Algorithms

FOURTH EDITION

ROBERT SEDGEWICK | KEVIN WAYNE

Algorithms

FOURTH EDITION

This page intentionally left blank

Algorithms

FOURTH EDITION

Robert Sedgewick
and
Kevin Wayne

Princeton University

Addison-Wesley

Upper Saddle River, NJ • Boston • Indianapolis • San Francisco
New York • Toronto • Montreal • London • Munich • Paris • Madrid
Capetown • Sydney • Tokyo • Singapore • Mexico City

The authors and publisher have taken care in the preparation of this book, but make no expressed or implied warranty of any kind and assume no responsibility for errors or omissions. No liability is assumed for incidental or consequential damages in connection with or arising out of the use of the information or programs contained herein.

The publisher offers excellent discounts on this book when ordered in quantity for bulk purchases or special sales, which may include electronic versions and/or custom covers and content particular to your business, training goals, marketing focus, and branding interests. For more information, please contact:

U.S. Corporate and Government Sales
(800) 382-3419
corpsales@pearsontechgroup.com

For sales outside the United States, please contact:

International Sales
international@pearson.com

Visit us on the Web: informit.com/aw

Cataloging-in-Publication Data is on file with the Library of Congress.

Pearson Education, Inc.
Rights and Contracts Department
501 Boylston Street, Suite 900
Boston, MA 02116
Fax: (617) 671-3447

ISBN-13: 978-0-321-57351-3
ISBN-10: 0-321-57351-X
Text printed in the United States on recycled paper at Courier in Westford, Massachusetts.
First printing, March 2011

*To Adam, Andrew, Brett, Robbie
and especially Linda*

To Jackie and Alex

CONTENTS

PREFACE

This book is intended to survey the most important computer algorithms in use today, and to teach fundamental techniques to the growing number of people in need of knowing them. It is intended for use as a textbook for a second course in computer science, after students have acquired basic programming skills and familiarity with computer systems. The book also may be useful for self-study or as a reference for people engaged in the development of computer systems or applications programs, since it contains implementations of useful algorithms and detailed information on performance characteristics and clients. The broad perspective taken makes the book an appropriate introduction to the field.

THE STUDY OF ALGORITHMS AND DATA STRUCTURES is fundamental to any computer-science curriculum, but it is not just for programmers and computer-science students. Everyone who uses a computer wants it to run faster or to solve larger problems. The algorithms in this book represent a body of knowledge developed over the last 50 years that has become indispensable. From *N*-body simulation problems in physics to genetic-sequencing problems in molecular biology, the basic methods described here have become essential in scientific research; from architectural modeling systems to aircraft simulation, they have become essential tools in engineering; and from database systems to internet search engines, they have become essential parts of modern software systems. And these are but a few examples—as the scope of computer applications continues to grow, so grows the impact of the basic methods covered here.

Before developing our fundamental approach to studying algorithms, we develop data types for stacks, queues, and other low-level abstractions that we use throughout the book. Then we survey fundamental algorithms for sorting, searching, graphs, and strings. The last chapter is an overview placing the rest of the material in the book in a larger context.

Distinctive features The orientation of the book is to study algorithms likely to be of practical use. The book teaches a broad variety of algorithms and data structures and provides sufficient information about them that readers can confidently implement, debug, and put them to work in any computational environment. The approach involves:

Algorithms. Our descriptions of algorithms are based on complete implementations and on a discussion of the operations of these programs on a consistent set of examples. Instead of presenting pseudo-code, we work with real code, so that the programs can quickly be put to practical use. Our programs are written in Java, but in a style such that most of our code can be reused to develop implementations in other modern programming languages.

Data types. We use a modern programming style based on data abstraction, so that algorithms and their data structures are encapsulated together.

Applications. Each chapter has a detailed description of applications where the algorithms described play a critical role. These range from applications in physics and molecular biology, to engineering computers and systems, to familiar tasks such as data compression and searching on the web.

A scientific approach. We emphasize developing mathematical models for describing the performance of algorithms, using the models to develop hypotheses about performance, and then testing the hypotheses by running the algorithms in realistic contexts.

Breadth of coverage. We cover basic abstract data types, sorting algorithms, searching algorithms, graph processing, and string processing. We keep the material in algorithmic context, describing data structures, algorithm design paradigms, reduction, and problem-solving models. We cover classic methods that have been taught since the 1960s and new methods that have been invented in recent years.

Our primary goal is to introduce the most important algorithms in use today to as wide an audience as possible. These algorithms are generally ingenious creations that, remarkably, can each be expressed in just a dozen or two lines of code. As a group, they represent problem-solving power of amazing scope. They have enabled the construction of computational artifacts, the solution of scientific problems, and the development of commercial applications that would not have been feasible without them.

Booksite An important feature of the book is its relationship to the booksite `algs4.cs.princeton.edu`. This site is freely available and contains an extensive amount of material about algorithms and data structures, for teachers, students, and practitioners, including:

An online synopsis. The text is summarized in the booksite to give it the same overall structure as the book, but linked so as to provide easy navigation through the material.

Full implementations. All code in the book is available on the booksite, in a form suitable for program development. Many other implementations are also available, including advanced implementations and improvements described in the book, answers to selected exercises, and client code for various applications. The emphasis is on testing algorithms in the context of meaningful applications.

Exercises and answers. The booksite expands on the exercises in the book by adding drill exercises (with answers available with a click), a wide variety of examples illustrating the reach of the material, programming exercises with code solutions, and challenging problems.

Dynamic visualizations. Dynamic simulations are impossible in a printed book, but the website is replete with implementations that use a graphics class to present compelling visual demonstrations of algorithm applications.

Course materials. A complete set of lecture slides is tied directly to the material in the book and on the booksite. A full selection of programming assignments, with check lists, test data, and preparatory material, is also included.

Links to related material. Hundreds of links lead students to background information about applications and to resources for studying algorithms.

Our goal in creating this material was to provide a complementary approach to the ideas. Generally, you should read the book when learning specific algorithms for the first time or when trying to get a global picture, and you should use the booksite as a reference when programming or as a starting point when searching for more detail while online.

Use in the curriculum The book is intended as a textbook in a second course in computer science. It provides full coverage of core material and is an excellent vehicle for students to gain experience and maturity in programming, quantitative reasoning, and problem-solving. Typically, one course in computer science will suffice as a prerequisite—the book is intended for anyone conversant with a modern programming language and with the basic features of modern computer systems.

The algorithms and data structures are expressed in Java, but in a style accessible to people fluent in other modern languages. We embrace modern Java abstractions (including generics) but resist dependence upon esoteric features of the language.

Most of the mathematical material supporting the analytic results is self-contained (or is labeled as beyond the scope of this book), so little specific preparation in mathematics is required for the bulk of the book, although mathematical maturity is definitely helpful. Applications are drawn from introductory material in the sciences, again self-contained.

The material covered is a fundamental background for any student intending to major in computer science, electrical engineering, or operations research, and is valuable for any student with interests in science, mathematics, or engineering.

Context The book is intended to follow our introductory text, *An Introduction to Programming in Java: An Interdisciplinary Approach*, which is a broad introduction to the field. Together, these two books can support a two- or three-semester introduction to computer science that will give any student the requisite background to successfully address computation in any chosen field of study in science, engineering, or the social sciences.

The starting point for much of the material in the book was the Sedgewick series of *Algorithms* books. In spirit, this book is closest to the first and second editions of that book, but this text benefits from decades of experience teaching and learning that material. Sedgewick's current *Algorithms in C/C++/Java, Third Edition* is more appropriate as a reference or a text for an advanced course; this book is specifically designed to be a textbook for a one-semester course for first- or second-year college students and as a modern introduction to the basics and a reference for use by working programmers.

Acknowledgments This book has been nearly 40 years in the making, so full recognition of all the people who have made it possible is simply not feasible. Earlier editions of this book list dozens of names, including (in alphabetical order) Andrew Appel, Trina Avery, Marc Brown, Lyn Dupré, Philippe Flajolet, Tom Freeman, Dave Hanson, Janet Incerpi, Mike Schidlowsky, Steve Summit, and Chris Van Wyk. All of these people deserve acknowledgement, even though some of their contributions may have happened decades ago. For this fourth edition, we are grateful to the hundreds of students at Princeton and several other institutions who have suffered through preliminary versions of the work, and to readers around the world for sending in comments and corrections through the booksite.

We are grateful for the support of Princeton University in its unwavering commitment to excellence in teaching and learning, which has provided the basis for the development of this work.

Peter Gordon has provided wise counsel throughout the evolution of this work almost from the beginning, including a gentle introduction of the "back to the basics" idea that is the foundation of this edition. For this fourth edition, we are grateful to Barbara Wood for her careful and professional copyediting, to Julie Nahil for managing the production, and to many others at Pearson for their roles in producing and marketing the book. All were extremely responsive to the demands of a rather tight schedule without the slightest sacrifice to the quality of the result.

Robert Sedgewick
Kevin Wayne

Princeton, NJ
January, 2011

This page intentionally left blank

ONE

Fundamentals

The objective of this book is to study a broad variety of important and useful *algorithms*—methods for solving problems that are suited for computer implementation. Algorithms go hand in hand with *data structures*—schemes for organizing data that leave them amenable to efficient processing by an algorithm. This chapter introduces the basic tools that we need to study algorithms and data structures.

First, we introduce our *basic programming model*. All of our programs are implemented using a small subset of the Java programming language plus a few of our own libraries for input/output and for statistical calculations. SECTION 1.1 is a summary of language constructs, features, and libraries that we use in this book.

Next, we emphasize *data abstraction*, where we define *abstract data types* (ADTs) in the service of modular programming. In SECTION 1.2 we introduce the process of implementing an ADT in Java, by specifying an *applications programming interface* (API) and then using the Java class mechanism to develop an implementation for use in client code.

As important and useful examples, we next consider three fundamental ADTs: the *bag*, the *queue*, and the *stack*. SECTION 1.3 describes APIs and implementations of bags, queues, and stacks using arrays, resizing arrays, and linked lists that serve as models and starting points for algorithm implementations throughout the book.

Performance is a central consideration in the study of algorithms. SECTION 1.4 describes our approach to analyzing algorithm performance. The basis of our approach is the *scientific method*: we develop hypotheses about performance, create mathematical models, and run experiments to test them, repeating the process as necessary.

We conclude with a case study where we consider solutions to a *connectivity* problem that uses algorithms and data structures that implement the classic *union-find* ADT.

Algorithms When we write a computer program, we are generally implementing a *method* that has been devised previously to solve some problem. This method is often independent of the particular programming language being used—it is likely to be equally appropriate for many computers and many programming languages. It is the method, rather than the computer program itself, that specifies the steps that we can take to solve the problem. The term *algorithm* is used in computer science to describe a finite, deterministic, and effective problem-solving method suitable for implementation as a computer program. Algorithms are the stuff of computer science: they are central objects of study in the field.

We can define an algorithm by describing a procedure for solving a problem in a natural language, or by writing a computer program that implements the procedure, as shown at right for *Euclid's algorithm* for finding the greatest common divisor of two numbers, a variant of which was devised over 2,300 years ago. If you are not familiar with Euclid's algorithm, you are encouraged to work EXERCISE 1.1.24 and EXERCISE 1.1.25, perhaps after reading SECTION 1.1. In this book, we use computer programs to describe algorithms. One important reason for doing so is that it makes easier the task of checking whether they are finite, deterministic, and effective, as required. But it is also important to recognize that a program in a particular language is just one way to express an algorithm. The fact that many of the algorithms in this book have been expressed in multiple programming languages over the past several decades reinforces the idea that each algorithm is a method suitable for implementation on any computer in any programming language.

English-language description

Compute the greatest common divisor of two nonnegative integers p and q as follows: If q is 0, the answer is p. If not, divide p by q and take the remainder r. The answer is the greatest common divisor of q and r.

Java-language description

```
public static int gcd(int p, int q)
{
   if (q == 0) return p;
   int r = p % q;
   return gcd(q, r);
}
```

Euclid's algorithm

Most algorithms of interest involve organizing the data involved in the computation. Such organization leads to *data structures*, which also are central objects of study in computer science. Algorithms and data structures go hand in hand. In this book we take the view that data structures exist as the byproducts or end products of algorithms and that we must therefore study them in order to understand the algorithms. Simple algorithms can give rise to complicated data structures and, conversely, complicated algorithms can use simple data structures. We shall study the properties of many data structures in this book; indeed, we might well have titled the book *Algorithms and Data Structures*.

When we use a computer to help us solve a problem, we typically are faced with a number of possible approaches. For small problems, it hardly matters which approach we use, as long as we have one that correctly solves the problem. For huge problems (or applications where we need to solve huge numbers of small problems), however, we quickly become motivated to devise methods that use time and space efficiently.

The primary reason to learn about algorithms is that this discipline gives us the potential to reap huge savings, even to the point of enabling us to do tasks that would otherwise be impossible. In an application where we are processing millions of objects, it is not unusual to be able to make a program millions of times faster by using a well-designed algorithm. We shall see such examples on numerous occasions throughout the book. By contrast, investing additional money or time to buy and install a new computer holds the potential for speeding up a program by perhaps a factor of only 10 or 100. Careful algorithm design is an extremely effective part of the process of solving a huge problem, whatever the applications area.

When developing a huge or complex computer program, a great deal of effort must go into understanding and defining the problem to be solved, managing its complexity, and decomposing it into smaller subtasks that can be implemented easily. Often, many of the algorithms required after the decomposition are trivial to implement. In most cases, however, there are a few algorithms whose choice is critical because most of the system resources will be spent running those algorithms. These are the types of algorithms on which we concentrate in this book. We study fundamental algorithms that are useful for solving challenging problems in a broad variety of applications areas.

The sharing of programs in computer systems is becoming more widespread, so although we might expect to be *using* a large fraction of the algorithms in this book, we also might expect to have to *implement* only a small fraction of them. For example, the Java libraries contain implementations of a host of fundamental algorithms. However, implementing simple versions of basic algorithms helps us to understand them better and thus to more effectively use and tune advanced versions from a library. More important, the opportunity to reimplement basic algorithms arises frequently. The primary reason to do so is that we are faced, all too often, with completely new computing environments (hardware and software) with new features that old implementations may not use to best advantage. In this book, we concentrate on the simplest reasonable implementations of the best algorithms. We do pay careful attention to coding the critical parts of the algorithms, and take pains to note where low-level optimization effort could be most beneficial.

The choice of the best algorithm for a particular task can be a complicated process, perhaps involving sophisticated mathematical analysis. The branch of computer science that comprises the study of such questions is called *analysis of algorithms*. Many of the algorithms that we study have been shown through analysis to have excellent theoretical performance; others are simply known to work well through experience. Our primary goal is to learn reasonable algorithms for important tasks, yet we shall also pay careful attention to comparative performance of the methods. We should not use an algorithm without having an idea of what resources it might consume, so we strive to be aware of how our algorithms might be expected to perform.

Summary of topics As an overview, we describe the major parts of the book, giving specific topics covered and an indication of our general orientation toward the material. This set of topics is intended to touch on as many fundamental algorithms as possible. Some of the areas covered are core computer-science areas that we study in depth to learn basic algorithms of wide applicability. Other algorithms that we discuss are from advanced fields of study within computer science and related fields. The algorithms that we consider are the products of decades of research and development and continue to play an essential role in the ever-expanding applications of computation.

Fundamentals (CHAPTER 1) in the context of this book are the basic principles and methodology that we use to implement, analyze, and compare algorithms. We consider our Java programming model, data abstraction, basic data structures, abstract data types for collections, methods of analyzing algorithm performance, and a case study.

Sorting algorithms (CHAPTER 2) for rearranging arrays in order are of fundamental importance. We consider a variety of algorithms in considerable depth, including insertion sort, selection sort, shellsort, quicksort, mergesort, and heapsort. We also encounter algorithms for several related problems, including priority queues, selection, and merging. Many of these algorithms will find application as the basis for other algorithms later in the book.

Searching algorithms (CHAPTER 3) for finding specific items among large collections of items are also of fundamental importance. We discuss basic and advanced methods for searching, including binary search trees, balanced search trees, and hashing. We note relationships among these methods and compare performance.

Graphs (CHAPTER 4) are sets of objects and connections, possibly with weights and orientation. Graphs are useful models for a vast number of difficult and important problems, and the design of algorithms for processing graphs is a major field of study. We consider depth-first search, breadth-first search, connectivity problems, and several algorithms and applications, including Kruskal's and Prim's algorithms for finding minimum spanning tree and Dijkstra's and the Bellman-Ford algorithms for solving shortest-paths problems.

Strings (CHAPTER 5) are an essential data type in modern computing applications. We consider a range of methods for processing sequences of characters. We begin with faster algorithms for sorting and searching when keys are strings. Then we consider substring search, regular expression pattern matching, and data-compression algorithms. Again, an introduction to advanced topics is given through treatment of some elementary problems that are important in their own right.

Context (CHAPTER 6) helps us relate the material in the book to several other advanced fields of study, including scientific computing, operations research, and the theory of computing. We survey event-based simulation, B-trees, suffix arrays, maximum flow, and other advanced topics from an introductory viewpoint to develop appreciation for the interesting advanced fields of study where algorithms play a critical role. Finally, we describe search problems, reduction, and NP-completeness to introduce the theoretical underpinnings of the study of algorithms and relationships to material in this book.

THE STUDY OF ALGORITHMS IS INTERESTING AND EXCITING because it is a new field (almost all the algorithms that we study are less than 50 years old, and some were just recently discovered) with a rich tradition (a few algorithms have been known for hundreds of years). New discoveries are constantly being made, but few algorithms are completely understood. In this book we shall consider intricate, complicated, and difficult algorithms as well as elegant, simple, and easy ones. Our challenge is to understand the former and to appreciate the latter in the context of scientific and commercial applications. In doing so, we shall explore a variety of useful tools and develop a style of *algorithmic thinking* that will serve us well in computational challenges to come.

1.1 BASIC PROGRAMMING MODEL

OUR STUDY OF ALGORITHMS is based upon implementing them as *programs* written in the Java programming language. We do so for several reasons:

- Our programs are concise, elegant, and complete descriptions of algorithms.
- You can run the programs to study properties of the algorithms.
- You can put the algorithms immediately to good use in applications.

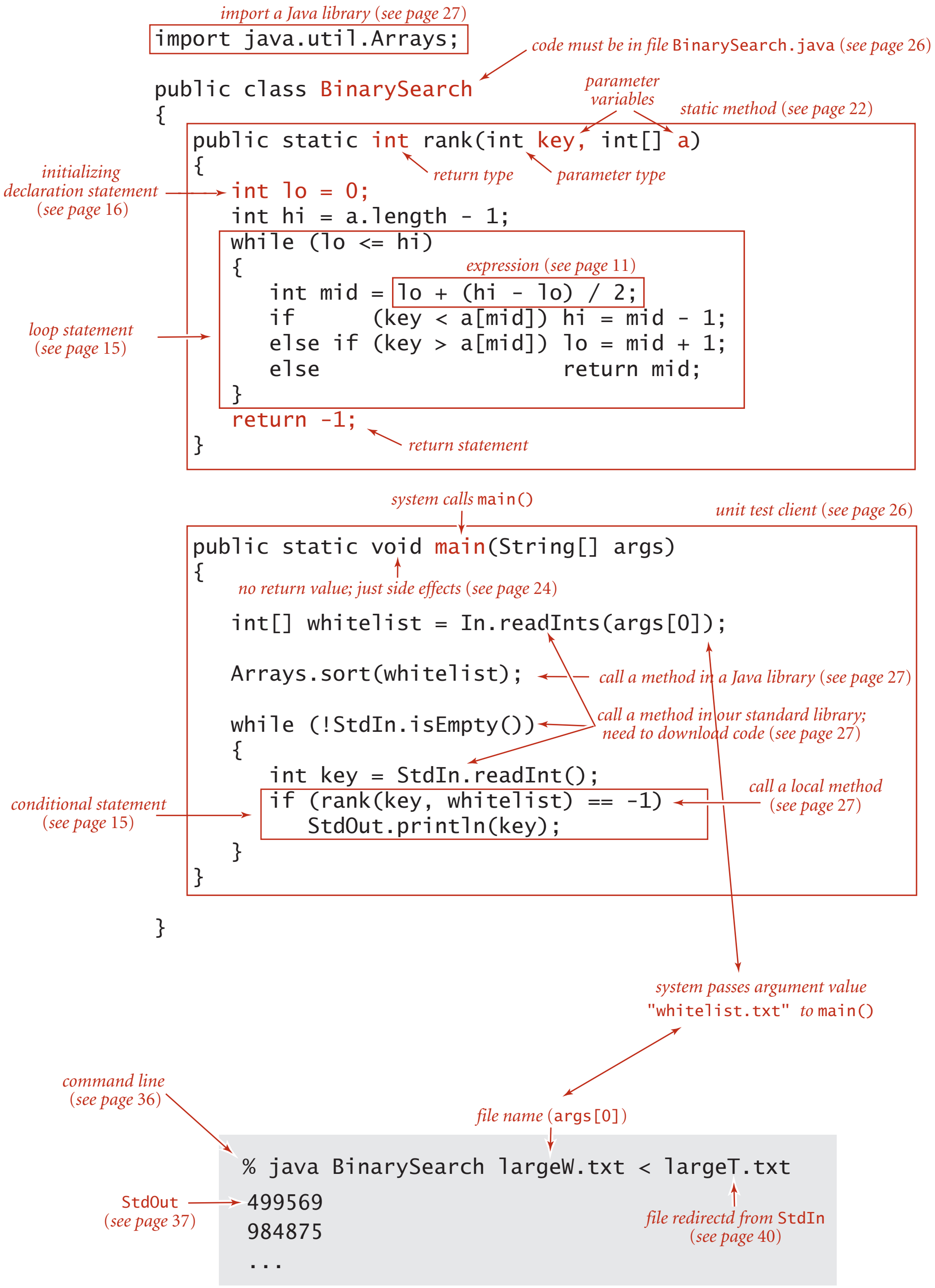
These are important and significant advantages over the alternatives of working with English-language descriptions of algorithms.

A potential downside to this approach is that we have to work with a specific programming language, possibly making it difficult to separate the idea of the algorithm from the details of its implementation. Our implementations are designed to mitigate this difficulty, by using programming constructs that are both found in many modern languages and needed to adequately describe the algorithms.

We use only a small subset of Java. While we stop short of formally defining the subset that we use, you will see that we make use of relatively few Java constructs, and that we emphasize those that are found in many modern programming languages. The code that we present is complete, and our expectation is that you will download it and execute it, on our test data or test data of your own choosing.

We refer to the programming constructs, software libraries, and operating system features that we use to implement and describe algorithms as our *programming model*. In this section and SECTION 1.2, we fully describe this programming model. The treatment is self-contained and primarily intended for documentation and for your reference in understanding any code in the book. The model we describe is the same model introduced in our book *An Introduction to Programming in Java: An Interdisciplinary Approach*, which provides a slower-paced introduction to the material.

For reference, the figure on the facing page depicts a complete Java program that illustrates many of the basic features of our programming model. We use this code for examples when discussing language features, but defer considering it in detail to page 46 (it implements a classic algorithm known as *binary search* and tests it for an application known as *whitelist filtering*). We assume that you have experience programming in some modern language, so that you are likely to recognize many of these features in this code. Page references are included in the annotations to help you find answers to any questions that you might have. Since our code is somewhat stylized and we strive to make consistent use of various Java idioms and constructs, it is worthwhile even for experienced Java programmers to read the information in this section.



```
import java.util.Arrays;

public class BinarySearch
{
   public static int rank(int key, int[] a)
   {
      int lo = 0;
      int hi = a.length - 1;
      while (lo <= hi)
      {
         int mid = lo + (hi - lo) / 2;
         if      (key < a[mid]) hi = mid - 1;
         else if (key > a[mid]) lo = mid + 1;
         else                   return mid;
      }
      return -1;
   }

   public static void main(String[] args)
   {
      int[] whitelist = In.readInts(args[0]);

      Arrays.sort(whitelist);

      while (!StdIn.isEmpty())
      {
         int key = StdIn.readInt();
         if (rank(key, whitelist) == -1)
            StdOut.println(key);
      }
   }

}
```

```
% java BinarySearch largeW.txt < largeT.txt
499569
984875
...
```

import a Java library (see page 27)
code must be in file BinarySearch.java (see page 26)
parameter variables
static method (see page 22)
initializing declaration statement (see page 16)
return type
parameter type
expression (see page 11)
loop statement (see page 15)
return statement
system calls main()
unit test client (see page 26)
no return value; just side effects (see page 24)
call a method in a Java library (see page 27)
call a method in our standard library; need to download code (see page 27)
call a local method (see page 27)
conditional statement (see page 15)
system passes argument value "whitelist.txt" to main()
command line (see page 36)
file name (args[0])
StdOut (see page 37)
file redirectd from StdIn (see page 40)

Anatomy of a Java program and its invocation from the command line

Basic structure of a Java program A Java program (*class*) is either a *library of static methods* (functions) or a *data type definition*. To create libraries of static methods and data-type definitions, we use the following five components, the basis of programming in Java and many other modern languages:

- *Primitive data types* precisely define the meaning of terms like *integer*, *real number*, and *boolean value* within a computer program. Their definition includes the set of possible values and *operations* on those values, which can be combined into *expressions* like mathematical expressions that define values.
- *Statements* allow us to define a computation by creating and assigning values to *variables*, controlling execution flow, or causing side effects. We use six types of statements: *declarations*, *assignments*, *conditionals*, *loops*, *calls*, and *returns*.
- *Arrays* allow us to work with multiple values of the same type.
- *Static methods* allow us to encapsulate and reuse code and to develop programs as a set of independent modules.
- *Strings* are sequences of characters. Some operations on them are built in to Java.
- *Input/output* sets up communication between programs and the outside world.
- *Data abstraction* extends encapsulation and reuse to allow us to define non-primitive data types, thus supporting object-oriented programming.

In this section, we will consider the first five of these in turn. Data abstraction is the topic of the next section.

Running a Java program involves interacting with an operating system or a program development environment. For clarity and economy, we describe such actions in terms of a *virtual terminal*, where we interact with programs by typing commands to the system. See the booksite for details on using a virtual terminal on your system, or for information on using one of the many more advanced program development environments that are available on modern systems.

