimate how long it will take to solve a problem of size 5,000. Is the

algorithm have linear, linearithmic, quadratic, cubic, or exponential?

2. Write a program [OneSum.java](#) that takes a command-line argument  $N$ , reads in  $N$  integers from standard input, and finds the value that is closest to 0. How many instructions are executed in the data processing loop?
3. Write a program [TwoSum.java](#) that takes a command-line argument  $N$ , reads in  $N$  integers from standard input, and finds the pair of values whose sum is closest to 0. How many instructions are executed in the data processing loop?
4. Analyze the following code fragment mathematically and determine whether the running time is linear, quadratic, or cubic as a function of  $N$ .

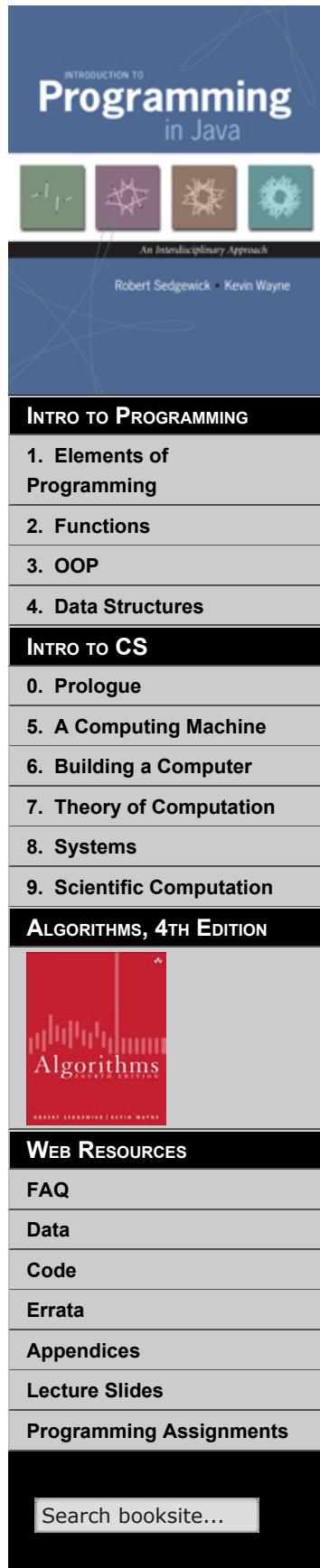
```
for (int i = 0; i < N; i++)
    for (int j = 0; j < N; j++)
        for (int k = 0; k < N; k++)
            c[i][j] += a[i][k] * b[k][j];
```

5. The following function returns a random string of length  $N$ . How long does it take?

```
public static String random(int N) {
    if (N == 0) return "";
    int r = (int) (26 * Math.random()); // between 0 and 25
    char c = 'a' + r; // between 'a' and 'z'
    return random(N/2) + c + random(N - N/2 - 1);
}
```

*Last modified on August 05, 2011.*

Copyright © 2002–2012 [Robert Sedgewick](#) and [Kevin Wayne](#). All rights reserved.



## 4.2 SORTING AND SEARCHING

This section under construction.

The sorting problem is to rearrange a set of items in ascending order. One reason that it is so useful is that it is much easier to search for something in a sorted list than an unsorted one. In this section, we will consider in detail two classical algorithms for sorting and searching, along with several applications where their efficiency plays a critical role.

**Binary search.** In the game of "twenty questions", your task is to guess the value of a hidden number that is one of the  $N$  integers between 0 and  $N-1$ . (For simplicity, we will assume that  $N$  is a power of two.) Each time that you make a guess, you are told whether your guess is too high or too low. An effective strategy is to maintain an interval that contains the hidden number, guess the number in the middle of the interval, and then use the answer to halve the interval size. [TwentyQuestions.java](#) implements this strategy, which is an example of the general problem-solving method known as *binary search*.

interval	size	Q	A
[0, 128]	128	< 64 ?	no
[64, 128]	64	< 96 ?	yes
[64, 96]	32	< 80 ?	yes
[64, 80]	16	< 72 ?	no
[72, 80]	8	< 76 ?	no
[76, 80]	4	< 78 ?	yes
[76, 78]	2	< 77 ?	no
[77, 77]	1	= 77	

*Finding a hidden number with binary search*

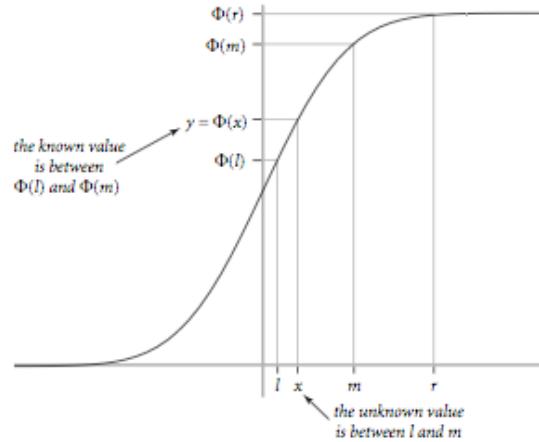
- *Correctness proof.* First, we have to convince ourselves that the method is correct: that it always leads us to the hidden number. We do so by establishing the following facts:

- The interval always contains the hidden number.
- The interval sizes are the powers of two, decreasing from  $N$ .

The first of these facts is enforced by the code; the second follows by noting that if the interval size ( $hi-lo$ ) is a power of two, then the next interval size is  $(hi-lo)/2$ , which is the next smaller power of two. These facts are the basis of an induction proof that the method operates as intended. Eventually, the interval size becomes 1, so we are guaranteed to find the number.

- *Running time analysis.* Since the size of the interval decreases by a factor of 2 at each iteration (and the base case is reached when  $N = 1$ ), the running time of binary search is  $\lg N$ .
- *Linear-logarithm chasm.* The alternative to using binary search is to guess 0, then 1, then 2, then 3, and so forth, until hitting the hidden number. We refer to such an algorithm as a *brute-force* algorithm: it seems to get the job done, but without much regard to the cost (which might prevent it from actually getting the job done for large problems). In this case, the running time of the brute-force algorithm is sensitive to the input value, but could be as much as  $N$  and has expected value  $N/2$  if the input value is chosen at random. Meanwhile, binary search is guaranteed to use no more than  $\lg N$  steps.
- *Binary representation.* If you look back to Program 1.3.7, you will recognize that binary search is nearly the same computation as converting a number to binary! Each guess determines one bit of the answer. In our example, the information that the number is between 0 and 127 says that the number of bits in its binary representation is 7, the answer to the first question (is the number less than 64?) tells us the value of the leading bit, the answer to the second question tells us the value of the next bit, and so forth. For example, if the number is 77, the sequence of answers no yes yes no no yes no immediately yields 1001101, the binary representation of 77.

- *Inverting a function.*  
As an example of the utility of binary search in scientific computing, we revisit a problem that we consider the problem of inverting an increasing function. To fix ideas, we refer to the Gaussian distribution  $\Phi$  when describing the method. Given a value  $y$ , our task is to find a value  $x$  such that  $\Phi(x) = y$ . In this situation, we use real numbers as the



Binary search (bisection) to invert an increasing function (one step)

endpoints of our interval, not integers, but we use the same essential method as for guessing a hidden integer: we halve the size of the interval at each step, keeping  $x$  in the interval, until the interval is sufficiently small that we know the value of  $x$  to within a desired precision  $\delta$ . We start with an interval  $(lo, hi)$  known to contain  $x$  and use the following recursive strategy:

- Compute  $m = lo + (hi - lo) / 2$
- Base case: If  $(hi - lo)$  is less than  $\delta$ , then return  $m$  as an estimate of  $x$
- Recursive step: otherwise, test whether  $\Phi(m) < y$ . If so look for  $x$  in  $(lo, m)$ ; if not look for  $x$  in  $(m, hi)$

The key to this method is the idea that the function is increasing - for any values  $a$  and  $b$ , knowing that  $\Phi(a) < \Phi(b)$  tells us that  $a < b$ , and vice versa. In this context, binary search is often called *bisection search* because we bisect the interval at each stage.

- *Binary search in a sorted array.* During much of the last century people would use a publication known as a *phone book* to look up a person's phone number. Entries appear in order, sorted by a key that identifies it (the person's name) in both cases). A brute-force solution would be to start at the beginning, examine each entry one at a time, and continue until you find the name. No one uses that method: instead, you open the book to some interior page and look for the name on that page. If it is there, you are done; otherwise, you eliminate either the part of the book before the current page or the part of the book after the current page from consideration and repeat.
- *Exception filter.* We now use binary search to solve the *existence problem*: is a given key in a sorted database of keys? For example, when checking the spelling of a word, you need only know whether your word is in the dictionary and are not interested in the definition. In a computer search, we keep the information in an array, sorted in order of the key. The binary search code in [BinarySearch.java](#) differs from our other applications in two details. First, the file size  $N$  need not be a power of two. Second, it has to allow the possibility that the item sought is not in the array. The client program implements an *exception filter*: it reads a sorted list of strings from a file (which we refer to as the *whitelist*) and an arbitrary sequence of strings from standard input and prints those in the sequence that are *not* in the whitelist.

**Insertion sort.** Insertion sort is a brute-force sorting algorithm that is based on a simple method that people often use to arrange hands of playing cards. Consider the cards one at a time and insert each into its proper place among those already considered (keeping them sorted). The following code mimics this process in a Java method that sorts strings in an array:

```
public static void sort(String[] a) {
    int N = a.length;
    for (int i = 1; i < N; i++)
        for (int j = i; j > 0; j--)
            if (a[j-1].compareTo(a[j]) > 0)
```

```

        exch(a, j, j-1);
        else break;
    }
}

```

The outer `for` loop sorts the first  $i$  entries in the array; the inner `for` loop can complete the sort by putting  $a[i]$  into its proper position in the array.

i	j	a							
		0	1	2	3	4	5	6	7
		was	had	him	and	you	his	the	but
1	0	had	was	him	and	you	his	the	but
2	1	had	him	was	and	you	his	the	but
3	0	and	had	him	was	you	his	the	but
4	4	and	had	him	was	you	his	the	but
5	3	and	had	him	his	was	you	the	but
6	4	and	had	him	his	the	was	you	but
7	1	and	but	had	him	his	the	was	you
		and	but	had	him	his	the	was	you

*Inserting a[1] through a[N-1] into position (insertion sort)*

- *Mathematical analysis.* The inner loop of the insertion sort code is within a double for loop, which suggests that the running time is quadratic, but we cannot immediately draw this conclusion because of the `break`.
  - *Best case.* When the input array is already in sorted order, the inner for loop amounts to nothing more than a comparison (to learn that  $a[j-1]$  is less than  $a[j]$ ) and the `break`, so the total running time is linear.
  - *Worst case.* When the input is reverse sorted, the inner loop fully completes without a `break`, so the frequency of execution of the instructions in the inner loop is  $1 + 2 + \dots + N-1 \sim N^2$  and the running time is quadratic.
  - *Average case.* When the input is *randomly* ordered To understand the performance of insertion sort for randomly ordered, we expect that each new element to be inserted is equally likely to fall into any position, so that element will move halfway to the left on average. Thus, we expect the running time to be  $1/2 + 2/2 + \dots + (N-1)/2 \sim N^2 / 2$ .
- *Sorting other types of data.* We want to be able to sort all types of data, not just strings. For sorting objects in an array, we need only assume that we can compare two elements to see whether the first is bigger than, smaller than, or equal to the second. Java provides the `Comparable` interface for precisely this purpose. Simply put, a class that implements the `Comparable` interface promises to implement a method `compareTo()` for objects of its type so that that `a.compareTo(b)` returns a negative integer if `a` is less than `b`, a positive integer if `a` is greater than `b`, and 0 if `a` is equal to `b`. The precise meanings of *less than*, *greater than*, and *equal to* are up to the data type, though implementations that do not respect the natural laws of mathematics surrounding these concepts will yield unpredictable results. With this convention, `Insertion.java` implements insertion sort so that it sorts arrays of `Comparable` objects.
- *Empirical analysis.* Program `InsertionTest.java` tests our hypothesis that insertion sort is quadratic for randomly ordered files. It relies on the helper data type `Stopwatch.java`.
- *Sensitivity to input.* Note that `InsertionTest.java` takes a command-line parameter `M` and runs `M` experiments for each array size, not just one. One reason for doing so is that the running time of insertion sort is sensitive to its input values. It is not correct to flatly predict that the running time of insertion sort will be quadratic, because your application might involve input for which the running time is linear.

**Mergesort.** To develop a faster sorting method, we use a *divide-and-conquer* approach to algorithm design that every programmer needs to understand. This nomenclature refers to the idea that one way to solve a problem is to *divide* it into independent parts, *conquer* them independently, and then use the solutions for the parts to develop a solution for the full problem. To sort an array with this strategy, we divide it into two halves, sort the two halves independently, and then merge the results to sort the full array. This method is known as *mergesort*. To sort  $a[lo, hi]$ , we use the following recursive strategy:

- *base case:* If the subarray size is 0 or 1, it is already sorted.
- *recursive step:* Otherwise, compute  $m = lo + (hi - lo)/2$ , sort (recursively) the two subarrays  $a[lo, m]$  and  $a[m, hi]$ , and merge them to produce a sorted result.

```

input
was had him and you his the but

sort left
and had him was you his the but

sort right
and had him was but his the you

merge
and but had him his the was you

Mergesort overview

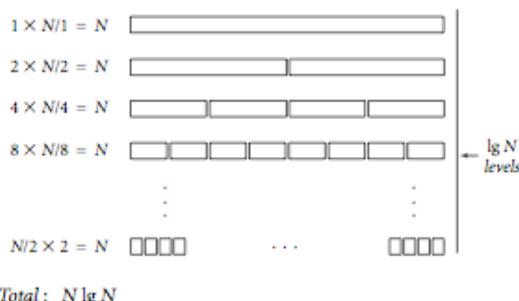
```

[Merge.java](#) is an implementation. As usual, the easiest way to understand the merge process is to study a trace of the contents of the array during the merge.

i	j	k	aux[k]	a							
				0	1	2	3	4	5	6	7
0	4	0	and	and	had	him	was	but	his	the	you
1	4	1	but	and	had	him	was	but	his	the	you
1	5	2	had	and	had	him	was	but	his	the	you
2	5	3	him	and	had	him	was	but	his	the	you
3	5	4	his	and	had	him	was	but	his	the	you
3	6	5	the	and	had	him	was	but	his	the	you
3	6	6	was	and	had	him	was	but	his	the	you
4	7	7	you	and	had	him	was	but	his	the	you

*Trace of the merge of the sorted left half with the sorted right half*

- *Mathematical analysis.* The inner loop of mergesort is centered on the auxiliary array. The two for loops involve  $N$  iterations (and creating the array takes time proportional to  $N$ ), so the frequency of execution of the instructions in the inner loop is proportional to the sum of the subarray sizes for all calls to the recursive function. The value of this quantity emerges when we arrange the calls on levels according to their size. On the first level, we have 1 call for size  $N$ , on the second level, we have 2 calls for size  $N/2$ , on the third level, we have 4 calls for size  $N/4$ , and so forth, down to the last level with  $N/2$  calls of size 2. There are precisely  $\lg N$  levels, giving the grand total  $N \lg N$  for the frequency of execution of the instructions in the inner loop of mergesort. This equation justifies a hypothesis that the running time of mergesort is linearithmic.



*Total:  $N \lg N$*

*Mergesort inner loop frequency count (when  $N$  is a power of 2)*

- *Quadratic-linearithmic chasm.* The difference between  $N^2$  and  $N \lg N$  makes a huge difference in practical applications. *Understanding the enormousness of this difference is another critical step to understanding the importance of the design and analysis of algorithms.* For a great many important computational problems, a speedup from quadratic to linearithmic makes the difference between being able to solve a problem involving a huge amount of data and not being able to effectively address it at all.
- *Divide-and-conquer algorithms.* The same basic approach is effective for many important problems, as you will learn if you take a course on algorithm design.
- *Reduction to sorting.* A problem  $A$  reduces to a problem  $B$  if we can use a solution to  $B$  to solve  $A$ . Designing a new divide-and-conquer algorithm from scratch is sometimes akin to solving a puzzle that requires some experience and ingenuity, so you may not feel confident that you can do so at first. But it is often the case that a simpler approach is effective: given a new problem that lends itself to a quadratic brute-force solution, ask yourself how you would solve it if the data were sorted in some way. For example, consider the problem of determining whether the elements in an array are all different. This problem reduces to sorting because we can sort the array, then make a linear pass through the sorted array to check whether any entry is equal to the next (if not, the elements are all different.)

**Frequency counts.** [FrequencyCount.java](#) reads a sequence of strings from standard input and then prints a table of the distinct values found and the number of times each was found, in decreasing order of the frequencies. We accomplish this by two sorts.

- *Computing the frequencies.* Our first step is to sort the strings on standard input. In this case, we are not so much interested in the fact that the strings are put into sorted order, but in the fact that sorting brings equal strings together. If the input is

to be or not to be to

then the result of the sort is

be be not or to to to

with equal strings like the three occurrences of `to` brought together in the array. Now, with equal strings all together in the array, we can make a single pass through the array to compute all the frequencies. The [Counter.java](#) data type that we considered in Section 3.x is the perfect tool for the job.

- *Sorting the frequencies.* Next, we sort the `Counter` objects. We can do so in client code without any special arrangements because `Counter` implements the `Comparable` interface.
- *Zipf's law.* The application highlighted in [FrequencyCount](#) is elementary linguistic analysis: which words appear most frequently in a text? A phenomenon known as [Zipf's law](#) says that the frequency of the  $i$ th most frequent word in a text of  $M$  distinct words is proportional to  $1/i$ .

**Longest repeated substring.** Another important computational task that reduces to sorting is the problem of finding the *longest repeated substring* in a given string. This problem is simple to state and has many important applications, including computer-assisted music analysis, cryptography, and data compression. Think briefly about how you might solve it. Could you find the longest repeated substring in a string that has millions of characters? Program [LRS.java](#) is a clever solution that uses *suffix sorting*.

**Quicksort.** Quicksort is a divide-and-conquer method for sorting. It works by *partitioning* an array of elements into two parts, then sorting the parts independently. As we shall see, the precise position of the partition depends on the initial order of the elements in the input file. The crux of the method is the

partitioning process, which rearranges the array to make the following three conditions hold:

- The element  $a[i]$  is in its final place in the array for  $i$ .
- None of the elements  $a[left], \dots, a[i-1]$  is greater than  $a[i]$ .
- None of the elements in  $a[i+1], \dots, a[right]$  is less than  $a[i]$ .

We achieve a complete sort by partitioning, then recursively applying the method to the subfiles.

We use the following general strategy to implement partitioning. First, we arbitrarily choose  $a[right]$  to be the *partitioning element* - the one that will go into its final position. Next, we scan from the left end of the array until we find an element greater than the partitioning element, and we scan from the right end of the array until we find an element less than the partitioning element. The two elements that stopped the scans are obviously out of place in the final partitioned array, so we exchange them. Continuing in this way, we ensure that no array elements to the left of the left index are greater than the partitioning element, and no array elements to the right of the right index are less than the partitioning element, as depicted in the following diagram.



When the scan indices cross, all that we need to do to complete the partitioning process is to exchange  $a[right]$  with the leftmost element of the right subfile (the element pointed to by the left index  $i$ ).

Program [QuickSort.java](#) implements this algorithm.

## Q + A

**Q.** Why do we need to go to such lengths to prove a program correct?

**A.** To spare ourselves considerable pain. Binary search is a notable example. For example, you now understand binary search; a classic programming exercise is to write a version that uses a while loop instead of recursion. Try solving Exercise 4.2.2 without looking back at the code in the book. In a famous experiment, [J. Bentley](#) once asked several professional programmers to do so, and most of their solutions were not correct. According to Knuth, the first binary search algorithm was published in 1946, but the first published binary search without bugs did not appear until 1962.

**Q.** Are there implementations for sorting and searching in the Java library?

**A.** Yes. The Java library `java.util.Arrays` contains the methods `Arrays.sort()` and `Arrays.binarySearch()` that implement mergesort and binary search for `Comparable` types and a sorting implementation for primitive types based on a version of the *quicksort* algorithm, which is faster than mergesort and also sorts an array in place (without using any extra space). [SystemSort.java](#) illustrates how to use `Arrays.sort()`.

**Q.** So why not just use them?

**A.** Feel free to do so. As with many topics we have studied, you will be able to use such tools more effectively if you understand the background behind them.

**Q.** Why doesn't the Java library use a randomized version of quicksort?

**A.** Good question. At the very least, the library should cutoff to some guaranteed  $N \log N$  algorithm if it "realizes" it is in trouble. Perhaps to avoid side effects. Programmers may want their libraries to be deterministic for debugging. But the library only uses quicksort for primitive types when stability is not an issue, so the programmer probably wouldn't notice the randomness, except in running time.