For example, `BinarySearch` is two static methods, `rank()` and `main()`. The first static method, `rank()`, is four statements: two declarations, a loop (which is itself an assignment and two conditionals), and a return. The second, `main()`, is three statements: a declaration, a call, and a loop (which is itself an assignment and a conditional).

To invoke a Java program, we first *compile* it using the `javac` command, then *run* it using the `java` command. For example, to run `BinarySearch`, we first type the command `javac BinarySearch.java` (which creates a file `BinarySearch.class` that contains a lower-level version of the program in Java *bytecode* in the file `BinarySearch.class`). Then we type `java BinarySearch` (followed by a whitelist file name) to transfer control to the bytecode version of the program. To develop a basis for understanding the effect of these actions, we next consider in detail primitive data types and expressions, the various kinds of Java statements, arrays, static methods, strings, and input/output.

Primitive data types and expressions A *data type* is a set of values and a set of operations on those values. We begin by considering the following four *primitive* data types that are the basis of the Java language:

- *Integers*, with arithmetic operations (`int`)
- *Real numbers*, again with arithmetic operations (`double`)
- *Booleans*, the set of values { *true*, *false* } with logical operations (`boolean`)
- *Characters*, the alphanumeric characters and symbols that you type (`char`)

Next we consider mechanisms for specifying values and operations for these types.

A Java program manipulates *variables* that are named with *identifiers*. Each variable is associated with a data type and stores one of the permissible data-type values. In Java code, we use *expressions* like familiar mathematical expressions to apply the operations associated with each type. For primitive types, we use identifiers to refer to variables, *operator* symbols such as `+ - * /` to specify operations, *literals* such as `1` or `3.14` to specify values, and expressions such as `(x + 2.236)/2` to specify operations on values. The purpose of an expression is to define one of the data-type values.

term	examples		definition
primitive data type	`int double boolean char`		a set of values and a set of operations on those values (built in to the Java language)
identifier	`a abc Ab$ a_b ab123 lo hi`		a sequence of letters, digits, _, and `$`, the first of which is not a digit
variable	[*any identifier*]		names a data-type value
operator	`+ - * /`		names a data-type operation
literal	`int`	`1 0 -42`	source-code representation of a value
	`double`	`2.0 1.0e-15 3.14`	
	`boolean`	`true false`	
	`char`	`'a' '+' '9' '\n'`	
expression	`int`	`lo + (hi - lo)/2`	a literal, a variable, or a sequence of operations on literals and/or variables that produces a value
	`double`	`1.0e-15 * t`	
	`boolean`	`lo <= hi`	

Basic building blocks for Java programs

To define a data type, we need only specify the values and the set of operations on those values. This information is summarized in the table below for Java's `int`, `double`, `boolean`, and `char` data types. These data types are similar to the basic data types found in many programming languages. For `int` and `double`, the operations are familiar arithmetic operations; for `boolean`, they are familiar logical operations. It is important to note that `+`, `-`, `*`, and `/` are *overloaded*—the same symbol specifies operations in multiple different types, depending on context. The key property of these primitive operations is that *an operation involving values of a given type has a value of that type*. This rule highlights the idea that we are often working with approximate values, since it is often the case that the exact value that would seem to be defined by the expression is not a value of the type. For example, `5/3` has the value `1` and `5.0/3.0` has a value very close to `1.66666666666667` but neither of these is exactly equal to 5/3. This table is far from complete; we discuss some additional operators and various exceptional situations that we occasionally need to consider in the Q&A at the end of this section.

type	set of values	operators	typical expressions	
			expression	value
`int`	integers between 2^{31} and$+2^{31}-1$ (32-bit two's complement)	`+` (add)	`5 + 3`	`8`
		`-` (subtract)	`5 - 3`	`2`
		`*` (multiply)	`5 * 3`	`15`
		`/` (divide)	`5 / 3`	`1`
		`%` (remainder)	`5 % 3`	`2`
`double`	double-precision real numbers (64-bit IEEE 754 standard)	`+` (add)	`3.141 - .03`	`3.111`
		`-` (subtract)	`2.0 - 2.0e-7`	`1.9999998`
		`*` (multiply)	`100 * .015`	`1.5`
		`/` (divide)	`6.02e23 / 2.0`	`3.01e23`
`boolean`	`true` or `false`	`&&` (and)	`true && false`	`false`
		`||` (or)	`false || true`	`true`
		`!` (not)	`!false`	`true`
		`^` (xor)	`true ^ true`	`false`
`char`	characters (16-bit)	[*arithmetic operations, rarely used*]		

Primitive data types in Java

Expressions. As illustrated in the table at the bottom of the previous page, typical expressions are *infix*: a literal (or an expression), followed by an operator, followed by another literal (or another expression). When an expression contains more than one operator, the order in which they are applied is often significant, so the following *precedence* conventions are part of the Java language specification: The operators `*` and `/` (and `%`) have higher precedence than (are applied before) the `+` and `-` operators; among logical operators, `!` is the highest precedence, followed by `&&` and then `||`. Generally, operators of the same precedence are applied left to right. As in standard arithmetic expressions, you can use parentheses to override these rules. Since precedence rules vary slightly from language to language, we use parentheses and otherwise strive to avoid dependence on precedence rules in our code.

Type conversion. Numbers are automatically promoted to a more inclusive type if no information is lost. For example, in the expression `1 + 2.5` , the `1` is promoted to the double value `1.0` and the expression evaluates to the `double` value `3.5` . A *cast* is a type name in parentheses within an expression, a directive to convert the following value into a value of that type. For example `(int) 3.7` is `3` and `(double) 3` is `3.0`. Note that casting to an `int` is truncation instead of rounding—rules for casting within complicated expressions can be intricate, and casts should be used sparingly and with care. A best practice is to use expressions that involve literals or variables of a single type.

Comparisons. The following operators compare two values of the same type and produce a `boolean` value: *equal* (`==`), *not equal* (`!=`), *less than* (`<`), *less than or equal* (`<=`), *greater than* (`>`), and *greater than or equal* (`>=`). These operators are known as *mixed-type* operators because their value is `boolean`, not the type of the values being compared. An expression with a boolean value is known as a *boolean expression*. Such expressions are essential components in conditional and loop statements, as we will see.

Other primitive types. Java's `int` has 2^{32} different values by design, so it can be represented in a 32-bit machine word (many machines have 64-bit words nowadays, but the 32-bit `int` persists). Similarly, the `double` standard specifies a 64-bit representation. These data-type sizes are adequate for typical applications that use integers and real numbers. To provide flexibility, Java has five additional primitive data types:

- 64-bit integers, with arithmetic operations (`long`)
- 16-bit integers, with arithmetic operations (`short`)
- 16-bit characters, with arithmetic operations (`char`)
- 8-bit integers, with arithmetic operations (`byte`)
- 32-bit single-precision real numbers, again with arithmetic operations (`float`)

We most often use `int` and `double` arithmetic operations in this book, so we do not consider the others (which are very similar) in further detail here.

Statements A Java program is composed of *statements*, which define the computation by creating and manipulating variables, assigning data-type values to them, and controlling the flow of execution of such operations. Statements are often organized in blocks, sequences of statements within curly braces.

- *Declarations* create variables of a specified type and name them with identifiers.
- *Assignments* associate a data-type value (defined by an expression) with a variable. Java also has several *implicit assignment* idioms for changing the value of a data-type value relative to its current value, such as incrementing the value of an integer variable.
- *Conditionals* provide for a simple change in the flow of execution—execute the statements in one of two blocks, depending on a specified condition.
- *Loops* provide for a more profound change in the flow of execution—execute the statements in a block as long as a given condition is true.
- *Calls* and *returns* relate to static methods (see page 22), which provide another way to change the flow of execution and to organize code.

A program is a sequence of statements, with declarations, assignments, conditionals, loops, calls, and returns. Programs typically have a *nested* structure: a statement among the statements in a block within a conditional or a loop may itself be a conditional or a loop. For example, the `while` loop in `rank()` contains an `if` statement. Next, we consider each of these types of statements in turn.

Declarations. 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 checks for consistency (for example, it does not permit us to multiply a `boolean` and a `double`). Declarations can appear anywhere before a variable is first used—most often, we put them *at* the point of first use. The *scope* of a variable is the part of the program where it is defined. Generally the scope of a variable is composed of the statements that follow the declaration in the same block as the declaration.

Assignments. An *assignment* statement associates a data-type value (defined by an expression) 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 a value of the type.

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

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

This description introduces a formal notation known as a *template* that we use occasionally 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 `<block statements>` represents a sequence of Java statements. It is possible to make formal definitions of `<boolean expression>` and `<block statements>`, but we refrain from going into that level of detail. The meaning of an `if` statement is self-explanatory: the statement(s) in the block are to be executed if and only if the boolean expression is `true`. The `if-else` statement:

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

allows for choosing between two alternative blocks of statements.

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

```
while (<boolean expression>) { <block 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 boolean expression is `false`, do nothing; if the boolean expression is `true`, execute the sequence of statements in the block (just as with `if`) but then check the boolean expression again, execute the sequence of statements in the block again if the boolean expression is `true`, and continue as long as the boolean expression is `true`. We refer to the statements in the block in a loop as the *body* of the loop.

Break and continue. Some situations call for slightly more complicated control flow than provide by the basic `if` and `while` statements. Accordingly, Java supports two additional statements for use within `while` loops:

- The `break` statement, which immediately exits the loop
- The `continue` statement, which immediately begins the next iteration of the loop

We rarely use these statements in the code in this book (and many programmers never use them), but they do considerably simplify code in certain instances.

Shortcut notations There are several ways to express a given computation; we seek clear, elegant, and efficient code. Such code often takes advantage of the following widely used shortcuts (that are found in many languages, not just Java).

Initializing declarations. We can combine a declaration with an assignment to initialize a variable at the same time that it is declared (created). For example, the code `int i = 1;` creates an `int` variable named `i` *and* assigns it the initial value `1`. A best practice is to use this mechanism close to first use of the variable (to limit scope).

Implicit assignments. The following shortcuts are available when our purpose is to modify a variable's value relative to its current value:

- Increment/decrement operators: `i++` is the same as `i = i + 1` and has the value `i` in an expression. Similarly, `i--` is the same as `i = i - 1`. The code `++i` and `--i` are the same except that the expression value is taken *after* the increment/decrement, not before.
- Other compound operations: Prepending a binary operator to the `=` in an assignment is equivalent to using the variable on the left as the first operand. For example, the code `i/=2;` is equivalent to the code `i = i/2;` Note that `i += 1;` has the same effect as `i = i+1;` (and `i++`).

Single-statement blocks. If a block of statements in a conditional or a loop has only a single statement, the curly braces may be omitted.

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 compactly with Java's `for` notation:

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

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

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

We use `for` loops to support this initialize-and-increment programming idiom.

statement	examples	definition

declaration	`int i;`
	`double c;`
		create a variable of a specified type, named with a given identifier

assignment	`a = b + 3;`
	`discriminant = b*b - 4.0*c;`
		assign a data-type value to a variable

initializing declaration	`int i = 1;`
	`double c = 3.141592625;`
		declaration that also assigns an initial value

implicit assignment	`i++;`
	`i += 1;`
		`i = i + 1;`

conditional (`if`)	`if (x < 0) x = -x;`	execute a statement, depending on boolean expression

conditional (`if-else`)	`if (x > y) max = x;`
	`else       max = y;`
		execute one or the other statement, depending on boolean expression

loop (`while`)	`int v = 0;`
	`while (v <= N)`
	`   v = 2*v;`
	`double t = c;`
	`while (Math.abs(t - c/t) > 1e-15*t)`
	`   t = (c/t + t) / 2.0;`
		execute statement until boolean expression is `false`

loop (`for`)	`for (int i = 1; i <= N; i++)`
	`   sum += 1.0/i;`
	`for (int i = 0; i <= N; i++)`
	`   StdOut.println(2*Math.PI*i/N);`
		compact version of `while` statement

call	`int key = StdIn.readInt();`	invoke other methods (see page 22)
return	`return false;`	return from a method (see page 24)

Java statements

Arrays An *array* stores a sequence of values that are all of the same type. We want not only to store values but also to 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 in Java code by using the notation `a[i]` to refer to the `i`th value for any value of `i` from `0` to `N-1`. This Java construct is known as a *one-dimensional array*.

Creating and initializing an array. 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 length (the number of values). For example, the "long form" code shown at right makes an array of `N` numbers of type `double`, all initialized to `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 keyword *new* in the second statement is a Java directive to create the array. The reason that we need to explicitly create arrays at run time 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). The `for` statement initializes the `N` array values. 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.

long form

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

short form

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

initializing declaration

```
int[] a = { 1, 1, 2, 3, 5, 8 };
```

Declaring, creating and initializing an array

Short form. For economy in code, we often take advantage of Java's default array initialization convention and combine all three steps into a single statement, as in the "short form" code in our example. 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 numeric types and false for type boolean. The third option shown for our example is to specify the initialization values at compile time, by listing literal values between curly braces, separated by commas.

Using an array. Typical array-processing code is shown on page 21. 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 index in square brackets after the array name. Once we create an array, its size is fixed. A program can refer to the length of an array a[] with the code a.length. The last element of an array a[] is always a[a.length-1]. Java does *automatic bounds checking*—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 ArrayOutOfBoundsException runtime exception.

Aliasing. Note carefully that *an array name refers to the whole array*—if we assign one array name to another, then both refer to the same array, as illustrated in the following code fragment.

```
int[] a = new int[N];
...
a[i] = 1234;
...
int[] b = a;
...
b[i] = 5678;  // a[i] is now 5678.
```

This situation is known as *aliasing* and can lead to subtle bugs. If your intent is to make a copy of an array, then you need to declare, create, and initialize a new array and then copy all of the entries in the original array to the new array, as in the third example on page 21.

Two-dimensional arrays. A *two-dimensional array* in Java is an array of one-dimensional arrays. A two-dimensional array may be *ragged* (its arrays may all be of differing lengths), but we most often work with (for appropriate parameters *M* and *N*) *M*-by-*N* two-dimensional arrays that are arrays of *M rows*, each an array of length *N* (so it also makes sense to refer to the array as having *N columns*). Extending Java array constructs to handle two-dimensional arrays is straightforward. To refer to the entry 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 square brackets; and to create the array, we specify the number of rows followed by the number of columns after the type name (both within square 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 numeric types to zero and in arrays of `boolean` values to `false`. Default initialization of two-dimensional arrays is useful because it masks more code than for one-dimensional arrays. The following code is equivalent to the single-line create-and-initialize idiom that we just considered:

```
double[][] a;
a = new double[M][N];
for (int i = 0; i < M; i++)
   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 other value(s).

task	implementation (code fragment)

find the maximum of the array values	`double max = a[0];`
	`for (int i = 1; i < a.length; i++)`
	`   if (a[i] > max) max = a[i];`

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

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

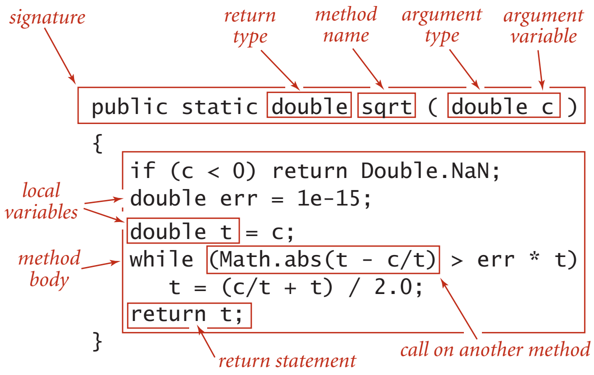
reverse the elements within an array	`int N = a.length;`
	`for (int i = 0; i < N/2; i++)`
	`{`
	`   double temp = a[i];`
	`   a[i] = a[N-1-i];`
	`   a[N-i-1] = temp;`
	`}`

matrix-matrix multiplication (square matrices)
`a[][]*b[][] = c[][]`
	`int N = a.length;`
	`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];`
	`   }`

Typical array-processing code

Static methods Every Java program in this book is either a *data-type definition* (which we describe in detail in Section 1.2) or a *library of static methods* (which we describe here). Static methods are called *functions* in many programming languages, since they can behave like mathematical functions, as described next. Each static method is a sequence of statements that are executed, one after the other, when the static method is *called*, in the manner described below. The modifier *static* distinguishes these methods from *instance methods*, which we discuss in Section 1.2. We use the word *method* without a modifier when describing characteristics shared by both kinds of methods.

Defining a static method. A *method* encapsulates a computation that is defined as a sequence of statements. A method takes *arguments* (values of given data types) and computes a *return value* of some data type that depends upon the arguments (such as a value defined by a mathematical function) or causes a *side effect* that depends on the arguments (such as printing a value). The static method `rank()` in `BinarySearch` is an example of the first; `main()` is an example of the second. Each static method is composed of a *signature* (the keywords `public static` followed by a return type, the method name, and a sequence of arguments, each with a declared type) and a *body* (a statement block: a sequence of statements, enclosed in curly braces). Examples of static methods are shown in the table on the facing page.



```
public static double sqrt ( double c )
{
   if (c < 0) return Double.NaN;
   double err = 1e-15;
   double t = c;
   while (Math.abs(t - c/t) > err * t)
      t = (c/t + t) / 2.0;
   return t;
}
```

Anatomy of a static method

Invoking a static method. A *call* on a static method is its name followed by expressions that specify argument values in parentheses, separated by commas. When the method call is part of an expression, the method computes a value and that value is used in place of the call in the expression. For example the call on `rank()` in `BinarySearch()` returns an `int` value. A method call followed by a semicolon is a *statement* that generally causes side effects. For example, the call `Arrays.sort()` in `main()` in `BinarySearch` is a call on the system method `Arrays.sort()` that has the side effect of putting the entries in the array in sorted order. When a method is called, its argument variables are initialized with the values of the corresponding expressions in the call. A `return` statement terminates a static method, returning control to the caller. If the static method is to compute a value, that value must be specified in a `return` statement (if such a static method can reach the end of its sequence of statements without a `return`, the compiler will report the error).

task | **implementation**

absolute value of an `int` *value*

```
public static int abs(int x)
{
   if (x < 0) return -x;
   else       return  x;
}
```

absolute value of a `double` *value*

```
public static double abs(double x)
{
   if (x < 0.0) return -x;
   else         return  x;
}
```

primality test

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

square root (Newton's method)

```
public static double sqrt(double c)
{
   if (c > 0) return Double.NaN;
   double err = 1e-15;
   double t = c;
   while (Math.abs(t - c/t) > err * t)
      t = (c/t + t) / 2.0;
   return t;
}
```

hypotenuse of a right triangle

```
public static double hypotenuse(double a, double b)
{  return Math.sqrt(a*a + b*b);  }
```

Harmonic number (see page 185)

```
public static double H(int N)
{
   double sum = 0.0;
   for (int i = 1; i <= N; i++)
      sum += 1.0 / i;
   return sum;
}
```

Typical implementations of static methods

Properties of methods. A complete detailed description of the properties of methods is beyond our scope, but the following points are worth noting:

- *Arguments are passed by value.* You can use argument variables anywhere in the code in the body of the method in the same way you use local variables. The only difference between an argument variable and a local variable is that the argument variable is initialized with the argument value provided by the calling code. The method works with the value of its arguments, not the arguments themselves. One consequence of this approach is that changing the value of an argument variable within a static method has no effect on the calling code. Generally, we do not change argument variables in the code in this book. The pass-by-value convention implies that array arguments are aliased (see page 19)—the method uses the argument variable to refer to the caller's array and can change the contents of the array (though it cannot change the array itself). For example, `Arrays.sort()` certainly changes the contents of the array passed as argument: it puts the entries in order.
- *Method names can be overloaded.* For example, the Java `Math` library uses this approach to provide implementations of `Math.abs()`, `Math.min()`, and `Math.max()` for all primitive numeric types. Another common use of overloading is to define two different versions of a function, one that takes an argument and another that uses a default value of that argument.
- *A method has a single return value but may have multiple return statements.* A Java method can provide only one return value, of the type declared in the method signature. 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. 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.
- *A method can have side effects.* A 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. A `void` static method is said to produce side effects (consume input, produce output, change entries in an array, or otherwise change the state of the system). For example, the `main()` static method in our programs has a `void` return type because its purpose is to produce output. Technically, `void` methods do not implement mathematical functions (and neither does `Math.random()`, which takes no arguments but does produce a return value).

The instance methods that are the subject of SECTION 2.1 share these properties, though profound differences surround the issue of side effects.

Recursion. A method can call itself (if you are not comfortable with this idea, known as *recursion*, you are encouraged to work Exercises 1.1.16 through 1.1.22). For example, the code at the bottom of this page gives an alternate implementation of the `rank()` method in `BinarySearch`. We often use recursive implementations of methods because they can lead to compact, elegant code that is easier to understand than a corresponding implementation that does not use recursion. For example, the comment in the implementation below provides a succinct description of what the code is supposed to do. We can use this comment to convince ourselves that it operates correctly, by mathematical induction. We will expand on this topic and provide such a proof for binary search in Section 3.1. There are three important rules of thumb in developing recursive programs:

- The recursion has a *base case*—we always include a conditional statement as the first statement in the program that has a `return`.
- Recursive calls must address subproblems that are *smaller* in some sense, so that recursive calls converge to the base case. In the code below, the difference between the values of the fourth and the third arguments always decreases.
- Recursive calls should not address subproblems that *overlap*. In the code below, the portions of the array referenced by the two subproblems are disjoint.

Violating any of these guidelines is likely to lead to incorrect results or a spectacularly inefficient program (see Exercises 1.1.19 and 1.1.27). Adhering to them is likely to lead to a clear and correct program whose performance is easy to understand. Another reason to use recursive methods is that they lead to mathematical models that we can use to understand performance. We address this issue for binary search in Section 3.2 and in several other instances throughout the book.

```
public static int rank(int key, int[] a)
{  return rank(key, a, 0, a.length - 1);  }

public static int rank(int key, int[] a, int lo, int hi)
{  // Index of key in a[], if present, is not smaller than lo
   //                                  and not larger than hi.
   if (lo > hi) return -1;
   int mid = lo + (hi - lo) / 2;
   if      (key < a[mid]) return rank(key, a, lo, mid - 1);
   else if (key > a[mid]) return rank(key, a, mid + 1, hi);
   else                   return mid;
}
```

Recursive implementation of binary search

Basic programming model. A *library of static methods* is a set of static methods that are defined in a Java class, by creating a file with the keywords `public class` followed by the class name, followed by the static methods, enclosed in braces, kept in a file with the same name as the class and a `.java` extension. A basic model for Java programming is to develop a program that addresses a specific computational task by creating a library of static methods, one of which is named `main()`. Typing `java` followed by a class name followed by a sequence of strings leads to a call on `main()` in that class, with an array containing those strings as argument. After the last statement in `main()` executes, the program terminates. In this book, when we talk of a *Java program* for accomplishing a task, we are talking about code developed along these lines (possibly also including a data-type definition, as described in SECTION 1.2). For example, `BinarySearch` is a Java program composed of two static methods, `rank()` and `main()`, that accomplishes the task of printing numbers on an input stream that are not found in a whitelist file given as command-line argument.

Modular programming. Of critical importance in this model is that libraries of static methods enable *modular programming* where we build libraries of static methods (*modules*) and a static method in one library can call static methods defined in other libraries. This approach has many important advantages. It allows us to

- Work with modules of reasonable size, even in program involving a large amount of code
- Share and reuse code without having to reimplement it
- Easily substitute improved implementations
- Develop appropriate abstract models for addressing programming problems
- Localize debugging (see the paragraph below on unit testing)

For example, `BinarySearch` makes use of three other independently developed libraries, our `StdIn` and `In` library and Java's `Arrays` library. Each of these libraries, in turn, makes use of several other libraries.

Unit testing. A best practice in Java programming is to include a `main()` in every library of static methods that tests the methods in the library (some other programming languages disallow multiple `main()` methods and thus do not support this approach). Proper unit testing can be a significant programming challenge in itself. At a minimum, every module should contain a `main()` method that exercises the code in the module and provides some assurance that it works. As a module matures, we often refine the `main()` method to be a *development client* that helps us do more detailed tests as we develop the code, or a *test client* that tests all the code extensively. As a client becomes more complicated, we might put it in an independent module. In this book, we use `main()` to help illustrate the purpose of each module and leave test clients for exercises.

External libraries. We use static methods from four different kinds of libraries, each requiring (slightly) differing procedures for code reuse. Most of these are libraries of static methods, but a few are data-type definitions that also include some static methods.

- The standard system libraries `java.lang.*`. These include `Math`, which contains methods for commonly used mathematical functions; `Integer` and `Double`, which we use for converting between strings of characters and `int` and `double` values; `String` and `StringBuilder`, which we discuss in detail later in this section and in CHAPTER 5; and dozens of other libraries that we do not use.
- Imported system libraries such as `java.util.Arrays`. There are thousands of such libraries in a standard Java release, but we make scant use of them in this book. An `import` statement at the beginning of the program is needed to use such libraries (and signal that we are doing so).
- Other libraries in this book. For example, another program can use `rank()` in `BinarySearch`. To use such a program, download the source from the booksite into your working directory.
- The standard libraries `Std*` that we have developed for use in this book (and our introductory book *An Introduction to Programming in Java: An Interdisciplinary Approach*). These libraries are summarized in the following several pages. Source code and instructions for downloading them are available on the booksite.

To invoke a method from another library (one in the same directory or a specified directory, a standard system library, or a system library that is named in an `import` statement before the class definition), we prepend the library name to the method name for each call. For example, the `main()` method in `BinarySearch` calls the `sort()` method in the system library `java.util.Arrays`, the `readInts()` method in our library `In`, and the `println()` method in our library `StdOut`.

standard system libraries

`Math`

`Integer`†

`Double`†

`String`†

`StringBuilder`

`System`

imported system libraries

`java.util.Arrays`

our standard libraries

`StdIn`

`StdOut`

`StdDraw`

`StdRandom`

`StdStats`

`In`†

`Out`†

† *data type definitions that include some static methods*

Libraries with static methods used in this book

LIBRARIES OF METHODS IMPLEMENTED BY OURSELVES AND BY OTHERS in a modular programming environment can vastly expand the scope of our programming model. Beyond all of the libraries available in a standard Java release, thousands more are available on the web for applications of all sorts. To limit the scope of our programming model to a manageable size so that we can concentrate on algorithms, we use just the libraries listed in the table at right on this page, with a subset of their methods listed in *APIs*, as described next.