## Exercises

1. Develop an implementation of [TwentyQuestions.java](#) that takes the maximum number  $N$  as command-line input. Prove that your implementation is correct.
2. Add code to `Insertion` to produce the trace given in the text.
3. Add code to `Merge` to produce the trace given in the text.
4. Give traces of insertion sort and mergesort in the style of the traces in the text, for the input

```
it was the best of times it was
```

5. Describe why it is desirable to use immutable keys with binary search.
6. Explain why we use  $lo + (hi - lo) / 2$  to compute the index midway between  $lo$  and  $hi$  instead of using  $(lo + hi) / 2$ . This can happen if the array contains around a billion elements.

*Solution.* The latter fails when  $lo + hi$  overflows an `int`. [Joshua Bloch reports](#) that Sun's Java 1.5 implementation contained this bug. Here is another correct version using bit-shifting.

```
int mid = lo + (hi - lo) / 2;
int mid = (lo + hi) >>> 1;
```

7. Modify [BinarySearch.java](#) so that if the search key is in the array, it returns the smallest index  $i$  for which  $a[i]$  is equal to key, and otherwise, it returns  $-i$ , where  $i$  is the smallest index such that  $a[i]$  is greater than key.
8. Describe what happens if you apply binary search to an unorderd array. Why shouldn't you check whether the array is sorted before each call to binary search? Could you check that the elements binary search examines are in ascending order?
9. Write a program [Dedup.java](#) that reads strings from standard input and prints them on standard output with all duplicates removed (in sorted order).
10. Find the frequency distribution of words in your favorite book. Does it obey Zipf's law?
11. Find the longest repeated substring in your favorite book.
12. Add code to [LRS.java](#) to make it print indices in the original string where the longest repeated substring occurs.
13. Find a pathological input for which [LRS.java](#) runs in quadratic time (or worse).
14. Show that binary search in a sorted array is logarithmic as long as it eliminates at least a constant fraction of the array at each step.
15. Modify [BinarySearch.java](#) so that if the search key is not in the array, it returns the largest index  $i$  for which  $a[i]$  is smaller than key (or  $-1$  if no such index exists).
16. Analyze mergesort mathematically when  $N$  is a power of 2, as we did for binary search.
17. Analyze mergesort for the case when  $N$  is not a power of two.

## Creative Exercises

*This list of exercises is intended to give you experience in developing fast solutions to typical problems. Think about using binary search, mergesort, or devising your own divide-and-conquer algorithm. Implement and test your algorithm.*

18. **Median.** Add to [StdStats.java](#) a method `median()` that computes in linearithmic time the median of a sequence of  $N$  integers.

*Hint:* reduce to sorting.

19. **Mode.** Add to [StdStats.java](#) a method `mode()` that computes in linearithmic time the mode (value that occurs most frequently) of a sequence of  $N$  integers.

*Hint:* reduce to sorting.

20. **Integer sort.** Write a *linear-time* filter [IntegerSort.java](#) that takes from standard input a sequence of integers that are between 0 and 99 and prints the same integers in sorted order on standard output. For example, presented with the input sequence

```
98 2 3 1 0 0 0 3 98 98 2 2 2 0 0 0 2
```

your program should print the output sequence

```
0 0 0 0 0 0 1 2 2 2 2 3 3 98 98 98
```

21. **Floor and ceiling.** Given a sorted array of `Comparable` items, write methods `floor()` and `ceil()` that returns the index of the largest (or smallest) item not larger (or smaller) than an argument item in logarithmic time.
22. **Closest pair.** Given an array of  $N$  real numbers, write a static method to find in linearithmic time the pair of integers that are closest in value.
23. **Farthest pair.** Given an array of  $N$  real numbers, write a static method to find in linear time the pair of integers that are farthest apart in value.
24. **Two sum.** Write a static method that takes as argument an array of  $N$  `int` values and determines in linearithmic time whether any *two* of them sum to 0.
25. **Three sum.** Write a static method that takes as argument an array of  $N$  `int` values and determines whether any *three* of them sum to 0. Your program should run in time proportional to  $N^2 \log N$ . *Extra credit:* develop a program that solves the problem in quadratic time.
26. **Majority.** An element is a *majority* if it appears more than  $N/2$  times. Write a static method that takes an array of  $N$  strings as argument and identifies a majority (if it exists) in linear time.

27. **Common element.** Write a static method that takes as argument three arrays of strings, determines whether there is any string common to all three arrays, and if so, returns one such string. The running time of your method should be linearithmic in the total number of strings.

*Hint:* sort each of the three lists, then describe how to do a "3-way" merge.

28. **Prefix-free codes.** In data compression, a set of strings is *prefix free* if no string is a prefix of another. For example, the set of strings 01, 10, 0010, 1010 is not prefix free because 10 is a prefix of 1010. Write a program that reads in a set of strings from standard input and determines whether the set is prefix free.

29. **Longest common substring.** Write a static method that finds the longest common substring of two given strings `s` and `t`.

*Hint:* Suffix sort each string. Then merge the two sorted suffixes together.

30. **Longest repeated, non-overlapping string.** Modify [LRS.java](#) to find the longest repeated substring that *does not overlap*.

31. **Partitioning.** Write a static method that sorts a `Comparable` array that is known to have at most two different values.

*Hint:* Maintain two pointers, one starting at the left end and moving right, the other starting at the right end and moving left. Maintain the invariant that all elements to the left of the left pointer are equal to the smaller of the two values and all elements to the right of the right pointer are equal to the larger of the two values.

32. **Dutch national flag problem.** Write a static method that sorts a `Comparable` array that is known to have at most three different values.

(Edsger Dijkstra named this the *Dutch-national-flag problem* because the result is three "stripes" of values like the three stripes in the flag.)

33. **Quicksort.** Write a recursive program [QuickSort.java](#) that sorts an array of randomly ordered distinct Comparable elements.

*Hint:* Use a method like the one described in the previous exercise. First, partition the array into a left part with all elements less than  $v$ , followed by  $v$ , followed by a right part with all elements greater than  $v$ . Then, recursively sort the two parts.

*Extra credit:* Modify your method (if necessary) to work properly when the elements are not necessarily distinct.

34. **Reverse domain.** Write a program to read in a list of domain names from standard input, and print the reverse domain names in sorted order. For example, the reverse domain of `cs.princeton.edu` is `edu.princeton.cs`. This computation is useful for web log analysis. To do so, create a data type [Domain.java](#) that implements the Comparable interface, using reverse domain name order.

35. **Local minimum in an array.** Given an array of  $N$  real numbers, write a static method to find in logarithmic time a *local minimum* (an index  $i$  such that  $a[i-1] < a[i] < a[i+1]$ ).

*Solution.* Query middle value  $a[n/2]$ , and two neighbors  $a[n/2 - 1]$  and  $a[n/2 + 1]$ . If  $a[n/2]$  is local minimum, stop; otherwise search in half with smaller neighbor.

36. **Discrete distribution.** Design a fast algorithm to repeatedly generate numbers from the discrete distribution: Given an array  $a[]$  of nonnegative real numbers that sum to 1, the goal is to return index  $i$  with probability  $a[i]$ . Form an array  $s[]$  of cumulated sums such that  $s[i]$  is the sum of the first  $i$  elements of  $a[]$ . Now, generate a random real number  $r$  between 0 and 1, and use binary search to return the index  $i$  for which  $s[i] \leq r < s[i+1]$ .

37. **Rhyming words.** Write a program [Rhymers.java](#) that tabulates a list that you can use to find words that rhyme. Use the following approach:

- Read in a dictionary of words into an array of strings.
- Reverse the letters in each word (`confound` becomes `dnuofnuc`, for example).
- Sort the resulting array.
- Reverse the letters in each word back to their original order.

For example, `confound` is adjacent to words such as `astound` and `surround` in the resulting list.

## Web Exercises

1. **Union of intervals.** Given  $N$  intervals on the real line, determine the length of their union in  $O(N \log N)$  time. For example the union of the four intervals  $[1, 3]$ ,  $[2, 4.5]$ ,  $[6, 9]$ , and  $[7, 8]$  is 6.5.
2. **Coffee can problem.** (David Gries). Suppose you have a coffee can which contains an unknown number of black beans and an unknown number of white beans. Repeat the following process until exactly one bean remains: Select two beans from the can at random. If they are both the same color, throw them both out, but insert another black bean. If they are different colors, throw the black one away, but return the white one. Prove that this process terminates with exactly one bean left. What can you deduce about the color of the last bean as a function of the initial number of black and white beans? *Hint:* find a useful invariant maintained by the process.
3. **Spam campaign.** To initiate an illegal spam campaign, you have a list of email addresses from various domains (the part of the email address that follows the @ symbol). To better forge the return addresses, you want to send the email from another user at the same domain. For example, you might want to forge an email from `nobody@princeton.edu` to `somebody@princeton.edu`. How would you process the email list to make

this an efficient task?

4. **Order statistics.** Given an array of  $N$  elements, not necessarily in ascending order, devised an algorithm to find the  $k$ th largest one. It should run in  $O(N)$  time on random inputs.
5. **Kendall's tau distance.** Given two permutations, Kendall's tau distance is the number of pairs out of position. "Bubblesort metric." Give an  $O(N \log N)$  algorithm to compute the Kendall tau distance between two permutations of size  $N$ . Useful in top- $k$  lists, social choice and voting theory, comparing genes using expression profiles, and [ranking search engine results](#).
6. **Antipodal points.** Given  $N$  points on a circle, centered at the origin, design an algorithm that determines whether there are two points that are *antipodal*, i.e., the line connecting the two points goes through the origin. Your algorithm should run in time proportional to  $N \log N$ .
7. **Antipodal points.** Repeat the previous question, but assume the points are given in clockwise order. Your algorithm should run in time proportional to  $N$ .
8. **Identity.** Given an array  $a[]$  of  $N$  distinct integers (positive or negative) in ascending order. Devise an algorithm to find an index  $i$  such that  $a[i] = i$  if such an index exists. Hint: binary search.
9. **L1 norm.** There are  $N$  circuit elements in the plane. You need to run a special wire (parallel to the  $x$ -axis) across the circuit. Each circuit element must be connected to the special wire. Where should you put the special wire? Hint: median minimizes L1 norm.
10. **Finding common elements.** Given two arrays of  $N$  64-bit integers, design an algorithm to print out all elements that appear in both lists. The output should be in sorted order. Your algorithm should run in  $N \log N$ . Hint: mergesort, mergesort, merge. Remark: not possible to do better than  $N \log N$  in comparison based model.
11. **Finding common elements.** Repeat the above exercise but assume the first array has  $M$  integers and the second has  $N$  integers where  $M$  is much less than  $N$ . Give an algorithm that runs in  $N \log M$  time. Hint: sort and binary search.
12. **Anagrams.** Design a  $O(N \log N)$  algorithm to read in a list of words and print out all anagrams. For example, the strings "comedian" and "demoniac" are anagrams of each other. Assume there are  $N$  words and each word contains at most 20 letters. Designing a  $O(N^2)$  algorithms should not be too difficult, but getting it down to  $O(N \log N)$  requires some cleverness.
13. **Pattern recognition.** Given a list of  $N$  points in the plane, find all subset of 3 or more [points that are collinear](#).
14. **Pattern recognition.** Given a list of  $N$  points in the plane in general position (no three are collinear), find a new point  $p$  that is not collinear with any pair of the  $N$  original points.
15. **Search in a sorted, rotated list.** Given a sorted list of  $N$  integers that has been rotated an unknown number of positions, e.g., 15 36 1 7 12 13 14, design an  $O(\log N)$  algorithm to determine if a given integer is in the list.
16. **Counting inversions.** Each user ranks  $N$  songs in order of preference. Given a preference list, find the user with the *closest* preferences. Measure "closest" according to the number of inversions. Devise an  $N \log N$  algorithm for the problem.
17. **Throwing cats from an  $N$ -story building.** Suppose that you have an  $N$  story building and a bunch of cats. Suppose also that a cat dies if it is thrown off floor  $F$  or higher, and lives otherwise. Devise a strategy to determine the floor  $F$ , while killing  $O(\log N)$  cats.
18. **Throwing cats from a building.** Repeat the previous exercise, but devise a strategy that kills  $O(\log F)$  cats. Hint: repeated doubling and binary search.
19. **Throwing two cats from an  $N$ -story building.** Repeat the previous question, but now assume you only have two cats. Now your goal is to minimize the number of throws. Devise a strategy to determine  $F$  that involves throwing cats  $O(\sqrt{N})$  times (before killing them both). This application might occur in practice if search hits (cat surviving fall) are much cheaper than misses (cat dying).
20. **Throwing two cats from a building.** Repeat the previous question, but only throw  $O(\sqrt{F})$  cats. Reference: ???.
21. **Nearly sorted.** Given an array of  $N$  elements, each which is at most  $k$

positions from its target position, devise an algorithm that sorts in  $O(N \log k)$  time.

*Solution 1:* divide the file into  $N/k$  pieces of size  $k$ , and sort each piece in  $O(k \log k)$  time, say using mergesort. Note that this preserves the property that no element is more than  $k$  elements out of position. Now, merge each blocks of  $k$  elements with the block to its left.

*Solution 2:* insert the first  $k$  elements into a binary heap. Insert the next element from the array into the heap, and delete the minimum element from the heap. Repeat.

22. **Merging  $k$  sorted lists.** Suppose you have  $k$  sorted lists with a total of  $N$  elements. Give an  $O(N \log k)$  algorithm to produce a sorted list of all  $N$  elements.
23. **Longest common reverse complemented substring.** Given two DNA strings, find the longest substring that appears in one, and whose reverse Watson-Crick complement appears in the other. Two strings  $s$  and  $t$  are reverse complements if  $t$  is the reverse of  $s$  except with the following substitutions A $\leftrightarrow$ T, C $\leftrightarrow$ G. For example ATTCGG and CCGAAAT are reverse complements of each other. *Hint:* suffix sort.
24. **Circular string linearization.** Plasmids contain DNA in a circular molecule instead of a linear one. To facilitate search in a database of DNA strings, we need a place to break it up to form a linear string. A natural choice is the place that leaves the lexicographically smallest string. Devise an algorithm to compute this canonical representation of the circular string *Hint:* suffix sort.
25. **Find all matches.** Given a text string, find *all* matches of the query string. *Hint:* combine suffix sorting and binary search.
26. **Longest repeated substring with less memory.** Instead of using an array of substrings where  $\text{suffixes}[i]$  refers to the  $i$ th sorted suffix, maintain an array of integers so that  $\text{index}[i]$  refers to the offset of the  $i$ th sorted suffix. To compare the substrings represented by  $a = \text{index}[i]$  and  $b = \text{index}[j]$ , compare the character  $s.\text{charAt}(a)$  against  $s.\text{charAt}(b)$ ,  $s.\text{charAt}(a+1)$  against  $s.\text{charAt}(b+1)$ , and so forth. How much memory do you save? Is your program faster?
27. **Idle time.** Suppose that a parallel machine processes  $n$  jobs. Job  $j$  is processed from  $s_j$  to  $t_j$ . Given the list of start and finish times, find the largest interval where the machine is idle. Find the largest interval where the machine is non-idle.
28. **Local minimum of a matrix.** Given an  $N$ -by- $N$  array  $a$  of  $N^2$  distinct integers, design an  $O(N)$  algorithm to find a *local minimum*: a pair of indices  $i$  and  $j$  such that  $a[i][j] < a[i+1][j]$ ,  $a[i][j] < a[i][j+1]$ ,  $a[i][j] < a[1][j]$ , and  $a[i][j] < a[i][j-1]$ .
29. **Monotone 2d array.** Give an  $n$ -by- $n$  array of elements such that each row is in ascending order and each column is in ascending order, devise an  $O(n)$  algorithm to determine if a given element  $x$  in the array. You may assume all elements in the  $n$ -by- $n$  array are distinct.
30. **2D maxima.** Given a set of  $n$  points in the plane, point  $(x_i, y_i)$  dominates  $(x_j, y_j)$  if  $x_i > x_j$  and  $y_i > y_j$ . A maxima is a point that is not dominated by any other point in the set. Devise an  $O(n \log n)$  algorithm to find all maxima. Application: on x-axis is space efficiency, on y-axis is time efficiency. Maxima are useful algorithms. *Hint:* sort in ascending order according to x-coordinate; scan from right to left, recording the highest y-value seen so far, and mark these as maxima.
31. **Compound words.** Read in a list of words from standard input, and print out all two-word *compound* words. If *after*, *thought*, and *afterthought* are in the list, then *afterthought* is a compound word. Note: the components in the compound word need not have the same length.
32. **Smith's rule.** The following problem arises in supply chain management. You have a bunch of jobs to schedule on a single machine. (Give example.) Job  $j$  requires  $p[j]$  units of processing time. Job  $j$  has a positive weight  $w[j]$  which represents its relative importance - think of it as the inventory cost of storing the raw materials for job  $j$  for 1 unit of time. If job  $j$  finishes being processed at time  $t$ , then it costs  $t * w[j]$  dollars. The goal is to sequence the jobs so as to minimize the sum of the weighted completion times of each job. Write a program `SmithsRule.java` that reads in a command line parameter  $N$  and a list of  $N$  jobs specified by their processing time  $p[j]$  and

their weight  $w[j]$ , and output an optimal sequence in which to process their jobs. *Hint: Use Smith's rule:* schedule the jobs in order of their ratio of processing time to weight. This greedy rule turns out to be optimal.

33. **Sum of four primes.** The Goldbach conjecture says that all positive even integers greater than 2 can be expressed as the sum of two primes. Given an input parameter  $N$  (odd or even), express  $N$  as the sum of four primes (not necessarily distinct) or report that it is impossible to do so. To make your algorithm fast for large  $N$ , do the following steps:
- Compute all primes less than  $N$  using the Sieve of Eratosthenes.
  - Tabulate a list of sums of two primes.
  - Sort the list.
  - Check if there are two numbers in the list that sum to  $N$ . If so, print out the corresponding four primes.
34. **Typing monkeys and power laws.** (Micahel Mitzenmacher) Suppose that a [typing monkey](#) creates random words by appending each of 26 possible letter with probability  $p$  to the current word, and finishes the word with probability  $1 - 26p$ . Write a program to estimate the frequency spectrum of the words produced.
35. **Typing monkeys and power laws.** Repeat the previous exercise, but assume that the letters a-z occur proportional to the following probabilities, which are typical of English text.

CHAR	FREQ	CHAR	FREQ	CHAR	FREQ	CHAR	FREQ	CHAR	FREQ
A	8.04	G	1.96	L	4.14	Q	0.11	V	0.99
B	1.54	H	5.49	M	2.53	R	6.12	W	1.92
C	3.06	I	7.26	N	7.09	S	6.54	X	0.19
D	3.99	J	0.16	O	7.60	T	9.25	Y	1.73
E	12.51	K	0.67	P	2.00	U	2.71	Z	0.09
F	2.30								

36. **Binary search.** Justify why the following modified version of `binarySearch()` works. Prove that if the key is in the array, it correctly returns the smallest index  $i$  such that  $a[i] = \text{key}$ ; if the key is not in the array, it returns  $-i$  where  $i$  is the smallest index such that  $a[i] > \text{key}$ .

```
// precondition array a in ascending order
public static int binarySearch(long[] a, long key) {
    int bot = -1;
    int top = a.length;
    while (top - bot > 1) {
        int mid = bot + (top - bot) / 2;
        if (key > a[mid]) bot = mid;
        else top = mid;
    }
    if (a[top] == key) return top;
    else return -top - 1;
}
```

*Answer.* The while loop invariant says  $\text{top} \geq \text{bot} + 2$ . This implies  $\text{bot} < \text{mid} < \text{top}$ . Hence length of interval strictly decreases in each iteration. While loop also maintains the invariant:  $a[\text{bot}] < \text{key} \leq a[\text{top}]$ , with the contention that  $a[-1]$  is  $-\infty$  and  $a[N]$  is  $+\infty$ .

37. **Range search.** Given a database of all tolls collected in NJ road system in 2006, devise a scheme to answer queries of the form: extract sum of all tolls collected in a given time interval. Use a `Toll` data type that implements the `Comparable` interface, where the key is the time that the toll was collected.

*Hint:* sort by time, compute a cumulative sum of the first  $i$  tolls, then use binary search to find the desired interval.

38. **Longest repeated substrings.** Modify [LRS.java](#) to find *all* longest repeated substrings.
39. **Non-recursive binary search.** Write a non-recursive version of binary search.

```
public static int binarySearch(long[] a, long key) {  
    int bot = 0;  
    int top = a.length - 1;  
    while (bot <= top) {  
        int mid = bot + (top - bot) / 2;  
        if (key < a[mid]) top = mid - 1;  
        else if (key > a[mid]) bot = mid + 1;  
        else return mid;  
    }  
    return -1;  
}
```

40. **Two sum to x.** Given a sorted list of  $N$  integers and a target integer  $x$ , determine in  $O(N)$  time whether there are any two that sum to exactly  $x$ .