APIs A critical component of modular programming is *documentation* that explains the operation of library methods that are intended for use by others. We will consistently describe the library methods that we use in this book in *application programming interfaces (APIs)* that list the library name and the signatures and short descriptions of each of the methods that we use. We use the term *client* to refer to a program that calls a method in another library and the term *implementation* to describe the Java code that implements the methods in an API.

Example. The following example, the API for commonly used static methods from the standard `Math` library in `java.lang`, illustrates our conventions for APIs:

`public class Math`

`static double abs(double a)`	*absolute value of a*
`static double max(double a, double b)`	*maximum of a and b*
`static double min(double a, double b)`	*minimum of a and b*

Note 1: `abs()`, `max()`, *and* `min()` *are defined also for* `int`, `long`, *and* `float`.

`static double sin(double theta)`	*sine function*
`static double cos(double theta)`	*cosine function*
`static double tan(double theta)`	*tangent function*

Note 2: Angles are expressed in radians. Use `toDegrees()` *and* `toRadians()` *to convert.*
Note 3: Use `asin()`, `acos()`, *and* `atan()` *for inverse functions.*

`static double exp(double a)`	*exponential* (e^a)
`static double log(double a)`	*natural log* ($\log_e a$, *or* $\ln a$)
`static double pow(double a, double b)`	*raise a to the bth power* (a^b)
`static double random()`	*random number in* $[0, 1)$
`static double sqrt(double a)`	*square root of a*
`static double E`	*value of e (constant)*
`static double PI`	*value of* π *(constant)*

See booksite for other available functions.

API for Java's mathematics library (excerpts)

These methods implement mathematical functions—they use their arguments to compute a value of a specified type (except `random()`, which does not implement a mathematical function because it does not take an argument). Since they all operate on `double` values and compute a `double` result, you can consider them as extending the `double` data type—extensibility of this nature is one of the characteristic features of modern programming languages. 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 `Math` library also defines the precise constant values `PI` (for π) and `E` (for e), so that you can use those names to refer to those constants in your programs. For example, the value of `Math.sin(Math.PI/2)` is `1.0` and the value of `Math.log(Math.E)` is `1.0` (because `Math.sin()` takes its argument in radians and `Math.log()` implements the natural logarithm function).

Java libraries. Extensive online descriptions of thousands of libraries are part of every Java release, but we excerpt just a few methods that we use in the book, in order to clearly delineate our programming model. For example, `BinarySearch` uses the `sort()` method from Java's `Arrays` library, which we document as follows:

```
public class Arrays
------------------------------------------------------------------------
      static void  sort(int[] a)          put the array in increasing order
```

Note: This method is defined also for other primitive types and `Object`.

Excerpt from Java's Arrays library (`java.util.Arrays`)

The `Arrays` library is not in `java.lang`, so an `import` statement is needed to use it, as in `BinarySearch`. Actually, CHAPTER 2 of this book is devoted to implementations of `sort()` for arrays, including the mergesort and quicksort algorithms that are implemented in `Arrays.sort()`. Many of the fundamental algorithms that we consider in this book are implemented in Java and in many other programming environments. For example, `Arrays` also includes an implementation of binary search. To avoid confusion, we generally use our own implementations, although there is nothing wrong with using a finely tuned library implementation of an algorithm that you understand.

Our standard libraries. We have developed a number of libraries that provide useful functionality for introductory Java programming, for scientific applications, and for the development, study, and application of algorithms. Most of these libraries are for input and output; we also make use of the following two libraries to test and analyze our implementations. The first extends `Math.random()` to allow us to draw random values from various distributions; the second supports statistical calculations:

`public class StdRandom`

`static`	`void`	`initialize(long seed)`	*initialize*
`static`	`double`	`random()`	*real between* `0` *and* `1`
`static`	`int`	`uniform(int N)`	*integer between* `0` *and* `N-1`
`static`	`int`	`uniform(int lo, int hi)`	*integer between* `lo` *and* `hi-1`
`static`	`double`	`uniform(double lo, double hi)`	*real between* `lo` *and* `hi`
`static`	`boolean`	`bernoulli(double p)`	*true with probability* `p`
`static`	`double`	`gaussian()`	*normal, mean* `0`*, std dev* `1`
`static`	`double`	`gaussian(double m, double s)`	*normal, mean* `m`*, std dev* `s`
`static`	`int`	`discrete(double[] a)`	`i` *with probability* `a[i]`
`static`	`void`	`shuffle(double[] a)`	*randomly shuffle the array* `a[]`

Note: overloaded implementations of `shuffle()` *are included for other primitive types and for* `Object`.

API for our library of static methods for random numbers

`public class StdStats`

`static`	`double`	`max(double[] a)`	*largest value*
`static`	`double`	`min(double[] a)`	*smallest value*
`static`	`double`	`mean(double[] a)`	*average*
`static`	`double`	`var(double[] a)`	*sample variance*
`static`	`double`	`stddev(double[] a)`	*sample standard deviation*
`static`	`double`	`median(double[] a)`	*median*

API for our library of static methods for data analysis

The `initialize()` method in `StdRandom` allows us to *seed* the random number generator so that we can reproduce experiments involving random numbers. For reference, implementations of many of these methods are given on page 32. Some of these methods are extremely easy to implement; why do we bother including them in a library? Answers to this question are standard for well-designed libraries:

- They implement a level of abstraction that allow us to focus on implementing and testing the algorithms in the book, not generating random objects or calculating statistics. Client code that uses such methods is clearer and easier to understand than homegrown code that does the same calculation.
- Library implementations test for exceptional conditions, cover rarely encountered situations, and submit to extensive testing, so that we can count on them to operate as expected. Such implementations might involve a significant amount of code. For example, we often want implementations for various types of data. For example, Java's `Arrays` library includes multiple overloaded implementations of `sort()`, one for each type of data that you might need to sort.

These are bedrock considerations for modular programming in Java, but perhaps a bit overstated in this case. While the methods in both of these libraries are essentially self-documenting and many of them are not difficult to implement, some of them represent interesting algorithmic exercises. Accordingly, you are well-advised to *both* study the code in `StdRandom.java` and `StdStats.java` on the booksite *and* to take advantage of these tried-and-true implementations. The easiest way to use these libraries (and to examine the code) is to download the source code from the booksite and put them in your working directory; various system-dependent mechanisms for using them without making multiple copies are also described on the booksite.

Your own libraries. It is worthwhile to consider *every program that you write* as a library implementation, for possible reuse in the future.

- Write code for the client, a top-level implementation that breaks the computation up into manageable parts.
- Articulate an API for a library (or multiple APIs for multiple libraries) of static methods that can address each part.
- Develop an implementation of the API, with a `main()` that tests the methods independent of the client.

Not only does this approach provide you with valuable software that you can later reuse, but also taking advantage of modular programming in this way is a key to successfully addressing a complex programming task.

intended result	implementation

random `double`
value in `[a, b)`
	`public static double uniform(double a, double b)`
	`{  return a + StdRandom.random() * (b-a);  }`

random `int`
value in `[0..N)`
	`public static int uniform(int N)`
	`{  return (int) (StdRandom.random() * N);  }`

random `int`
value in `[lo..hi)`
	`public static int uniform(int lo, int hi)`
	`{  return lo + StdRandom.uniform(hi - lo);  }`

random `int` *value drawn*
from discrete distribution
(`i` *with probability* `a[i]`*)*
	`public static int discrete(double[] a)`
	`{  // Entries in a[] must sum to 1.`
	`     double r = StdRandom.random();`
	`     double sum = 0.0;`
	`     for (int i = 0; i < a.length; i++)`
	`     {`
	`        sum = sum + a[i];`
	`        if (sum >= r) return i;`
	`     }`
	`     return -1;`
	`}`

randomly shuffle the
elements in an array of
`double` *values*
(See Exercise 1.1.36)
	`public static void shuffle(double[] a)`
	`{`
	`   int N = a.length;`
	`   for (int i = 0; i < N; i++)`
	`   {  // Exchange a[i] with random element in a[i..N-1]`
	`      int r = i + StdRandom.uniform(N-i);`
	`      double temp = a[i];`
	`      a[i] = a[r];`
	`      a[r] = temp;`
	`   }`
	`}`

Implementations of static methods in StdRandom library

THE PURPOSE OF AN API is to *separate* the client from the implementation: the client should know nothing about the implementation other than information given in the API, and the implementation should not take properties of any particular client into account. APIs enable us to separately develop code for various purposes, then reuse it widely. No Java library can contain all the methods that we might need for a given computation, so this ability is a crucial step in addressing complex programming applications. Accordingly, programmers normally think of the API as a *contract* between the client and the implementation that is a clear specification of what each method is to do. Our goal when developing an implementation is to honor the terms of the contract. Often, there are many ways to do so, and separating client code from implementation code gives us the freedom to substitute new and improved implementations. In the study of algorithms, this ability is an important ingredient in our ability to understand the impact of algorithmic improvements that we develop.

Strings A String is a sequence of characters (char values). A literal String is a sequence of characters within double quotes, such as "Hello, World". The data type String is a Java data type but it is *not* a primitive type. We consider String now because it is a fundamental data type that almost every Java program uses.

Concatenation. Java has a built-in *concatenation* operator (+) for String like the built-in operators that it has for primitive types, justifying the addition of the row in the table below to the primitive-type table on page 12. The result of concatenating two String values is a single String value, the first string followed by the second.

type	set of values	typical literals	operators	typical expressions	
				expression	value
String	character sequences	"AB" "Hello" "2.5"	+ (concatenate)	"Hi, " + "Bob" "12" + "34" "1" + "+" + "2"	"Hi, Bob" "1234" "1+2"

Java's String data type

Conversion. Two primary uses of strings are to convert values that we can enter on a keyboard into data-type values and to convert data-type values to values that we can read on a display. Java has built-in operations for String to facilitate these operations. In particular, the language includes libraries Integer and Double that contain static methods to convert between String values and int values and between String values and double values, respectively.

```
public class Integer
--------------------------------------------------------------------------
     static    int  parseInt(String s)             convert s to an int value
     static String  toString(int i)                convert i to a String value

public class Double
--------------------------------------------------------------------------
     static double  parseDouble(String s)          convert s to a double value
     static String  toString(double x)             convert x to a String value
```

APIs for conversion between numbers and String values

Automatic conversion. We rarely explicitly use the static `toString()` methods just described because Java has a built-in mechanism that allows us to convert from any data type value to a `String` value by using concatenation: if *one* of the arguments of `+` is a `String`, Java *automatically* converts the other argument to a `String` (if it is not already a `String`). Beyond usage like `"The square root of 2.0 is " + Math.sqrt(2.0)` this mechanism enables conversion of any data-type value to a `String`, by concatenating it with the empty string `""`.

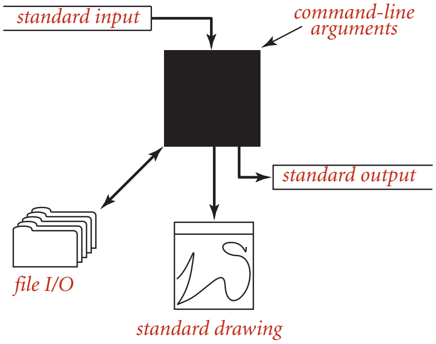
Command-line arguments. One important use of strings in Java programming is to enable a mechanism for passing information from the command line to the program. The mechanism is simple. When you type the `java` command followed by a library name followed by a sequence of strings, the Java system invokes the `main()` method in that library with an *array of strings* as argument: the strings typed after the library name. For example, the `main()` method in `BinarySearch` takes one command-line argument, so the system creates an array of size one. The program uses that value, `args[0]`, to name the file containing the whitelist, for use as the argument to `In.readInts()`. Another typical paradigm that we often use in our code is when a command-line argument is intended to represent a number, so we use `parseInt()` to convert to an `int` value or `parseDouble()` to convert to a `double` value.

COMPUTING WITH STRINGS is an essential component of modern computing. For the moment, we make use of `String` just to convert between external representation of numbers as sequences of characters and internal representation of numeric data-type values. In SECTION 1.2, we will see that Java supports many, many more operations on `String` values that we use throughout the book; in SECTION 1.4, we will examine the internal representation of `String` values; and in CHAPTER 5, we consider in depth algorithms that process `String` data. These algorithms are among the most interesting, intricate, and impactful methods that we consider in this book.

Input and output The primary purpose of our standard libraries for input, output, and drawing is to support a simple model for Java programs to interact with the outside world. These libraries are built upon extensive capabilities that are available in Java libraries, but are generally much more complicated and much more difficult to learn and use. We begin by briefly reviewing the model.

In our model, a Java program takes input values from *command-line arguments* or from an abstract stream of characters known as the *standard input stream* and writes to another abstract stream of characters known as the *standard output stream*.

Necessarily, we need to consider the interface between Java and the operating system, so we need to briefly discuss basic mechanisms that are provided by most modern operating systems and program-development environments. You can find more details about your particular system on the booksite. By default, command-line arguments, standard input, and standard output are associated with an application supported by either the operating system or the program development environment that takes commands. We use the generic term *terminal window* to refer to the window maintained by this application, where we type and read text. Since early Unix systems in the 1970s this model has proven to be a convenient and direct way for us to interact with our programs and data. We add to the classical model a *standard drawing* that allows us to create visual representations for data analysis.



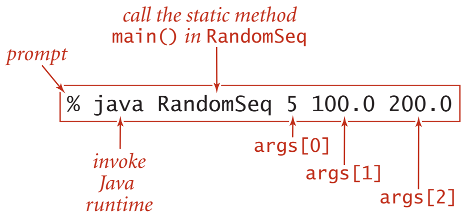
A bird's-eye view of a Java program

Commands and arguments. In the terminal window, we see a prompt, where we type *commands* to the operating system that may take *arguments*. We use only a few commands in this book, shown in the table below. Most often, we use the `.java` command, to run our programs. As mentioned on page 35, Java classes have a `main()` static 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. If we intend for an argument to be a number, we use a method such as `Integer.parseInt()` to convert it from `String` to the appropriate type.

command	arguments	purpose
`javac`	`.java` file name	compile Java program
`java`	`.class` file name (no extension) and command-line arguments	run Java program
`more`	any text file name	print file contents

Typical operating-system commands

Standard output. Our `StdOut` library provides support for standard output. By default, the system connects standard output to the terminal window. The `print()` method puts its argument on standard output; the `println()` method adds a newline; and the `printf()` method supports formatted output, as described next. Java provides a similar method in its `System.out` library; we use `StdOut` to treat standard input and standard output in a uniform manner (and to provide a few technical improvements).



Anatomy of a command

`public class StdOut`

`static void print(String s)`	*print* `s`
`static void println(String s)`	*print* `s`*, followed by newline*
`static void println()`	*print a new line*
`static void printf(String f, ... )`	*formatted print*

Note: overloaded implementations are included for primitive types and for `Object`.

API for our library of static methods for standard output

To use these methods, download into your working directory `StdOut.java` from the booksite and use code such as `StdOut.println("Hello, World");` to call them. A sample client is shown at right.

```
public class RandomSeq
{
   public static void main(String[] args)
   {  // Print N random values in (lo, hi).
      int N = Integer.parseInt(args[0]);
      double lo = Double.parseDouble(args[1]);
      double hi = Double.parseDouble(args[2]);
      for (int i = 0; i < N; i++)
      {
         double x = StdRandom.uniform(lo, hi);
         StdOut.printf("%.2f\n", x);
      }
   }
}
```

Sample StdOut client

Formatted output. In its simplest form, `printf()` takes two arguments. The first argument is a *format string* that describes how the second argument is to be converted to 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 value that specifies the *field width* of the

```
% java RandomSeq 5 100.0 200.0
123.43
153.13
144.38
155.18
104.02
```

converted value (the number of characters in the converted output string). By default, blank spaces are added on the left to make the length of the converted output equal to the field width; if we want the spaces on the right, we can insert a minus sign before the field width. (If the converted output string is bigger 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. 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()`. 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. You can also use the static method `String.format()` with arguments exactly as just described for `printf()` to get a formatted string without printing it. Formatted printing is a convenient mechanism that allows us to develop compact code that can produce tabulated experimental data (our primary use in this book).

type	code	typical literal	sample format strings	converted string values for output

`int`	`d`	`512`	`"%14d"`
			`"%-14d"`
				`"           512"`
				`"512           "`

`double`	`f`
	`e`
		`1595.1680010754388`	`"%14.2f"`
			`"%.7f"`
			`"%14.4e"`
				`"       1595.17"`
				`"1595.1680011"`
				`"    1.5952e+03"`

`String`	`s`	`"Hello, World"`	`"%14s"`
			`"%-14s"`
			`"%-14.5s"`
				`"  Hello, World"`
				`"Hello, World  "`
				`"Hello         "`

Format conventions for `printf()` (see the booksite for many other options)

Standard input. Our StdIn library takes data from the standard input stream that may be empty or may contain a sequence of values separated by whitespace (spaces, tabs, newline characters, and the like). By default, the system connects standard output to the terminal window—what you type is the input stream (terminated by <ctrl-d> or <ctrl-z>, depending on your terminal window application). 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. Within the input stream model, the static methods in this library are largely self-documenting (described by their signatures).

```
public class Average
{
   public static void main(String[] args)
   {  // Average the numbers on StdIn.
      double sum = 0.0;
      int cnt = 0;
      while (!StdIn.isEmpty())
      {  // Read a number and cumulate the sum.
         sum += StdIn.readDouble();
         cnt++;
      }
      double avg = sum / cnt;
      StdOut.printf("Average is %.5f\n", avg);
   }
}
```

Sample StdIn client

```
% java Average
1.23456
2.34567
3.45678
4.56789
<ctrl-d>
Average is 2.90123
```

`public class StdIn`

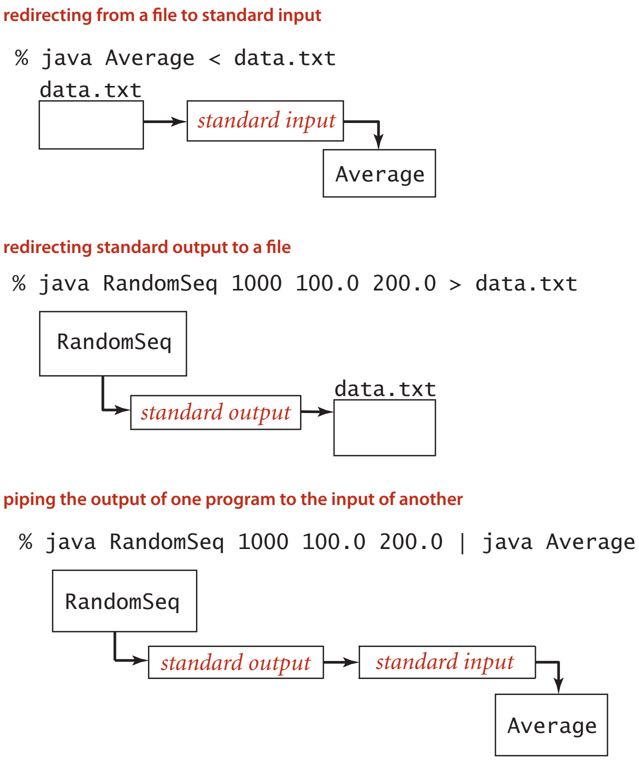
static	boolean	isEmpty()	true *if no more values,* false *otherwise*
static	int	readInt()	*read a value of type* int
static	double	readDouble()	*read a value of type* double
static	float	readFloat()	*read a value of type* float
static	long	readLong()	*read a value of type* long
static	boolean	readBoolean()	*read a value of type* boolean
static	char	readChar()	*read a value of type* char
static	byte	readByte()	*read a value of type* byte
static	String	readString()	*read a value of type* String
static	boolean	hasNextLine()	*is there another line in the input stream?*
static	String	readLine()	*read the rest of the line*
static	String	readAll()	*read the rest of the input stream*

API for our library of static methods for standard input

Redirection and piping. Standard input and output enable us to take advantage of command-line extensions supported by many operating-systems. 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:

```
% java RandomSeq 1000 100.0 200.0 > data.txt
```

This command 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 `StdOut.print()` or `StdOut.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. Not that we do not have to change `RandomSeq` in any way—it is using the standard output abstraction and is unaffected by our use of a different implementation of that abstraction. Similarly, we can redirect standard input so that `StdIn` reads data from a file instead of the terminal application:



redirecting from a file to standard input

```
% java Average < data.txt
```

redirecting standard output to a file

```
% java RandomSeq 1000 100.0 200.0 > data.txt
```

piping the output of one program to the input of another

```
% java RandomSeq 1000 100.0 200.0 | java Average
```

Redirection and piping from the command line

```
% 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. Combining these to redirect the output of one program to the input of another is known as *piping*:

```
% java RandomSeq 1000 100.0 200.0 | java Average
```

This command specifies that standard output for `RandomSeq` and standard input 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 difference is profound, because it removes the 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). When `RandomSeq` calls `StdOut.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.

Input and output from a file. Our `In` and `Out` libraries provide static methods that implement the abstraction of reading from and writing to a file the contents of an array of values of a primitive type (or `String`). We use `readInts()`, `readDoubles()`, and `readStrings()` in the `In` library and `writeInts()`, `writeDoubles()`, and `writeStrings()` in the `Out` library. The named argument can be a file or a web page. For example, this ability allows us to use a file and standard input for two different purposes in the same program, as in `BinarySearch`. The `In` and `Out` libraries also implement data types with instance methods that allow us the more general ability to treat multiple files as input and output streams, and web pages as input streams, so we will revisit them in SECTION 1.2.

`public class In`

`static int[]`	`readInts(String name)`	*read* `int` *values*
`static double[]`	`readDoubles(String name)`	*read* `double` *values*
`static String[]`	`readStrings(String name)`	*read* `String` *values*

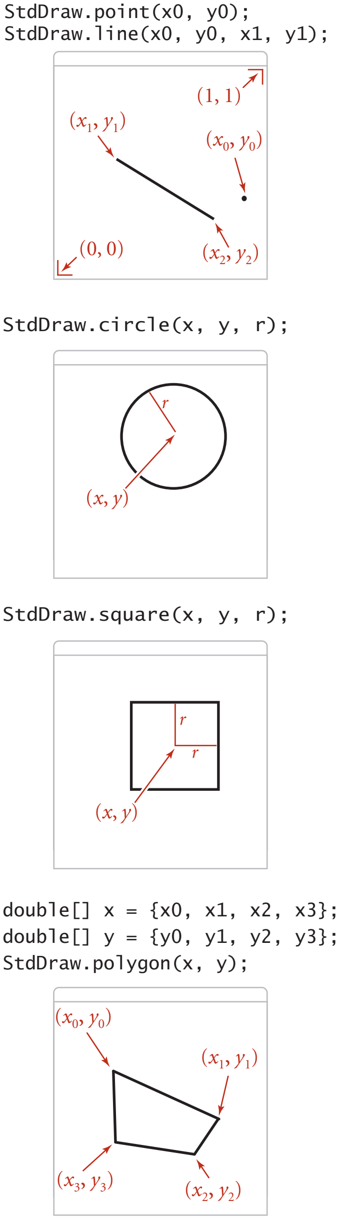
`public class Out`

`static void`	`write(int[] a, String name)`	*write* `int` *values*
`static void`	`write(double[] a, String name)`	*write* `double` *values*
`static void`	`write(String[] a, String name)`	*write* `String` *values*

Note 1: Other primitive types are supported.
Note 2: `StdIn` *and* `StdOut` *are supported (omit* `name` *argument).*

APIs for our static methods for reading and writing arrays

Standard drawing (basic methods). 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/output, our standard drawing abstraction is implemented in a library `StdDraw` that you can access by downloading the file `StdDraw.java` from the booksite into your working directory. Standard draw 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 to draw basic geometric shapes that our programs issue in the form of calls to static methods in `StdDraw`, including methods for drawing lines, points, text strings, circles, rectangles, and polygons. Like the methods for standard input and standard output, these methods are nearly self-documenting: `StdDraw.line()` draws a straight line segment connecting the 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, and so forth, as illustrated in the diagrams at right. Geometric shapes can be filled (in black, by default). The default scale is the unit square (all coordinates are 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.



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

```
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);
```

StdDraw examples

```
public class StdDraw

   static void  line(double x0, double y0, double x1, double y1)
   static void  point(double x, double y)
   static void  text(double x, double y, String s)
   static void  circle(double x, double y, double r)
   static void  filledCircle(double x, double y, double r)
   static void  ellipse(double x, double y, double rw, double rh)
   static void  filledEllipse(double x, double y, double rw, double rh)
   static void  square(double x, double y, double r)
   static void  filledSquare(double x, double y, double r)
   static void  rectangle(double x, double y, double rw, double rh)
   static void  filledRectangle(double x, double y, double rw, double rh)
   static void  polygon(double[] x, double[] y)
   static void  filledPolygon(double[] x, double[] y)
```

API for our library of static methods for standard drawing (drawing methods)

Standard drawing (control methods). The library also includes methods to change the scale and size of the canvas, the color and width of the lines, the text font, and the timing of drawing (for use in animation). As arguments for `setPenColor()` you can use one of the predefined colors `BLACK`, `BLUE`, `CYAN`, `DARK_GRAY`, `GRAY`, `GREEN`, `LIGHT_GRAY`, `MAGENTA`, `ORANGE`, `PINK`, `RED`, `BOOK_RED`, `WHITE`, and `YELLOW` that are defined as constants in `StdDraw` (so we refer to one of them with code like `StdDraw.RED`). The window also includes a menu option to save your drawing to a file, in a format suitable for publishing on the web.

`public class StdDraw`

`static void  setXscale(double x0, double x1)`	*reset x range to* (x_0, x_1)
`static void  setYscale(double y0, double y1)`	*reset y range to* (y_0, y_1)
`static void  setPenRadius(double r)`	*set pen radius to r*
`static void  setPenColor(Color c)`	*set pen color to c*
`static void  setFont(Font f)`	*set text font to f*
`static void  setCanvasSize(int w, int h)`	*set canvas to w-by-h window*
`static void  clear(Color c)`	*clear the canvas; color it c*
`static void  show(int dt)`	*show all; pause dt milliseconds*

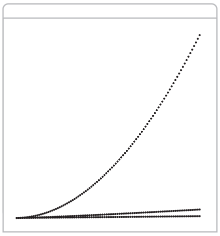
API for our library of static methods for standard drawing (control methods)

IN THIS BOOK, we use StdDraw for data analysis and for creating visual representations of algorithms in operation. The table at on the opposite page indicates some possibilities; we will consider many more examples in the text and the exercises throughout the book. The library also supports *animation*—of course, this topic is treated primarily on the booksite.

data **plot implementation (code fragment)** **result**

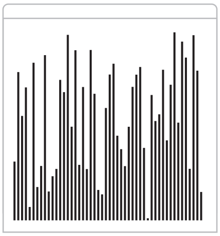
function values

```
int N = 100;
StdDraw.setXscale(0, N);
StdDraw.setYscale(0, N*N);
StdDraw.setPenRadius(.01);
for (int i = 1; i <= N; i++)
{
   StdDraw.point(i, i);
   StdDraw.point(i, i*i);
   StdDraw.point(i, i*Math.log(i));
}
```



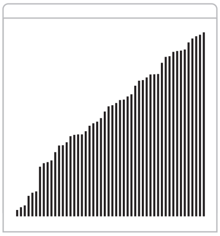
array of random values

```
int N = 50;
double[] a = new double[N];
for (int i = 0; i < N; i++)
   a[i] = StdRandom.random();
for (int i = 0; i < N; i++)
{
   double x = 1.0*i/N;
   double y = a[i]/2.0;
   double rw = 0.5/N;
   double rh = a[i]/2.0;
   StdDraw.filledRectangle(x, y, rw, rh);
}
```



sorted array of random values

```
int N = 50;
double[] a = new double[N];
for (int i = 0; i < N; i++)
   a[i] = StdRandom.random();
Arrays.sort(a);
for (int i = 0; i < N; i++)
{
   double x = 1.0*i/N;
   double y = a[i]/2.0;
   double rw = 0.5/N;
   double rh = a[i]/2.0;
   StdDraw.filledRectangle(x, y, rw, rh);
}
```

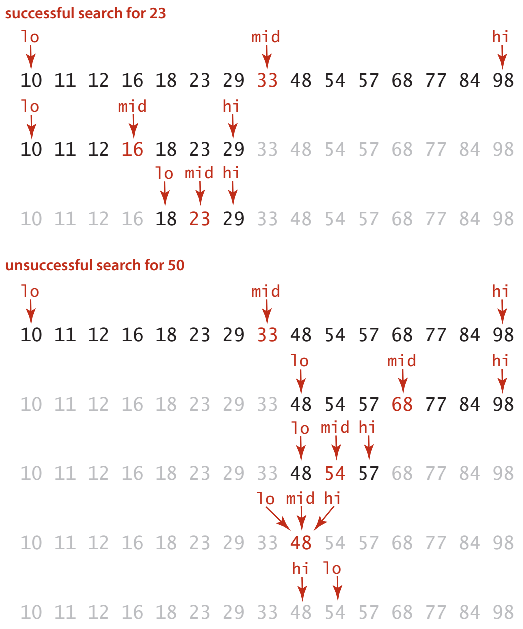


StdDraw plotting examples

Binary search

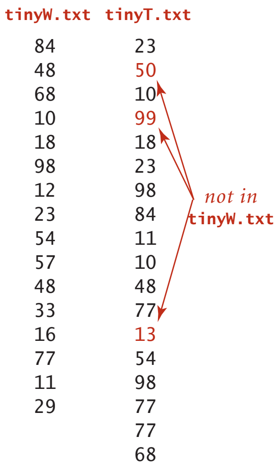
The sample Java program that we started with, shown on the facing page, is based on the famous, effective, and widely used *binary search* algorithm. This example is a prototype of the way in which we will examine new algorithms throughout the book. As with all of the programs we consider, it is both a precise definition of the method and a complete Java implementation that you can download from the booksite.

Binary search. We will study the binary search algorithm in detail in SECTION 3.2, but a brief description is appropriate here. The algorithm is implemented in the static method `rank()`, which takes an integer key and a *sorted* array of `int` values as arguments and returns the index of the key if it is present in the array, `-1` otherwise. It accomplishes this task by maintaining variables `lo` and `hi` such that the key is in `a[lo..hi]` if it is in the array, then entering into a loop that tests the middle entry in the interval (at index `mid`). If the key is equal to `a[mid]`, the return value is `mid`; otherwise the method cuts the interval size about in half, looking at the left half if the key is less than `a[mid]` and at the right half if the key is greater than `a[mid]`. The process terminates when the key is found or the interval is empty. Binary search is effective because it needs to examine just a few array entries (relative to the size of the array) to find the key (or determine that it is not there).



Binary search in an ordered array

Development client. For every algorithm implementation, we include a development client `main()` that you can use with sample input files provided in the book and on the booksite to learn about the algorithm and to test its performance. In this example, the client reads integers from the file named on the command line, then prints any integers on standard input that do not appear in the file. We use small test files such as those shown at right to demonstrate this behavior, and as the basis for traces and examples such as those at left above. We use large test files to model real-world applications and to test performance (see page 48).



tinyW.txt	tinyT.txt
84	23
48	50
68	10
10	99
18	18
98	23
12	98
23	84
54	11
57	10
48	48
33	77
16	13
77	54
11	98
29	77
	77
	68

Small test files for BinarySearch test client

Binary Search

```
import java.util.Arrays;

public class BinarySearch
{
   public static int rank(int key, int[] a)
   {  // Array must be sorted.
      int lo  = 0;
      int hi = a.length - 1;
      while (lo <= hi)
      {  // Key is in a[lo..hi] or not present.
         int mid = lo + (hi - lo) / 2;
         if      (key < a[mid]) hi = mid - 1;
         else if (key > a[mid]) lo = mid + 1;
         else                   return mid;
      }
      return -1;
   }

   public static void main(String[] args)
   {
      int[] whitelist = In.readInts(args[0]);

      Arrays.sort(whitelist);

      while (!StdIn.isEmpty())
      {  // Read key, print if not in whitelist.
         int key = StdIn.readInt();
         if (rank(key, whitelist) < 0)
            StdOut.println(key);
      }

   }
}
```

This program takes the name of a whitelist file (a sequence of integers) as argument and filters any entry that is on the whitelist from standard input, leaving only integers that are not on the whitelist on standard output. It uses the binary search algorithm, implemented in the static method `rank()`, to accomplish the task efficiently. See SECTION 3.1 for a full discussion of the binary search algorithm, its correctness, its performance analysis, and its applications.

```
% java BinarySearch tinyW.txt < tinyT.txt
50
99
13
```

Whitelisting. When possible, our development clients are intended to mirror practical situations and demonstrate the need for the algorithm at hand. In this case, the process is known as *whitelisting*. Specifically, imagine a credit card company that needs to check whether customer transactions are for a valid account. To do so, it can

- Keep customers account numbers in a file, which we refer to as a *whitelist*.
- Produce the account number associated with each transaction in the standard input stream.
- Use the test client to put onto standard output the numbers that are *not* associated with any customer. Presumably the company would refuse such transactions.

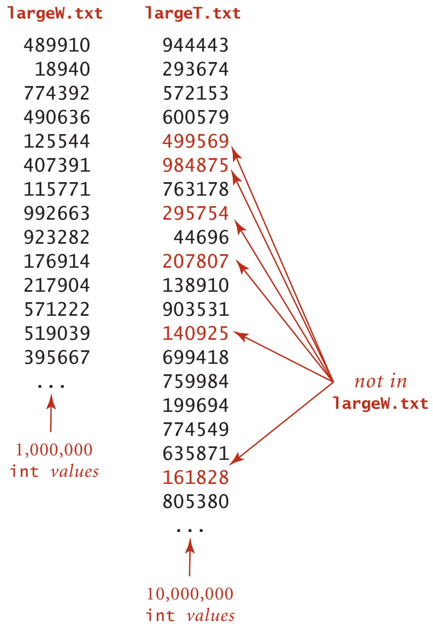
It would not be unusual for a big company with millions of customers to have to process millions of transactions or more. To model this situation, we provide on the booksite the files `largeW.txt` (1 million integers) and `largeT.txt` (10 million integers).

Performance. A working program is often not sufficient. For example, a much simpler implementation of `rank()`, which does not even require the array to be sorted, is to check every entry, as follows:

```
public static int rank(int key, int[] a)
{
   for (int i = 0; i < a.length; i++)
      if (a[i] == key) return i;
   return -1;
}
```

Given this simple and easy-to-understand solution, why do we use mergesort and binary search? If you work EXERCISE 1.1.38, you will see that your computer is too slow to run this brute-force implementation of `rank()` for large numbers of inputs (say, 1 million whitelist entries and 10 million transactions). *Solving the whitelist problem for a large number of inputs is not feasible without efficient algorithms such as binary search and mergesort.* Good performance is often of critical importance, so we lay the groundwork for studying performance in SECTION 1.4 and analyze the performance characteristics of all of our algorithms (including binary search, in SECTION 3.1 and mergesort, in SECTION 2.2).

IN THE PRESENT CONTEXT, our goal in thoroughly outlining our programming model is to ensure that you can run code like `BinarySearch` on your computer, use it on test data like ours, and modify it to adapt to various situations (such as those described in the exercises at the end of this section), in order to best understand its applicability. The programming model that we have sketched is designed to facilitate such activities, which are crucial to our approach to studying algorithms.



```
largeW.txt   largeT.txt

  489910       944443
   18940       293674
  774392       572153
  490636       600579
  125544       499569
  407391       984875
  115771       763178
  992663       295754
  923282        44696
  176914       207807
  217904       138910
  571222       903531
  519039       140925
  395667       699418
    ...        759984
               199694
               774549
               635871
               161828
               805380
                 ...
```

```
% java BinarySearch largeW.txt < largeT.txt
499569
984875
295754
207807
140925
161828
  ...
```

3,675,966
int values

Large files for BinarySearch test client

Perspective In this section, we have described a fine and complete programming model that served (and still serves) many programmers for many decades. Modern programming, however, goes one step further. This next level is called *data abstraction*, sometimes known as *object-oriented programming*, and is the subject of the next section. Simply put, the idea behind data abstraction is to allow a program to define *data types* (sets of values and sets of operations on those values), not just static methods that operate on predefined data types.

Object-oriented programming has come into widespread use in recent decades, and data abstraction is central to modern program development. We embrace data abstraction in this book for three primary reasons:

- It enables us to expand our ability to reuse code through modular programming. For example, our sorts in CHAPTER 2 and binary search and other algorithms in CHAPTER 3 allow clients to make use of the same code for any type of data (not just integers), including one defined by the client.
- It provides a convenient mechanism for building so-called *linked* data structures that provide more flexibility than arrays and are the basis of efficient algorithms in many settings.
- It enables us to precisely define the algorithmic challenges that we face. For example, our union-find algorithms in SECTION 1.5, our priority-queue algorithms in SECTION 2.4, and our symbol-table algorithms in CHAPTER 3 are all oriented toward defining data structures that enable efficient implementations of a *set* of operations. This challenge aligns perfectly with data abstraction.

Despite all of these considerations, our focus remains on the study of algorithms. In this context, we proceed to consider next the essential features of object-oriented programming that are relevant to our mission.

Q&A

Q. What is Java bytecode?

A. A low-level version of your program that runs on the Java *virtual machine*. This level of abstraction makes it easier for the developers of Java to ensure that our programs run on a broad variety of devices.

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

A. This issue is a contentious one among programmers. The short answer 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. We use the `int` type for small numbers (less than ten decimal digits), and the `long` type when values run into the billions or more.

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. 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. 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. What happens if I use a variable before initializing it to a value?

A. Java will report a compile-time error if there is any path through your code that would lead to use of an uninitialized variable.

Q. What are the values of `1/0` and `1.0/0.0` as Java expressions?

A. The first generates a runtime *exception* for division by zero (which stops your program because the value is undefined); the second has the value `Infinity`.

Q&A *(continued)*

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

A. No. Those operators are defined only for primitive types. See page 80.

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

A. The operators `&`, `|`, and `^` are *bitwise* logical operations for integer types that do *and*, *or*, and *exclusive or* (respectively) on each bit position. Thus the value of `10&6` is `14` and the value of `10^6` is `12`. We use these operators rarely (but occasionally) in this book. The operators `&&` and `||` are valid only in boolean expressions are included separately because of *short-circuiting*: an expression is evaluated left-to-right and the evaluation stops when the value is known.

Q. Is ambiguity in nested `if` statements a problem?

A. Yes. In Java, when you write

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

it is equivalent to

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

even if you might have been thinking

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

Using explicit braces is a good way to avoid this *dangling else* pitfall.

Q. What is the difference between a `for` loop and its `while` formulation?

A. The code in the `for` loop header is considered to be in the same block as the `for` loop body. 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.

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. If `a[]` is an array, why does `StdOut.println(a)` print out a hexadecimal integer, such as `@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. Why are we not using the standard Java libraries for input and graphics?

A. We *are* using them, but we prefer to work with simpler abstract models. The Java libraries behind `StdIn` and `StdDraw` 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` and `StdDraw.java`.

Q. Can my program reread data from standard input?

A. No. You only get one shot at it, in the same way that you cannot undo `println()`.

Q. What happens if my program attempts to read after standard input 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 this error message mean?

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

A. You probably forgot to put `StdIn.java` in your working directory.

Q. Can a static method take another static method as an argument in Java?

A. No. Good question, since many other languages do support this capability.

EXERCISES

1.1.1 Give the value of each of the following expressions:

a. `( 0 + 15 ) / 2`

b. `2.0e-6 * 100000000.1`

c. `true && false || true && true`

1.1.2 Give the type and value of each of the following expressions:

a. `(1 + 2.236)/2`

b. `1 + 2 + 3 + 4.0`

c. `4.1 >= 4`

d. `1 + 2 + "3"`

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

1.1.4 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;`

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

1.1.6 What does the following program print?

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

1.1.7 Give the value printed by each of the following code fragments:

a.

```
double t = 9.0;
while (Math.abs(t - 9.0/t) > .001)
   t = (9.0/t + t) / 2.0;
StdOut.printf("%.5f\n", t);
```

b.

```
int sum = 0;
for (int i = 1; i < 1000; i++)
    for (int j = 0; j < i; j++)
        sum++;
StdOut.println(sum);
```

c.

```
int sum = 0;
for (int i = 1; i < 1000; i *= 2)
   for (int j = 0; j < N; j++)
       sum++;
StdOut.println(sum);
```

1.1.8 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.

1.1.9 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. Here is a particularly concise solution:

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

EXERCISES *(continued)*

1.1.10 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.

1.1.11 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.

1.1.12 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(i);
```

1.1.13 Write a code fragment to print the *transposition* (rows and columns changed) of a two-dimensional array with *M* rows and *N* columns.

1.1.14 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`.

1.1.15 Write a static method `histogram()` that takes an array `a[]` of `int` values and an integer `M` as arguments 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`.

1.1.16 Give the value of `exR1(6)`:

```
public static String exR1(int n)
{
   if (n <= 0) return "";
   return exR1(n-3) + n + exR1(n-2) + n;
}
```

1.1.17 Criticize the following recursive function:

```
public static String exR2(int n)
{
   String s = exR2(n-3) + n + exR2(n-2) + n;
   if (n <= 0) return "";
   return s;
}
```

Answer: The base case will never be reached. A call to `exR2(3)` will result in calls to `exR2(0)`, `exR2(-3)`, `exR3(-6)`, and so forth until a `StackOverflowError` occurs.

1.1.18 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`.

1.1.19 Run the following program on your computer:

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

   public static void main(String[] args)
   {
      for (int N = 0; N < 100; N++)
         StdOut.println(N + " " + F(N));
   }
}
```

EXERCISES *(continued)*

What is the largest value of `N` for which this program takes less 1 hour to compute the value of `F(N)`? Develop a better implementation of `F(N)` that saves computed values in an array.

1.1.20 Write a recursive static method that computes the value of $\ln(N!)$

1.1.21 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.

1.1.22 Write a version of `BinarySearch` that uses the recursive `rank()` given on page 25 and *traces* the method calls. Each time the recursive method is called, print the argument values `lo` and `hi`, indented by the depth of the recursion. *Hint*: Add an argument to the recursive method that keeps track of the depth.

1.1.23 Add to the `BinarySearch` test client the ability to respond to a second argument: `+` to print numbers from standard input that *are not* in the whitelist, `-` to print numbers that *are* in the whitelist.

1.1.24 Give the sequence of values of p and q that are computed when Euclid's algorithm is used to compute the greatest common divisor of 105 and 24. Extend the code given on page 4 to develop a program `Euclid` that takes two integers from the command line and computes their greatest common divisor, printing out the two arguments for each call on the recursive method. Use your program to compute the greatest common divisor or 1111111 and 1234567.

1.1.25 Use mathematical induction to prove that Euclid's algorithm computes the greatest common divisor of any pair of nonnegative integers p and q.

CREATIVE PROBLEMS

1.1.26 *Sorting three numbers.* Suppose that the variables a, b, c, and t are all of the same numeric primitive type. Show 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; }
```

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

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

to compute binomial(100, 50). Develop a better implementation that is based on saving computed values in an array.

1.1.28 *Remove duplicates.* Modify the test client in BinarySearch to remove any duplicate keys in the whitelist after the sort.

1.1.29 *Equal keys.* Add to BinarySearch a static method rank() that takes a key and a sorted array of int values (some of which may be equal) as arguments and returns the number of elements that are smaller than the key and a similar method count() that returns the number of elements equal to the key. *Note*: If i and j are the values returned by rank(key, a) and count(key, a) respectively, then a[i..i+j-1] are the values in the array that are equal to key.

1.1.30 *Array exercise.* Write a code fragment that 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.

1.1.31 *Random connections.* 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 dots of size .05 on the circumference of a circle, and then, with probability p for each pair of points, draws a gray line connecting them.

CREATIVE PROBLEMS *(continued)*

1.1.32 *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.

1.1.33 *Matrix library.* Write a library `Matrix` that implements the following API:

`public class Matrix`

`static`	`double`	`dot(double[] x, double[] y)`	*vector dot product*
`static`	`double[][]`	`mult(double[][] a, double[][] b)`	*matrix-matrix product*
`static`	`double[][]`	`transpose(double[][] a)`	*transpose*
`static`	`double[]`	`mult(double[][] a, double[] x)`	*matrix-vector product*
`static`	`double[]`	`mult(double[] y, double[][] a)`	*vector-matrix product*

Develop a test client that reads values from standard input and tests all the methods.

1.1.34 *Filtering.* 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 and arrays of fixed size (not dependent on N)? 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 median of the numbers.
- Print the kth smallest value, for k less than 100.
- 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.

EXPERIMENTS

1.1.35 *Dice simulation.* The following code computes the exact probability distribution for the sum of two dice:

```
int SIDES = 6;
double[] dist = new double[2*SIDES+1];
for (int i = 1; i <= SIDES; i++)
   for (int j = 1; j <= SIDES; j++)
      dist[i+j] += 1.0;

for (int k = 2; k <= 2*SIDES; k++)
   dist[k] /= 36.0;
```

The value `dist[i]` 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?

1.1.36 *Empirical shuffle check.* Run computational experiments to check that our shuffling code on page 32 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*.

1.1.37 *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.

1.1.38 *Binary search versus brute-force search.* Write a program `BruteForceSearch` that uses the brute-force search method given on page 48 and compare its running time on your computer with that of `BinarySearch` for `largeW.txt` and `largeT.txt`.

EXPERIMENTS *(continued)*

1.1.39 *Random matches.* Write a `BinarySearch` client that takes an `int` value `T` as command-line argument and runs T trials of the following experiment for $N = 10^3$, 10^4, 10^5, and 10^6: generate two arrays of N randomly generated positive six-digit `int` values, and find the number of values that appear in both arrays. Print a table giving the average value of this quantity over the T trials for each value of N.

This page intentionally left blank

1.2 DATA ABSTRACTION

A DATA TYPE is a set of values and a set of operations on those values. So far, we have discussed in detail Java's *primitive* data types: for example, the *values* of the primitive data type `int` are integers between -2^{31} and $2^{31} - 1$; the *operations* of `int` include `+`, `*`, `-`, `/`, `%`, `<`, and `>`. In principle, we could write all of our programs using only the built-in primitive types, but it is much more convenient to write programs at a higher level of abstraction. In this section, we focus on the process of defining and using data types, which is known as *data abstraction* (and supplements the *function abstraction* style that is the basis of SECTION 1.1).

Programming in Java is largely based on building data types known as *reference types* with the familiar Java `class`. This style of programming is known as *object-oriented programming*, as it revolves around the concept of an *object*, an entity that holds a data type value. With Java's primitive types we are largely confined to programs that operate on numbers, but with reference types we can write programs that operate on strings, pictures, sounds, any of hundreds of other abstractions that are available in Java's standard libraries or on our booksite. Even more significant than libraries of predefined data types is that the range of data types available in Java programming is open-ended, because *you can define your own data types* to implement any abstraction whatsoever.

An *abstract data type* (ADT) is a data type whose representation is hidden from the client. Implementing an ADT as a Java class is not very different from implementing a function library as a set of static methods. The primary difference is that we associate *data* with the function implementations and we hide the representation of the data from the client. When *using* an ADT, we focus on the *operations* specified in the API and pay no attention to the data representation; when *implementing* an ADT, we focus on the *data*, then implement operations on that data.

Abstract data types are important because they support encapsulation in program design. In this book, we use them as a means to

- Precisely specify problems in the form of APIs for use by diverse clients
- Describe algorithms and data structures as API implementations

Our primary reason for studying different algorithms for the same task is that performance characteristics differ. Abstract data types are an appropriate framework for the study of algorithms because they allow us to put knowledge of algorithm performance to immediate use: we can substitute one algorithm for another to improve performance for all clients without changing any client code.

Using abstract data types *You do not need to know how a data type is implemented in order to be able to use it*, so we begin by describing how to write programs that use a simple data type named `Counter` whose values are a name and a nonnegative integer and whose operations are *create and initialize to zero*, *increment by one*, and *examine the current value.* This abstraction is useful in many contexts. For example, it would be reasonable to use such a data type in electronic voting software, to ensure that the only thing that a voter can do is increment a chosen candidate's tally by one. Or, we might use a `Counter` to keep track of fundamental operations when analyzing the performance of algorithms. To use a `Counter`, you need to learn our mechanism for specifying the operations defined in the data type and the Java language mechanisms for creating and manipulating data-type values. Such mechanisms are critically important in modern programming, and we use them throughout this book, so this first example is worthy of careful attention.

API for an abstract data type. To specify the behavior of an abstract data type, we use an *application programming interface* (API), which is a list of *constructors* and *instance methods* (operations), with an informal description of the effect of each, as in this API for `Counter`:

`public class Counter`

	`Counter(String id)`	*create a counter named* `id`
`void`	`increment()`	*increment the counter by one*
`int`	`tally()`	*number of increments since creation*
`String`	`toString()`	*string representation*

An API for a counter

Even though the basis of a data-type definition is a set of values, the role of the values is not visible from the API, only the operations on those values. Accordingly, an ADT definition has many similarities with a library of static methods (see page 24):

- Both are implemented as a Java `class`.
- Instance methods may take zero or more arguments of a specified type, separated by commas and enclosed in parentheses.
- They may provide a return value of a specified type or no return value (signified by `void`).

And there are three significant differences:

- Some entries in the API have the same name as the class and lack a return type. Such entries are known as *constructors* and play a special role. In this case, `Counter` has a constructor that takes a `String` argument.

- Instance methods lack the `static` modifier. They are *not* static methods—their purpose is to operate on data type values.
- Some instance methods are present so as to adhere to Java conventions—we refer to such methods as *inherited methods* and shade them gray in the API.

As with APIs for libraries of static methods, an API for an abstract data type is a contract with all clients and, therefore, the starting point both for developing any client code and for developing any data-type implementation. In this case, the API tells us that to use `Counter`, we have available the `Counter()` constructor, the `increment()` and `tally()` instance methods, and the inherited `toString()` method.

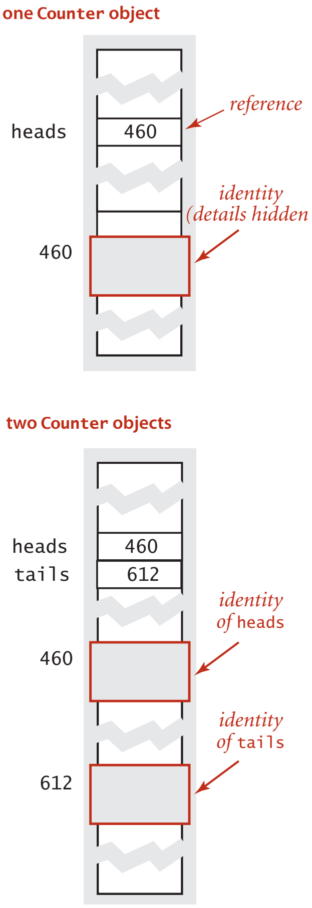
Inherited methods. Various Java conventions enable a data type to take advantage of built-in language mechanisms by including specific methods in the API. For example, all Java data types *inherit* a `toString()` method that returns a `String` representation of the data-type values. Java calls this method when any data-type value is to be concatenated with a `String` value with the + operator. The default implementation is not particularly useful (it gives a string representation of the memory address of the data-type value), so we often provide an implementation that overrides the default, and include `toString()` in the API whenever we do so. Other examples of such methods include `equals()`, `compareTo()`, and `hashCode()` (see page 101).

Client code. As with modular programming based on static methods, the API allows us to write client code without knowing details of the implementation (and to write implementation code without knowing details of any particular client). The mechanisms introduced on page 28 for organizing programs as independent modules are useful for all Java classes, and thus are effective for modular programming with ADTs as well as for libraries of static methods. Accordingly, we can use an ADT in any program provided that the source code is in a `.java` file in the same directory, or in the standard Java library, or accessible through an `import` statement, or through one of the classpath mechanisms described on the booksite. All of the benefits of modular programming follow. By encapsulating all the code that implements a data type within a single Java class, we enable the development of client code at a higher level of abstraction. To develop client code, you need to be able to *declare variables*, *create objects* to hold data-type values, and *provide access* to the values for instance methods to operate on them. These processes are different from the corresponding processes for primitive types, though you will notice many similarities.