*Hint:* maintain an index  $lo = 0$  and  $hi = N-1$  and compute  $a[lo] + a[hi]$ . If the sum equals  $x$ , you are done; if the sum is less than  $x$ , decrement  $hi$ ; if the sum is greater than  $x$ , increment  $lo$ . Be careful if one (or more) of the integers are 0.

41. **Zero of a monotonic function.** Let  $f$  be a monotonically increasing function with  $f(0) < 0$  and  $f(N) > 0$ . Find the smallest integer  $i$  such that  $f(i) > 0$ . Devise an algorithm that makes  $O(\log N)$  calls to  $f()$ .

*Hint:* assuming we know  $N$ , maintaining an interval  $[lo, hi]$  such that  $f[lo] < 0$  and  $f[hi] > 0$  and apply binary search. If we don't know  $N$ , repeatedly compute  $f(1), f(2), f(4), f(8), f(16)$ , and so on until you find a value of  $N$  such that  $f(N) > 0$ .

42. **Bitonic max.** Let  $a[]$  be an array that starts out increasing, reaches a maximum, and then decreases. Design an  $O(\log N)$  algorithm to find the index of the maximum value.
43. **Bitonic search.** An array is *bitonic* if it is comprised of an increasing sequence of integers followed immediately by a decreasing sequence of integers. Given a bitonic array  $a$  of  $N$  distinct integers, describe how to determine whether a given integer is in the array in  $O(\log N)$  steps. Hint: find the maximum, then binary search in each piece.
44. **Median in two sorted arrays.** Given two sorted arrays of size  $N_1$  and  $N_2$ , find the median of all elements in  $O(\log N)$  time where  $N = N_1 + N_2$ . *Hint:* design a more general algorithm that finds the  $k$ th largest element for any  $k$ . Compute the median element in the large of the two lists and; throw away at least 1/4 of the elements and recur.
45. **Element distinctness.** Give an array of  $N$  long integers, devise an  $O(N \log N)$  algorithm to determine if any two are equal. *Hint:* sorting brings equal values together.
46. **Duplicate count.** Give a sorted array of  $N$  elements, possibly with duplicates, find the index of the first and last occurrence of  $k$  in  $O(\log N)$  time. Give a sorted array of  $N$  elements, possibly with duplicates, find the number of occurrences of element  $k$  in  $O(\log N)$  time. *Hint:* modify binary search.

**Scientific example of sorting.** Google display search results in descending order of "importance", a spreadsheet displays columns sorted by a particular field, Matlab sorts the real eigenvalues of a symmetric matrix in descending order. Sorting also arises as a critical subroutine in many applications that appear to have nothing to do with sorting at all including: data compression (see the Burrows-Wheeler programming assignment), computer graphics (convex hull, closest pair), computational biology (longest common substring discussed below), supply chain management (schedule jobs to minimize weighted sum of completion times),

combinatorial optimization (Kruskal's algorithm), social choice and voting (Kendall's tau distance), Historically, sorting was most important for commercial applications, but sorting also plays a major role in the scientific computing infrastructure. [NASA](#) and the fluids mechanics community use sorting to study problems in rarefied flow; these collision detection problems are especially challenging since they involve ten of billions of particles and can only be solved on supercomputers in parallel. Similar sorting techniques are used in some fast N-body simulation codes. Another important scientific application of sorting is for load balancing the processors of a parallel supercomputers. Scientists rely on clever sorting algorithm to perform load-balancing on such systems.

*Last modified on June 07, 2012.*

Copyright © 2002–2012 [Robert Sedgewick](#) and [Kevin Wayne](#). All rights reserved.



An Interdisciplinary Approach

Robert Sedgewick • Kevin Wayne

## INTRO TO PROGRAMMING

### 1. Elements of Programming

### 2. Functions

### 3. OOP

### 4. Data Structures

## INTRO TO CS

### 0. Prologue

### 5. A Computing Machine

### 6. Building a Computer

### 7. Theory of Computation

### 8. Systems

### 9. Scientific Computation

## ALGORITHMS, 4TH EDITION



## WEB RESOURCES

### FAQ

### Data

### Code

### Errata

### Appendices

### Lecture Slides

### Programming Assignments

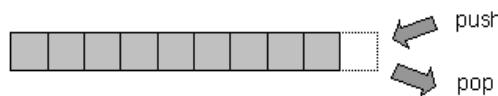
Search booksite...

## 4.3 STACKS AND QUEUES

This section under major construction.

**Stacks and queues.** In this section, we introduce two closely-related data types for manipulating arbitrarily large collections of objects: the *stack* and the *queue*. Each is defined by two basic operations: *insert* a new item, and *remove* an item. When we insert an item, our intent is clear. But when we remove an item, which one do we choose? The rule used for a queue is to always remove the item that has been in the collection the *most* amount of time. This policy is known as *first-in-first-out* or *FIFO*. The rule used for a stack is to always remove the item that has been in the collection the *least* amount of time. This policy is known as *last-in first-out* or *LIFO*.

**Pushdown stacks.** A *pushdown stack* (or just a *stack*) is a collection that is based on the last-in-first-out (LIFO) policy. When you click a hyperlink, your browser displays the new page (and inserts it onto a stack). You can keep clicking on hyperlinks to visit new pages. You can always revisit the previous page by clicking the back button (remove it from a stack). The last-in-first-out policy offered by a pushdown stack provides just the behavior that you expect.



By tradition, we name the stack *insert* method `push()` and the stack *remove* operation `pop()`. We also include a method to test whether the stack is empty. The following API summarizes the operations:

```
public class *StackOfStrings (pushdown stack for strings)
    *StackOfStrings() create a stack
    boolean isEmpty() is the stack empty?
    void push(String item) push a string onto the stack
    String pop() pop the stack
```

The asterisk indicates that we will be considering more than one implementation of this API.

**Array implementation.** Representing stacks with arrays is a natural idea. The first problem that you might encounter is implementing the constructor `ArrayStackOfStrings()`. An instance variable `a[]` with an array of strings to hold the stack items is clearly needed, but how big should it be? For the moment, We will finesse this problem by having the client provide an argument for the constructor that gives the maximum stack size. We keep the items in *reverse* order of their arrival. This policy allows us to add and remove items at the end without moving any of the other items in the stack.

We could hardly hope for a simpler implementation of `ArrayStackOfStrings.java`: all of the methods are one-liners! The instance variables are an array `a[]` that hold the items in the stack and an integer `N` that counts the number of items in the stack. To remove an item, we decrement `N` and then return `a[N]`; to insert a new item, we set `a[N]` equal to the new item and then increment `N`. These operations preserve the following properties: the items in the array are in their insertion order the stack is empty when the value of `N` is 0 the top of the stack (if it is nonempty) is at `a[N-1]`

StdIn	StdOut	N	a[]				
			0	1	2	3	4
		0					
to	1	to					
be	2	to be					
or	3	to be or					
not	4	to be or not					
to	5	to be or not to					
-	4	to be or not to					
be	5	to be or not be					
-	4	to be or not be					
-	3	to be or not be					
that	4	to be or that be					
-	3	to be or that be					
-	2	to be or that be					
-	1	to be or that be					
is	2	to is or not to					

Trace of `ArrayStackOfStrings` test client

The primary characteristic of this implementation is that *the push and pop operations take constant time*. The drawback of this implementation is that it requires the client to estimate the maximum size of the stack ahead of time and always uses space proportional to that maximum, which may be unreasonable in some situations.

**Linked lists.** For classes such as stacks that implement collections of objects, an important objective is to ensure that *the amount of space used is always proportional to the number of items in the collection*. Now we consider the use of a fundamental data structure known as a *linked list* that can provide

implementations of collections (and, in particular, stacks) that achieves this important objective.

A linked list is a recursive data structure defined as follows: *a linked list is either empty (null) or a reference to a node having a reference to a linked list*. The *node* in this definition is an abstract entity that might hold any kind of data in addition to the node reference that characterizes its role in building linked lists. With object-oriented programming, implementing linked lists is not difficult. We start with a simple example of a class for the node abstraction:

```
class Node {
    String item;
    Node next;
}
```

A *Node* has two instance variables: a *String* and a *Node*. The *String* is a placeholder in this example for any data that we might want to structure with a linked list (we can use any set of instance variables); the instance variable of type *Node* characterizes the linked nature of the data structure. Now, from the recursive definition, we can represent a linked list by a variable of type *Node* just by ensuring that its value is either *null* or a reference to a *Node* whose *next* field is a reference to a linked list.

We create an object of type *Node* by invoking its (no-argument) constructor. This creates a reference to a *Node* object whose instance variables are both initialized to the value *null*. For example, to build a linked list that contains the items "to", "be", and "or", we create a *Node* for each item:

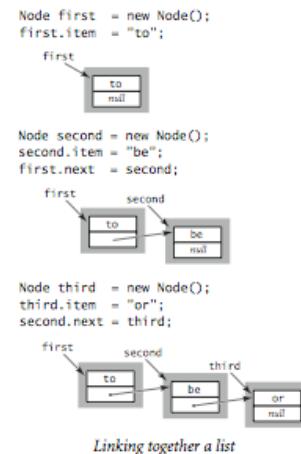
```
Node first = new Node();
Node second = new Node();
Node third = new Node();
```

and set the *item* field in each of the nodes to the desired item value:

```
first.item = "to";
second.item = "be";
third.item = "or";
```

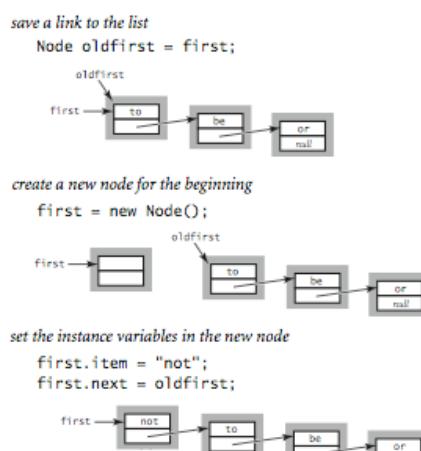
and set the *next* fields to build the linked list:

```
first.next = second;
second.next = third;
third.next = null;
```

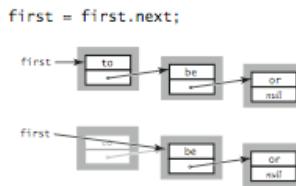


When tracing code that uses linked lists and other linked structures, we use a visual representation of the changes where we draw a rectangle to represent each object we put the values of instance variables within the rectangle we depict references as arrows that point to the referenced object This visual representation captures the essential characteristic of linked lists and allows us to focus on the links.

- *Insert*. Suppose that you want to insert a new node into a linked list. The easiest place to do so is at the beginning of the list. For example, to insert the string "not" at the beginning of a given linked list whose first node is *first*, we save *first* in *oldfirst*, assign to *first* a new *Node*, assign its *item* field to "not" and its *next* field to *oldfirst*.



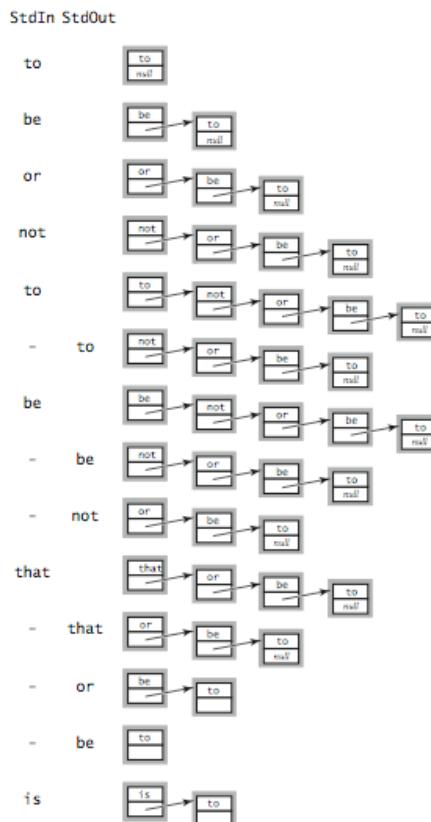
- **Remove.** Suppose that you want to remove the first node from a list. This operation is even easier: simply assign to `first` the value `first.next`. Normally, you would retrieve the value of the item (by assigning it to some `String` variable) before doing this assignment, because once you change the value of `first`, you may not have any access to the node to which it was referring. Typically, the node object becomes an orphan, and the memory it occupies is eventually reclaimed by the Java memory management system.



*Removing the first node in a linked list*

These two operations take constant time (independent of the length of the list).

**Implementing stacks with linked lists.** Program `LinkedStackOfStrings.java` uses a linked list to implement a stack of strings. The implementation is based on a *nested class* `Node` like the one we have been using. Java allows us to define and use other classes within class implementations in this natural way. The class is *private* because clients do not need to know any of the details of the linked lists.



*Trace of LinkedStackOfStrings test client*

**List traversal.** One of the most common operations we perform on collections is to iterate through the items in the collection. For example, we might wish to implement the `toString()` method to facilitate debugging our stack code with traces. For `ArrayStackOfStrings`, this implementation is familiar:

```

public String toString() {
    String s = "";
    for (int i = 0; i < N; i++)
        s += a[i] + " ";
    return s;
}

```

As usual, this solution is intended for use only when `N` is small - it takes quadratic time because string concatenation takes linear time. Our focus now is just on the process of examining every item. There is a corresponding idiom for visiting the items in a linked list: We initialize a loop index variable `x` that references the first `Node` of the linked list. Then, we find the value of the item associated with `x` by accessing `x.item`, and then update `x` to refer to the next `Node` in the linked list assigning to it the value of `x.next`, repeating this process until `x` is `null` (which indicates that we have reached the end of the

linked list). This process is known as *traversing* the list, and is succinctly expressed in this implementation of `toString()` for `LinkedStackOfStrings`:

```
public String toString() {
    String s = "";
    for (Node x = first; x != null; x = x.next)
        s += x.item + " ";
    return s;
}
```

**Array doubling.** Next, we consider an approach to accommodating arbitrary growth and shrinkage in a data structure that is an attractive alternative to linked lists. As with linked lists, the idea is to modify the array implementation to dynamically adjust the size of the array `a[]` so that it is (i) both sufficiently large to hold all of the items and (ii) not so large as to waste an excessive amount of space. Program `DoublingStackOfStrings.java` is a modification of `ArrayStackOfStrings.java` that achieves these objectives.

First, in `push()`, we check whether the array is too small. In particular, we check whether there is room for the new item in the array by checking whether the stack size `N` is equal to the array size `a.length`. If not, we just insert it with `a[N++] = item` as before; if so, we *double* the size of the array, by creating a new array of twice the size, copying the stack items to the new array, and resetting the `a[]` instance variable to reference the new array. Similarly, in `pop()`, we begin by checking whether the array is too large, and we *halve* its size if that is the case.

StdIn	StdOut	N	a.length	a							
				0	1	2	3	4	5	6	7
		0	1								
to		1	2	to		null					
be		2	2	to		be					
or		3	4	to		be	or		null		
not		4	4	to		be	or		not		
to		5	8	to		be	or	not	to	null	null
-	to	4	8	to		be	or	not	null	null	null
be		5	8	to		be	or	not	be	null	null
-	be	4	8	to		be	or	not	null	null	null
-	not	3	8	to		be	or		null	null	null
that		4	8	to		be	or		that	null	null
-	that	3	8	to		be	or		null	null	null
-	or	2	4	to		be		null	null		
-	be	1	2	to		null					
is		2	2	to		is					

*Trace of ArrayDoublingStackOfStrings test client*

**Parameterized data types.** We have developed one stack implementation that allows us to build a stack of one particular type (`String`). In other applications we might need a stack of integers or a stack of oranges or a queue of customers.

- *Create a stack of Objects.* We could develop one stack implementation `StackOfObjects.java` whose elements are of type `Object`. Using inheritance, we can insert an object of any type. However, when we pop it, we must cast it back to the appropriate type. This approach can expose us to subtle bugs in our programs that cannot be detected until runtime. For example, there is nothing to stop a programmer from putting different types of objects on the same stack, then encountering a runtime type-checking error, as in the following example:

```
StackOfObjects stack = new StackOfObjects();
Apple a = new Apple();
Orange b = new Orange();
stack.push(a);
stack.push(b);
a = (Apple) (stack.pop()); // throws a ClassCastException
b = (Orange) (stack.pop());
```

This toy example illustrates a basic problem. When we use type casting with an implementation such as `Stack` for different types of items, we are assuming that clients will cast objects popped from the stack to the proper type. This implicit assumption contradicts our requirement for ADTs that operations are to be accessed only through an explicit interface. One reason that programmers use precisely defined ADTs is to protect future clients against errors that arise from such implicit assumptions. The code cannot be type-checked at compile time: there might be an incorrect cast that occurs in a complex piece of code that could escape detection until some particular runtime circumstance arises. Such an error is to be avoided at all costs because it could happen long after an implementation is delivered to a client, who would have no way to fix it.

- *Java generics.* We use Java generics to *limit* the objects on a stack or queue to all be of the same type within a given application. The primary benefit is to discover type mismatch errors at compile-time instead of run-time. This involves a small bit of new Java syntax. We name the generic class

Stack. It is identical to StackOfStrings except that we replace every occurrence of String with Item and declare the class as follows:

```
public class Stack<Item>
```

Program [Stack.java](#) implements a generic stack using this approach. The client

```
Stack<Apple> stack = new Stack<Apple>();
Apple a = new Apple();
Orange b = new Orange();
stack.push(a);
stack.push(b);      // compile-time error
```

Program [DoublingStack.java](#) implements a generic stack using an array. For technical reasons, one cast is needed when allocating the array of generics.

**Autoboxing.** We have designed our stacks so that they can store any generic *object* type. We now describe the Java language feature, known as *auto-boxing* and *auto-unboxing*, that enables us to reuse the same code with primitive types as well. Associated with each primitive type, e.g. int, is a full blown object type, e.g., Integer. When we assign a primitive to the corresponding object type (or vice versa), Java automatically performs the transformation. This enables us to write code like the following.

```
Stack<Integer> stack = new Stack<Integer>();
stack.push(17);           // auto-boxing (converts int to Integer)
int a = stack.pop();     // auto-unboxing (converts Integer to int)
```

The value 17 is automatically cast to be of type Integer when we pass it to the push() method. The pop() method returns an Integer, and this value is cast to an int when we assign it to the variable a. We should be aware of what is going on behind the scenes since this can affect performance.

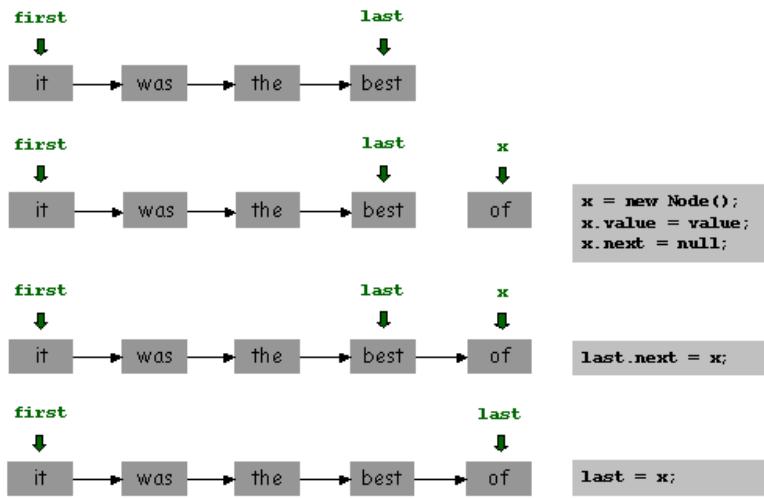
Java supplies built-in *wrapper types* for all of the primitive types: Boolean, Byte, Character, Double, Float, Integer, Long, and Short. These classes consist primarily of static methods (e.g., Integer.parseInt(), Integer.reverse()), but they also include some non-static methods (compareTo(), equals(), doubleValue()).

**Queue.** A queue supports the insert and remove operations using a FIFO discipline. By convention, we name the queue insert operation *enqueue* and the remove operation *dequeue*. Lincoln tunnel. Student has tasks that must be completed. Put on a queue. Do the tasks in the same order that they arrive.



```
public class Queue<Item> {
    public boolean isEmpty();
    public void enqueue(Item item);
    public Item dequeue();
}
```

- *Linked list implementation.* Program [Queue.java](#) implements a FIFO queue of strings using a linked list. Like Stack, we maintain a reference first to the least-recently added Node on the queue. For efficiency, we also maintain a reference last to the least-recently added Node on the queue.



- *Array implementation.* Similar to array implementation of stack, but a little trickier since need to wrap-around. Program [DoublingQueue.java](#) implements the queue interface. The array is dynamically resized using repeated doubling.