Objects. Naturally, you can declare that a variable heads is to be associated with data of type Counter with the code

```
Counter heads;
```

but how can you assign values or specify operations? The answer to this question involves a fundamental concept in data abstraction: an *object* is an entity that can take on a data-type value. Objects are characterized by three essential properties: *state, identity,* and *behavior*. The *state* of an object is a value from its data type. The *identity* of an object distinguishes one object from another. It is useful to think of an object's identity as the place where its value is stored in memory. The *behavior* of an object is the effect of data-type operations. The implementation has the sole responsibility for maintaining an object's identity, so that client code can use a data type without regard to the representation of its state by conforming to an API that describes an object's behavior. An object's state might be used to provide information to a client or cause a side effect or be changed by one of its data type's operations, but the details of the representation of the data-type value are not relevant to client code. A *reference* is a mechanism for accessing an object. Java nomenclature makes clear the distinction from primitive types (where variables are associated with values) by using the term *reference types* for nonprimitive types. The details of implementing references vary in Java implementations, but it is useful to think of a reference as a memory address, as shown at right (for brevity, we use three-digit memory addresses in the diagram).

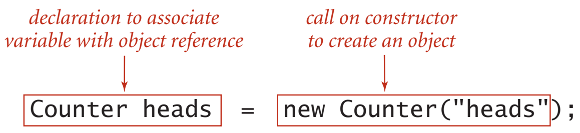


Object representation

Creating objects. Each data-type value is stored in an object. To create (or *instantiate*) an individual object, we invoke a constructor by using the keyword new, followed by the class name, followed by () (or a list of argument values enclosed in parentheses, if the constructor takes arguments). A constructor has no return type because it always returns a reference to an object of its data type. Each time that a client uses new(), the system

- Allocates memory space for the object
- Invokes the constructor to initialize its value
- Returns a reference to the object

In client code we typically create objects in an initializing declaration that associates a variable with the object, as we often do with variables of primitive types. Unlike primitive types, variables are associated with references to objects, not the data-type values



```
Counter heads = new Counter("heads");
```

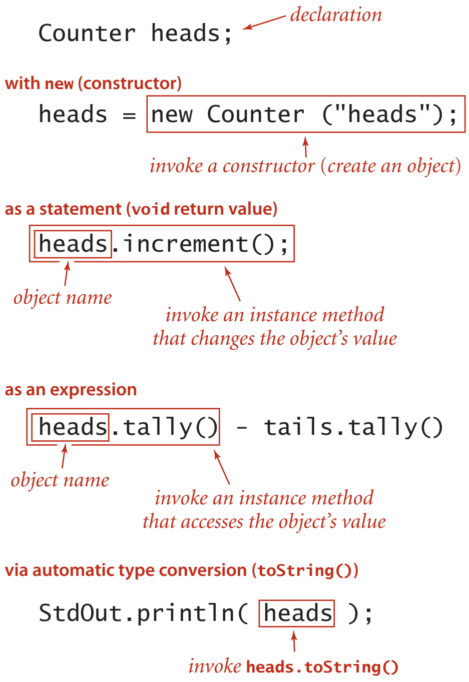
declaration to associate variable with object reference
call on constructor to create an object

Creating an object

themselves. We can create any number of objects from the same class—each object has its own identity and may or may not store the same value as another object of the same type. For example, the code

```
Counter heads = new Counter("heads");
Counter tails = new Counter("tails");
```

creates two different `Counter` objects. In an abstract data type, details of the representation of the value are hidden from client code. You might assume that the value associated with each `Counter` object is a `String` name and an `int` tally, but *you cannot write code that depends on any specific representation* (or even know whether that assumption is true—perhaps the tally is a `long` value).



```
Counter heads;
```

declaration
with new (constructor)

```
heads = new Counter ("heads");
```

invoke a constructor (create an object)
as a statement (void return value)

```
heads.increment();
```

object name
invoke an instance method that changes the object's value
as an expression

```
heads.tally() - tails.tally()
```

object name
invoke an instance method that accesses the object's value
via automatic type conversion (toString())

```
StdOut.println( heads );
```

Invoking instance methods

Invoking instance methods. The purpose of an instance method is to operate on data-type values, so the Java language includes a special mechanism to invoke instance methods that emphasizes a connection to an object. Specifically, we invoke an instance method by writing a variable name that refers to an object, followed by a period, followed by an instance method name, followed by 0 or more arguments, enclosed in parentheses and separated by commas. An instance method might *change* the data-type value or just *examine* the data-type value. Instance methods have all of the properties of static methods that we considered on page 24—arguments are passed by value, method names can be overloaded, they may have a return value, and they may cause side effects—but they have an additional property that characterizes them: *each invocation is associated with an object*. For example, the code

```
heads.increment();
```

invokes the instance method `increment()` to operate on the `Counter` object `heads` (in this case the operation involves incrementing the tally), and the code

```
heads.tally() - tails.tally();
```

invokes the instance method `tally()` twice, first to operate on the `Counter` object `heads` and then to operate on the `Counter` object `tails` (in this case the operation involves returning the tally as an `int` value). As these examples illustrate, you can use calls on instance methods in client code in the same way as you use calls on static methods—as statements (`void` methods) or values in expressions (methods that return a value). The primary purpose of static methods is to implement functions; the primary purpose of non-static (instance) methods is to implement data-type operations. Either type of method may appear in client code, but you can easily distinguish between them, because a static method call starts with a *class* name (uppercase, by convention) and a non-static method call always starts with an *object* name (lowercase, by convention). These differences are summarized in the table at right.

	instance method	static method
sample call	`head.increment()`	`Math.sqrt(2.0)`
invoked with	object name	class name
parameters	reference to object and argument(s)	argument(s)
primary purpose	examine or change object value	compute return value

Instance methods versus static methods

Using objects. Declarations give us variable names for objects that we can use in code not just to create objects and invoke instance methods, but also in the same way as we use variable names for integers, floating-point numbers, and other primitive types. To develop client code for a given data type, w:

- Declare variables of the type, for use in referring to objects
- Use the keyword `new` to invoke a constructor that creates objects of the type
- Use the object name to invoke instance methods, either as statements or within expressions

For example, the class `Flips` shown at the top of the next page is a `Counter` client that takes a command-line argument `T` and simulates `T` coin flips (it is also a `StdRandom` client). Beyond these direct uses, we can use variables associated with objects in the same way as we use variables associated with primitive-type values:

- In assignment statements
- To pass or return objects from methods
- To create and use arrays of object.

Understanding the behavior of each of these types of uses requires thinking in terms of *references*, not values, as you will see when we consider them, in turn.

Assignment statements. An assignment statement with a reference type creates a copy of the reference. The assignment statement does not create a new object, just another reference to an existing object. This situation is known as *aliasing*: both variables refer to the same object. The effect of aliasing is a bit unexpected, because it is different for variables holding values of a primitive type. Be sure that you understand the difference.

```
public class Flips
{
   public static void main(String[] args)
   {
      int T = Integer.parseInt(args[0]);
      Counter heads = new Counter("heads");
      Counter tails = new Counter("tails");
      for (int t = 0; t < T; t++)
         if (StdRandom.bernoulli(0.5))
              heads.increment();
         else tails.increment();
      StdOut.println(heads);
      StdOut.println(tails);
      int d = heads.tally() - tails.tally();
      StdOut.println("delta: " + Math.abs(d));
   }
}
```

```
% java Flips 10
5 heads
5 tails
delta: 0

% java Flips 10
8 heads
2 tails
delta: 6

% java Flips 1000000
499710 heads
500290 tails
delta: 580
```

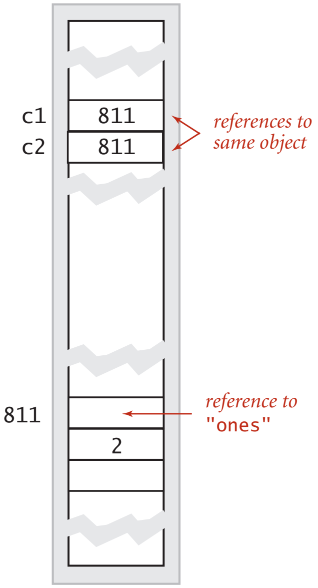
Counter client that simulates T coin flips

If x and y are variables of a primitive type, then the assignment x = y copies the value of y to x. For reference types, the *reference* is copied (not the value). Aliasing is a common source of bugs in Java programs, as illustrated by the following example:

```
Counter c1 = new Counter("ones");
c1.increment();
Counter c2 = c1;
c2.increment();
StdOut.println(c1);
```

With a typical toString() implementation this code would print the string "2 ones" which may or may not be what was intended and is counterintuitive at first. Such bugs are common in programs written by people without much experience in using objects (that may be you, so pay attention here!). Changing the state of an object impacts all code involving aliased variables referencing that object. We are used to thinking of two different variables of primitive types as being independent, but that intuition does not carry over to variables of reference types.

```
Counter c1;
c1 = new Counter("ones");
c1.increment();
Counter c2 = c1;
c2.increment();
```



Aliasing

Objects as arguments. You can pass objects as *arguments* to methods. This ability typically simplifies client code. For example, when we use a `Counter` as an argument, we are essentially passing both a name and a tally, but need only specify one variable. When we call a method with arguments, the effect in Java is as if each argument value were to appear on the right-hand side of an assignment statement with the corresponding argument name on the left. That is, Java passes a *copy* of the argument value from the calling program to the method. This arrangement is known as *pass by value* (see page 24). One important consequence is that the method cannot change the value of a caller's variable. For primitive types, this policy is what we expect (the two variables are independent), but each time that we use a reference type as a method argument we create an alias, so we must be cautious. In other words, the convention is to pass the *reference* by value (make a copy of it) but to pass the *object* by reference. For example, if we pass a reference to an object of type `Counter`, the method cannot change the original reference (make it point to a different `Counter`), but it *can* change the value of the object, for example by using the reference to call `increment()`.

Objects as return values. Naturally, you can also use an object as a *return value* from a method. The method might return an object passed to it as an argument, as in the example below, or it might create an object and return a reference to it. This capability is important because Java methods allow only one return value—using objects enables us to write code that, in effect, returns multiple values.

```
public class FlipsMax
{
   public static Counter max(Counter x, Counter y)
   {
      if (x.tally() > y.tally()) return x;
      else                       return y;
   }

   public static void main(String[] args)
   {
      int T = Integer.parseInt(args[0]);
      Counter heads = new Counter("heads");
      Counter tails = new Counter("tails");
      for (int t = 0; t < T; t++)
         if (StdRandom.bernoulli(0.5))
              heads.increment();
         else tails.increment();

      if (heads.tally() == tails.tally())
           StdOut.println("Tie");
      else StdOut.println(max(heads, tails) + " wins");
   }
}
```

```
% java FlipsMax 1000000
500281 tails wins
```

Example of a static method with object arguments and return values

Arrays are objects. In Java, every value of any nonprimitive type is an object. In particular, arrays are objects. As with strings, there is special language support for certain operations on arrays: declarations, initialization, and indexing. As with any other object, when we pass an array to a method or use an array variable on the right hand side of an assignment statement, we are making a copy of the array reference, not a copy of the array. This convention is appropriate for the typical case where we expect the method to be able to modify the array, by rearranging its entries, as, for example, in `java.util.Arrays.sort()` or the `shuffle()` method that we considered on page 32.

Arrays of objects. Array entries can be of any type, as we have already seen: `args[]` in our `main()` implementations is an array of `String` objects. When we create an array of objects, we do so in two steps:

- Create the array, using the bracket syntax for array constructors.
- Create each object in the array, using a standard constructor for each.

For example, the code below simulates rolling a die, using an array of `Counter` objects to keep track of the number of occurrences of each possible value. An array of objects in Java is an array of references to objects, not the objects themselves. If the objects are large, then we may gain efficiency by not having to move them around, just their references. If they are small, we may lose efficiency by having to follow a reference each time we need to get to some information.

```
public class Rolls
{
   public static void main(String[] args)
   {
      int T = Integer.parseInt(args[0]);
      int SIDES = 6;
      Counter[] rolls = new Counter[SIDES+1];
      for (int i = 1; i <= SIDES; i++)
         rolls[i] = new Counter(i + "'s");

      for (int t = 0; t < T; t++)
      {
         int result = StdRandom.uniform(1, SIDES+1);
         rolls[result].increment();
      }
      for (int i = 1; i <= SIDES; i++)
         StdOut.println(rolls[i]);
   }
}
```

```
% java Rolls 1000000
167308 1's
166540 2's
166087 3's
167051 4's
166422 5's
166592 6's
```

Counter client that simulates T rolls of a die

WITH THIS FOCUS ON OBJECTS, writing code that embraces data abstraction (defining and using data types, with data-type values held in objects) is widely referred to as *object-oriented programming*. The basic concepts that we have just covered are the starting point for object-oriented programming, so it is worthwhile to briefly summarize them. A *data type* is a set of values and a set of operations defined on those values. We implement data types in independent Java `class` modules and write client programs that use them. An *object* is an entity that can take on a data-type value or an *instance* of a data type. Objects are characterized by three essential properties: *state, identity,* and *behavior*. A data-type implementation supports clients of the data type as follows:

- Client code can *create objects* (establish identity) by using the `new` construct to invoke a constructor that creates an object, initializes its instance variables, and returns a reference to that object.
- Client code can *manipulate data-type values* (control an object's behavior, possibly changing its state) by using a variable associated with an object to invoke an instance method that operates on that object's instance variables.
- Client code can *manipulate objects* by creating arrays of objects and passing them and returning them to methods, in the same way as for primitive-type values, except that variables refer to references to values, not the values themselves.

These capabilities are the foundation of a flexible, modern, and widely useful programming style that we will use as the basis for studying algorithms in this book.

Examples of abstract data types The Java language has thousands of built-in ADTs, and we have defined many other ADTs to facilitate the study of algorithms. Indeed, every Java program that we write is a data-type implementation (or a library of static methods). To control complexity, we will specifically cite APIs for any ADT that we use in this book (not many, actually).

In this section, we introduce as examples several data types, with some examples of client code. In some cases, we present excerpts of APIs that may contain dozens of instance methods or more. We articulate these APIs to present real-world examples, to specify the instance methods that we will use in the book, and to emphasize that you do not need to know the details of an ADT implementation in order to be able to use it.

For reference, the data types that we use and develop in this book are shown on the facing page. These fall into several different categories:

- Standard system ADTs in `java.lang.*`, which can be used in any Java program.
- Java ADTs in libraries such as `java.awt`, `java.net`, and `java.io`, which can also be used in any Java program, but need an `import` statement.
- Our I/O ADTs that allow us to work with multiple input/output streams similar to `StdIn` and `StdOut`.
- Data-oriented ADTs whose primary purpose is to facilitate organizing and processing data by encapsulating the representation. We describe several examples for applications in computational geometry and information processing later in this section and use them as examples in client code later on.
- Collection ADTs whose primary purpose is to facilitate manipulation collections of data of the same. We describe the basic `Bag`, `Stack`, and `Queue` types in SECTION 1.3, `PQ` types in CHAPTER 2, and the `ST` and `SET` types in CHAPTERS 3 and 5.
- Opertions-oriented ADTs that we use to analyze algorithms, as described in SECTION 1.4 and SECTION 1.5.
- ADTs for graph algorithms, including both data-oriented ADTs that focus on encapsulating representations of various kinds of graphs and operations-oriented ADTs that focus on providing specifications for graph-processing algorithms.

This list does not include the dozens of types that we consider in exercises, which may be found in the index. Also, as described on page 90, we often distinguish multiple implementations of various ADTs with a descriptive prefix. As a group, the ADTs that we use demonstrate that organizing and understanding the data types that you use is an important factor in modern programming.

A typical application might use only five to ten of these ADTs. A prime goal in the development and organization of the ADTs in this book is to enable programmers to easily take advantage of a relatively small set of them in developing client code.

standard Java system types in java.lang	
Integer	int *wrapper*
Double	double *wrapper*
String	*indexed* chars
StringBuilder	*builder for strings*

other Java types	
java.awt.Color	*colors*
java.awt.Font	*fonts*
java.net.URL	*URLs*
java.io.File	*files*

our standard I/O types	
In	*input stream*
Out	*output stream*
Draw	*drawing*

data-oriented types for client examples	
Point2D	*point in the plane*
Interval1D	*1D interval*
Interval2D	*2D interval*
Date	*date*
Transaction	*transaction*

types for the analysis of algorithms	
Counter	*counter*
Accumulator	*accumulator*
VisualAccumulator	*visual version*
Stopwatch	*stopwatch*

collection types	
Stack	*pushdown stack*
Queue	*FIFO queue*
Bag	*bag*
MinPQ MaxPQ	*priority queue*
IndexMinPQ IndexMinPQ	*priority queue (indexed)*
ST	*symbol table*
SET	*set*
StringST	*symbol table (string keys)*

data-oriented graph types	
Graph	*graph*
Digraph	*directed graph*
Edge	*edge (weighted)*
EdgeWeightedGraph	*graph (weighted)*
DirectedEdge	*edge (directed, weighted)*
EdgeWeightedDigraph	*graph (directed, weighted)*

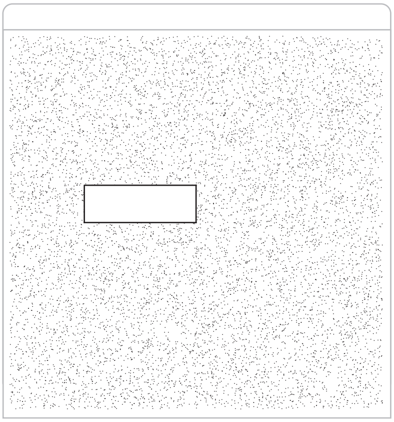
operations-oriented graph types	
UF	*dynamic connectivity*
DepthFirstPaths	*DFS path searcher*
CC	*connected components*
BreadthFirstPaths	*BFS path search*
DirectedDFS	*DFS digraph path search*
DirectedBFS	*BFS digraph path search*
TransitiveClosure	*all paths*
Topological	*topological order*
DepthFirstOrder	*DFS order*
DirectedCycle	*cycle search*
SCC	*strong components*
MST	*minimum spanning tree*
SP	*shortest paths*

Selected ADTs used in this book

Geometric objects. A natural example of object-oriented programming is designing data types for geometric objects. For example, the APIs on the facing page define abstract data types for three familiar geometric objects: `Point2D` (points in the plane), `Interval1D` (intervals on the line), and `Interval2D` (two-dimensional intervals in the plane, or axis-aligned rectangles). As usual, the APIs are essentially self-documenting and lead immediately to easily understood client code such as the example at left, which reads the boundaries of an `Interval2D` and an integer T from the command line, generates T random points in the unit square, and counts the number of points that fall in the interval (an estimate of the area of the rectangle). For dramatic effect, the client also draws the interval and the points that fall outside the interval. This computation is a model for a method that reduces the problem of computing the area and volume of geometric shapes to the problem of determining whether a point falls within the shape or not (a less difficult but not trivial problem). Of course, we can define APIs for other geometric objects such as line segments, triangles, polygons, circles, and so forth, though implementing operations on them can be challenging. Several examples are addressed in the exercises at the end of this section.

```
public static void main(String[] args)
{
   double xlo = Double.parseDouble(args[0]);
   double xhi = Double.parseDouble(args[1]);
   double ylo = Double.parseDouble(args[2]);
   double yhi = Double.parseDouble(args[3]);
   int T = Integer.parseInt(args[4]);

   Interval1D x = new Interval1D(xlo, xhi);
   Interval1D y = new Interval1D(ylo, yhi);
   Interval2D box = new Interval2D(x, y);
   box.draw();

   Counter c = new Counter(“hits”);
   for (int t = 0; t < T; t++)
   {
      double x = Math.random();
      double y = Math.random();
      Point p = new Point(x, y);
      if (box.contains(p)) c.increment();
      else                 p.draw();
   }

   StdOut.println(c);
   StdOut.println(box.area());
}
```

Interval2D test client



```
% java Interval2D .2 .5 .5 .6 10000
297 hits
.03
```

PROGRAMS THAT PROCESS GEOMETRIC OBJECTS have wide application in computing with models of the natural world, in scientific computing, video games, movies, and many other applications. The development and study of such programs and applications has blossomed into a far-reaching field of study known as *computational geometry*, which is a `public class Point2D`

	`Point2D(double x, double y)`	*create a point*
`double`	`x()`	*x coordinate*
`double`	`y()`	*y coordinate*
`double`	`r()`	*radius (polar coordinates)*
`double`	`theta()`	*angle (polar coordinates)*
`double`	`distTo(Point2D that)`	*Euclidean distance from this point to* `that`
`void`	`draw()`	*draw the point on* `StdDraw`

An API for points in the plane

`public class Interval1D`

	`Interval1D(double lo, double hi)`	*create an interval*
`double`	`length()`	*length of the interval*
`boolean`	`contains(double x)`	*does the interval contain* `x`*?*
`boolean`	`intersects(Interval1D that)`	*does the interval intersect* `that`*?*
`void`	`draw()`	*draw the interval on* `StdDraw`

An API for intervals on the line

`public class Interval2D`

	`Interval2D(Interval1D x, Interval1D y)`	*create a 2D interval*
`double`	`area()`	*area of the 2D interval*
`boolean`	`contains(Point p)`	*does the 2D interval contain* `p`*?*
`boolean`	`intersects(Interval2D that)`	*does the 2D interval intersect* `that`*?*
`void`	`draw()`	*draw the 2D interval on* `StdDraw`

An API for two dimensional intervals in the plane

fertile area of examples for the application of the algorithms that we address in this book, as you will see in examples throughout the book. In the present context, our interest is to suggest that abstract data types that directly represent geometric abstractions are not difficult to define and can lead to simple and clear client code. This idea is reinforced in several exercises at the end of this section and on the booksite.

Information processing Whether it be a bank processing millions of credit card transactions or a web analytics company processing billions of touchpad taps or a scientific research group processing millions of experimental observations, a great many applications are centered around processing and organizing information. Abstract data types provide a natural mechanism for organizing the information. Without getting into details, the two APIs on the facing page suggest a typical approach for a commercial application. The idea is to define data types that allow us to keep information in objects that correspond to things in the real world. A date is a day, a month, and a year and a transaction is a customer, a date, and an amount. These two are just examples: we might also define data types that can hold detailed information for customers, times, locations, goods and services, or whatever. Each data type consists of constructors that create objects containing the data and methods for use by client code to access it. To simplify client code, we provide two constructors for each type, one that presents the data in its appropriate type and another that parses a string to get the data (see EXERCISE 1.2.19 for details). As usual, there is no reason for client code to know the representation of the data. Most often, the reason to organize the data in this way is to treat the data associated with an object as a single entity: we can maintain arrays of `Transaction` values, use `Date` values as a argument or a return value for a method, and so forth. The focus of such data types is on encapsulating the data, while at the same time enabling the development of client code that does not depend on the representation of the data. We do not dwell on organizing information in this way, except to take note that doing so and including the inherited methods `toString()`, `compareTo()`, `equals()`, and `hashCode()` allows us to take advantage of algorithm implementations that can process *any type of data*. We will discuss inherited methods in more detail on page 100. For example, we have already noted Java's convention that enables clients to print a string representation of every value if we include `toString()` implementation in a data type. We consider conventions corresponding to the other inherited methods in SECTION 1.3, SECTION 2.5, SECTION 3.4, and SECTION 3.5, using `Date` and `Transaction` as examples. SECTION 1.3 gives classic examples of data types and a Java language mechanism known as *parameterized types,* or *generics*, that takes advantage of these conventions, and CHAPTER 2 and CHAPTER 3 are also devoted to taking advantage of generic types and inherited methods to develop implementations of sorting and searching algorithms that are effective for any type of data.

WHENEVER YOU HAVE DATA OF DIFFERENT TYPES that logically belong together, it is worthwhile to contemplate defining an ADT as in these examples. The ability to do so helps to organize the data, can greatly simplify client code in typical applications, and is an important step on the road to data abstraction.