**Iteration.** Sometimes the client needs to access all of the items of a collection, one at a time, without deleting them. To maintain encapsulation, we do not want to reveal the internal representation of the queue (array or linked list) to the client. "Decouple the thing that needs to traverse the list from the details of getting each element from it." We solve this design challenge by using Java's `java.util.Iterator` interface:

```

public interface Iterator<Item> {
    boolean hasNext();
    Item next();
    void remove();      // optional
}

```

That is, any data type that implements the `Iterator` interface promises to implement two methods: `hasNext()` and `next()`. The client uses these methods to access the list elements one a time using the following idiom.

```

Queue<String> queue = new Queue<String>();
...
Iterator<String> i = queue.iterator();
while (i.hasNext()) {
    String s = i.next();
    StdOut.println(s);
}

```

- *Queue iterator in Java.* [Queue.java](#) illustrates how to implement an `Iterator` when the items are stored in a linked list.

```

public Iterator iterator() { return new QueueIterator(); }

private class QueueIterator implements Iterator<Item> {
    Node current = first;

    public boolean hasNext() { return current != null; }

    public Item next() {
        Item item = current.item;
        current = current.next;
        return item;
    }
}

```

It relies on a `private nested subclass` `QueueIterator` that implements the `Iterator` interface. The method `iterator()` creates an instance of type `QueueIterator` and returns it as an `Iterator`. This enforces the iteration abstraction since the client will only see the items through the `hasNext()` and `next()` methods. The client has no access to the internals of the `Queue` or even the `QueueIterator`. It is the client's responsibility to only add elements to the list when no iterator is in action.

- *Enhanced for loop.* Iteration is such a useful abstraction that Java provides compact syntax (known as the *enhanced for loop*) to iterate over the elements of a collection (or array).

```
Iterator<String> i = queue.iterator();
while (i.hasNext()) {
    String s = i.next();
    StdOut.println(s);
}

for (String s : queue)
    StdOut.println(s);
```

To take advantage of Java's enhanced foreach syntax, the data type must implement Java's `Iterable` interface.

```
public interface Iterable<Item> {
    Iterator<Item> iterator();
}
```

That is, the data type must implement a method named `iterator()` that returns an `Iterator` to the underlying collection. Since our `Queue` ADT now includes such a method, we simply need to declare it as implementing the `Iterable` interface and we are ready to use the foreach notation.

```
public class Queue<Item> implements Iterable<Item>
```

**Stack and queue applications.** Stacks and queues have numerous useful applications.

- Queue applications: Computing applications: serving requests of a single shared resource (printer, disk, CPU), transferring data asynchronously (data not necessarily received at same rate as sent) between two processes (IO buffers), e.g., pipes, file IO, sockets. Buffers on MP3 players and portable CD players, iPod playlist. Playlist for jukebox - add songs to the end, play from the front of the list. Interrupt handling: When programming a real-time system that can be interrupted (e.g., by a mouse click or wireless connection), it is necessary to attend to the interrupts immediately, before proceeding with the current activity. If the interrupts should be handled in the same order they arrive, then a FIFO queue is the appropriate data structure.
- *Arithmetic expression evaluation.* Program [Evaluate.java](#) evaluates a fully parenthesized arithmetic expression.

An important application of stacks is in *parsing*. For example, a compiler must parse arithmetic expressions written using *infix notation*. For example the following infix expression evaluates to 212.

```
( 2 + ( ( 3 + 4 ) * ( 5 * 6 ) ) )
```

We break the problem of parsing infix expressions into two stages. First, we convert from infix to a different representation called *postfix*. Then we parse the postfix expression, which is a somewhat easier problem than directly parsing infix.

- *Evaluating a postfix expression.* A postfix expression is....

```
2 3 4 + 5 6 * * +
```

First, we describe how to parse and evaluate a postfix expression. We read the tokens in one at a time. If it is an integer, push it on the stack; if it is a binary operator, pop the top two elements from the stack, apply the operator to the two elements, and push the result back on the stack. Program [Postfix.java](#) reads in and evaluates postfix expressions using this algorithm.

- *Converting from infix to postfix.* Now, we describe how to convert from infix to postfix. We read in the tokens one at a time. If it is an operator, we push it on the stack; if it is an integer, we print it out; if it is a right parentheses, we pop the topmost element from the stack and print it out; if it is a left parentheses, we ignore it. Program [Infix.java](#) reads in an infix expression, and uses a stack to output an equivalent postfix expression using the algorithm described above. Relate back to the parse tree example in Section 4.3.
- *Function calls.* Perhaps the most important application of stacks is to implement function calls. Most compilers implement function calls by using a stack. This also provides a technique for eliminating recursion from a program: instead of calling a function recursively, the programmer uses a stack to simulate the function calls in the same way that the compiler would have done so. Conversely, we can often use recursion instead of using an explicit stack. Some programming languages provide a

mechanism for recursion, but not for calling functions.

Programming languages have built in support for stacks (recursion), but no analogous mechanism for dealing with queues.

Postscript and FORTH programming languages are stack based. Java bytecode is interpreted on (virtual) stack based processor. Microsoft Intermediate Language (MSIL) that .NET applications are compiled to.

- *M/M/1 queue.* The [Markov/Markov/Single-Server model](#) is a fundamental queueing model in operations research and probability theory. Tasks arrive according to a *Poisson process* at a certain rate  $\lambda$ . This means that  $\lambda$  customers arrive per hour. More specifically, the arrivals follow an exponential distribution with mean  $1 / \lambda$ : the probability of  $k$  arrivals between time 0 and  $t$  is  $(\lambda t)^k e^{(-\lambda t)} / k!$ . Tasks are serviced in FIFO order according to a Poisson process with rate  $\mu$ . The two M's standard for Markov: it means that the system is *memoryless*: the time between arrivals is independent, and the time between departures is independent.

Analysis of M/M/1 model. We are interested in understanding the queueing system. If  $\lambda > \mu$  the queue size increases without limit. For simple models like M/M/1 we can analyze these quantities analytically using probability theory. Assuming  $\mu > \lambda$ , the probability of exactly  $n$  customers in the system is  $(\lambda / \mu)^n (1 - \lambda / \mu)^{n-1}$ .

- $L = \text{average number of customers in the system} = \lambda / (\mu - \lambda)$ .
- $L_Q = \text{average number of customers in the queue} = \lambda^2 / (\mu (\mu - \lambda))$ .
- $W = \text{average time a customer spends in the system} = 1 / (\mu - \lambda)$ .
- $W_Q = \text{average time a customer spends in the queue} = W - 1 / \mu$ .

Program [MM1Queue.java](#) For more complex models we need to resort to simulation like this. Variants: multiple queues, multiple servers, sequential multi-stage servers, using a finite queue and measuring number of customers that are turned away. Applications: customers in McDonalds, packets in an internet router,

[Little's law](#) asserts that the average number of customers in a (stable) queueing system equals the average arrival rate times their average time in the system. But the variance of customer waiting times satisfies:  $\text{Var(FIFO)} < \text{Var(SIRO)} < \text{Var(LIFO)}$ .

The distribution of the number of customers in the system does not depend on the queueing discipline (so long as it is independent of their service times). Same for expected waiting time.

- *M/D/1 queue.* Program [MD1Queue.java](#) is similar but the service occurs at a fixed rate (rather than random).
- *Load balancing.* Write a program [LoadBalance.java](#) that performs a load-balancing simulation.

## Q + A.

**Q.** When do I use new with Node?

**A.** Just as with any other class, you should only use new when you want to create a new `Node` object (a new element in the linked list). You should not use new to create a new reference to an existing `Node` object. For example, the code

```
Node oldfirst = new Node();
oldfirst = first;
```

creates a new `Node` object, then immediately loses track of the only reference to it. This code does not result in an error, but it is a bit untidy to create orphans for no reason.

**Q.** Why declare Node as a nested class? Why private?

**A.** By declaring the subclass `Node` to be `private` we restrict access to methods within the enclosing class. One characteristic of a private nested class is that its instance variables can be directly accessed from within the enclosing class, but nowhere else, so there is no need to declare them public or private. Note : A nested class that is not static is known as an *inner class*, so technically our `Node` classes are inner classes, though the ones that are not generic could be static.

**Q.** Why does javac `LinkedStackOfStrings.java` creates a file

`LinkedStackOfStrings$Node.class` as well as `LinkedStackOfStrings.class`?

**A.** That file is for the nested class `Node`. Java's naming convention is to use \$ to separate the name of the outer class from the nested class.

**Q.** Should a client be allowed to insert `null` items onto a stack or queue? A. This question arises frequently when implementing collections in Java. Our implementation (and Java's stack and queue libraries) do permit the insertion of `null` values.

**Q.** Are there Java libraries for stacks and queues?

**A.** Yes and no. Java has a built in library called `java.util.Stack`, but you should avoid using it when you want a stack. It has several additional operations that are not normally associated with a stack, e.g., getting the  $i$ th element. It also allows adding an element to the bottom of the stack (instead of the top), so it can implement a queue! Although having such extra operations may appear to be a bonus, it is actually a curse. We use data types not because they provide every available operation, but rather

because they allow us to precisely specify the operations we need. The prime benefit of doing so is that the system can prevent us from performing operations that we do not actually want. The `java.util.Stack` API is an example of a *wide interface*, which we generally strive to avoid.

**Q.** I want to use an array representation for a generic stack, but code like the following will not compile. What is the problem?

```
private Item[] a = new Item[max];
oldfirst = first;
```

**A.** Good try. Unfortunately, creating arrays of generics is not allowed in Java 1.5. Experts still are vigorously debating this decision. As usual, complaining too loudly about a language feature puts you on the slippery slope towards becoming a language designer. There is a way out, using a cast: you can write:

```
private Item[] a = (Item[]) new Object[max];
oldfirst = first;
```

The underlying cause is that arrays in Java are *covariant*, but generics are not. In other words, `String[]` is a subtype of `Object[]`, but `Stack<String>` is not a subtype of `Stack<Object>`. To get around this defect, you need to perform an unchecked cast as in [DoublingStack.java](#). Many programmers consider covariant arrays to be a defect in Java's type system (and this resulted in the need for "reifiable types" and "type erasure"). However, in a world without generics, covariant arrays are useful, e.g., to implement `Arrays.sort(Comparable[])` and have it be callable with an input array of type `String[]`.

**Q.** Can I use the `foreach` construction with arrays?

**A.** Yes (even though arrays do not implement the `Iterator` interface). The following prints out the command-line arguments:

```
public static void main(String[] args) {
    for (String s : args)
        StdOut.println(s);
}
```

**Q.** Is iterating over a linked list more efficient with a loop or recursion?

**A.** An optimizing compiler will likely translate a tail-recursive function into the equivalent loop, so there may be no observable performance overhead of using recursion.

**Q.** How does auto-boxing handle the following code fragment?

```
Integer a = null;
int b = a;
```

**A.** It results in a run-time error. Primitive type can store every value of their corresponding wrapper type except `null`.

**Q.** Why does the first group of statements print `true`, but the second two print `false`?

```
Integer a1 = 100;
Integer a2 = 100;
System.out.println(a1 == a2);    // true

Integer b1 = new Integer(100);
Integer b2 = new Integer(100);
System.out.println(b1 == b2);    // false

Integer c1 = 150;
Integer c2 = 150;
System.out.println(c1 == c2);    // false
```

**A.** The second prints `false` because `b1` and `b2` are references to different `Integer` objects. The first and third code fragments rely on autoboxing. Surprisingly the first prints `true` because values between -128 and 127 appear to refer to the same immutable `Integer` objects (presumably there is a pool of them that are reused), while Java creates new objects for each integer outside this range. Lesson: as usual, don't use `==` to compare whether two objects have the same value.

**Q.** Are generics solely for auto-casting?

**A.** No, but this will be the only thing we use them for. This is known as "pure generics" or "concrete parameterized types." Concrete parameterized types work almost like normal types with a few exceptions (array creation, exception handling, with `instanceof`, and in a class literal). More advanced uses of generics, including "wildcards", are useful for handling subtypes and inheritance. Here is a [generics tutorial](#).

**Q.** Why do I get an incompatible types compile-time error with the following code?

```
Stack stack = new Stack<String>();
stack.push("Hello");
String s = stack.pop();
```

**A.** You forgot to specify the concrete type when declaring `stack`. It should be `Stack<String>`.

**Q.** Why do I get a uses unchecked or unsafe operations compile-time warning with the following code?

```
Stack<String> stack = new Stack();
stack.push("Hello");
String s = stack.pop();
```

**A.** You forgot to specify the concrete type when calling the constructor. It should be `new Stack<String>()`.

## Exercises

1. Add a method `isFull()` to [ArrayStackOfStrings.java](#).
2. Give the output printed by `java ArrayStackOfStrings 5` for the input

```
it was - the best - of times - - - it was - the - -
```

3. Suppose that a client performs an intermixed sequence of (stack) push and pop operations. The push operations put the integers 0 through 9 in order on to the stack; the pop operations print out the return value. Which of the following sequence(s) could not occur?

- (a) 4 3 2 1 0 9 8 7 6 5
- (b) 4 6 8 7 5 3 2 9 0 1
- (c) 2 5 6 7 4 8 9 3 1 0
- (d) 4 3 2 1 0 5 6 7 8 9
- (e) 1 2 3 4 5 6 9 8 7 0
- (f) 0 4 6 5 3 8 1 7 2 9
- (g) 1 4 7 9 8 6 5 3 0 2
- (h) 2 1 4 3 6 5 8 7 9 0

4. Write a stack client [Reverse.java](#) that reads in strings from standard input and prints them in reverse order.
5. Assuming that standard input has some unknown number `N` of `double` values. Write a method that reads all the values and returns an array of length `N` containing them, in the order they appear on standard input.
6. Write a stack client [Parentheses.java](#) that reads in a text stream from standard input and uses stack to determine whether its parentheses are properly balanced. For example, your program should print `true` for `[(()){}{{()()}}()` and `false` for `[(])`. Hint : Use a stack.
7. What does the following code fragment print when `N` is 50? Give a high-level description of what the code fragment does when presented with a positive integer `N`.

```
Stack stack = new Stack();
while (N > 0) {
    stack.push(N % 2);
    N = N / 2;
}
while (!stack.isEmpty())
    StdOut.print(stack.pop());
StdOut.println();
```

Answer: prints the binary representation of N (110010 when N is 50).

8. What does the following code fragment do to the queue `q`?

```
Stack stack = new Stack();
while (!q.isEmpty())
    stack.push(q.dequeue());
while (!stack.isEmpty())
    q.enqueue(stack.pop());
```

9. Add a method `peek()` to [Stack.java](#) that returns the most recently inserted element on the stack (without popping it).

10. Give the contents and size of the array for `DoublingStackOfStrings` with the input

```
it was - the best - of times - - - it was - the - -
```

11. Add a method `length()` to [Queue.java](#) that returns the number of elements on the queue. Hint: Make sure that your method takes constant time by maintaining an instance variable `N` that you initialize to 0, increment in `enqueue()`, decrement in `dequeue()`, and return in `length()`.
12. Draw a memory usage diagram in the style of the diagrams in Section 4.1 for the three-node example used to introduce linked lists in this section.
13. Write a program that takes from standard input an expression without left parentheses and prints the equivalent infix expression with the parentheses inserted. For example, given the input

```
1 + 2 ) * 3 - 4 ) * 5 - 6 ) ) )
```

your program should print

```
( ( 1 + 2 ) * ( ( 3 - 4 ) * ( 5 - 6 ) ) )
```

14. Write a filter [InfixToPostfix.java](#) that converts an arithmetic expression from infix to postfix.
15. Write a program [EvaluatePostfix.java](#) that takes a postfix expression from standard input, evaluates it, and prints the value. (Piping the output of your program from the previous exercise to this program gives equivalent behavior to [Evaluate.java](#).)
16. Suppose that a client performs an intermixed sequence of (queue) `enqueue` and `dequeue` operations. The enqueue operations put the integers 0 through 9 in order on to the queue; the dequeue operations print out the return value. Which of the following sequence(s) could *not* occur?

```
(a) 0 1 2 3 4 5 6 7 8 9
(b) 4 6 8 7 5 3 2 9 0 1
(c) 2 5 6 7 4 8 9 3 1 0
(d) 4 3 2 1 0 5 6 7 8 9
```

17. Write an iterable `Stack` *client* that has a static methods `copy()` that takes a stack of strings as argument and returns a copy of the stack. *Note:* This ability is a prime example of the value of having an iterator, because it allows development of such functionality without changing the basic API.
18. Develop a class [DoublingQueueOfStrings.java](#) that implements the queue abstraction with a fixed-size array, and then extend your implementation to use array doubling to remove the size restriction.
19. Write a [Queue.java](#) client that takes a command-line argument `k` and prints the `k`th from the last string found on standard input.
20. (For the mathematically inclined.) Prove that the array in [DoublingStackOfStrings.java](#) is never less than one-quarter full. Then prove that, for any `DoublingStackOfStrings` client, the total cost of all of the stack operations divided by the number of operations is a constant.
21. Modify [MD1Queue.java](#) to make a program [MM1Queue.java](#) that simulates a queue for which both the arrival and service times are Poisson processes. Verify Little's law for this model.
22. Develop a class [StackOfInts.java](#) that uses a linked-list representation (but no generics). Write a client that compares the performance of your implementation with `Stack<Integer>` to determine the performance penalty from autoboxing on your system.

## Linked List Exercises

23. Write a method `delete()` that takes an `int` argument `k` and deletes the `k`th element in a linked list, if it exists.

*Solution.*

```
// we assume that first is a reference to the first Node in the list
public void delete(int k) {
    if (k <= 0) throw new RuntimeException("Invalid value of k");

    // degenerate case - empty linked list
    if (first == null) return;

    // special case - removing the first node
    if (k == 1) {
        first = first.next;
        return;
    }

    // general case, make temp point to the (k-1)st node
    Node temp = first;
    for (int i = 2; i < k; i++) {
        temp = temp.next;
        if (temp == null) return; // list has < k nodes
    }

    if (temp.next == null) return; // list has < k nodes

    // change temp.next to skip kth node
    temp.next = temp.next.next;
}
```

24. Write a method `find()` that takes a linked list and a string `key` as arguments and returns `true` if some node in the list has `key` as its `item` field, `false` otherwise.
25. Suppose `x` is a linked-list node. What is the effect of the following code fragment?

```
x.next = x.next.next;
```

*Answer:* Deletes from the list the node immediately following `x`.

26. Suppose that `x` is a linked-list node. What is the effect of the following code fragment?

```
t.next = x.next;
x.next = t;
```

*Answer:* Inserts node `t` immediately after node `x`.

27. Why does the following code fragment not have the same effect as in the previous question?

```
x.next = t;
t.next = x.next;
```

*Answer:* When it comes time to update `t.next`, `x.next` is no longer the original node following `x`, but is instead `t` itself!

28. Write a method `removeAfter()` that takes a linked-list `Node` as argument and removes the node following the given one (and does nothing if the argument or the next field in the argument node is null).
29. Write a method `insertAfter()` that takes two linked-list `Node` arguments and inserts the second after the first on its list (and does nothing if either argument is null).
30. Write a method `remove()` that takes a linked list and a string `key` as arguments and removes all of the nodes in the list that have `key` as its `item` field.
31. Write a method `max()` that takes a reference to the first node in a linked list as argument and returns the value of the maximum key in the list. Assume that all keys are positive integers, and return 0 if the list is empty.
32. Develop a recursive solution to the previous exercise.
33. Write a recursive method to print the elements of a linked list in reverse order. Do not modify any of the links. *Easy:* Use quadratic time, constant extra space. *Also easy:* Use linear time, linear extra space. *Not so easy:* Develop a divide-and-conquer algorithm that uses linearithmic time and logarithmic extra space.
34. Write a recursive method to randomly shuffle the elements of a linked list by modifying the links.

*Easy:* Use quadratic time, constant extra space. *Not so easy:* Develop a divide-and-conquer algorithm that uses linearithmic time and logarithmic extra space.

## Creative Exercises

1. **Deque** A *double-ended queue* or *deque* (pronounced deck) is a combination of a stack and a queue. It stores a parameterized collection of items and supports the following API:

<u>public class Deque&lt;Item&gt; (generic double-ended queue)</u>	
Deque()	<i>create a deque</i>
boolean isEmpty()	<i>is the deque empty?</i>
void enqueue(Item item)	<i>add an item to the end</i>
void push(Item item)	<i>add an item to the beginning</i>
Item pop()	<i>remove an item from the beginning</i>
Item dequeue()	<i>remove an item from the end</i>

Write a data type `Deque.java` that implements the deque API using a singly linked list.

2. **Random queue.** Create an abstract data type `RandomizedQueue.java` that supports the following operations: `isEmpty()`, `insert()`, `random()`, and `removeRandom()`, where the deletion operation deletes and returns a random object. *Hint:* maintain an array of objects. To delete an object, swap a random object (indexed 0 through  $N-1$ ) with the last object (index  $N-1$ ). Then, delete and return the last object.

<u>public class RandomQueue&lt;Item&gt; (generic random queue)</u>	
RandomQueue()	<i>create a random queue</i>
boolean isEmpty()	<i>is the queue empty?</i>
void enqueue(Item item)	<i>add an item</i>
Item dequeue()	<i>remove a random item (sample without replacement)</i>
Item sample()	<i>return a random item (do not remove) (sample with replacement)</i>

3. **Listing files.** A Unix directory is a list of files and directories. Program `Directory.java` takes the name of a directory as a command line parameter and prints out all of the files contained in that directory (and any subdirectories) in level-order. It uses a queue.
4. **Josephus problem** Program `Josephus.java` uses a queue to solve the Josephus problem.
5. **Delete ith element.** Create an ADT that supports the following operations: `isEmpty`, `insert`, and `remove(int i)`, where the deletion operation deletes and returns the  $i$ th least recently added object on the queue. Do it with an array, then do it with a linked list. See Exercise XYZ for a more efficient implementation that uses a BST.
6. **Dynamic shrinking.** With the array implementations of stack and queue, we doubled the size of the array when it wasn't big enough to store the next element. If we perform a number of doubling operations, and then delete a lot of elements, we might end up with an array that is much bigger than necessary. Implement the following strategy: whenever the array is 1/4 full or less, shrink it to half the size. Explain why we don't shrink it to half the size when it is 1/2 full or less.
7. **Ring buffer.** A *ring buffer* or *circular queue* is a FIFO data structure of a fixed size  $N$ . It is useful for transferring data between asynchronous processes or storing log files. When the buffer is empty, the consumer waits until data is deposited; when the buffer is full, the producer waits to deposit data. A ring buffer has the following methods: `isEmpty()`, `isFull()`, `enqueue()`, and `dequeue()`. Write an generic data type `RingBuffer` using an array (with circular wrap-around for efficiency).
8. **Merging two sorted queues.** Given two queues with strings in ascending order, move all of the strings to a third queue so that the third queue ends up with the strings in ascending order.
9. **Mergesort.** Given  $N$  strings, create  $N$  queues, each containing one of the strings. Create a queue of the  $N$  queues. Then repeatedly apply the sorted merging operation to the first two queues and reinsert the merged queue at the end. Repeat until the queue of queues contains only one queue.
10. **Queue with two stacks.** Show how to implement a queue using two stacks. *Hint:* If you push elements onto a stack and then pop them all, they appear in reverse order. If you repeat this process, they're now back in order.
11. **Move-to-front.** Read in a sequence of characters from standard input and maintain the characters in a linked list with no duplicates. When you read in a previously unseen character, insert it at the front of the list. When you read in a duplicate character, delete it from the list and re-insert it at the beginning. This *move-to-front* strategy is useful for caching and data compression (Burrows-Wheeler) algorithms where items that have been recently accessed are more likely to be re-accessed.
12. **Text editor buffer.** Implement an ADT for a buffer in a text editor. It should support the following operations:
  - `insert(c)`: insert character  $c$  at cursor
  - `delete()`: delete and return the character at the cursor
  - `left()`: move the cursor one position to the left

- `right()`: move the cursor one position to the right
- `get(i)`: return the *i*th character in the buffer

*Hint:* use two stacks.

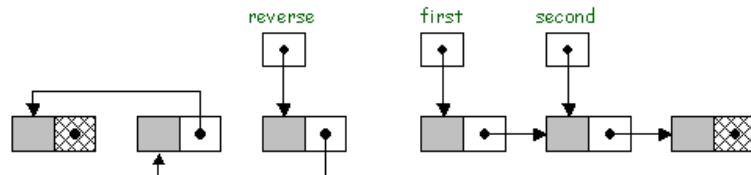
13. **Topological sort.** You have to sequence the order of *N* jobs on a processor. Some of the jobs must complete before others can begin. Specifically, you are given a list of order pairs of jobs  $(i, j)$ . Find a sequence of the jobs such that for each pair  $(i, j)$  job *i* is scheduled before job *j*. Use the following algorithm.... For each node, maintain a list of outgoing arcs using a queue. Also, maintain the indegree of each node. Finally, maintain a queue of all nodes whose indegree is 0. Repeatedly delete a node with zero indegree, and delete all of its outgoing arcs. Write a program [TopologicalSorter.java](#) to accomplish this.

Alternate application: prerequisites for graduating in your major. Must take COS 126 and COS 217 before COS 341, etc. Can you graduate?

14. **PERT / CPM.** Modify the previous exercise to handle weights  $(i, j, w)$  means job *i* is scheduled at least *w* units of time before job *j*.
15. **Set of integers.** Create a data type that represents a set of integers (no duplicates) between 0 and *N*-1. Support `add(i)`, `exists(i)`, `remove(i)`, `size()`, `intersect`, `difference`, `symmetricDifference`, `union`, `isSubset`, `isSuperSet`, and `isDisjointFrom`.
16. **Indexing a book.** Write a program that reads in a text file from standard input and compiles an alphabetical index of which words appear on which lines, as in the following input. Ignore case and punctuation. Similar to `FrequencyCount`, but for each word maintain a list of location on which it appears.

**Reverse a linked list.** Write a function that takes the first `Node` in a linked list, reverse it, and returns the first `Node` in the resulting linked list.

*Solution.* To accomplish this, we maintain references to three consecutive nodes in the linked list, `reverse`, `first`, and `second`. At each iteration we extract the node `first` from the original linked list and insert it at the beginning of the reversed list. We maintain the invariant that `first` is the first node of what's left of the original list, `second` is the second node of what's left of the original list, and `reverse` is the first node of the resulting reversed list.



```
public static Node reverse(Node list) {
    Node first = list;
    Node reverse = null;
    while (first != null) {
        Node second = first.next;
        first.next = reverse;
        reverse = first;
        first = second;
    }
    return reverse;
}
```

When writing code involving linked lists, we must always be careful to properly handle the exceptional cases (when the linked list is empty, when the list has only one or two nodes) and the boundary cases (dealing with the first or last items). This is usually the trickiest part, as opposed to handling the normal cases.

*Recursive solution.* Assuming the linked list has *N* elements, we recursively reverse the last *N*-1 elements, then carefully append the first element to the end.

```
public Node reverse(Node first) {
    if (first == null || first.next == null) return first;
    Node second = first.next;
    Node rest = reverse(second);
    second.next = first;
    first.next = null;
    return rest;
}
```

## Web Exercises

1. **Quote.** Develop a data type [Quote.java](#) that implements the following API:

```

public class Quote
-----
    Quote()
    void addWord(String w)
    int count()
    String getWord(int i)
    void insertWord(int i, String w)
    String toString()

```

create an empty quote  
 add w to the end of the quote  
 return number of words in quote  
 return the ith word starting at 1  
 add w after the ith word  
 return the entire quote as a String

To do so, define a nested class `Card` that holds one word of the quote and a link to the next word:

```

private class Card {
    private String word;
    private Card next;

    public Card(String word) {
        this.word = word;
        this.next = null;
    }
}

```

2. **Circular quote.** Repeated the previous exercise, but write a program [CircularQuote.java](#) that use a circular linked list.
3. Write a recursive function that takes as input a queue, and rearranges it so that it is in reverse order. Hint: `dequeue()` the first element, recursively reverse the queue, and the enqueue the first element.
4. Add a method `Item[] multiPop(int k)` to `Stack` that pops k elements from the stack and returns them as an array of objects.
5. Add a method `Item[] toArray()` to `Queue` that returns all N elements on the queue as an array of length N.
6. What does the following code fragment do?

```

IntQueue q = new IntQueue();
q.enqueue(0);
q.enqueue(1);
for (int i = 0; i < 10; i++) {
    int a = q.dequeue();
    int b = q.dequeue();
    q.enqueue(b);
    q.enqueue(a + b);
    System.out.println(a);
}

```

### Fibonacci

7. What data type would you choose to implement an "Undo" feature in a word processor?
8. Suppose you have a single array of size N and want to implement two stacks so that you won't get overflow until the total number of elements on both stacks is N+1. How would you proceed?
9. Suppose that you implemented `push` in the linked list implementation of `StackList` with the following code. What is the mistake?

```

public void push(Object value) {
    Node second = first;
    Node first = new Node();
    first.value = value;
    first.next = second;
}

```

**Answer:** By redeclaring `first`, you are create a new local variable named `first`, which is different from the instance variable named `first`.

10. **Copy a queue.** Create a new constructor so that `LinkedQueue r = new LinkedQueue(q)` makes `r` reference a new and independent queue. Hint: delete all of the elements from `q` and add to both `q` and `this`.
11. **Copy a stack.** Create a new constructor for the linked list implementation of `Stack.java` so that `Stack t = new Stack(s)` makes `t` reference a new and independent copy of the stack `s`. You should be able to push and pop from `s` or `t` without influencing the other.

Should it work if argument is `null`? **Recursive solution:** create a copy constructor for a `Node` and

use this to create the new stack.

```
Node(Node x) {  
    item = x.item;  
    if (x.next != null) next = new Node(x.next);  
}  
  
public Stack(Stack s) { first = new Node(s.first); }
```

*Nonrecursive solution (untested):*

```
Node(Node x, Node next) { this.x = x; this.next = next; }  
  
public Stack(Stack s) {  
    if (s.first != null) {  
        first = new Node(s.first.value, s.first.next);  
        for (Node x = first; x.next != null; x = x.next)  
            x.next = new Node(x.next.value, x.next.next);  
    }  
}
```

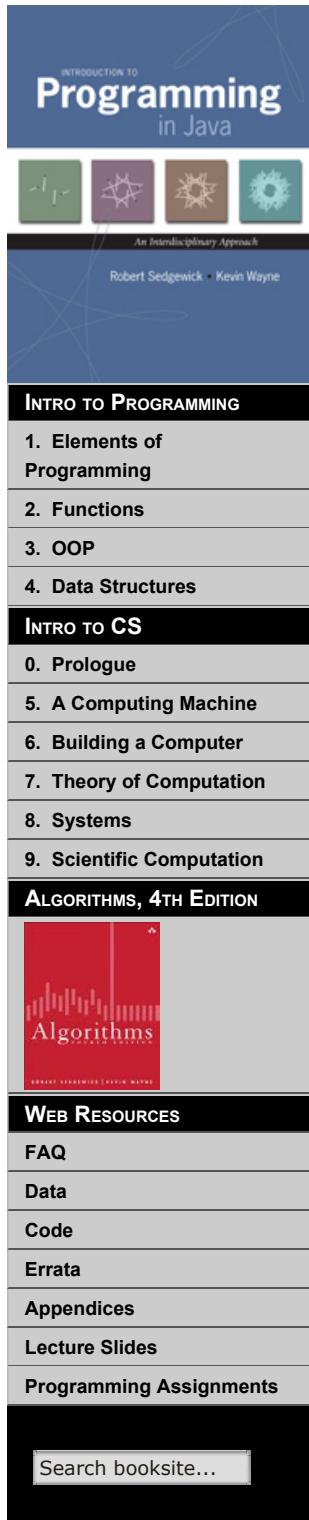
12. **Stack with one queue.** Show how to implement a stack using one queue. *Hint:* to delete an item, get all of the elements on the queue one at a time, and put them at the end, except for the last one which you should delete and return.
13. **Listing files with a stack.** Write a program that takes the name of a directory as a command line argument, and prints out all of the files contained in this directory and any subdirectories. Also prints out the file size (in bytes) of each file. Use a stack instead of a queue. Repeat using recursion and name your program [DirectoryR.java](#). Modify [DirectoryR.java](#) so that it prints out each subdirectory and its total size. The size of a directory is equal to the sum of all of the files it contains or that its subdirectories contain.
14. **Stack + max.** Create a data structure that efficiently supports the stack operations (pop and push) and also return the maximum element. Assume the elements are integers or reals so that you can compare them. *Hint:* use two stacks, one to store all of the elements and a second stack to store the maximums.
15. **Tag systems.** Write a program that reads in a binary string from the command line and applies the following (00, 1101) tag-system: if the first bit is 0, delete the first three bits and append 00; if the first bit is 1, delete the first three bits and append 1101. Repeat as long as the string has at least 3 bits. Try to determine whether the following inputs will halt or go into an infinite loop: 10010, 100100100100100. Use a queue.
16. **Reverse.** Write a method to read in an arbitrary number of strings from standard input and print them in reverse order.

```
public static void main(String[] args) {  
    Stack<String> stack = new Stack<String>();  
    while (!StdIn.isEmpty()) {  
        String s = StdIn.readString();  
        stack.push(s);  
    }  
    while (!stack.isEmpty()) {  
        String s = stack.pop();  
        StdOut.println(s);  
    }  
}
```

17. Add a method `int size()` to [DoublingStack.java](#) and [Stack.java](#) that returns the number of elements on the stack.
18. Add a method `reverse()` to [Queue](#) that reverses the order of the elements on the queue.
19. Add a method `copy()` to [ArrayStackOfStrings.java](#)

*Last modified on April 11, 2012.*

Copyright © 2002–2012 Robert Sedgewick and Kevin Wayne. All rights reserved.



## 4.4 SYMBOL TABLES

This section under major construction.

**Symbol tables.** A *symbol table* is a data type that we use to associate *values* with *keys*. Clients can store (*put*) an entry into the symbol table by specifying a key-value pair and then can retrieve (*get*) the value corresponding to a particular key from the symbol table. For example, a university might associate information such as a student's name, home address, and grades (the *value*) with that student's social security number (the *key*), so that each student's records can be accessed by specifying a social security number. The same approach might be used by a scientist to organize data, by a business to keep track of customer transactions, or by an internet search engine to associate keywords with web pages.

**API.** A symbol table is a collection of key-value pairs. We use a generic type *Key* for keys and a generic type *Value* for values: every symbol-table entry associates a *Value* with a *Key*. In most applications, the keys have some natural ordering, so (as we did with sorting) we require the key type *Key* to implement Java's *Comparable* interface.

```
public class *ST<Key extends Comparable<Key>, Value>
    *ST()
    void put(Key key, Value v)    // put key-value pair into the table
    Value get(Key key)           // return value paired with key
                                // or null if no such value
    boolean contains(Key key)    // is there a value paired with key?
```

This API reflects several design decisions, which we now enumerate:

- *Comparable keys.* We take advantage of key ordering to develop efficient implementations of *put* and *get*. We also assume the keys do not change their value while in the symbol table. The simplest and most commonly used types of keys, *String* and built-in wrapper types like *Integer* and *Double*, are immutable.
- *Replace-the-old-value policy.* If when a key-value pair is inserted into the symbol table that already associates another value with the given key, we adopt the convention that the new value replaces the old one (just as with an array assignment statement).
- *Not found.* The method *get()* returns *null* if no entry with the given key has previously been put into the table. This choice has two implications, discussed next.
- *Null keys and values.* Clients are not permitted to use *null* as a key or value. This convention enables us to implement *contains()* as follows:

```
public boolean contains(Key key) {
    return get(key) != null;
}
```

- *Remove.* We do not include a method for removing keys from the symbol table. Many applications do require such a method, but we leave implementations as an exercise or for more advanced courses in data structures in algorithms. Since clients cannot associate *null* with a key, one simple interface is to take a *put* command with *null* as the value to mean *remove*.
- *Iterable.* As with most collections, it is best to implement *Iterable* and to provide an appropriate iterator that allows clients to visit the contents of the table. The natural order of iteration is in order of the keys, so we implement *Iterable<Key>* and use *get* to get values, if desired.
- *Variations.* Numerous other useful operations on symbol tables have been identified, and APIs based on various subsets of them have been widely studied. We will consider several of these at the end of this section.

**Symbol table clients.** We consider two prototypical examples.

- *Dictionary lookup.* The most basic kind of symbol-table client builds a symbol table with successive *put* operations in order to be able to support *get* requests. The following list of familiar examples illustrates the utility of this approach.

Application	Action	Key	Value
phone book	look up phone number	person's name	phone number
dictionary	look up word	word	definition

Internet DNS	Look up website by IP address	website	IP address
Reverse DNS	Look up IP address by web site	IP address	Website
genomics	amino acid dictionary	codon	amino acid
Java compiler	Find properties of variable	Variable name	Value and type
stock quote	Look up price of stock	stock symbol	price
file share	find song to download	song name	machine
file system	find file on hard drive	file name	location on hard drive

Program [Lookup.java](#) builds a set of key-value pairs from a file of comma-separated values and then prints out values corresponding to keys read from standard input. The command-line arguments are the file name and two integers, one specifying the field to serve as the key and the other specifying the field to serve as the value.

Here are some sample data files: [amino.csv](#) (codons and amino acids), [DJIA.csv](#) (Dow Jones Industrial average by date), [ip.csv](#) (hostnames and IP addresses), [morse.csv](#) (Morse code), [elements.csv](#) (Periodic table of elements), [mktsymbols.csv](#) (market symbols and names), [toplevel-domain.txt](#) (top-level domain names and their country).

- *Indexing.* Program [Index.java](#) is a prototypical example of a symbol table client that uses an intermixed sequence of calls to `get()` and `put()`: it reads in a list of strings from standard input and prints a sorted table of all the different strings along with a list of integers specifying the positions where each string appeared in the input. We have a large amount of data and want to know where certain strings of interest are found. In this case, we seem to be associating multiple values with each key, but we actually associating just one: a queue. Again, this approach is familiar:

Application	Action	Key	Value
book index	search for terms	term	page numbers
genomics	find genetic markers	DNA substring	locations
web search	find information on web	keyword	websites
business	find transactions	customer name	transactions

**Elementary implementations.** Symbol-table implementations have been heavily studied. Many different algorithms and data structures have been invented for this purpose. We first describe two simple approaches.

- *Binary search implementation.* Program [BinarySearchST.java](#) implements a symbol table by maintaining two parallel arrays of keys and values, keeping them in key-sorted order. It uses *binary search* for `get`. To maintain the keys in sorted order, it moves all of the larger elements over for `put`. The `get` operation is logarithmic, but the `put` operation takes linear time per operation.
- *Linked list implementation.* Program [LinkedListST.java](#) implements a symbol table with an (unordered) linked list. Both `put` and `get` take linear time per operation: to search for a key, we need to traverse its links; to put a key-value pair, we need to search for the given key.

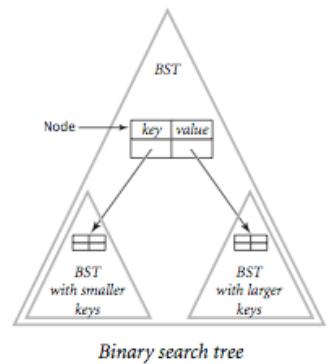


To develop a symbol-table implementation that is feasible for use with clients like `Lookup` and `Index`, we need the flexibility of linked lists and the efficiency of binary search. Binary search trees, which we consider next, provide just this combination.

**Binary search trees.** The *binary tree* is a mathematical abstraction that plays a central role in the efficient organization of information. Like arrays and linked lists, a binary tree is a data type that stores a collection of data. Binary trees play an important role in computer programming because they strike an efficient balance between flexibility and ease of implementation.

For symbol-table implementations, we use special type of binary tree to organize the data and to provide a basis for efficient implementations of the symbol-table `put` operations and `get` requests. A *binary search tree (BST)* associates Comparable keys with values, in a structure defined recursively as follows: A BST is either

- empty (`null`) or



- a node having a key-value pair and two references to BSTs, a left BST with smaller keys and a right BST with larger keys.

The key type must be `Comparable`, but the type of the value is not specified, so a BST node can hold any kind of data in addition to the (characteristic) references to BSTs. To implement BSTs, we start with a class for the node abstraction, which has references to a key, a value, and left and right BSTs:

```
private class Node {
    Key key;
    Value val;
    Node left, right;

    Node(Key key, Value val) {
        this.key = key;
        this.val = val;
    }
}
```

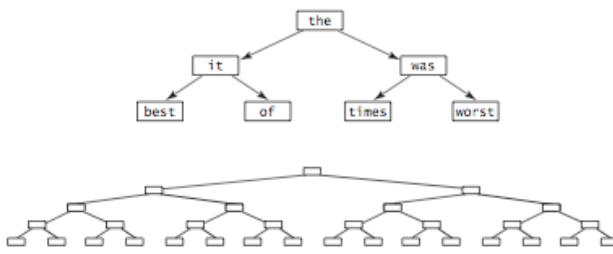
This definition is like our definition of nodes for linked lists, except that it has *two* links, not just one.