`public class Date implements Comparable<Date>`

	`Date(int month, int day, int year)`	*create a date*
	`Date(String date)`	*create a date (parse constructor)*
`int`	`month()`	*month*
`int`	`day()`	*day*
`int`	`year()`	*year*
`String`	`toString()`	*string representation*
`boolean`	`equals(Object that)`	*is this the same date as* `that`*?*
`int`	`compareTo(Date that)`	*compare this date to* `that`
`int`	`hashCode()`	*hash code*

`public class Transaction implements Comparable<Transaction>`

	`Transaction(String who, Date when, double amount)`	
	`Transaction(String transaction)`	*create a transaction (parse constructor)*
`String`	`who()`	*customer name*
`Date`	`when()`	*date*
`double`	`amount()`	*amount*
`String`	`toString()`	*string representation*
`boolean`	`equals(Object that)`	*is this the same transaction as* `that`*?*
`int`	`compareTo(Transaction that)`	*compare this transaction to* `that`
`int`	`hashCode()`	*hash code*

Sample APIs for commercial applications (dates and transactions)

Strings. Java's String is an important and useful ADT. A String is an indexed sequence of char values. String has dozens of instance methods, including the following:

public class String

	String()	*create an empty string*
int	length()	*length of the string*
int	charAt(int i)	i*th character*
int	indexOf(String p)	*first occurrence of* p (-1 *if none*)
int	indexOf(String p, int i)	*first occurrence of* p *after* i (-1 *if none*)
String	concat(String t)	*this string with* t *appended*
String	substring(int i, int j)	*substring of this string (*i*th to* j-1*st chars)*
String[]	split(String delim)	*strings between occurrences of* delim
int	compareTo(String t)	*string comparison*
boolean	equals(String t)	*is this string's value the same as* t*'s ?*
int	hashCode()	*hash code*

Java String API (partial list of methods)

String values are similar to arrays of characters, but the two are not the same. Arrays have built-in Java language syntax for accessing a character; String has instance methods for indexed access, length, and many other operations. On the other hand, String has special language support for initialization and concatenation: instead of creating and initializing a string with a constructor, we can use a string literal; instead of invoking the method concat() we can use the + operator. We do not need to consider the details of the implementation, though understanding performance characteristics of some of the methods is important when developing string-processing algorithms, as you will see in Chapter 5. Why not just use arrays of characters instead of String values? The answer to this question is the same as for any ADT: *to simplify and clarify client code*. With String, we can write clear and simple client code that uses numerous convenient instance methods without regard to the way in which strings are represented (see facing page). Even this short list contains powerful operations that require advanced algorithms such

```
String a = "now is ";
String b = "the time ";
String c = "to"
```

call	*value*
a.length()	7
a.charAt(4)	i
a.concat(c)	"now is to"
a.indexOf("is")	4
a.substring(2, 5)	"w i"
a.split(" ")[0]	"now"
a.split(" ")[1]	"is"
b.equals(c)	false

Examples of string operations

task **implementation**

is the string a palindrome?

```
public static boolean isPalindrome(String s)
{
   int N = s.length();
   for (int i = 0; i < N/2; i++)
      if (s.charAt(i) != s.charAt(N-1-i))
         return false;
   return true;
}
```

extract file name and extension from a command-line argument

```
String s = args[0];
int dot = s.rank(".");
String base      = s.substring(0, dot);
String extension = s.substring(dot + 1, s.length());
```

print all lines in standard input that contain a string specified on the command line

```
String query = args[0];
while (!StdIn.isEmpty())
{
   String s = StdIn.readLine();
   if (s.contains(query)) StdOut.println(s);
}
```

create an array of the strings on `StdIn` *delimited by whitespace*

```
String input = StdIn.readAll();
String[] words = input.split("\\s+");
```

check whether an array of strings is in alphabetical order

```
public boolean isSorted(String[] a)
{
   for (int i = 1; i < a.length; i++)
   {
      if (a[i-1].compareTo(a[i]) > 0)
         return false;
   }
   return true;
}
```

Typical string-processing code

as those considered in CHAPTER 5. For example, the argument of `split()` can be a *regular expression* (see SECTION 5.4)—the `split()` example on page 81 uses the argument "\\s+", which means "one or more tabs, spaces, newlines, or returns."

Input and output revisited. A disadvantage of the `StdIn`, `StdOut`, and `StdDraw` standard libraries of SECTION 1.1 is that they restrict us to working with just one input file, one output file, and one drawing for any given program. With object-oriented programming, we can define similar mechanisms that allow us to work with *multiple* input streams, output streams, and drawings within one program. Specifically, our standard libary includes the data types `In`, `Out`, and `Draw` with the APIs shown on the facing page, When invoked with a constructor having a `String` argument, `In` and `Out` will first try to find a file in the current directory of your computer that has that name. If it cannot do so, it will assume the argument to be a website name and will try to connect to that website (if no such website exists, it will issue a runtime exception). In either case, the specified file or website becomes the source/target of the input/output for the stream object thus created, and the `read*()` and `print*()` methods will refer to that file or website. (If you use the no-argument constructor, then you obtain the standard streams.) This arrangement makes it possible for a single program to process multiple files and drawings. You also can assign such objects to variables, pass them as arguments or return values from methods, create arrays of them, and manipulate them just as you manipulate objects of any type. The program `Cat` shown at left is a sample client of `In` and `Out` that uses multiple input streams to concatenate several input files into a single output file. The `In` and `Out` classes also contain static methods for reading files containing values that are all `int`, `double`, or `String` types into an array (see page 126 and EXERCISE 1.2.15).

```
public class Cat
{
   public static void main(String[] args)
   {  // Copy input files to out (last argument).
      Out out = new Out(args[args.length-1]);
      for (int i = 0; i < args.length - 1; i++)
      {  // Copy input file named on ith arg to out.
         In in = new In(args[i]);
         String s = in.readAll();
         out.println(s);
         in.close();
      }
      out.close();
   }
}
```

A sample `In` and `Out` client

```
% more in1.txt
This is

% more in2.txt
a tiny
test.

% java Cat in1.txt in2.txt out.txt

% more out.txt
This is
a tiny
test.
```

`public class In`

	`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`
	`...`	
`void`	`close()`	*close the input stream*

Note: all operations supported by `StdIn` *are also supported for* `In` *objects.*

API for our data type for input streams

`public class Out`

	`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*
`void`	`println()`	*append a newline to the output stream*
`void`	`printf(String f, ...)`	*formatted print to the output steam*
`void`	`close()`	*close the output stream*

Note: all operations supported by `StdOut` *are also supported for* `Out` *objects.*

API for our data type for output streams

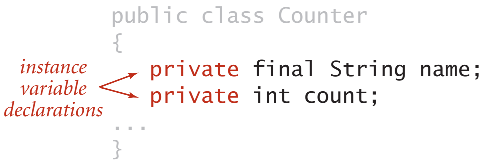
`public class Draw`

	`Draw()`
`void`	`line(double x0, double y0, double x1, double y1)`
`void`	`point(double x, double y)`
	`...`

Note: all operations supported by `StdDraw` *are also supported for* `Draw` *objects.*

API for our data type for drawings

Implementing an abstract data type. As with libraries of static methods, we implement ADTs with a Java `class`, putting the code in a file with the same name as the class, followed by the `.java` extension. The first statements in the file declare *instance variables* that define the data-type values. Following the instance variables are the *constructor* and the *instance methods* that implement operations on data-type values. Instance methods may be *public* (specified in the API) or *private* (used to organize the computation and not available to clients). A data-type definition may have multiple constructors and may also include definitions of static methods. In particular, a unit-test client `main()` is normally useful for testing and debugging. As a first example, we consider an implementation of the `Counter` ADT that we defined on page 65. A full annotated implementation is shown on the facing page, for reference as we discuss its constituent parts. Every ADT implementation that you will develop has the same basic ingredients as this simple example.

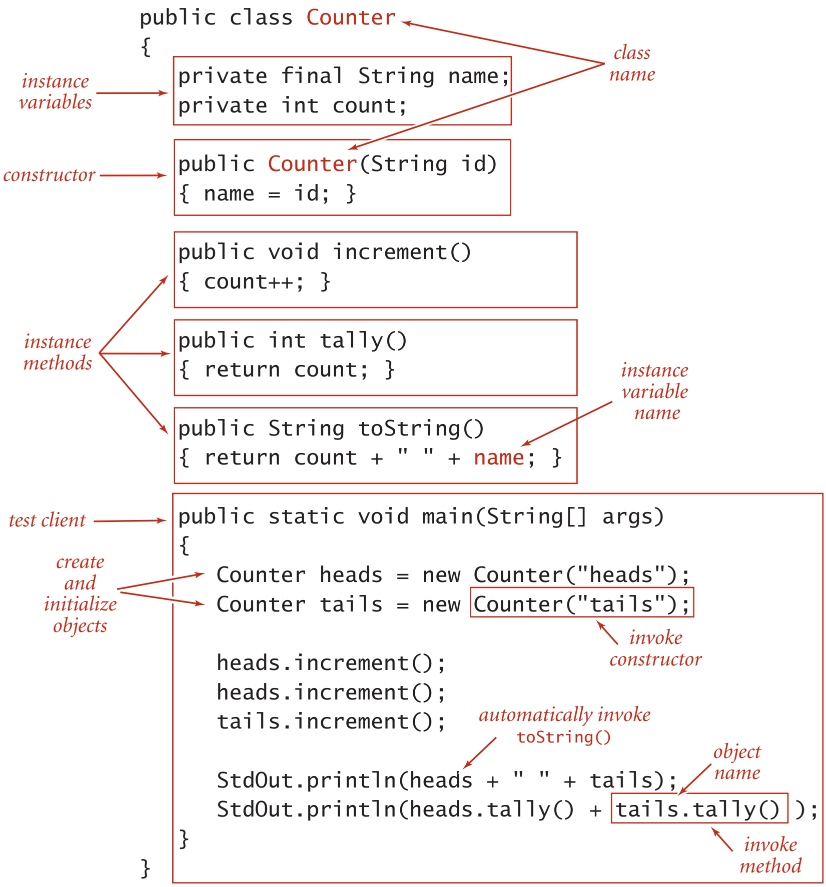


```
public class Counter
{
   private final String name;
   private int count;
   ...
}
```

Instance variables in ADTs are `private`

Instance variables. To define data-type values (the *state* of each object), we declare *instance variables* in much the same way as we declare local variables. There is a critical distinction between instance variables and the local variables within a static method or a block that you are accustomed to: there is just *one* value corresponding to each local variable at a given time, but there are *numerous* values corresponding to each instance variable (one for each object that is an instance of the data type). There is no ambiguity with this arrangement, because each time that we access an instance variable, we do so with an object name—that object is the one whose value we are accessing. Also, each declaration is qualified by a *visibility modifier*. In ADT implementations, we use `private`, using a Java language mechansim to enforce the idea that the representation of an ADT is to be hidden from the client, and also `final`, if the value is not to be changed once it is initialized. `Counter` has two instance variables: a `String` value `name` and an `int` value `count`. If we were to use `public` instance variables (allowed in Java) the data type would, by definition, not be abstract, so we do not do so.

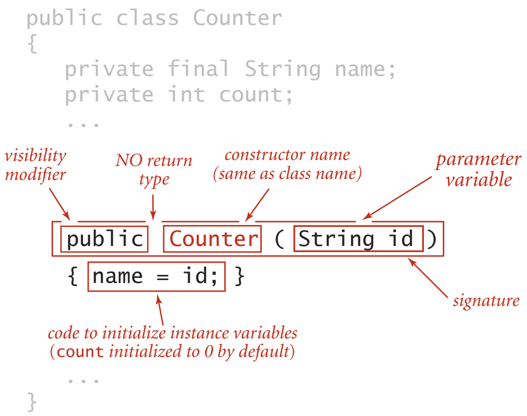
Constructors. Every Java class has at least one *constructor* that establishes an object's *identity*. A constructor is like a static method, but it can refer directly to instance variables and has no return value. Generally, the purpose of a constructor is to initialize the instance variables. Every constructor creates an object and provides to the client a reference to that object. Constructors always share the same name as the class. We can overload the name and have multiple constructors with different signatures, just as with methods. If no other constructor is defined, a default no-argument constructor is



```
public class Counter
{
   private final String name;
   private int count;

   public Counter(String id)
   { name = id; }

   public void increment()
   { count++; }

   public int tally()
   { return count; }

   public String toString()
   { return count + " " + name; }

   public static void main(String[] args)
   {
      Counter heads = new Counter("heads");
      Counter tails = new Counter("tails");

      heads.increment();
      heads.increment();
      tails.increment();

      StdOut.println(heads + " " + tails);
      StdOut.println(heads.tally() + tails.tally() );
   }
}
```

Anatomy of a class that defines a data type

implicit, has no arguments, and initializes instance values to default values. The default values of instance variables are `0` for primitive numeric types, `false` for `boolean`, and `null` for reference types. These defaults may be changed by using initializing declarations for instance variables. Java automatically invokes a constructor when a client program uses the keyword `new`. Overloaded constructors are typically used to initialize instance variables to client-supplied values other than the defaults. For example, `Counter` has a one-argument constructor that initializes the `name` instance variable to the value given as argument (leaving the `count` instance variable to be initialized to the default value `0`).

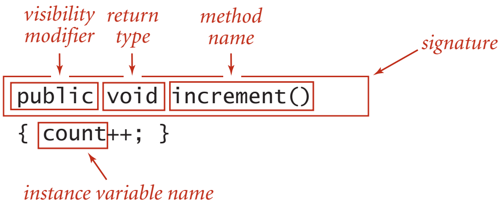


```
public class Counter
{
   private final String name;
   private int count;
   ...
   public Counter ( String id )
   { name = id; }
   ...
}
```

visibility modifier
NO return type
constructor name (same as class name)
parameter variable
signature
code to initialize instance variables (count initialized to 0 by default)

Anatomy of a constructor

Instance methods. To implement data-type instance methods (the *behavior* of each object), we implement *instance methods* with code that is precisely like the code that you learned in SECTION 1.1 to implement static methods (functions). Each instance method has a return type, a *signature* (which specifies its name 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 (if any) are initialized with client values, the statements 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 access and perform operations on instance variables.* How do we specify which object's instance variables we want to use? If you think about this question for a moment, you will see the logical answer: a reference to a variable in an instance method refers to the value *for the object that was used to invoke the method.* When we say `heads.increment()` the code in `increment()` is referring to the instance variables for `heads`. In other words,



```
public void increment()
{ count++; }
```

Anatomy of an instance method

object-oriented programming adds one critically important additional way to use variables in a Java program:

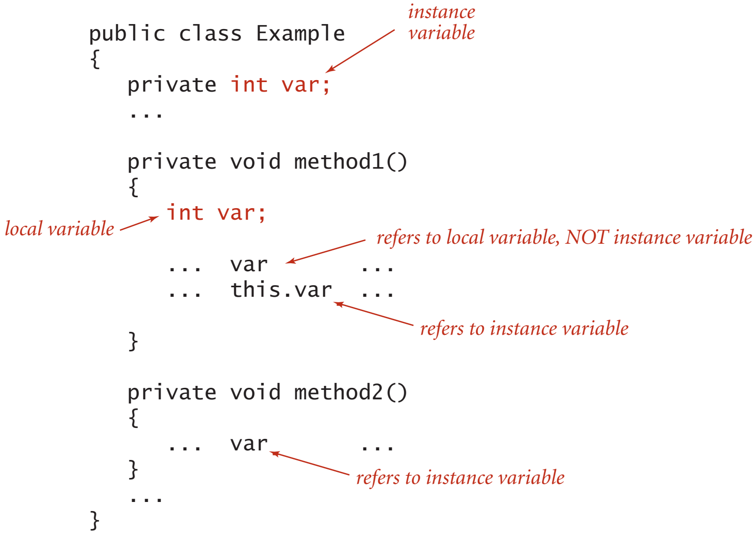
- to invoke an instance method that operates on the object's values.

The difference from working solely with static methods is semantic (see the Q&A), but has reoriented the way that modern programmers think about developing code in many situations. As you will see, it also dovetails well with the study of algorithms and data structures.

Scope. In summary, the Java code that we write to implement instance methods uses *three* kinds of variables:

- Parameter variables
- Local variables
- *Instance variables*

The first two of these are the same as for static methods: 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. The scope of parameter variables is the entire method; the scope of local variables is the following statements in 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 (whenever there is an ambiguity, you can use the `this` prefix to identify instance variables). Understanding the distinctions among these three kinds of variables in instance methods is a key to success in object-oriented programming.



```
public class Example
{
   private int var;
   ...

   private void method1()
   {
      int var;

      ...  var       ...
      ...  this.var  ...

   }

   private void method2()
   {
      ...  var       ...
   }
   ...
}
```

instance variable
local variable
refers to local variable, NOT instance variable
refers to instance variable
refers to instance variable

Scope of instance and local variables in an instance method

API, clients, and implementations. These are the basic components that you need to understand to be able to build and use abstract data types in Java. Every ADT implementation that we will consider will be a Java class with private instance variables, constructors, instance methods, and a client. To fully understand a data type, we need the API, typical client code, and an implementation, summarized for `Counter` on the facing page. To emphasize the separation of client and implementation, we normally present each client as a separate class containing a static method `main()` and reserve test client's `main()` in the data-type definition for minimal unit testing and development (calling each instance method at least once). In each data type that we develop, we go through the same steps. Rather than thinking about what action we need to take next to accomplish a computational goal (as we did when first learning to program), we think about the needs of a client, then accommodate them in an ADT, following these three steps:

- Specify an API. The purpose of the API is to *separate clients from implementations*, to enable modular programming. We have two goals when specifying an API. First, we want to enable clear and correct client code. Indeed, it is a good idea to write some client code before finalizing the API to gain confidence that the specified data-type operations are the ones that clients need. Second, we want to be able to implement the operations. There is no point specifying operations that we have no idea how to implement.
- Implement a Java class that meets the API specifications. First we choose the instance variables, then we write constructors and the instance methods.
- Develop multiple test clients, to validate the design decisions made in the first two steps.

What operations do clients need to perform, and what data-type values can best support those operations? These basic decisions are at the heart of every implementation that we develop.

API `public class Counter`

	`Counter(String id)`	*create a counter named* `id`
`void`	`increment()`	*increment the counter*
`int`	`tally()`	*number of increments since creation*
`String`	`toString()`	*string representation*

typical client

```
public class Flips
{
   public static void main(String[] args)
   {
      int T = Integer.parseInt(args[0]);

      Counter heads = new Counter("heads");
      Counter tails = new Counter("tails");

      for (int t = 0; t < T; t++)
         if (StdRandom.bernoulli(0.5))
              heads.increment();
         else tails.increment();

      StdOut.println(heads);
      StdOut.println(tails);
      int d = heads.tally() - tails.tally();
      StdOut.println("delta: " + Math.abs(d));
   }
}
```

implementation

```
public class Counter
{
   private final String name;
   private int count;

   public Counter(String id)
   { name = id; }

   public void increment()
   { count++; }

   public int tally()
   { return count; }

   public String toString()
   { return count + " " + name; }
}
```

application

```
% java Flips 1000000
500172 heads
499828 tails
delta: 344
```

An abstract data type for a simple counter

More ADT implementations As with any programming concept, the best way to understand the power and utility of ADTs is to consider carefully more examples and more implementations. There will be ample opportunity for you to do so, as much of this book is devoted to ADT implementations, but a few more simple examples will help us lay the groundwork for addressing them.

Date. Shown on the facing page are two implementations of the `Date` ADT that we considered on page 79. To reduce clutter, we omit the parsing constructor (which is described in EXERCISE 1.2.19) and the inherited methods `equals()` (see page 103), `compareTo()` (see page 247), and `hashCode()` (see EXERCISE 3.4.22). The straightforward implementation on the left maintains the day, month, and year as instance variables, so that the instance methods can just return the appropriate value; the more space-efficient implementation on the right uses only a single `int` value to represent a date, using a mixed-radix number that represents the date with day d, month m, and year y as $512y + 32m + d$. One way that a client might notice the difference between these implementations is by violating implicit assumptions: the second implementation depends for its correctness on the day being between 0 and 31, the month being between 0 and 15, and the year being positive (in practice, both implementations should check that months are between 1 and 12, days are between 1 and 31, and that dates such as June 31 and February 29, 2009, are illegal, though that requires a bit more work). This example highlights the idea that we rarely *fully* specify implementation requirements in an API (we normally do the best we can, and could do better here). Another way that a client might notice the difference between the two implementations is *performance:* the implementation on the right uses less space to hold data-type values at the cost of more time to provide them to the client in the agreed form (one or two arithmetic operations are needed). Such tradeoffs are common: one client may prefer one of the implementations and another client might prefer the other, so we need to accommodate both. Indeed, one of the recurring themes of this book is that we need to understand the space and time requirements of various implementations and their suitability for use by various clients. One of the key advantages of using data abstraction in our implementations is that we can normally change from one implementation to another *without changing any client code.*

Maintaining multiple implementations. Multiple implementations of the same API can present maintainence and nomenclature issues. In some cases, we simply want to replace an old implementation with an improved one. In others, we may need to maintain two implementations, one suitable for some clients, the other suitable for others. Indeed, a prime goal of this book is to consider in depth several implementations of each of a number of fundamental ADTs, generally with different performance characteristics. In this book, we often compare the performance of a single client using two

API `public class Date`

	`Date(int month, int day, int year)`	*create a date*
`int`	`month()`	*month*
`int`	`day()`	*day*
`int`	`year()`	*year*
`String`	`toString()`	*string representation*

test client

```
public static void main(String[] args)
{
   int m = Integer.parseInt(args[0]);
   int d = Integer.parseInt(args[1]);
   int y = Integer.parseInt(args[2]);
   Date date = new Date(m, d, y);
   StdOut.println(date);
}
```

application

```
% java Date 12 31 1999
12/31/1999
```

implementation

```
public class Date
{
   private final int month;
   private final int day;
   private final int year;

   public Date(int m, int d, int y)
   {  month = m; day = d; year = y; }

   public int month()
   {  return month;  }

   public int day()
   {  return day;  }

   public int year()
   {  return day;  }

   public String toString()
   {  return month() + "/" + day()
                     + "/" + year();  }

}
```

alternate implementation

```
public class Date
{
   private final int value;

   public Date(int m, int d, int y)
   { value = y*512 + m*32 + d; }

   public int month()
   { return (value / 32) % 16; }

   public int day()
   { return value % 32; }

   public int year()
   { return value / 512; }

   public String toString()
   {  return month() + "/" + day()
                     + "/" + year();  }
}
```

An abstract data type to encapsulate dates, with two implementations

different implementations of the same API. For this reason, we generally adopt an informal naming convention where we:

- Identify different implementations of the same API by prepending a descriptive modifier. For example, we might name our `Date` implementations on the previous page `BasicDate` and `SmallDate`, and we might wish to develop a `SmartDate` implementation that can validate that dates are legal.
- Maintain a reference implementation with no prefix that makes a choice that should be suitable for most clients. That is, most clients should just use `Date`.

In a large system, this solution is not ideal, as it might involve changing client code. For example, if we were to develop a new implementation `ExtraSmallDate`, then our only options are to change client code or to make it the reference implementation for use by all clients. Java has various advanced language mechanisms for maintaining multiple implementations without needing to change client code, but we use them sparingly because their use is challenging (and even controversial) even for experts, especially in conjuction with other advanced language features that we do value (generics and iterators). These issues are important (for example, ignoring them led to the celebrated *Y2K problem* at the turn of the millennium, because many programs used their own implementations of the date abstraction that did not take into account the first two digits of the year), but detailed consideration of these issues would take us rather far afield from the study of algorithms.

Accumulator. The *accumulator* API shown on the facing page defines an abstract data type that provides to clients the ability to maintain a running average of data values. For example, we use this data type frequently in this book to process experimental results (see SECTION 1.4). The implementation is straightforward: it maintains a `int` instance variable counts the number of data values seen so far and a `double` instance variable that keeps track of the sum of the values seen so far; to compute the average it divides the sum by the count. Note that the implementation does not save the data values—it could be used for a huge number of them (even on a device that is not capable of holding that many), or a huge number of accumulators could be used on a big system. This performance characteristic is subtle and might be specified in the API, because an implementation that does save the values might cause an application to run out of memory.

API

```
public class Accumulator
----------------------------------------------------------------------
                Accumulator()                   create an accumulator
           void addDataValue(double val)        add a new data value
         double mean()                          mean of all data values
         String toString()                      string representation
```

typical client

```
public class TestAccumulator
{
   public static void main(String[] args)
   {
      int T = Integer.parseInt(args[0]);
      Accumulator a = new Accumulator();
      for (int t = 0; t < T; t++)
         a.addDataValue(StdRandom.random());
      StdOut.println(a);
   }
}
```

application

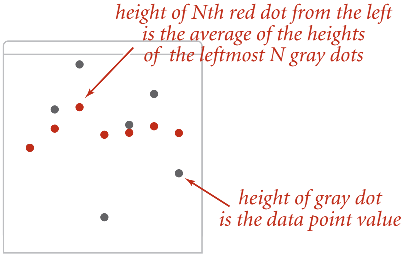
```
% java TestAccumulator 1000
Mean (1000 values): 0.51829

% java TestAccumulator 1000000
Mean (1000000 values): 0.49948

% java TestAccumulator 1000000
Mean (1000000 values): 0.50014
```

implementation

```
public class Accumulator
{
   private double total;
   private int N;

   public void addDataValue(double val)
   {
       N++;
       total += val;
   }

   public double mean()
   {  return total/N;  }

   public String toString()
   { return "Mean (" + N + " values): "
                 + String.format("%7.5f", mean()); }

}
```

An abstract data type for accumulating data values

Visual accumulator. The *visual accumulator* implementation shown on the facing page extends `Accumulator` to present a useful side effect: it draws on `StdDraw` all the data (in gray) and the running average (in red). The easiest way to do so is to add a constructor that provides the number of points to be plotted and the maximum value, for rescaling the plot. `VisualAccumulator` is not technically an implementation of the `Accumulator` API (its constructor has a different signature and it causes a different prescribed side effect). Generally, we are careful to fully specify APIs and are loath to make *any* changes in an API once articulated, as it might involve changing an unknown amount of client (and implementation) code, but adding a constructor to gain functionality can sometimes be defended because it involves changing the same line in client code that we change when changing a class name. In this example, if we have developed a client that uses an `Accumulator` and perhaps has many calls to `addDataValue()` and `avg()`, we can enjoy the benefits of `VisualAccumulator` by just changing one line of client code.

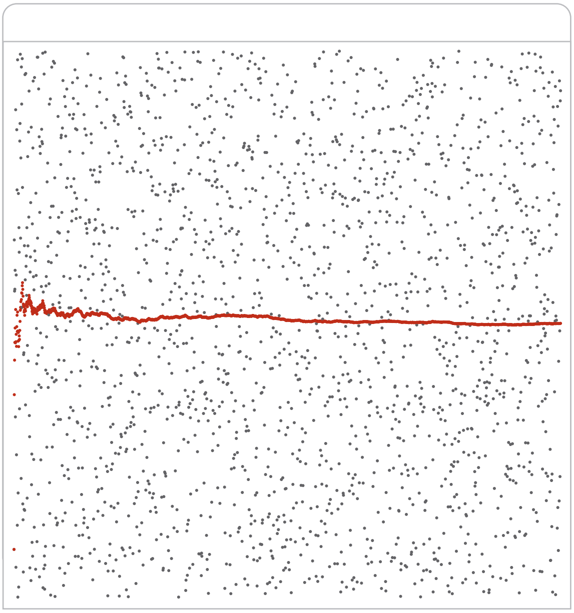


height of Nth red dot from the left is the average of the heights of the leftmost N gray dots

height of gray dot is the data point value

Visual accumulator plot

application



```
% java TestVisualAccumulator 2000
Mean (2000 values): 0.509789
```

API

```
public class VisualAccumulator
---------------------------------------------------------------------------
                VisualAccumulator(int trials, double max)
          void  addDataValue(double val)        add a new data value
        double  avg()                           average of all data values
        String  toString()                      string representation
```

typical client

```
public class TestVisualAccumulator
{
   public static void main(String[] args)
   {
      int T = Integer.parseInt(args[0]);
      VisualAccumulator a = new VisualAccumulator(T, 1.0);
      for (int t = 0; t < T; t++)
         a.addDataValue(StdRandom.random());
      StdOut.println(a);
   }
}
```

implementation

```
public class VisualAccumulator
{
   private double total;
   private int N;

   public VisualAccumulator(int trials, double max)
   {
      StdDraw.setXscale(0, trials);
      StdDraw.setYscale(0, max);
      StdDraw.setPenRadius(.005);
   }

   public void addDataValue(double val)
   {
      N++;
      total += val;
      StdDraw.setPenColor(StdDraw.DARK_GRAY);
      StdDraw.point(N, val);
      StdDraw.setPenColor(StdDraw.RED);
      StdDraw.point(N, total/N);
   }

   public double mean()
   public String toString()
   // Same as Accumulator.

}
```

An abstract data type for accumulating data values (visual version)

Data-type design

An abstract data type is a data type whose representation is hidden from the client. This idea has had a powerful effect on modern programming. The various examples that we have considered give us the vocabulary to address advanced characteristics of ADTs and their implementation as Java classes. Many of these topics are, on the surface, tangential to the study of algorithms, so it is safe for you to skim this section and refer to it later in the context of specific implementation problems. Our goal is to put important information related to designing data types in one place for reference and to set the stage for implementations throughout this book.