- *Search in a BST.* Suppose that you want to *search* for a node with a given key in a BST (or to *get* a value with a given key in a symbol table). There are two possible outcomes: the search might be successful (we find the key in the BST; in a symbol-table implementation we return the associated value) or it might be unsuccessful (there is no key in the BST with the given key; in a symbol-table implementation, we return `null`). A recursive algorithm is immediate: Given a BST (a reference to a `Node`), first check whether the tree is empty (the reference is `null`). If so, then terminate the search as unsuccessful (in a symbol-table implementation, return `null`). If the tree is non-empty, check whether the key in the node is equal to the search key. If so, then terminate the search as successful (in a symbol-table implementation, return the value associated with the key). If not, compare the search key with the key in the node. If it is smaller, search (recursively) in the left subtree; if it is greater, search (recursively) in the right subtree.
- *Inserting into a BST.* Suppose that you want to insert a new node into a BST (in a symbol-table implementation, *put* a new key-value pair into the data structure). The logic is similar to searching for a key, but the code is trickier: If the BST is empty, we create and return a new `Node` containing the key-value pair; if the search key is less than the key at the root, we set the left link to the result of inserting the key-value pair into the left subtree; if the search key is less, we set the right link to the result of inserting the key-value pair into the right subtree; otherwise if the search key is equal, we overwrite the existing value with the new value. Resetting the link after the recursive call in this way is usually unnecessary, because the link changes only if the subtree is empty, but it is as easy to set the link as to test to avoid setting it.

Program [BST.java](#) is a symbol-table implementation based on these two recursive algorithms. Here is a useful [binary search tree application](#).

**Performance characteristics of BST.** The running times of algorithms on BSTs are dependent on the shape of the trees, and the shape of the trees is dependent on the order in which the keys are inserted.

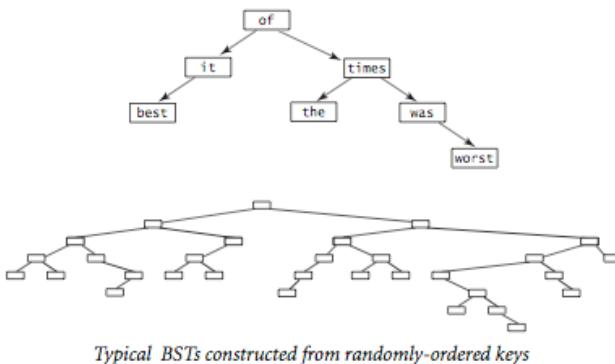
- *Best case.* In the best case, the tree is perfectly balanced (each `Node` has exactly two non-null children), with  $\lg N$  nodes between the root and each leaf node. In such a tree, it is easy to see that the cost of an unsuccessful search is logarithmic, because that cost satisfies the same recurrence relation as the cost of binary search (see Section 4.2) so that the cost of every *put* operation and *get* request is proportional to  $\lg N$  or less.



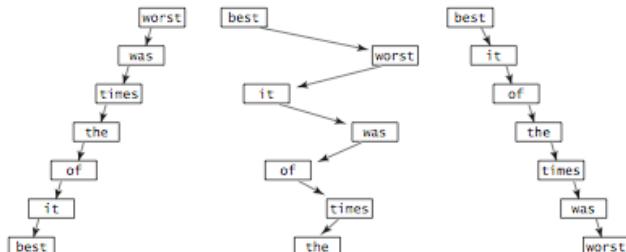
*Best case (perfectly balanced) BSTs*

- *Average case.* If we insert random keys, we might expect the search times to be logarithmic as well, because the first element becomes the root of the tree and should divide the keys roughly in half. Applying the same argument to the subtrees, we expect to get about the same

result as for the best case. This intuition is indeed validated by careful analysis: a classic mathematical derivation shows that the time required for put and get in a tree constructed from randomly ordered keys is logarithmic. More precisely, the expected number of key comparisons is  $\sim 2 \ln N$  for a random *put* or *get* in a key built from  $N$  randomly ordered keys.



- *Worst case.* In the worst case, each node has exactly one null link, so the BST is like a linked list, where *put* operations and *get* requests take linear time. Unfortunately this worst case is not rare in practice - it arises, for example, when we insert the keys in order. Thus, good performance of the basic BST implementation is dependent on the keys being sufficiently similar to random keys that the tree is not likely to contain many long paths.



*Worst case (totally unbalanced) BSTs*

Remarkably, there are BST variants that eliminate this worst case and guarantee logarithmic performance per operation, by making all trees nearly perfectly balanced. One popular variant is known as a *red-black tree*. The Java library `java.util.TreeMap` implements a symbol table using this approach. Our symbol-table implementation `ST.java` uses this data structure to implement our symbol-table API. It is remarkably efficient: typically, it only accesses a small number of the nodes in the BST (those on the path from the root to the node sought or to the leaf to which the new node is attached) and it only creates one new `Node` and adds one new link for the *put* operation. Next, we show that *put* operations and *get* requests take logarithmic time (under certain assumptions).

**Traversing a BST.** Perhaps the most basic tree-processing function is known as *tree traversal*: Given a (reference to) a tree, we want to systematically process every key-value pair in the tree. For linked lists, we accomplish this task by following the single link to move from one node to the next. For trees, however, we have decisions to make, because there are generally two links to follow. But recursion comes immediately to the rescue. To process every key in a BST:

- process every key-value pair in the left subtree
- process the key-value pair at the root
- process every key-value pair in the right subtree

This approach not only processes every key pair in the BST, but it does so in key-sorted order. For example, the following method prints the key-value pairs in the tree rooted at its argument in key-sorted order.

```
private static void traverse(Node x) {
    if (x == null) return;
    traverse(x.left);
    StdOut.println(x.key + " " + x.val);
    traverse(x.right);
}
```

This remarkably simple method is worthy of careful study. It can be used as a basis for a `toString()` implementation for BSTs and also a starting point for developing an iterator.

**Extended symbol table operations.** The flexibility of BSTs enable the implementation of many useful additional operations beyond those dictated by the symbol table API.

- *Minimum and maximum.* To find the smallest key in a BST, follow left links from the root until reaching null. The last key encountered is the smallest in the BST. The same procedure following right links lead to the largest key.
- *Size.* To keep track of the number of nodes in a BST, keep an extra instance variable `N` in BST that counts the number of nodes in the tree. Initialize it to 0 and increment it whenever creating a new `Node`.
- *Remove.* Many applications demand the ability to remove a key-value pair with a given key. You can find explicit code for removing a node from a BST in a book on algorithms and data structures. An easy lazy way to implement `remove()` relies on the fact that values cannot be null:

```
public void remove(Key key) {  
    if (contains(key))  
        put(key, null);  
}
```

This approach necessitates periodically cleaning out nodes in the BST with `null` values, because performance will degrade unless the size of the data structure stays proportional to the number of key-value pairs in the table.

- *Range search.* With a recursive method like `traverse()`, we can count the number of keys that fall within a given range or return all the keys falling into a given range.
- *Order statistics.* If we maintain an instance variable in each node having the size of the subtree rooted at each node, we can implement a recursive method that returns the  $k$ th largest key in the BST.

**Set data type.** As a final example, we consider a data type that is simpler than a symbol table, still broadly useful, and easy to implement with BSTs. A *set* is an (unordered) collection of distinct comparable keys, defined by the following API:

<code>public class SET&lt;Key extends Comparable&gt;</code>	<hr/>
<code>    SET()</code>	<i>create a set</i>
<code>    boolean isEmpty()</code>	<i>is the set empty?</i>
<code>    void add(Key key)</code>	<i>add key to the set</i>
<code>    boolean contains(Key key)</code>	<i>is key in the set?</i>

A set is a symbol table with no values. We could use BST to implement SET, but a direct implementation is simpler, and client code that uses SET is simpler and clearer than it would be to use placeholder values and ignore them. Program [SET.java](#) implements the set API. Program [DeDup.java](#) is a SET client that reads in a sequence of strings from standard input and prints out the first occurrence of each string (thereby removing duplicates).

**Perspective.** The use of binary search trees to implement symbol tables is a sterling example of exploiting the tree abstraction, which is ubiquitous and familiar. Trees lie at the basis of many scientific topics, and are widely used in computer science. We are accustomed to many tree structures in everyday life including family trees, sports tournaments, the organization chart of a company, the [tree of life](#), and parse trees in grammar. Trees also arise in numerous computational applications including function call trees, parse trees, and file systems. Many important applications are rooted in science and engineering, including phylogenetic trees in computational biology, multidimensional trees in computer graphics, minimax game trees in economics, and quad trees in molecular dynamics simulations.

## Q + A.

- Q. Why use immutable symbol table keys?
- A. If we changed a key while it was in the BST, it would invalidate the ordering restriction.
- Q. Why not use the Java library methods for symbol tables?
- A. Now that you understand how a symbol table works, you are certainly welcome to use the industrial strength versions `java.util.TreeMap` and `java.util.HashMap`. They follow the same basic API as BST, but they allow null keys and use the names `containsKey()` and `keySet()` instead of `contains()` and `iterator()`, respectively. They also contain additional methods such as `remove()`, but they do not provide any efficient way to add some of the additional methods that we mentioned, such as order statistics. You can also use `java.util.TreeSet` and `java.util.HashSet` which implement an API like our SET.

## Exercises

1. Modify `Lookup` to make a program `LookupAndPut` that allows `put` operations to be specified on standard input. Use the convention that a plus sign indicates that the next two strings typed are the key-value pair to be inserted.
2. Modify `Lookup` to make a program `LookupMultiple` that handles multiple values having the same key by putting the values on a queue, as in `Index`, and then printing them all out on a `get` request, as follows:

```
% java LookupMultiple amino.csv 3 0
Leucine
TTA TTG CTT CTC CTA CTG
```

3. Modify `Index` to make a program `IndexByKeyword` that takes a file name from the command line and makes an index from standard input using only the keywords in that file. Note : using the same file for indexing and keywords should give the same result as `Index`.
4. Modify `Index` to make a program `IndexLines` that considers only consecutive sequences of letters as keys and uses line numbers instead of word position as the value and to . This functionality is useful for programs, as follows:

```
% java IndexLines 6 0 < Index.java
continue 12
enqueue 15
Integer 4 5 7 8 14
parseInt 4 5
println 22
```

5. Develop an implementation `BinarySearchST.java` of the symbol-table API that maintains parallel arrays of keys and values, keeping them in key-sorted order. Use binary search for get and move larger elements to the right one position for put (use array doubling to keep the array size proportional to the number of key-value pairs in the table). Test your implementation with `Index`, and validate the hypothesis that using such an implementation for `Index` takes time proportional to the product of the number of strings and the number of distinct strings in the input.
6. Develop an implementation `LinkedListST.java` of the symbol-table API that maintains a linked list of nodes containing keys and values, keeping them in key-sorted order. Test your implementation with `Index`, and validate the hypothesis that using such an implementation for `Index` takes time proportional to the product of the number of strings and the number of distinct strings in the input.
7. Draw all the different BSTs that can represent the key sequence

best of it the time was.

8. Draw the BST that results when you insert items with keys

E A S Y Q U E S T I O N

- in that order into an initially empty tree.
9. Suppose we have int values between 1 and 1000 in a BST and search for 363. Which of the following cannot be the sequence of keys examined.
    - a. 2 252 401 398 330 363
    - b. 399 387 219 266 382 381 278 363
    - c. 3 923 220 911 244 898 258 362 363
    - d. 4 924 278 347 621 299 392 358 363
    - e. 5 925 202 910 245 363
  10. Suppose that the following 31 keys appear (in some order) in a BST of height 5:

```
10 15 18 21 23 24 30 30 38 41
42 45 50 55 59 60 61 63 71 77
78 83 84 85 86 88 91 92 93 94
98
```

Draw the top 3 nodes of the tree (the root and its two children).

11. Implement `toString()` for BST, using a recursive helper method like `traverse()`. As usual, you can accept quadratic performance because of the cost of string concatenation. Extra credit : Write a linear-time `toString()` method for BST.
12. True or false. Given a BST, let  $x$  be a leaf node, and let  $p$  be its parent. Then either (i) the key of  $p$  is the smallest key in the BST larger than the key of  $x$  or (ii) the key of  $p$  is the largest key in the BST smaller than the key of  $x$ . Answer: true.
13. Modify the symbol-table API to use a wrapper type `STentry` that holds the keys and values (with accessor methods `key()` and `value()` to access them). Your iterator should return `STentry` objects in key-sorted order. Implement BST and Index as dictated by this API. Discuss the pros and cons of this approach versus the one given in the text.
14. Modify the symbol-table API to handle values with duplicate keys by having `get()` return a iterator for the values having a given key. Implement BST and Index as dictated by this API. Discuss the pros and cons of this approach versus the one given in the text.
15. Prove that the expected number of key comparisons for a random `put` or `get` in a BST build from randomly ordered keys is  $\sim 2 \ln N$ .
16. Prove that the number of different BSTs that can be built from  $N$  keys is  $\sim$
17. Run experiments to validate the claims in the text that the `put` operations and `get` requests for Lookup and Index are logarithmic in the size of the table when using BST.
18. Modify BST to implement a symbol-table API where keys are arbitrary objects. Use `hashCode()` to convert keys to integers and use integer keys in the BST.
19. Modify BST to add methods `min()` and `max()` that return the smallest (largest) key in the table (or `null` if no such key exists).
20. Modify BST to add a method `size()` that returns the number of elements in the table. Use the approach of storing within each `Node` the number of nodes in the subtree rooted there.
21. Modify BST to add a method `rangeCount()` that takes two `Key` arguments and returns the number of keys in a BST between the two given keys. Your method should take time proportional to the height of the tree. Hint : First work the previous exercise.
22. Modify SET to add methods `floor()`, `ceil()`, and `nearest()` that take as argument a `Key` and return the largest (smallest, nearest) element in the set that is no larger than (no smaller than, closest to) the given `Key`.
23. Write a ST client [GPA.java](#) that creates a symbol table mapping letter grades to numerical scores, as in the table below, then reads from standard input a list of letter grades and computes their average (GPA).

A+	A	A-	B+	B	B-	C+	C	C-	D	F
4.33	4.00	3.67	3.33	3.00	2.67	2.33	2.00	1.67	1.00	0.00

## Binary Tree Exercises

This list of exercises is intended to give you experience in working with binary trees that are not necessarily BSTs. They all assume a `Node` class with three instance variables: a integer value and two `Node` references.

1. Implement the following methods for a binary tree that each take as argument a `Node` that is the root of a binary tree.
    - `size()`: number of nodes in the tree.
    - `leaves()`: number of nodes whose links are both `null`
    - `total()`: sum of the key values in all nodes
- Your methods should all run in linear time.
2. Implement a linear-time method `height()` that returns the maximum number of nodes on any path from the root to a leaf node (the height of the `null` tree is 0; the height of a 1-node tree is 1).
  3. A binary tree is *heap-ordered* if the key at the root is larger than the keys in all of its descendants. Implement a linear-time method `heapOrdered()` that returns `true` if the tree is heap-ordered, `false` otherwise.
  4. A binary tree is *balanced* if both its subtrees are balanced and the height of its two subtrees differ by at most 1. Implement a linear-time method `balanced()` that returns `true` if the tree is balanced, `false` otherwise.
  5. Two binary trees are *isomorphic* if only their key values differ (they have the same shape). Implement a linear-time static method `isomorphic()` that takes two tree references as parameters and returns `true` if they refer to isomorphic trees, `false` otherwise. Then implement a linear-time static method `eq()` that takes two tree references as parameters and returns `true` if they refer to identical trees (isomorphic with same key values), `false` otherwise.

```

public static boolean isomorphic(Node x, Node y) {
    if (x == null && y == null) return true; // both null
    if (x == null || y == null) return false; // exactly one null
    return isomorphic(x.left, y.left) && isomorphic(x.right, y.right);
}

```

6. Implement a linear-time method `isBST()` that returns `true` if the tree is a BST, `false` otherwise.

Solution : This task is a bit more difficult than it might seem. We use an overloaded recursive method `isBST()` that takes two additional arguments `min` and `max` and returns `true` if the tree is a BST *and* all its values are between `min` and `max`.

```

public static boolean isBST() {
    return isBST(root, Integer.MIN_VALUE, Integer.MAX_VALUE);
}

private boolean isBST(Node x, int min, int max) {
    if (x == null) return true;
    if (x.val < min || x.val > max) return false;
    return isBST(x.left, min, x.val) && isBST(x.right, x.val, max);
}

```

7. Write a method `levelOrder()` that prints BST keys in *level order*: First print the root, then the nodes one level below the root, left to right, then the nodes two levels below the root (left to right), and so forth. *Hint* : Use a `Queue<Node>`.
8. Compute the value returned by `mystery()` on some sample binary trees, then formulate an hypothesis about its behavior and prove it.

```

public int mystery(Node x) {
    if (x == null) return 0;
    else return mystery(x.left) + mystery(x.right);
}

```

Answer: Returns 0 for any binary tree.

## Creative Exercises

1. **Spell checking.** Write a `SET` client [SpellChecker.java](#) that takes as command-line argument the name of a file containing a dictionary of words, and then reads strings from standard input and prints out any string that is not in the dictionary. Use the 600,000+ word dictionary [words.utf-8.txt](#).
2. **Spell correction.** Write an `ST` client [SpellCorrector.java](#) that serves as a filter that replaces commonly misspelled words on standard input with a suggested replacement, printing the result to standard output. Take as command-line argument a file that contains common misspellings and corrections. Use the file [misspellings.txt](#), which contains many common misspellings.
3. **Web filter.** Write a `SET` client [WebBlocker](#) that takes as command-line argument the name of a file containing a list of [objectionable websites](#) and then reads strings from standard input and prints out only those websites that should not be filtered.
4. **Set operations.** Add methods `union()` and `intersection()` to the set ADT. Do not worry about efficient BST implementations.
5. **Frequency symbol table.** Develop a data type [FrequencyTable.java](#) that supports the following operations: `click()` and `count()`, both of which take `String` arguments. The data-type value is an integer that keeps track of the number of times the `click()` operation is been called with the given `String` as argument. The `click()` operation increments the count by one, and the `count()` operation returns the value, possibly 0. Clients of this data type might include a web traffic analyzer, a music player that counts the number of times each song has been played, phone software for counting calls, and so forth.
6. **1D range searching.** Develop a data type that supports the following operations: insert a date, search for a date, and count the number of dates in the data structure that lie in a particular interval. Use either Java's `java.util.Date` data type or `java.util.GregorianCalendar` data type to represent the date.
7. **Non-overlapping interval search.** Given a list of non-overlapping intervals of integers, write a function that takes an integer argument and determines in which if any interval that values lies, e.g., if the intervals are 1643-2033, 5532-7643, 8999-10332, 5666653-5669321, then the query point 9122 lies in the third interval and 8122 lies in no interval.

8. **IP lookup by country.** Write a BST client that uses the data file the data file [ip-to-country.csv](#) to determine what country a given IP address is coming from. The data file has five fields (beginning of IP address range, ending of IP address range, two character country code, three character country code, and country name. The IP addresses are non-overlapping. Such a database tool can be used for: credit card fraud detection, spam filtering, auto-selection of language on a web site, and web server log analysis.

See also [GeoIPCountryWhois.csv](#), the [IP-to-country website](#) or [MaxMind GeoIP](#).

9. **Inverted index of web.** Given a list of web pages, create a symbol table of words contained in those web pages. Associate with each word a set of web pages in which that word appears. Program [InvertedIndex.java](#) takes a list of filenames as command-line inputs, reads in the text files, creates a symbol table of sets, and supports single-word queries by returning the set of web pages in which that query word appears.
10. **Inverted index of web.** Extend the previous exercise so that it supports multi-word queries. In this case, output the list of web pages that contain at least one occurrence of each of the query words.
11. **Multiple word search.** Write a program that takes  $k$  words from the command-line, reads in sequence of words from standard input, and identifies the smallest subsequence of text that contains all of the  $k$  words (not necessarily in the same order). Don???t consider partial words.

*Hint:* for each index  $i$ , find the smallest subsequence  $[i, j]$  that contains the  $k$  query words. Keep a count of the number of times each of the  $k$  query words appear. Given  $[i, j]$ , compute  $[i+1, j]$  by decrementing the counter for word  $i$ . Then gradually increase  $j$  until the subsequence contains at least one copy of each of the  $k$  words.

12. **Repetition draw in chess.** In the game of chess, if a board position is repeated three times with the same side to move, the side to move can declare a draw. Describe how you could test this condition using a computer program.
13. **Registrar scheduling.** The Registrar at a prominent northeastern University recently scheduled an instructor to teach two different classes at the same exact time. Help the Registrar prevent future mistakes by describing a method to check for such conflicts. For simplicity, assume all classes run for 50 minutes starting at 9, 10, 11, 1, 2, or 3.
14. **Entropy.** We define the *relative entropy* of a text corpus with  $N$  words,  $k$  of which are distinct as

$$E = 1 / (N \lg N) (p_1 \lg(k / p_1) + \dots + p_k \lg(k / p_k))$$

where  $p_i$  is the fraction of times that word  $i$  appears. Write a program that reads in a text corpus and prints out the relative entropy. Convert all letters to lowercase and treat punctuation marks as whitespace.

15. **Order statistics.** Add to BST a method `select()` that takes an integer argument  $k$  and returns the  $k$ th largest key in the BST. Assume that the subtree sizes are maintained in each node. The running time should be proportional to the height of the tree.
16. **Delete ith element.** Create an ADT that supports the following operations: `isEmpty()`, `insert()`, and `delete(int i)`, where the deletion operation deletes and returns the  $i$ th least recently added object. Use a BST with a counter that records the total number of elements  $n$  inserted and give the  $n$ th element inserted the key  $n$ . Each BST node should also maintain the total number of nodes in the subtree rooted at that node. To find the  $i$ th least recently added item, search for the  $i$ th smallest element in the BST.
17. **Mutable string.** Create an ADT that supports the following operations on a string: `get(int i)`, `insert(int i, char c)`, and `delete(int i)`, where `get` returns the  $i$ th character of the string, `insert()` inserts the character  $c$  and makes it the  $i$ th character, and `delete()` deletes the  $i$ th character. Use a BST to implement operations in logarithmic time.
18. **Sparse vectors and matrices.** An  $N$ -by- $N$  matrix is sparse if its number of nonzeros is proportional to  $N$  (or less). Goal: represent sparse matrix with space proportional to  $N$  by only implicitly storing the zero entries. Add two sparse vectors/matrices in time proportional to the number of nonzeros. Design ADTs [SparseVector.java](#) and [SparseMatrix.java](#) that support the following APIs.

```

public class SparseVector
    public SparseVector(int N)                                // 0-vector of length N
    public void put(int i, double value)                      // a[i] = val
    public double get(int i)                                  // return a[i]
    public double dot(SparseVector b)                         // vector dot product
    public SparseVector plus(SparseVector b)                 // vector addition

public class SparseMatrix
    public SparseMatrix(int N)                                // 0-matrix of size N
    public void put(int i, int j, double val)                 // a[i][j] = val
    public double get(int i, int j)                           // return a[i][j]
    public SparseMatrix times(SparseMatrix b)                // matrix product
    public SparseMatrix plus(SparseMatrix b)                 // matrix addition
  
```

19. **Expression parser.** Write a program to parse and evaluate expressions of the following form. Store the variable names in a symbol table.

```

A = 5
B = 10
C = A + B
D = C * C
Result: A = 5, B = 10, C = 15, D = 225

```

Consider adding more complicated expressions, e.g.,  $E = 7 * (B + C * D)$ .

20. **Actor and actress aliases.** Given this 10MB file containing a [list of actors \(with canonical name\) and their aliases](#), write a program that reads in the name of an actor from standard input and prints out his canonical name.
21. **Database joins.** Given two tables, an [inner join](#) finds the "intersection" between the two tables.

Name	Dept ID	Dept	Dept ID
Smith	34	Sales	31
Jones	33	Engineering	33
Robinson	34	Clerical	34
Jasper	36	Marketing	35
Steinberg	33		
Rafferty	31		

The inner join on department ID is as follows.

Name	Dept ID	Dept
Smith	34	Clerical
Jones	33	Engineering
Robinson	34	Clerical
Steinberg	33	Engineering
Rafferty	31	Sales

22. **Molecular weight calculator.** Write a program `MolecularWeight.java` that reads in a list of [elements and their molecular weights](#), and then prompts the user for the molecular description of a chemical compound (e.g., CO2 or Na.Cl or N.H4.N.O3 = ammonium nitrate) and prints out its molecular weight.

## Web Exercises

1. **Codon usage table.** Describe how to hash codons (3 consecutive nucleotides) to an integer between 0 and 63. Print out summary statistics for each codon, e.g.,

UUU	13.2(	724)	UCU	19.6(	1079)	UAU	16.5(	909)	UGU	12.4(	680)
UUC	23.5(	1290)	UCC	10.6(	583)	UAC	14.7(	808)	UGC	8.0(	438)
UUA	5.8(	317)	UCA	16.1(	884)	UAA	0.7(	38)	UGA	0.3(	15)
UUG	17.6(	965)	UCG	11.8(	648)	UAG	0.2(	13)	UGG	9.5(	522)
CUU	21.2(	1166)	CCU	10.4(	571)	CAU	13.3(	733)	CGU	10.5(	578)
CUC	13.5(	739)	CCC	4.9(	267)	CAC	8.2(	448)	CGC	4.2(	233)
CUA	6.5(	357)	CCA	41.0(	2251)	CAA	24.9(	1370)	CGA	10.7(	588)
CUG	10.7(	590)	CCG	10.1(	553)	CAG	11.4(	625)	CGG	3.7(	201)
AUU	27.1(	1491)	ACU	25.6(	1405)	AAU	27.2(	1495)	AGU	11.9(	653)
AUC	23.3(	1279)	ACC	13.3(	728)	AAC	21.0(	1151)	AGC	6.8(	374)
AUA	5.9(	324)	ACA	17.1(	940)	AAA	32.7(	1794)	AGA	14.2(	782)
AUG	22.3(	1223)	ACG	9.2(	505)	AAG	23.9(	1311)	AGG	2.8(	156)
GUU	25.7(	1412)	GCU	24.2(	1328)	GAU	49.4(	2714)	GGU	11.8(	650)
GUC	15.3(	843)	GCC	12.6(	691)	GAC	22.1(	1212)	GGC	7.0(	384)
GUA	8.7(	478)	GCA	16.8(	922)	GAA	39.8(	2185)	GGA	47.2(	2592)

2. **Functions on trees.** Write a function `count()` that takes a `Node x` as an argument returns the number of nodes in the subtree rooted at `x` (including `x`). The number of elements in an empty binary tree is 0 (base case), and the number of elements in a non-empty binary tree is equal to one plus the number of elements in the left subtree plus the number of elements in the right subtree.

```
public static int count(TwoNode x) {
    if (x == null) return 0;
    return 1 + count(x.left) + count(x.right);
}
```

3. **Random element.** Add a symbol table function `random` to a BST that returns a random element. Assume that the nodes of your BST have integer size fields that contain the number of elements in the subtree rooted at that node. The running time should be proportional to the length of the path from the root to the node returned.
4. **Markov language model.** Create a data type that supports the following two operations: `add` and `random`. The `add` method should insert a new item into the data structure if it doesn't yet exists; if it already exists, it should increase its frequency count by one. The `random` method should return an element at random, where the probabilities are weighted by the frequency of each element.
5. **Queue with no duplicates.** Create a data type that is a queue, except that an element may only appear on the queue at most once at any given time. Ignore requests to insert an item if it is already on the queue.
6. **Bayesian spam filter.** Follow the ideas in [A Plan for Spam](#). Here is a place to get [test data](#).
7. **Symbol table with random access.** Create a data type that supports inserting a key-value pair, searching for a key and returning the associated value, and deleting and returning a random value. *Hint:* combine a symbol table and a randomized queue.
8. **Random phone numbers.** Write a program that takes a command line input `N` and prints `N` random phone numbers of the form `(xxx) xxx-xxxx`. Use a symbol table to avoid choosing the same number more than once. Use this [list of area codes](#) to avoid printing out bogus area codes.
9. **Unique substrings of length L.** Write a program that reads in text from standard input and calculate the number of unique substrings of length `L` that it contains. For example, if the input is `cgcgggcgcg` then there are 5 unique substrings of length 3: `cgc`, `cgg`, `gca`, `gcg`, and `ggg`. Applications to data compression. *Hint:* use the string method `substring(i, i + L)` to extract `i`th substring and insert into a symbol table. Test it out on the first [million digits of pi](#) or [10 million digits of pi](#).
10. **The great tree-list recursion problem.** A binary search tree and a circular doubly linked list are conceptually built from the same type of nodes - a data field and two references to other nodes. Given a binary search tree, rearrange the references so that it becomes a circular doubly-linked list (in sorted order). Nick Parlante describes this as [one of the neatest recursive pointer problems ever devised](#). *Hint:* create a circularly linked list `A` from the left subtree, a circularly linked list `B` from the right subtree, and make the root a one node circularly linked list. Then merge the three lists.
11. **Optimal BST.**
12. **Password checker.** Write a program that reads in a string from the command line and a dictionary of words from standard input, and checks whether it is a "good" password. Here, assume "good" means that it (i) is at least 8 characters long, (ii) is not a word in the dictionary, (iii) is not a word in the dictionary followed by a digit 0-9 (e.g., `hello5`), (iv) is not two words separated by a digit (e.g., `hello2world`)
13. **Reverse password checker.** Modify the previous problem so that (ii) - (v) are also satisfied for reverses of words in the dictionary (e.g., `olleh` and `olleh2world`). *Simple solution:* insert each word and its reverse into the symbol table.
14. **Cryptograms.** Write a program to read in a cryptogram and solve it. A cryptogram is ancient form of encryption known as a substitution cipher in which each letter in the original message is replaced by another letter. Assuming we use only lowercase letters, there are 26! possibilities, and your goal is to find one that results in a message where each word is a valid word in the dictionary. Use [Permutations.java](#) and backtracking.
15. **Frequency counter.** Write a program [FrequencyCounter.java](#) that reads in a sequence of strings from standard input and `counts` the number of times each string appears.
16. **Unordered array symbol table.** Write a program [SequentialSearchST.java](#) that implements a symbol-table using (unordered) parallel arrays.
17. **Exception filter.** The client program [ExceptionFilter.java](#) reads in a sequence of strings from a whitelist file specified as a command-line argument, then it prints out all words from standard input not in the whitelist.
18. Give the preorder traversal of some BST (not including null nodes). Ask the reader to reconstruct the tree.
19. Write an ADT [IterativeBST.java](#) that implements a symbol table using a BST, but uses iterative versions of `get()` and `put()`.
20. Largest absolute value.

21. **Tree reconstruction.** Given the following traversals of a binary tree (only the elements, not the null nodes), can you can you reconstruct the tree?

- a. Preorder and inorder.
- b. Postorder and inorder.
- c. Level order and inorder.
- d. Preorder and level order.
- e. Preorder and postorder.

Answers

- a. Yes. Scan the preorder from left to right, and use the inorder traversal to identify the left and right subtrees.
  - b. Yes. Scan the postorder from right to left.
  - c. Yes. Scan the level order from left to right.
  - d. No. Consider two trees with A as the root and B as either the left or right child. The preorder traversal of both is AB and the level order traversal of both is AB.
  - e. No. Same counterexample as above.
22. **Highlighting browser hyperlinks.** Browsers typically denote hyperlinks in blue, unless they've already been visited, in which case they're depicted in purple Write a program `HyperLinkColorer.java` that reads in a list of web addresses from standard input and output blue if it's the first time reading in that string, and purple otherwise.
23. **Spam blacklist.** Insert known spam email addresses into a `SET.java` ADT, and use to blacklist spam.
24. **Inverted index of a book.** Write a program that reads in a text file from standard input and compiles an alphabetical index of which words appear on which lines, as in the following input. Ignore case and punctuation.

```
It was the best of times,
it was the worst of times,
it was the age of wisdom,
it was the age of foolishness,

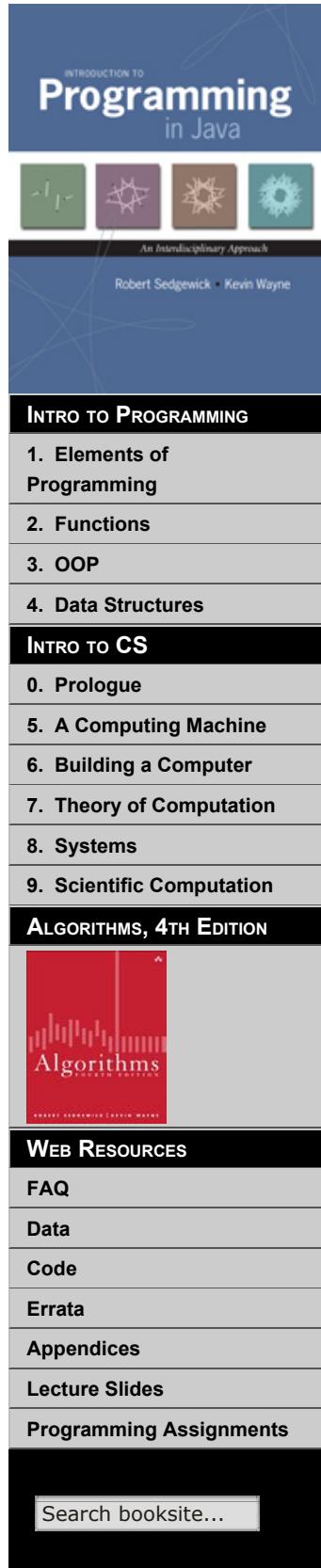
age 3-4
best 1
foolishness 4
it 1-4
of 1-4
the 1-4
times 1-2
was 1-4
wisdom 4
worst 2
```

*Hint:* create a symbol table whose key is a `String` that represents a word and whose value is a `Sequence<Integer>` that represents the list of pages on which the word appears.

25. **Randomly generated identities.** The files `names20k.csv` and The file `names20k-2.csv` each contain 20,000 randomly generated identities (number, gender, first name, middle initial, last name, street address, city, state, zip code, country, email address, telephone number, mother's maiden name, birthday, credit card type, credit card number, credit card expiration date, Social security number) from [fakenamegenerator.com](http://fakenamegenerator.com).
26. **Distance between zip codes.** Write a data type `Location.java` that represents a named location on the surface of the earth (a name, latitude, and longitude). Then, write a client program that takes the name of a ZIP code file (such as `zips.txt`) as a command-line argument, reads the data from the file, and stores it in a symbol table. Then, repeatedly read in pairs of ZIP codes from standard input and output the great-circle distance between them (in statute miles). This distance is used by the post office to calculate shipping rates.

*Last modified on August 05, 2011.*

Copyright © 2002–2012 [Robert Sedgewick](#) and [Kevin Wayne](#). All rights reserved.



## 4.5 A CASE STUDY: SMALL WORLD PHENOMENON

This section under major construction.

**Small world phenomenon.** The mathematical model that we use for studying the nature of pairwise connections among entities is known as the *graph*. Some graphs exhibit a specific property known as the *small-world phenomenon*. You may be familiar with this property, which is sometimes known as *six degrees of separation*. It is the basic idea that, even though each of us has relatively few acquaintances, there is a relatively short chain of acquaintances (the six degrees of separation) separating us from one another.

**Graphs.** A *graph* is comprised of a set of *vertices* and a set of *edges*. Each edge represents a connection between two vertices. The list below suggests the diverse range of systems where graphs are appropriate starting points for understanding structure.

GRAPH	VERTICES	EDGES
Communication	telephones, computers	fiber optic cable
Circuits	gates, registers, processors	wires
Mechanical	joints	rods, beams, springs
Hydraulic	reservoirs, pumping stations	pipelines
Financial	stocks, currency	transactions
Transportation	street intersections, airports	highways, air routes
Scheduling	tasks	precedence constraints
Software systems	functions	function calls
Internet	web pages	hyperlinks
Games	board positions	legal moves
Social networks	people, actors, terrorists	friendships, movie casts, associations
Protein interaction networks	proteins	protein-protein interactions
Genetic regulatory networks	genes	regulatory interactions
Neural networks	neurons	synapses
Infectious disease	people	infections
Electrical power grid	transmission stations	cable
Chemical compounds	molecules	chemical bonds

**Graph data type.** The following API supports graph processing:

## public class Graph

```

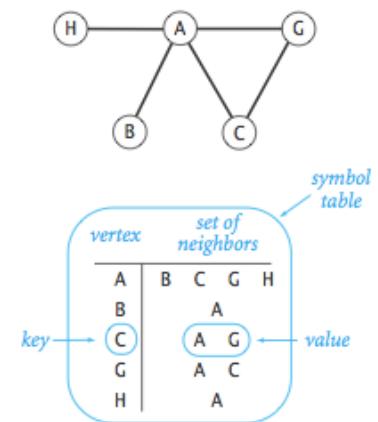
        Graph()
        Graph(In in, String delim)
        void addEdge(String v, String w)
            int V()
            int E()
        Iterable<String> vertices()
        Iterable<String> adjacentTo(String v)
            int degree(String v)
            boolean hasVertex(String v)
            boolean hasEdge(String v, String w)
    
```

*create an empty graph*  
*read graph from input stream*  
*add edge v-w*  
*number of vertices*  
*number of edges*  
*vertices in the graph*  
*neighbors of v*  
*number of neighbors of v*  
*is v a vertex in the graph?*  
*is v-w an edge in the graph?*

*API for a graph with String vertices*

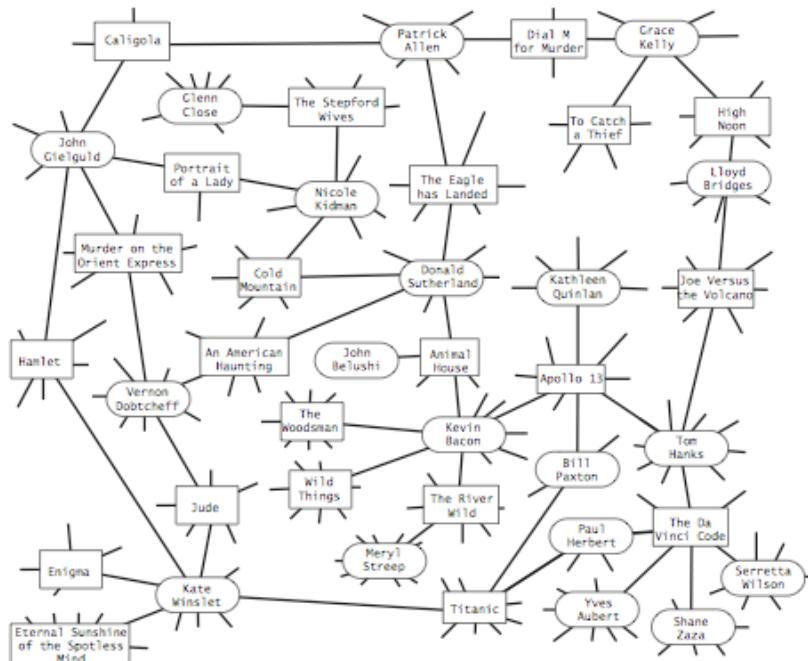
**Graph implementation.** Program [Graph.java](#) implements this API. Its internal representation is a symbol table of sets: the keys are vertices and the values are the sets of neighbors (the vertices adjacent to the key). This representation uses the two data types [ST.java](#) and [SET.java](#) that we introduced in [Section 4.4](#). Our implementation has three important properties:

- Clients can efficiently iterate through the graph vertices.
- Clients can efficiently iterate through a vertex's neighbors.
- Space usage is proportional to the number of edges.



*Symbol-table-of-sets graph representation*

**Performer-movie graphs.** The performer-movie graph includes a vertex for each performer and movie and an edge between a performer and a movie if the performer appeared in the movie.



*A tiny portion of the movie-performer relationship graph*

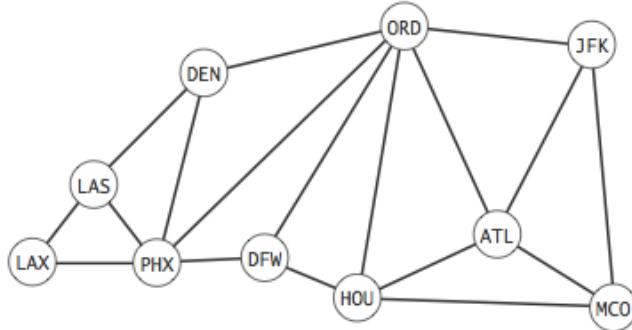
We provide a number of sample data files. Each line in a file consists of a movie, followed by a list of performers that appeared in the movie, separated by the / delimiter.

FILE	DESCRIPTION	MOVIES	ACTORS	EDGES
cast.06.txt	movies release in 2006	8780	84236	103815
cast.00-06.txt	movies release since 2000	52195	348497	606531
cast.G.txt	movies rated G by MPAA	1288	21177	28485
cast.PG.txt	movies rated PG by MPAA	3687	74724	116715
cast.PG13.txt	movies rated PG-13 by MPAA	2538	70325	103265
cast.mpaa.txt	movies rated by MPAA	21861	280624	627061
cast.action.txt	action movies	14938	139861	266485
cast.rated.txt	popular movies	4527	122406	217603
cast.all.txt	over 250,000 movies	285462	933864	3317266

These input files are as of November 2, 2006. The data is taken from the [Internet Movie Database](#).

Program [IndexGraph.java](#) takes a query which is the name of a movie (in which case it prints the list of performers that appear in that movie) or the name of a performer (in which case it prints the list of movies in which that performer has appeared).