Encapsulation. A hallmark of object-oriented programming is that it enables us to *encapsulate* data types within their implementations, to facilitate separate development of clients and data type implementations. Encapsulation enables modular programming, allowing us to

- Independently develop of client and implementation code
- Substitute improved implementations without affecting clients
- Support programs not yet written (the API is a guide for any future client)

Encapsulation also isolates data-type operations, which leads to the possibility of

- Limiting the potential for error
- Adding consistency checks and other debugging tools in implementations
- Clarifying client code

An encapsulated data type can be used by any client, so it extends the Java language. The programming style that we are advocating is predicated on the idea of breaking large programs into small modules that can be developed and debugged independently. This approach improves the resiliency of our software by limiting and localizing the effects of making changes, and it promotes code reuse by making it possible to substitute new implementations of a data type to improve performance, accuracy, or memory footprint. The same idea works in many settings. We often reap the benefits of encapsulation when we use system libraries. New versions of the Java system often include new implementations of various data types or static method libraries, but *the APIs do not change*. In the context of the study of algorithms and data structures, there is strong and constant motivation to develop better algorithms because we can improve performance for *all* clients by substituting an improved ADT implementation without changing the code of *any* client. The key to success in modular programming is to maintain *independence* among modules. We do so by insisting on the API being the *only* point of dependence between client and implementation. *You do not need to know how a data type is implemented in order to use it* and *you can assume that a client knows nothing but the API* when implementing a data type. Encapsulation is the key to attaining both of these advantages.

Designing APIs. One of the most important and most challenging steps in building modern software is designing APIs. This task takes practice, careful deliberation, and many iterations, but any time spent designing a good API is certain to be repaid in time saved debugging or code reuse. Articulating an API might seem to be overkill when writing a small program, but you should consider writing *every* program as though you will need to reuse the code someday. Ideally, an API would clearly articulate behavior for all possible inputs, including side effects, and then we would have software to check that implementations meet the specification. Unfortunately, a fundamental result from theoretical computer science known as the *specification problem* implies that this goal is actually *impossible* to achieve. Briefly, such a specification would have to be written in a formal language like a programming language, and the problem of determining whether two programs perform the same computation is known, mathematically, to be *undecidable*. Therefore, our APIs are brief English-language descriptions of the set of values in the associated abstract data type along with a list of constructors and instance methods, again with brief English-language descriptions of their purpose, including side effects. To validate the design, we always include examples of client code in the text surrounding our APIs. Within this broad outline, there are numerous pitfalls that every API design is susceptible to:

- An API may be *too hard to implement*, implying implementations that are difficult or impossible to develop.
- An API may be *too hard to use*, leading to client code that is more complicated than it would be without the API.
- An API may be *too narrow*, omitting methods that clients need.
- An API may be *too wide*, including a large number of methods not needed by any client. This pitfall is perhaps the most common, and one of the most difficult to avoid. The size of an API tends to grow over time because it is not difficult to add methods to an existing API, but it *is* difficult to remove methods without breaking existing clients.
- An API may be *too general*, providing no useful abstractions.
- An API may be *too specific*, providing abstractions so detailed or so diffuse as to be useless.
- An API may be *too dependent on a particular representation*, therefore not serving the purpose of freeing client code from the details of using that representation. This pitfall is also difficult to avoid, because the representation is certainly central to the development of the implementation.

These considerations are sometimes summarized in yet another motto: *provide to clients the methods they need and no others.*

Algorithms and abstract data types. Data abstraction is naturally suited to the study of algorithms, because it helps us provide a framework within which we can precisely specify both what an algorithm needs to accomplish and how a client can make use of an algorithm. Typically, in this book, an algorithm is an implementation of an instance method in an abstract data type. For example, our whitelisting example at the beginning of the chapter is naturally cast as an ADT client, based on the following operations:

- Construct a `SET` from an array of given values.
- Determine whether a given value is in the set.

These operations are encapsulated in the `StaticSETofInts` ADT, shown on the facing page along with `Whitelist`, a typical client. `StaticSETofInts` is a special case of the more general and more useful *symbol table* ADT that is the focus of Chapter 3. Binary search is one of several algorithms that we study that is suitable for implementing these ADTs. By comparison with the `BinarySearch` implementation on page 47, this implementation leads to clearer and more useful client code. For example, `StaticSETofInts` enforces the idea that the array must be sorted before `rank()` is called. With the abstract data type, we separate the client from the implementation making it easier for *any* client to benefit from the ingenuity of the binary search algorithm, just by following the API (clients of `rank()` in `BinarySearch` have to know to sort the array first). Whitelisting is one of many clients that can take advantage of binary search.

application

```
% java Whitelist largeW.txt < largeT.txt
499569
984875
295754
207807
140925
161828
 ...
```

Every Java program is a set of static methods and/or a data type implementation. In this book, we focus primarily on *abstract* data type implementations such as `StaticSETofInts`, where the focus is on operations and the representation of the data is hidden from the client. As this example illustrates, data abstraction enables us to

- Precisely specify what algorithms can provide for clients
- Separate algorithm implementations from the client code
- Develop layers of abstraction, where we make use of well-understood algorithms to develop other algorithms

These are desirable properties of *any* approach to describing algorithms, whether it be an English-language description or pseudo-code. By embracing the Java `class` mechanism in support of data abstraction, we have little to lose and much to gain: working code that we can test and use to compare performance for diverse clients.

API

`public class StaticSETofInts`

	`StaticSETofInts(int[] a)`	*create a set from the values in* `a[]`
`boolean`	`contains(int key)`	*is* `key` *in the set?*

typical client

```
public class Whitelist
{
   public static void main(String[] args)
   {
      int[] w = In.readInts(args[0]);
      StaticSETofInts set = new StaticSETofInts(w);
      while (!StdIn.isEmpty())
      {  // Read key, print if not in whitelist.
         int key = StdIn.readInt();
         if (set.rank(key) == -1)
            StdOut.println(key);
      }
   }
}
```

implementation

```
import java.util.Arrays;

public class StaticSETofInts
{
   private int[] a;

   public StaticSETofInts(int[] keys)
   {
      a = new int[keys.length];
      for (int i = 0; i < keys.length; i++)
         a[i] = keys[i]; // defensive copy
      Arrays.sort(a);
   }

   public boolean contains(int key)
   {  return rank(key) != -1;  }

   private int rank(int key)
   {  // Binary search.
      int lo  = 0;
      int hi = a.length - 1;
      while (lo <= hi)
      {  // Key is in a[lo..hi] or not present.
         int mid = lo + (hi - lo) / 2;
         if      (key < a[mid]) hi = mid - 1;
         else if (key > a[mid]) lo = mid + 1;
         else                   return mid;
      }
      return -1;
   }
}
```

Binary search recast as an object-oriented program (an ADT for search in a set of integers)

Interface inheritance. Java provides language support for defining relationships among objects, known as *inheritance*. These mechanisms are widely used by software developers, so you will study them in detail if you take a course in software engineering. The first inheritance mechanism that we consider is known as *subtyping*, which allows us to specify a relationship between otherwise unrelated classes by specifying in an *interface* a set of common methods that each implementing class must contain. An interface is nothing more than a list of instance methods. For example, instead of using our informal API, we might have articulated an interface for Date:

```
public interface Datable
{
   int month();
   int day();
   int year();
}
```

and then referred to the interface in our implementation code

```
public class Date implements Datable
{
   // implementation code (same as before)
}
```

so that the Java compiler will check that it matches the interface. Adding the code implements Datable to any class that implements month(), day(), and year() provides a guarantee to any client that an object of that class can invoke those methods. This arrangement is known as *interface inheritance*—an implementing class *inherits* the interface. Interface inheritance allows us to write client programs that can manipulate objects of *any* type that implements the interface (even a type to be created in the future), by invoking methods in the interface. We might have used interface inheritance in place of our more informal APIs, but chose not to do so to avoid dependence on specific high-level language mechanisms that are not critical to the understanding of algorithms and to avoid the extra baggage of interface files. But there are a few situations where Java conventions make

	interface	methods	section
comparison	java.lang.Comparable	compareTo()	2.1
	java.util.Comparator	compare()	2.5
iteration	java.lang.Iterable	iterator()	1.3

	java.util.Iterator	hasNext()
		next()
		remove()
			1.3

Java interfaces used in this book

it worthwhile for us to take advantage of interfaces: we use them for *comparison* and for *iteration*, as detailed in the table at the bottom of the previous page, and will consider them in more detail when we cover those concepts.

Implementation inheritance. Java also supports another inheritence mechanism known as *subclassing*, which is a powerful technique that enables a programmer to change behavior and add functionality without rewriting an entire class from scratch. The idea is to define a new class (*subclass*, or *derived class*) that inherits instance methods *and* instance variables from another class (*superclass*, or *base class*). The subclass contains more methods than the superclass. Moreover, the subclass can redefine or *override* methods in the superclass. Subclassing is widely used by systems programmers to build so-called *extensible* libraries—one programmer (even you) can add methods to a library built by another programmer (or, perhaps, a team of systems programmers), effectively reusing the code in a potentially huge library. For example, this approach is widely used in the development of graphical user interfaces, so that the large amount of code required to provide all the facilities that users expect (drop-down menus, cut-and-paste, access to files, and so forth) can be reused. The use of subclassing is controversial among systems and applications programmers (its advantages over interface inheritance are debatable), and we avoid it in this book because it generally works against encapsulation. Certain vestiges of the approach are built in to Java and therefore unavoidable: specifically, every class is a subtype of Java's `Object` class. This structure enables the "convention" that every class includes an implementation of `getClass()`, `toString()`, `equals()`, `hashCode()`, and several other methods that we do not use in this book. Actually, every class *inherits* these methods from `Object` through subclassing, so any client can use them for any object. We usually override `toString()`, `equals()`, `hashCode()` in new classes because the default `Object` implementation generally does not lead to the desired behavior. We now will consider `toString()` and `equals()`; we discuss `hashCode()` in SECTION 3.4.

method		purpose	section
`Class`	`getClass()`	*what class is this object?*	1.2
`String`	`toString()`	*string representation of this object*	1.1
`boolean`	`equals(Object that)`	*is this object equal to* `that`*?*	1.2
`int`	`hashCode()`	*hash code for this object*	3.4

Inherited methods from `Object` used in this book

String conversion. By convention, every Java type inherits `toString()` from `Object`, so any client can invoke `toString()` for any object. This convention is the basis for Java's automatic conversion of one operand of the concatenation operator + to a `String` whenever the other operand is a `String`. If an object's data type does not include an implementation of `toString()`, then the default implementation in `Object` is invoked, which is normally not helpful, since it typically returns a string representation of the memory address of the object. Accordingly, we generally include implementations of `toString()` that override the default in every class that we develop, as highlighted for `Date` on the facing page. As illustrated in this code, `toString()` implementations are often quite simple, implicitly (through +) using `toString()` for each instance variable.

Wrapper types. Java supplies built-in reference types known as *wrapper types*, one for each of the primitive types: `Boolean`, `Byte`, `Character`, `Double`, `Float`, `Integer`, `Long`, and `Short` correspond to `boolean`, `byte`, `char`, `double`, `float`, `int`, `long`, and `short`, respectively. These classes consist primarily of static methods such as `parseInt()` but they also include the inherited instance methods `toString()`, `compareTo()`, `equals()`, and `hashCode()`. Java automatically converts from primitive types to wrapper types when warranted, as described on page 122. For example, when an `int` value is concatenated with a `String`, it is converted to an `Integer` that can invoke `toString()`.

Equality. What does it mean for two objects to be equal? If we test equality with `(a == b)` where `a` and `b` are reference variables of the same type, we are testing whether they have the same identity: whether the *references* are equal. Typical clients would rather be able to test whether the *data-type values* (object state) are the same, or to implement some type-specific rule. Java gives us a head start by providing implementations both for standard types such as `Integer`, `Double`, and `String` and for more complicated types such as `File` and `URL`. When using these types of data, you can just use the built-in implementation. For example, if `x` and `y` are `String` values, then `x.equals(y)` is `true` if and only if `x` and `y` have the same length and are identical in each character position. When we define our own data types, such as `Date` or `Transaction`, we need to override `equals()`. Java's convention is that `equals()` must be an *equivalence relation*. It must be

- *Reflexive*: `x.equals(x)` is `true.`
- *Symmetric*: `x.equals(y)` is `true` if and only if `y.equals(x).`
- *Transitive*: if `x.equals(y)` and `y.equals(z)` are `true`, then so is `x.equals(z).`

In addition, it must take an `Object` as argument and satisfy the following properties.

- *Consistent*: multiple invocations of `x.equals(y)` consistently return the same value, provided neither object is modified.
- *Not null*: `x.equals(null)` returns `false.`

These are natural definitions, but ensuring that these properties hold, adhering to Java conventions, and avoiding unnecessary work in an implementation can be tricky, as illustrated for `Date` below. It takes the following step-by-step approach:

- If the reference to this object is the same as the reference to the argument object, return `true`. This test saves the work of doing all the other checks in this case.
- If the argument is null, return `false`, to adhere to the convention (and to avoid following a null reference in code to follow).
- If the objects are not from the same class, return `false`. To determine an object's class, we use `getClass()`. Note that we can use == to tell us whether two objects of type `Class` are equal because `getClass()` is guaranteed to return the same reference for all objects in any given class.
- Cast the argument from `Object` to `Date` (this cast must succeed because of the previous test).
- Return `false` if any instance variables do not match. For other classes, some other definition of equality might be appropriate. For example, we might regard two `Counter` objects as equal if their `count` instance variables are equal.

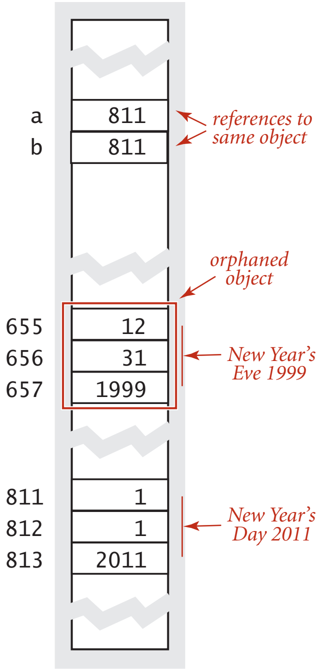
This implementation is a model that you can use to implement `equals()` for any type that you implement. Once you have implemented one `equals()`, you will not find it difficult to implement another.

```
public class Date
{
   private final int month;
   private final int day;
   private final int year;

   public Date(int m, int d, int y)
   {  month = m; day = d; year = y; }

   public int month()
   {  return month;  }

   public int day()
   {  return day;  }

   public int year()
   {  return year;  }

   public String toString()
   {  return month() + "/" + day() + "/" + year();  }

   public boolean equals(Object x)
   {
      if (this == x) return true;
      if (x == null) return false;
      if (this.getClass() != x.getClass()) return false;
      Date that = (Date) x;
      if (this.day != that.day)            return false;
      if (this.month != that.month)        return false;
      if (this.year != that.year)          return false;
      return true;
   }

}
```

Overriding `toString()` and `equals()` in a data-type definition

Memory management. The ability to assign a new value to a reference variable creates the possibility that a program may have created an object that can no longer be referenced. For example, consider the three assignment statements in the figure at left. After the third assignment statement, not only do a and b refer to the same Date object (1/1/2011), but also there is no longer a reference to the Date object that was created and used to initialize b. The only reference to that object was in the variable b, and this reference was overwritten by the assignment, so there is no way to refer to the object again. Such an object is said to be *orphaned*. Objects are also orphaned when they go out of scope. Java programs tend to create huge numbers of objects (and variables that hold primitive data-type values), but only have a need for a small number of them at any given point in time. Accordingly, programming languages and systems need mechanisms to *allocate* memory for data-type values during the time they are needed and to *free* the memory when they are no longer needed (for an object, sometime after it is orphaned). Memory management turns out to be easier for primitive types because all of the information needed for memory allocation is known at compile time. Java (and most other systems) takes care of reserving space for variables when they are declared and freeing that space when they go out of scope. Memory management for objects is more complicated: the system can allocate memory for an object when it is created, but cannot know precisely when to free the memory associated with each object because the dynamics of a program in execution determines when objects are orphaned. In many languages (such as C and C++) the programmer is responsible for both allocating and freeing memory. Doing so is tedious and notoriously error-prone. One of Java's most significant features is its ability to *automatically* manage memory. The idea is to free the programmers from the responsibility of managing memory by keeping track of orphaned objects and returning the memory they use to a pool of free memory. Reclaiming memory in this way is known as *garbage collection*. One of Java's characteristic features is its policy that references cannot be modified. This policy enables Java to do efficient automatic garbage collection. Programmers still debate whether the overhead of automatic garbage collection justifies the convenience of not having to worry about memory management.

```
Date a = new Date(12, 31, 1999);
Date b = new Date(1, 1, 2011);
b = a;
```



An orphaned object

Immutability. An *immutable* data type, such as `Date`, has the property that the value of an object never changes once constructed. By contrast, a *mutable* data type, such as `Counter` or `Accumulator`, manipulates object values that are intended to change. Java's language support for helping to enforce immutability is the `final` modifier. 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. Code that could modify the value of a `final` variable leads to a compile-time error. In our code, we use the modifier `final` with instance variables whose values never change. This policy serves as documentation that the value does not change, prevents accidental changes, and makes programs easier to debug. For example, you do not have to include a `final` value in a trace, since you know that its value never changes. A data type such as `Date` whose instance variables are all primitive and `final` is immutable (in code that does not use implementation inheritence, our convention). Whether to make a data type immutable is an important design decision and depends on the application at hand. For data types such as `Date`, the purpose of the abstraction is to encapsulate values that do not change so that we can use them in assignment statements and as arguments and return values from functions in the same way as we use primitive types (without having to worry about their values changing). A programmer implementing a `Date` client might reasonably expect to write the code `d = d0` for two `Date` variables, in the same way as for `double` or `int` values. But if `Date` were mutable and the value of `d` were to change *after* the assignment `d = d0`, then the value of `d0` would *also* change (they are both references to the same object)! On the other hand, for data types such as `Counter` and `Accumulator`, the very purpose of the abstraction is to encapsulate values as they change. You have already encountered this distinction as a client programmer, when using Java arrays (mutable) and Java's `String` data type (immutable). When you pass a `String` to a method, you do not worry about that method changing the sequence of characters in the `String`, but when you pass an array to a method, the method is free to change the contents of the array. `String` objects are immutable because we generally do *not* want `String` values to change, and Java arrays are mutable because we generally *do* want array values to change. There are also situations where we want to have mutable strings (that is the purpose of Java's `StringBuilder` class) and where we want to have immutable arrays (that is the purpose of the `Vector` class that we consider later in this section). Generally, immutable types are easier to use and harder to misuse than mutable types because the scope of code that can change their values is far smaller. It is easier to debug code that uses immutable types because it is easier to guarantee that variables in client code that uses them remain in a consistent state. When using mutable types,

mutable	**immutable**
`Counter`	`Date`
Java arrays	`String`

Mutable/immutable examples

you must always be concerned about where and when their values change. The downside of immutability is that *a new object must be created for every value*. This expense is normally manageable because Java garbage collectors are typically optimized for such situations. Another downside of immutability stems from the fact that, unfortunately, `final` guarantees immutability only when instance variables are primitive types, not reference types. If an instance variable of a reference type has the `final` modifier, the value of that instance variable (the reference to an object) will never change—it will always refer to the same object—but the value of the object itself *can* change. For example, this code does *not* implement an immutable type:

```
public class Vector
{
   private final double[] coords;

   public Vector(double[] a)
   {  coords = a; }
   ...
}
```

A client program could create a `Vector` by specifying the entries in an array, and then (bypassing the API) change the elements of the `Vector` after construction:

```
double[] a = { 3.0, 4.0 };
Vector vector = new Vector(a);
a[0] = 0.0;  // Bypasses the public API.
```

The instance variable `coords[]` is `private` and `final`, but `Vector` is mutable because the client holds a reference to the data. Immutability needs to be taken into account in any data-type design, and whether a data type is immutable should be specified in the API, so that clients know that object values will not change. In this book, our primary interest in immutability is for use in certifying the correctness of our algorithms. For example, if the type of data used for a binary search algorithm were mutable, then clients could invalidate our assumption that the array is sorted for binary search.

Design by contract. To conclude, we briefly discuss Java language mechanisms that enables you to verify assumptions about your program *as it is running*. We use two Java language mechanisms for this purpose:

- Exceptions, which generally handle unforeseen errors *outside* our control
- Assertions, which verify assumptions that we make *within* code we develop

Liberal use of both exceptions and assertions is good programming practice. We use them sparingly in the book for economy, but you will find them throughout the code on the booksite. This code aligns with a substantial amount of the surrounding commentary about each algorithm in the text that has to do with exceptional conditions and with asserted invariants.

Exceptions and errors. *Exceptions* and *errors* are disruptive events that occur while a program is running, often to signal an error. The action taken is known as *throwing an exception or throwing an error*. We have already encountered exceptions thrown by Java system methods in the course of learning basic features of Java: `StackOverflowError`, `ArithmeticException`, `ArrayIndexOutOfBoundsException`, `OutOfMemoryError`, and `NullPointerException` are typical examples. You can also create your own exceptions. The simplest kind is a `RuntimeException` that terminates execution of the program and prints an error message

```
throw new RuntimeException("Error message here.");
```

A general practice known as *fail fast* programming suggests that an error is more easily pinpointed if an exception is thrown as soon as an error is discovered (as opposed to ignoring the error and deferring the exception to sometime in the future).

Assertions. An *assertion* is a boolean expression that you are affirming is `true` at that point in the program. If the expression is `false`, the program will terminate and report an error message. We use assertions both to gain confidence in the correctness of programs and to document intent. For example, suppose that you have a computed value that you might use to index into an array. If this value were negative, it would cause an `ArrayIndexOutOfBoundsException` sometime later. But if you write the code `assert index >= 0;` you can pinpoint the place where the error occurred. You can also add an optional detail message such as

```
assert index >= 0 : "Negative index in method X";
```

to help you locate the bug. By default, assertions are disabled. You can enable them from the command line by using the `-enableassertions` flag (`-ea` for short). Assertions are for debugging: your program should not rely on assertions for normal operation since they may be disabled. When you take a course in systems programming, you will learn to use assertions to ensure that your code *never* terminates in a system error or goes into an infinite loop. One model, known as the *design-by-contract* model of programming expresses the idea. The designer of a data type expresses a *precondition* (the condition that the client promises to satisfy when calling a method), a *postcondition* (the condition that the implementation promises to achieve when returning from a method), and *side effects* (any other change in state that the method could cause). During development, these conditions can be tested with assertions.

Summary. The language mechanisms discussed throughout this section illustrate that effective data-type design leads to nontrivial issues that are not easy to resolve. Experts are still debating the best ways to support some of the design ideas that we are discussing. Why does Java not allow functions as arguments? Why does Matlab copy arrays passed as arguments to functions? As mentioned early in CHAPTER 1, it is a slippery slope from complaining about features in a programming language to becoming a programming-language designer. If you do not plan to do so, your best strategy is to use widely available languages. Most systems have extensive libraries that you certainly should use when appropriate, but you often can simplify your client code and protect yourself by building abstractions that can easily transport to other languages. Your main goal is to develop data types so that most of your work is done at a level of abstraction that is appropriate to the problem at hand.

The table on the facing page summarizes the various kinds of Java classes that we have considered.

kind of class	examples	characteristics
static methods	`Math` `StdIn` `StdOut`	no instance variables

immutable
abstract data type
	`Date` `Transaction`
	`String` `Integer`
		instance variables all `private`
		instance variables all `final`
		defensive copy for reference types
		Note: these are necessary but not sufficient.

mutable *abstract data type*	`Counter` `Accumulator`	instance variables all `private` not all instance variables `final`

abstract data type
with I/O side effects
	`VisualAccumulator`
	`In` `Out` `Draw`
		instance variables all `private`
		instance methods do I/O

Java classes (data-type implementations)

Q & A

Q. Why bother with data abstraction?

A. It helps us produce reliable and correct code. For example, in the 2000 presidential election, Al Gore received –16,022 votes on an electronic voting machine in Volusia County, Florida—the tally was clearly not properly encapsulated in the voting machine software!

Q. Why the distinction between primitive and reference types? Why not just have reference types?

A. Performance. Java provides the reference types `Integer`, `Double`, and so forth that correspond to primitive types that can be used by programmers who prefer to ignore the distinction. Primitive types are closer to the types of data that are supported by computer hardware, so programs that use them usually run faster than programs that use corresponding reference types.

Q. Do data types *have* to be abstract?

A. No. Java also allows `public` and `protected` to allow some clients to refer directly to instance variables. As described in the text, the advantages of allowing client code to directly refer to data are greatly outweighed by the disadvantages of dependence on a particular representation, so all instance variables are `private` in our code. We also occasionally use `private` instance methods to share code among public methods.