**Shortest paths in graphs.** Given two vertices in a graph, a *path* is a sequence of edges connecting them. A *shortest path* is one with minimal length over all such paths (there typically are multiple shortest paths).



source	destination	distance	shortest paths
JFK	LAX	3	JFK-ORD-PHX-LAX
LAS	MCO	4	LAS-PHX-DFW-HOU-MCO and four others
HOU	JFK	2	HOU-ATL-JFK and two others

### Examples of shortest paths in a graph

We use the following API.

```
public class PathFinder
    PathFinder(Graph G, String s)           constructor
    int distanceTo(String v)                length of shortest path
                                            from s to v in G
    Iterable<String> pathTo(String v)        shortest path
                                            from s to v in G
    API for single-source shortest paths in a Graph
```

[PathFinder.java](#) implements the API using a classic algorithm known as *breadth-first search*.

**Degrees of separation.** One of the classic applications of shortest-paths algorithms is to find the *degrees of separation* of individuals in social networks. To fix ideas, we discuss this application in terms of a recently popularized pastime known as the *Kevin Bacon game*, which uses the movie-performer graph that we just considered. Kevin Bacon is a prolific actor who appeared in many movies. We assign every performer who has appeared in a movie a [Bacon number](#): Bacon himself is 0, any performer who has been in the same cast as Bacon has a Kevin Bacon number of 1, any other performer (except Bacon) who has been in the same cast as a performer whose number is 1 has a Kevin Bacon number of 2, and so forth. For example, Meryl Streep has a Kevin Bacon

number of 1 because she appeared in *The River Wild* with Kevin Bacon. Nicole Kidman's number is 2: although she did not appear in any movie with Kevin Bacon, she was in *Cold Mountain* with Donald Sutherland, and Sutherland appeared in *Animal House* with Kevin Bacon.

Given the name of a performer, the simplest version of the game is to find some alternating sequence of movies and performers that lead back to Kevin Bacon. Remarkably, [PathFinder.java](#) is precisely the program that you need to find a shortest path that establishes the Bacon number of any performer in `movies.txt`.

```
% java PathFinder movies.txt "/" "Bacon, Kevin"
Kidman, Nicole
Bacon, Kevin
Animal House (1978)
Sutherland, Donald (I)
Cold Mountain (2003)
Kidman, Nicole
Delpy, Julia
Bacon, Kevin
The Air I Breathe (2007)
Delpy, Julia
```

The six degrees of separation rule suggests that most actors will have a Kevin Bacon number of 6 or less. In fact, most have a Kevin Bacon number of 3 or less!

## Q + A

**Q.** What is formally meant by a *small world network*?

**A.** High local clustering and short global separation. *Local clustering* means that two vertices are more likely to be adjacent if they share a neighbor. *Short global separation* means that all (or most) pairs of vertices are connected by "short" paths. Here short means the length of chain scales logarithmic (or polylogarithmic) with the number of vertices. For example, empirically, any two documents on the Web are around 19 clicks away (directed) on average. Empirical estimate =  $.3 + 2.06 \log_{10} N$ , where  $N = 1$  billion. If web grows by factor of 1,000, only 21 clicks according to model.

**Q.** Other small world applications.

**A.** The small world phenomenon has numerous applications in the natural and social sciences. The domain of the Kevin Bacon is actor-movie relationships, but we can ask the same questions with mathematicians and papers ([Erdős number](#)), musicians and rock groups, baseball players and teams, or people and online social network groups. Sociology: looking for a job via connections, marketing products or ideas, formation and spread of fame and fads, train of thought followed in a conversation, and defining representative-ness of political bodies. Science: synchronization of neurons, analysis of World Wide Web, secure peer-to-peer networks, routing algorithms for ad hoc wireless networks, design of electrical power grids, processing of information in the human brain, phase transitions in coupled Kuramoto oscillators, spread of infectious diseases and computer viruses, and evolution of cooperation in multi-player iterated Prisoner's Dilemma.

In [The Small-World Phenomenon: An Algorithmic Perspective](#).

**Q.** Other applications and extensions.

**A.** Another important application of breadth first search is crawling the web. Or routing packets on the Internet. An important generalization is when each edge has an associated distance or time and the goal is to find a path that minimizes the sum of the edge lengths. More sophisticated algorithm known as *Dijkstra's algorithm* solves that problem in linearithmic time. Basis for map routing applications on the web and in GPS-equipped automobiles.

## Exercises

1. Wikipedia graph - a nice dataset.
2. Add methods `vertices()` and `edges()` to `Graph` that returns the number of

vertices and edges in the graph, respectively.

3. Add a boolean method `exists(String v)` that returns `true` if `v` is a vertex in the graph, and `false` otherwise.
4. Add a boolean method `exists(String v, String w)` that returns `true` if `v-w` is an edge in the graph, and `false` otherwise.
5. Suppose you use a stack instead of a queue when running breadth first in `PathFinder.java`. Does it still compute Kevin Bacon numbers correctly?
6. Suppose you use a queue instead of a stack when forming the shortest path in `path()`. What is the effect?
7. Add a method `isReachable(v)` to `PathFinder.java` that returns `true` if `v` can reach `s` via some path, and `false` otherwise.
8. The *degree* of a vertex is the number of incident edges. Add a method `degree()` to `Graph` that takes a `Vertex` as input and returns the degree of that vertex. Use this method to find the actor node in the Kevin Bacon graph with the highest degree, i.e., the actor that has appeared in the most movies.

```
public int degree(String v) {  
    if (st.contains(v)) return st.get(v).size();  
    else return 0;  
}
```

9. Write a program `MovieStats.java` that reads in a movie data set and prints out the number of distinct actors and movies.
10. Modify `Bacon.java` so that the user types two actors (one per line) and the program prints a shortest chain between the two actors.
11. Create a copy constructor for `Graph` that takes as input a graph `G`, and creates and initializes a new, independent, copy of the graph. Any changes you make to `G` should not affect the newly created graph.
12. Array of `PathFinder`'s to save away previous queries?
13. Modify `Graph.java` to include a method `iterator()` that returns an `Iterator` to the graph's vertices. Also, make `Graph` implement the `Iterable<String>` interface so that you can use it with `foreach` to iterate over the graph's vertices.

```
public Iterator<String> iterator() {  
    return st.iterator();  
}
```

14. True or false. At some point during breadth first search the queue can contain two vertices, one whose distance from the source is 7 and one whose distance is 9.  
*Answer:* False. The queue can contain vertices of at most two distinct distances `d` and `d+1`. Breadth first search examines the vertices in increasing order of distance from the source. When examining a vertex at distance `d`, only vertices of distance `d+1` can be enqueued.

## Creative Exercises

1. **Histogram.** Write a program `Histogram.java` to print a histogram of Kevin Bacon numbers, indicating how many actors have a Bacon number of 0, 1, 2, 3, etc.

#	Freq
<hr/>	
0	1
1	2083
2	187072
3	515582
4	113741
5	8269
6	772
7	93
8	7

2. **Large Bacon numbers.** Find the actors in `cast.txt` with the largest, but finite, Kevin Bacon number. *Answer:* here are a few actors whose number is 8: Ozra Esmaili, Gabriela Pereyra, Yamil Turk, Alejandro Zain.
3. **Actor graph.** An alternate way to compute Kevin Bacon numbers is to build a graph where there is a vertex for each actor (but not for each movie). Instead, two actors are connected by an edge if they appear in a movie together. Calculate Kevin Bacon numbers by running BFS on the actor graph. Compare the running time versus the algorithm described in the text. Explain why this approach is so much slower (and more difficult to reconstruct the movies along the path).
4. **Connected components.** Given an undirected graph, a *connected component* is a maximal set of vertices that are mutually reachable. Write a program `CCFinder.java` that computes the connected components of a graph. Include a constructor that takes a `Graph` as an argument and computes all of the connected components using breadth first search. Also, include a method `isReachable(v, w)` that returns `true` if `v` and `w` are in the same connected component, and `false` otherwise. Also include a method `components()` that returns the number of connected components. *Note:* need a way to iterate over the vertices in the symbol table. Then run BFS from each vertex that hasn't yet been visited.
5. **Flood fill / image processing.** Blob = collection of neighboring pixels of the same color. How many blobs in picture? Given N-by-N image of color pixels, identify all clusters of pixels of the same color that are connected. Find connected components.
6. **Word ladder.** Write a program `WordLadder.java` that takes two 5 letter strings as command-line arguments, and reads in a list of 5 letter words from standard input, and prints out a shortest (or `word ladder`) connecting the two strings (if it exists). This word game was invented by Lewis Carroll and was originally known as a doublet. Two words can be connected in a word ladder chain if they differ in exactly one letter. As an example, the following word ladder connects `green` and `brown`.

green greet great groat groan grown brown

You can also try out your program on this list of [6 letter words](#).

7. **All paths.** Given an undirected graph and two vertices `s` and `t`, write a program `AllPaths.java` that can be used to count or print out all *simple* paths from `s` to `t`. A simple path does not revisit any vertex more than once. In two-dimensional grids, such paths are referred to as self-avoiding walks. It is a fundamental problem in statistical physics and theoretical chemistry, e.g., to model the spatial arrangement of linear polymer molecules in a solution. Warning: there might be exponentially paths, so don't run this on large graphs. [Reference](#).
8. **Garbage collection.** Reference counting vs. mark-and-sweep. Automatic memory management in languages like Java is a challenging problem. Allocating memory is easy, but discovering when a program is finished with memory (and reclaiming it) is more difficult. Reference counting: doesn't work with circular linked structure. Mark-and-sweep algorithm. Root = local variables and static variables. Run DFS from roots, marking all variables references from roots, and so on. Then, make second pass: free all unmarked objects and unmark all marked objects. Or a copying GC would then move all of the marked objects to a single memory area. Uses one extra bit per object. JVM must pause while garbage collection occurs. Fragments memory.
9. **Directed graph.** Create a data type `Digraph.java` for directed graph. It is almost identical to our `Graph` ADT, except that when we insert an edge `v-w`, we only insert `w` on the adjacently list of `v`, but not `v` onto the adjacency list of `w`. Include a method `outdegree(String v)`.
10. **Shortest path in a digraph.** Write a directed version of `PathFinder` that finds the shortest path from `s` to every other vertex in a directed graph. *Hint:* breadth search first works in digraphs.
11. **Transitive closure.** Write an ADT `TransitiveClosure.java` whose constructor takes a `Digraph` as an argument, and whose method `isReachable(v, w)` true if `w` is reachable from `v` along a directed path in `G`, and `false` otherwise. *Note:* need a way to iterate over the vertices in the symbol table. Then run BFS from each vertex. Might need to store an `V`-by-`V` table or compute queries on the fly.

12. **Topological order and cycle detection.** Given a directed graph, determine if there is a directed cycle. If so, print one out. Otherwise print out a topological order. Given a list of pairs of variables of the form  $x < y$ , find an ordering that is consistent with the partial order. For example, if  $x < z$ ,  $y < z$ ,  $z < v$ , and  $v < w$ , then  $x < y < z < v < w$  is one possible solution. Print that the input is inconsistent if there is a cycle, e.g.,  $x < y$ ,  $y < z$ ,  $z < x$ .
13. **Cover time.** Given a connected, undirected graph  $G$ , a random walk moves from vertex  $v$  to  $w$  with probability equal to  $1 / \text{degree}(v)$  if  $(v, w)$  is an edge (and 0 otherwise). Write a program to simulate how many steps it takes to visit every vertex in the graph. Find an infinite sequence graph where the cover time grows proportional to  $V$ , proportional to  $V^2$ . Can you find one where it grows proportional to  $V^3$  or  $2^V$ ?
14. **Center of the Hollywood universe.** We can measure how good of a center that Kevin Bacon is by computing their [Hollywood number](#) or average path length. The Hollywood number of Kevin Bacon is the average Bacon number of all the actors (in its connected component). The Hollywood number of another actor is computed the same way, but we make them be the source instead of Kevin Bacon. Compute Kevin Bacon's Hollywood number and find an actor and actress with better Hollywood numbers than Kevin Bacon. Find the actor or actress (in the same connected component as Kevin Bacon) with the worst Hollywood number.
15. **Diameter.** The distance between two vertices is the length of the shortest path between them. The *eccentricity* of a vertex is the greatest distance between it and any other vertex. The *diameter* of a graph is the greatest distance between any two vertices (the maximum eccentricity of any vertex). Write a program `Diameter.java` to compute the eccentricity of a vertex and the diameter of a graph. Use it to find the diameter of the Kevin Bacon graph.
16. **Parallel edges.** Modify `Graph` to allow *parallel edges*, i.e., two or more edges connecting the same pair of vertices. *Hint:* use a `Queue` for the adjacency list instead of a `SET`.
17. **Erdos-Renyi graph model.** Graph on  $V$  vertices. Edge possible edge is included independently with probability  $p$ . Verify the following properties.
  - *Connectivity thresholds.* If  $p < 1/n$  and  $n$  is large, then most of connected components are small, with largest having  $O(\log n)$  vertices. If  $p > 1/n$  then there is almost surely a giant component containing  $\Theta(n)$  vertices. If  $p < \ln n / n$ , the graph is disconnected with high probability; if  $p > \ln n / n$  the graph is connected with high probability.
  - *Distribution of degrees.* Distribution of degrees follows a binomial distribution. Most vertices have similar degrees. Called *exponential* since the probability that a vertex is connected to  $k$  other vertices decreases exponentially in  $k$ .
  - *No hubs.* Maximum degree if  $p$  is a constant is  $O(\log n)$ .
  - *Not many triangles.* Random graph model doesn't capture social networks. Typically the neighbors of a vertex are also neighbors of each other.
  - *Small diameter.* If  $p > \log n / n$ , then diameter is  $O(\log n)$ .
18. **Power law of web links.** (Micahel Mitzenmacher) The indegrees and outdegrees of World Wide Web obey a power law. Can be modeled by [preferred attachment](#) process. Suppose that each web page has exactly one outgoing link. Each page is created one at a time, starting with a single page that points to itself. With probability  $p < 1$ , it links to one of the existing pages, chosen uniformly at random. With probability  $1-p$  it links to an existing page with probability proportional to the number of incoming links of that page. This rule reflects the common tendency for new web pages to point to popular pages. Write a program to simulate this process and plot a histogram of the number of incoming links. *Answer:* you should observe that the fraction of pages with indegree  $k$  is proportional to  $k^{(-1 / (1 - p))}$ .

## Web Exercises

1. **All shortest pairs.** Write a program [AllShortestPaths.java](#) that builds a graph from a file, and reads source-destination requests from standard input and prints a shortest path in the graph from the source to the destination.
2. **Faster word ladders.** To speed things up (if the word list is very large), don't write a nested loop to try all pairs of words to see if they are adjacent. To handle 5 letter words, first sort the word list. Words that only differ in their last letter will appear consecutively in the sorted list. Sort 4 more times, but cyclically shift the letters one position to the right so that words that differ in the  $i$ th letter will appear consecutively in one of the sorted lists.

Try out this approach using a [larger word list](#) with words of different sizes. Two words of different lengths are neighbors if the smaller word is the same as the bigger word, minus the last letter, e.g., brow and brown.

3. **Checkers.** Extend the rules of checkers to an N-by-N checkerboard. Show how to determine whether a checker can become in king in the current move. (Use BFS or DFS.) Show how to determine whether black has a winning move. (Find an Euler path.)
4. **Combinational circuits.** Determining the truth value of a combinational circuit given its inputs is a graph reachability problem (on a directed acyclic graph).
5. **Hex.** The game of Hex is played on a trapezoidal grid of hexagons....
  - Describe how to detect whether white or black has won the game using breadth first search.
  - Prove that the game cannot end in a tie. *Hint:* consider the set of cells reachable from the left side of the board.
  - Prove that the first player in Hex can always win if they play optimally. *Hint:* if the second player had a winning strategy, you could choose a random cell initially, and then just copy the second players winning strategy. This is called *strategy stealing*.
6. Explain the design flaw in the following implementation of a ring graph.

```
public class RingGraph extends Graph {
    public RingGraph(int V) {
        for (int i = 0; i < V; i++) {
            String v = "" + i;
            String w = "" + (i + 1) % V;
            addEdge(v, w);
        }
    }
}
```

The client can create a ring graph on V vertices, then invoke the `addEdge()` method, which would result in something other than a ring graph.

**Topological robustness.** "Scale-free networks are robust to random damage, but vulnerable to malicious attack." In a power law distributed small world network, deletion of random node rarely has much effect on length of shortest paths (since most shortest paths are routed through hubs). This is in contrast to random network. However, small world networks can catastrophically fail if the hub is deleted or attacked (whereas random networks have no central point of failure). Random 95% of Internet links (6209 vertices and 24401 links) can be removed and graph stays connected, but destroying 2.3% of hubs would disconnect it. Scale-free networks are also vulnerable to spreading viruses. We should immunize the hubs. Barabasi hypothesized that biological systems may be evolutionarily predisposed to prefer small world networks since they are more robust to perturbations (mutation, viral infection). Viruses do adapt to attack the hubs. Internet will still function if we randomly delete 80% of vertices. Scale-free networks are topologically robust if  $\gamma \leq 3$ . "promiscuous individuals are the vulnerable nodes to target in safe-sex campaigns." Liljeros.

**Power laws and scale-free networks.** Create a histogram of the the degrees of vertices in a graph and look at distribution. Power laws model scale invariance observed in many natural phenomena, including physics, sociology, geography, and economics. Direct result of self-organization. US Road network is approximately random network with bell-shaped connectivity distribution; US airport network is scale-free with many hubs. Think hub = airline hub.

WWW, Internet, food webs, social webs, pairwise interactions between proteins in human body. Probability than an actor has  $k$  neighbors follows a power law-tail for large  $k$ :  $p(k) \sim k^{-2.3}$ . Web graph (in-degrees and out-degrees) obey a power law. Probability that  $k$  documents point to a given web page  $\sim k^{-2.1}$ .

**Generative models.** Reference: [powerlaws.media.mit.edu](http://powerlaws.media.mit.edu). Traditionally, scientists have modeled graphs using completed ordered graphs (a grid or crystal lattice) or completely random graphs. Most real world networks seem fall in between the two extremes. People meet friends via existing friends, so the networks have many local connections and highly clustered. If person moves to a different city or joins a group, they form long-range connections.

Such networks grow over time by the gradual addition of vertices and edges with preferential connectivity. New web page likely to connect to well-established web pages.

**Barabasi model:** rich get richer. Start at time  $t = 1$  with two vertices linked by  $d$  parallel edges. At every time  $t \geq 2$ , add a new vertex with  $d$  edges using preferential attachment. Probability that new vertex will be connected to vertex  $i$  is  $k_i / \text{sum } (k_j)$ , where  $k_i$  is the connectivity of vertex  $i$  (indegree for directed graphs). Add edges one at a time, with subsequent edges doing preferential attachment using updated degrees. (Has property that for general average outdegree  $d$ , can run  $d = 1$  model for  $dn$  steps and collapse vertices  $k_d, k_d - 1, \dots, (k-1)d + 1$  to make vertex  $k$ .) Fraction of vertices with degree  $k$  converges to  $2d(d+1)/(k(k+1)(k+2))$  as  $n$  gets large. In other words, a power law proportional with exponent 3, where  $d$  only influences constant. [reference](#). Variants lead to exponents between 2 and infinity. (Exercise: simulate and compute an estimate.) As a result, hubs are formed.

Lovasz. Fixed preferential attachment graph. Fix  $n$  vertices with self-loops and  $m$  steps. In step  $i$ , choose two random vertices independently with probability proportional to degree and connect them. Random graph model of Erdos and Renyi not model many large networks that arise in practice.

- **Watts-Strogatz model.** Watts-Strogatz model captures structural property. Watts and Strogatz proposed a hybrid model that contained typical links of vertices near each other (people know their geographic neighbors), plus some random long-range connection links.  $n$ -by- $n$  grid digraph (local contacts) with many random shortcuts (long-range contacts). Leads to abundance of short paths.
- **Kleinberg model.** No way for participants in Watts-Strogatz model to find short paths in a decentralized But Milgram experiment also had striking algorithmic component - individuals can find short paths! Kleinberg proposed making distribution of shortcuts obey a power law, with probability proportional to the  $d$ th power of the distance (in  $d$  dimensions). Each vertex has one long-range neighbor. Uniform over all distance scales (same number of links at distances 1-10 as 10-100 or 100-1000. Greedy algorithm (forward message along edge that brings it as close to target as possible in terms of lattice distance) leads to  $\log^2 N$  length paths. The only exponent that works is  $d$ .

Simple random model is design principle for some decentralized peer-to-peer networks.

**Cycle with random matching.** Bollobas and Chung proved that a cycle (of even length) plus a random matching has  $\log n$  diameter with high probability, making it a small world network.

*Last modified on August 30, 2011.*

Copyright © 2002–2012 [Robert Sedgewick](#) and [Kevin Wayne](#). All rights reserved.