Q. What happens if I forget to use `new` when creating an object?

A. To Java, it looks as though you want to call a static method with a return value of the object type. Since you have not defined such a method, the error message is the same as anytime you refer to an undefined symbol. If you compile the code

```
Counter c = Counter("test");
```

you get this error message:

```
cannot find symbol
symbol  : method Counter(String)
```

You get the same kind of error message if you provide the wrong number of arguments to a constructor.

Q. What happens if I forget to use `new` when creating an array of objects?

A. You need to use `new` for each object that you create, so when you create an array of N objects, you need to use `new` $N+1$ times: once for the array and once for each of the objects. If you forget to create the array:

```
Counter[] a;
a[0] = new Counter("test");
```

you get the same error message that you would get when trying to assign a value to any uninitialized variable:

```
variable a might not have been initialized
      a[0] = new Counter("test");
      ^
```

but if you forget to use `new` when creating an object within the array and then try to use it to invoke a method:

```
Counter[] a = new Counter[2];
a[0].increment();
```

you get a `NullPointerException`.

Q. Why not write `StdOut.println(x.toString())` to print objects?

A. That code works fine, but Java saves us the trouble of writing it by automatically invoking the `toString()` method for any object, since `println()` has a method that takes an `Object` as argument.

Q. What is a *pointer*?

A. Good question. Perhaps that should be `NullReferenceException`. Like a Java reference, you can think of a *pointer* as a machine address. In many programming languages, the pointer is a primitive data type that programmers can manipulate in many ways. But programming with pointers is notoriously error-prone, so operations provided for pointers need to be carefully designed to help programmers avoid errors. Java takes this point of view to an extreme (that is favored by many modern programming-language designers). In Java, there is only *one* way to create a reference (`new`) and only *one* way to change a reference (with an assignment statement). That is, the only things that a programmer can do with references are to create them and copy them. In

Q & A *(continued)*

programming-language jargon, Java references are known as *safe pointers*, because Java can guarantee that each reference points to an object of the specified type (and it can determine which objects are not in use, for garbage collection). Programmers used to writing code that directly manipulates pointers think of Java as having no pointers at all, but people still debate whether it is really desirable to have unsafe pointers.

Q. Where can I find more details on how Java implements references and does garbage collection?

A. One Java system might differ completely from another. For example, one natural scheme is to use a pointer (machine address); another is to use a *handle* (a pointer to a pointer). The former gives faster access to data; the latter provides for better garbage collection.

Q. What exactly does it mean to `import` a name?

A. Not much: it just saves some typing. You could type `java.util.Arrays` instead of `Arrays` everywhere in your code instead of using the `import` statement.

Q. What is the problem with implementation inheritance?

A. Subtyping makes modular programming more difficult for two reasons. First, any change in the superclass affects all subclasses. The subclass cannot be developed *independently* of the superclass; indeed, it is *completely dependent* on the superclass. This problem is known as the *fragile base class* problem. Second, the subclass code, having access to instance variables, can subvert the intention of the superclass code. For example, the designer of a class like `Counter` for a voting system may take great care to make it so that `Counter` can only increment the tally by one (remember Al Gore's problem). But a subclass, with full access to the instance variable, can change it to any value whatever.

Q. How do I make a class immutable?

A. To ensure immutability of a data type that includes an instance variable of a mutable type, we need to make a local copy, known as a *defensive copy*. And that may not be enough. Making the copy is one challenge; ensuring that none of the instance methods change values is another.

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`. If you get this error message, check to make sure that your constructor properly initializes all of its instance variables.

Q. Can I have a static method in a class that implements a data type?

A. Of course. For example, all of our classes have `main()`. Also, it is natural to consider adding 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 might define a static method like the following within `Point`:

```
public static double distance(Point a, Point b)
{
   return a.distTo(b);
}
```

Often, including such methods can serve to clarify client code.

Q. Are there other kinds of variables besides parameter, local, and instance variables?

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. When we do, we will call attention to them.

Q. What is a *deprecated* method?

A. A method that is no longer fully supported, but kept in an API to maintain compatibility. For example, Java once included a method `Character.isSpace()`, and programmers wrote programs that relied on using that method's behavior. When the designers of Java later wanted to support additional Unicode whitespace characters, they could not change the behavior of `isSpace()` without breaking client programs, so, instead, they added a new method, `Character.isWhiteSpace()`, and deprecated the old method. As time wears on, this practice certainly complicates APIs. Sometimes, entire classes are deprecated. For example, Java deprecated its `java.util.Date` in order to better support internationalization.

EXERCISES

1.2.1 Write a `Point2D` client that takes an integer value N from the command line, generates N random points in the unit square, and computes the distance separating the *closest pair* of points.

1.2.2 Write an `Interval1D` client that takes an `int` value N as command-line argument, reads N intervals (each defined by a pair of `double` values) from standard input, and prints all pairs that intersect.

1.2.3 Write an `Interval2D` client that takes command-line arguments `N`, `min`, and `max` and generates `N` random 2D intervals whose width and height are uniformly distributed between `min` and `max` in the unit square. Draw them on `StdDraw` and print the number of pairs of intervals that intersect and the number of intervals that are contained in one another.

1.2.4 What does the following code fragment print?

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

1.2.5 What does the following code fragment print?

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

Answer: `"Hello  World"`. `String` objects are immutable—string methods return a new `String` object with the appropriate value (but they do not change the value of the object that was used to invoke them). This code ignores the objects returned and just prints the original string. To print `"WORLD"`, use `s = s.toUpperCase()` and `s = s.substring(6, 11)`.

1.2.6 A string `s` is a *circular rotation* of a string `t` if it matches when the characters are circularly shifted by any number of positions; e.g., ACTGACG is a circular shift of TGACGAC, and vice versa. Detecting this condition is important in the study of genomic sequences. Write a program that checks whether two given strings `s` and `t` are circular shifts of one another. *Hint*: The solution is a one-liner with `indexOf()`, `length()`, and string concatenation.

1.2.7 What does the following recursive function return?

```
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);
}
```

1.2.8 Suppose that `a[]` and `b[]` are each integer arrays consisting of millions of integers. What does the follow code do? Is it reasonably efficient?

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

Answer. It swaps them. It could hardly be more efficient because it does so by copying references, so that it is not necessary to copy millions of elements.

1.2.9 Instrument `BinarySearch` (page 47) to use a `Counter` to count the total number of keys examined during all searches and then print the total after all searches are complete. *Hint*: Create a `Counter` in `main()` and pass it as an argument to `rank()`.

1.2.10 Develop a class `VisualCounter` that allows both increment and decrement operations. Take two arguments `N` and `max` in the constructor, where `N` specifies the maximum number of operations and `max` specifies the maximum absolute value for the counter. As a side effect, create a plot showing the value of the counter each time its tally changes.

1.2.11 Develop an implementation `SmartDate` of our `Date` API that raises an exception if the date is not legal.

1.2.12 Add a method `dayOfTheWeek()` to `SmartDate` that returns a `String` value `Monday`, `Tuesday`, `Wednesday`, `Thursday`, `Friday`, `Saturday`, or `Sunday`, giving the appropriate day of the week for the date. You may assume that the date is in the 21st century.

EXERCISES *(continued)*

1.2.13 Using our implementation of `Date` as a model (page 91), develop an implementation of `Transaction`.

1.2.14 Using our implementation of `equals()` in `Date` as a model (page 103), develop implementations of `equals()` for `Transaction`.

CREATIVE PROBLEMS

1.2.15 *File input.* Develop a possible implementation of the static `readInts()` method from `In` (which we use for various test clients, such as binary search on page 47) that is based on the `split()` method in `String`.

Solution:

```
public static int[] readInts(String name)
{
   In in = new In(name);
   String input = StdIn.readAll();
    String[] words = input.split("\\s+");
   int[] ints = new int[words.length;
   for int i = 0; i < word.length; i++)
      ints[i] = Integer.parseInt(words[i]);
   return ints;
}
```

We will consider a different implementation in Section 1.3 (see page 126).

1.2.16 *Rational numbers.* Implement an immutable data type `Rational` for rational numbers that supports addition, subtraction, multiplication, and division.

`public class Rational`

	`Rational(int numerator. int denominator)`	
`Rational`	`plus(Rational b)`	*sum of this number and b*
`Rational`	`minus(Rational b)`	*difference of this number and b*
`Rational`	`times(Rational b)`	*product of this number and b*
`Rational`	`divides(Rational b)`	*quotient of this number and b*
`boolean`	`equals(Rational that)`	*is this number equal to* `that` *?*
`String`	`toString()`	*string representation*

You do not have to worry about testing for overflow (see Exercise 1.2.17), but use as instance variables two `long` values that represent the numerator and denominator to limit the possibility of overflow. Use Euclid's algorithm (see page 4) to ensure that the numerator and denominator never have any common factors. Include a test client that exercises all of your methods.

CREATIVE PROBLEMS *(continued)*

1.2.17 *Robust implementation of rational numbers.* Use assertions to develop an implementation of Rational (see EXERCISE 1.2.16) that is immune to overflow.

1.2.18 *Variance for accumulator.* Validate that the following code, which adds the methods var() and stddev() to Accumulator, computes both the mean and variance of the numbers presented as arguments to addDataValue():

```
public class Accumulator
{
   private double m;
   private double s;
   private int N;

   public void addDataValue(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 var()
   {  return s/(N - 1);  }

   public double stddev()
   {  return Math.sqrt(this.var());  }

}
```

This implementation is less susceptible to roundoff error than the straightforward implementation based on saving the sum of the squares of the numbers.

1.2.19 *Parsing.* Develop the parse constructors for your Date and Transaction implementations of EXERCISE 1.2.13 that take a single String argument to specify the initialization values, using the formats given in the table below.

Partial solution:

```
public Date(String date)
{
   String[] fields = date.split("/");
   month = Integer.parseInt(fields[0]);
   day   = Integer.parseInt(fields[1]);
   year  = Integer.parseInt(fields[2]);
}
```

type	format	example
Date	integers separated by slashes	5/22/1939
Transaction	customer, date, and amount, separated by whitespace	Turing 5/22/1939 11.99

Formats for parsing

1.3 BAGS, QUEUES, AND STACKS

Several fundamental data types involve *collections* of objects. Specifically, the set of values is a collection of objects, and the operations revolve around adding, removing, or examining objects in the collection. In this section, we consider three such data types, known as the *bag*, the *queue*, and the *stack*. They differ in the specification of which object is to be removed or examined next.

Bags, queues, and stacks are fundamental and broadly useful. We use them in implementations throughout the book. Beyond this direct applicability, the client and implementation code in this section serves as an introduction to our general approach to the development of data structures and algorithms.

One goal of this section is to emphasize the idea that the way in which we represent the objects in the collection directly impacts the efficiency of the various operations. For collections, we design data structures for representing the collection of objects that can support efficient implementation of the requisite operations.

A second goal of this section is to introduce *generics* and *iteration*, basic Java constructs that substantially simplify client code. These are advanced programming-language mechanisms that are not necessarily essential to the understanding of algorithms, but their use allows us to develop client code (and implementations of algorithms) that is more clear, compact, and elegant than would otherwise be possible.

A third goal of this section is to introduce and show the importance of *linked* data structures. In particular, a classic data structure known as the *linked list* enables implementation of bags, queues, and stacks that achieve efficiencies not otherwise possible. Understanding linked lists is a key first step to the study of algorithms and data structures.

For each of the three types, we consider APIs and sample client programs, then look at possible representations of the data type values and implementations of the data-type operations. This scenario repeats (with more complicated data structures) throughout this book. The implementations here are models of implementations later in the book and worthy of careful study.

APIs As usual, we begin our discussion of abstract data types for collections by defining their APIs, shown below. Each contains a no-argument constructor, a method to add an item to the collection, a method to test whether the collection is empty, and a method that returns the size of the collection. `Stack` and `Queue` each have a method to remove a particular item from the collection. Beyond these basics, these APIs reflect two Java features that we will describe on the next few pages: *generics* and *iterable collections*.

Bag

`public class Bag<Item> implements Iterable<Item>`

	`Bag()`	*create an empty bag*
`void`	`add(Item item)`	*add an item*
`boolean`	`isEmpty()`	*is the bag empty?*
`int`	`size()`	*number of items in the bag*

FIFO queue

`public class Queue<Item> implements Iterable<Item>`

	`Queue()`	*create an empty queue*
`void`	`enqueue(Item item)`	*add an item*
`Item`	`dequeue()`	*remove the least recently added item*
`boolean`	`isEmpty()`	*is the queue empty?*
`int`	`size()`	*number of items in the queue*

Pushdown (LIFO) stack

`public class Stack<Item> implements Iterable<Item>`

	`Stack()`	*create an empty stack*
`void`	`push(Item item)`	*add an item*
`Item`	`pop()`	*remove the most recently added item*
`boolean`	`isEmpty()`	*is the stack empty?*
`int`	`size()`	*number of items in the stack*

APIs for fundamental generic iterable collections

Generics. An essential characteristic of collection ADTs is that we should be able to use them for any type of data. A specific Java mechanism known as *generics*, also known as *parameterized types*, enables this capability. The impact of generics on the programming language is sufficiently deep that they are not found in many languages (including early versions of Java), but our use of them in the present context involves just a small bit of extra Java syntax and is easy to understand. The notation `<Item>` after the class name in each of our APIs defines the name `Item` as a *type parameter*, a symbolic placeholder for some concrete type to be used by the client. You can read `Stack<Item>` as "stack of items." When implementing `Stack`, we do not know the concrete type of `Item`, but a client can use our stack for any type of data, including one defined long after we develop our implementation. The client code provides a concrete type when the stack is created: we can replace `Item` with the name of *any* reference data type (consistently, everywhere it appears). This provides exactly the capability that we need. For example, you can write code such as

```
Stack<String> stack = new Stack<String>();
stack.push("Test");
...
String next = stack.pop();
```

to use a stack for `String` objects and code such as

```
Queue<Date> queue = new Queue<Date>();
queue.enqueue(new Date(12, 31, 1999));
...
Date next = queue.dequeue();
```

to use a queue for `Date` objects. If you try to add a `Date` (or data of any other type than `String`) to `stack` or a `String` (or data of any other type than `Date`) to `queue`, you will get a compile-time error. Without generics, we would have to define (and implement) different APIs for each type of data we might need to collect; with generics, we can use one API (and one implementation) for all types of data, even types that are implemented in the future. As you will soon see, generic types lead to clear client code that is easy to understand and debug, so we use them throughout this book.

Autoboxing. Type parameters have to be instantiated as *reference* types, so Java has special mechanisms to allow generic code to be used with primitive types. Recall that Java's wrapper types are reference types that correspond to primitive types: `Boolean`, `Byte`, `Character`, `Double`, `Float`, `Integer`, `Long`, and `Short` correspond to `boolean`, `byte`, `char`, `double`, `float`, `int`, `long`, and `short`, respectively. Java automatically converts between these reference types and the corresponding primitive types—in assignments, method arguments, and arithmetic/logic expressions. In the present context, this conversion is helpful because it enables us to use generics with primitive types, as in the following code:

```
Stack<Integer> stack = new Stack<Integer>();
stack.push(17);      // auto-boxing (int -> Integer)
int i = stack.pop(); // auto-unboxing (Integer -> int)
```

Automatically casting a primitive type to a wrapper type is known as *autoboxing*, and automatically casting a wrapper type to a primitive type is known as *auto-unboxing*. In this example, Java automatically casts (autoboxes) the primitive value 17 to be of type `Integer` when we pass it to the `push()` method. The `pop()` method returns an `Integer`, which Java casts (auto-unboxes) to an `int` before assigning it to the variable `i`.

Iterable collections. For many applications, the client's requirement is just to process each of the items in some way, or to *iterate* through the items in the collection. This paradigm is so important that it has achieved first-class status in Java and many other modern languages (the programming language itself has specific mechanisms to support it, not just the libraries). With it, we can write clear and compact code that is free from dependence on the details of a collection's implementation. For example, suppose that a client maintains a collection of transactions in a `Queue`, as follows:

```
Queue<Transaction> collection = new Queue<Transaction>();
```

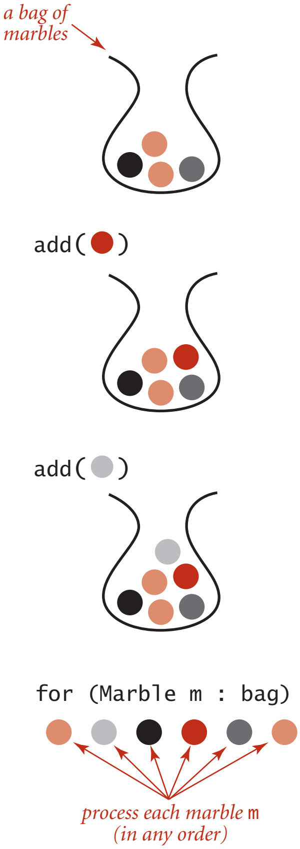
If the collection is iterable, the client can print a transaction list with a single statement:

```
for (Transaction t : collection)
{  StdOut.println(t);  }
```

This construct is known as the *foreach* statement: you can read the `for` statement as *for each transaction t in the collection, execute the following block of code*. This client code does not need to know anything about the representation or the implementation of the collection; it just wants to process each of the items in the collection. The same `for` loop would work with a `Bag` of transactions or any other iterable collection. We could hardly imagine client code that is more clear and compact. As you will see, supporting this capability requires extra effort in the implementation, but this effort is well worthwhile.

IT IS INTERESTING TO NOTE that the only differences between the APIs for `Stack` and `Queue` are their names and the names of the methods. This observation highlights the idea that we cannot easily specify all of the characteristics of a data type in a list of method signatures. In this case, the true specification has to do with the English-language descriptions that specify the rules by which an item is chosen to be removed (or to be processed next in the *foreach* statement). Differences in these rules are profound, *part of the API*, and certainly of critical importance in developing client code.

Bags. A *bag* is a collection where removing items is not supported—its purpose is to provide clients with the ability to collect items and then to iterate through the collected items (the client can also test if a bag is empty and find its number of items). The order of iteration is unspecified and should be immaterial to the client. To appreciate the concept, consider the idea of an avid marble collector, who might put marbles in a bag, one at a time, and periodically process all the marbles to look for one having some particular characteristic. With our `Bag` API, a client can add items to a bag and process them all with a *foreach* statement whenever needed. Such a client could use a stack or a queue, but one way to emphasize that the order in which items are processed is immaterial is to use a `Bag`. The class `Stats` at right illustrates a typical `Bag` client. The task is simply to compute the average and the sample standard deviation of the `double` values on standard input. If there are *N* numbers on standard input, their average is computed by adding the numbers and dividing by *N*; their sample standard deviation is computed by adding the squares of the difference between each number and the average, dividing by *N*–1, and taking the square root. The order in which the numbers are considered is not relevant for either of these calculations, so we save them in a `Bag` and use the *foreach* construct to compute each sum. *Note*: It is possible to compute the standard deviation without saving all the numbers (as we did for the average in `Accumulator`—see EXERCISE 1.2.18). Keeping the all numbers in a `Bag` *is* required for more complicated statistics.



Operations on a bag

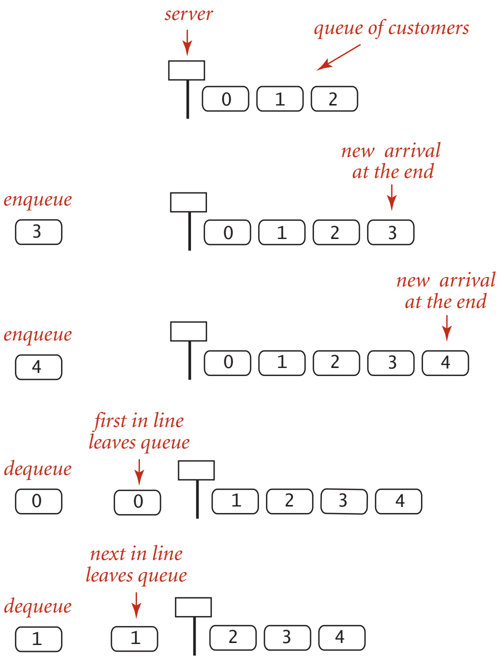
typical Bag client

```
public class Stats
{
   public static void main(String[] args)
   {
      Bag<Double> numbers = new Bag<Double>();

      while (!StdIn.isEmpty())
         numbers.add(StdIn.readDouble());
      int N = numbers.size();

      double sum = 0.0;
      for (double x : numbers)
         sum += x;
      double mean = sum/N;

      sum = 0.0;
      for (double x : numbers)
         sum += (x - mean)*(x - mean);
      double std = Math.sqrt(sum/(N-1));

      StdOut.printf("Mean: %.2f\n", mean);
      StdOut.printf("Std dev: %.2f\n", std);

   }
}
```

application

```
% java Stats
100
99
101
120
98
107
109
81
101
90

Mean: 100.60
Std dev: 10.51
```

FIFO queues. A *FIFO queue* (or just a *queue*) is a collection that is based on the *first-in-first-out* (FIFO) policy. The policy of doing tasks in the same order that they arrive is one that we encounter frequently in everyday life: from people waiting in line at a theater, to cars waiting in line at a toll booth, to tasks waiting to be serviced by an application on your computer. One bedrock principle of any service policy is the perception of fairness. The first idea that comes to mind when most people think about fairness is that whoever has been waiting the longest should be served first. That is precisely the FIFO discipline. Queues are a natural model for many everyday phenomena, and they play a central role in numerous applications. When a client iterates through the items in a queue with the *foreach* construct, the items are processed in the order they were added to the queue. A typical reason to use a queue in an application is to save items in a collection while at the same time *preserving their relative order*: they come out in the same order in which they were put in. For example, the client below is a possible implementation of the `readDoubles()` static method from our `In` class. The problem that this method solves for the client is that the client can get numbers from a file into an array *without knowing the file size ahead of time.* We *enqueue* the numbers from the file, use the `size()` method from `Queue` to find the size needed for the array, create the array, and then *dequeue* the numbers to move them to the array. A queue is appropriate because it puts the numbers into the array in the order in which they appear in the file (we might use a `Bag` if that order is immaterial). This code uses autoboxing and auto-unboxing to convert between the client's `double` primitive type and and the queue's `Double` wrapper type.

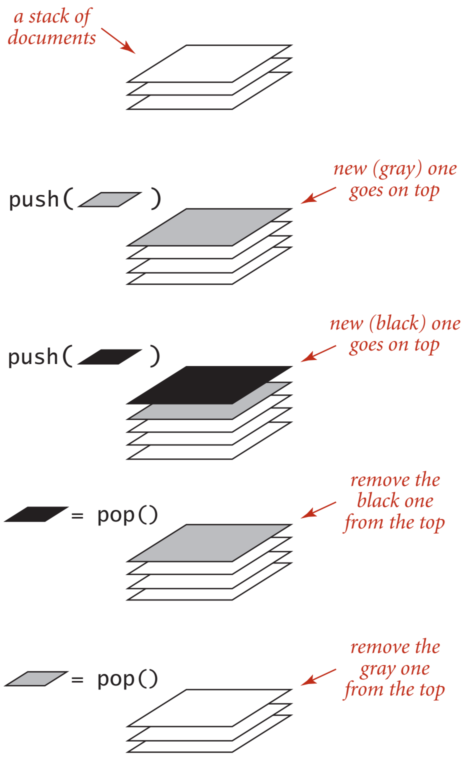


A typical FIFO queue

```
public static int[] readInts(String name)
{
   In in = new In(name);
   Queue<Integer> q = new Queue<Integer>();
   while (!in.isEmpty())
       q.enqueue(in.readInt());

   int N = q.size();
   int[] a = new int[N];
   for (int i = 0; i < N; i++)
      a[i] = q.dequeue();
   return a;
}
```

Sample Queue client

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 keep your mail in a pile on your desk, you are using a stack. You pile pieces of new mail on the top when they arrive and take each piece of mail from the top when you are ready to read it. People do not process as many papers as they did in the past, but the same organizing principle underlies several of the applications that you use regularly on your computer. For example, many people organize their email as a stack—they *push* messages on the top when they are received and *pop* them from the top when they read them, with most recently received first (last in, first out). The advantage of this strategy is that we see interesting email as soon as possible; the disadvantage is that some old email might never get read if we never empty the stack. You have likely encountered another common example of a stack when surfing the web. When you click a hyperlink, your browser displays the new page (and pushes onto a stack). You can keep clicking on hyperlinks to visit new pages, but you can always revisit the previous page by clicking the back button (popping it from the stack). The LIFO policy offered by a stack provides just the behavior that you expect. When a client iterates through the items in a stack with the *foreach* construct, the items are processed in the *reverse* of the order in which they were added. A typical reason to use a stack iterator in an application is to save items in a collection while at the same time *reversing* their relative order. For example, the client `Reverse` at right reverses the order of the integers on standard input, again without having to know ahead of time how many there are. The importance of stacks in computing is fundamental and profound, as indicated in the detailed example that we consider next.


a stack of documents
push( )
new (gray) one goes on top
push( )
new (black) one goes on top
= pop()
remove the black one from the top
= pop()
remove the gray one from the top

Operations on a pushdown stack

```
public class Reverse
{
   public static void main(String[] args)
   {
      Stack<Integer> stack;
      stack = new Stack<Integer>();
      while (!StdIn.isEmpty())
         stack.push(StdIn.readInt());

      for (int i : stack)
         StdOut.println(i);
   }
}
```

Sample Stack client

Arithmetic expression evaluation. As another example of a stack client, we consider a classic example that also demonstrates the utility of generics. Some of the first programs that we considered in SECTION 1.1 involved computing the value of arithmetic expressions like this one:

```
( 1 + ( ( 2 + 3 ) * ( 4 * 5 ) ) )
```

If you multiply `4` by `5`, add `3` to `2`, multiply the result, and then add `1`, you get the value `101`. But how does the Java system do this calculation? Without going into the details of how the Java system is built, we can address the essential ideas by writing a Java program that can take a string as input (the expression) and produce the number represented by the expression as output. For simplicity, we begin with the following explicit recursive definition: an *arithmetic expression* is either a number, or a left parenthesis followed by an arithmetic expression followed by an operator followed by another arithmetic expression followed by a right parenthesis. For simplicity, this definition is for *fully parenthesized* arithmetic expressions, which specify precisely which operators apply to which operands—you are a bit more familiar with expressions such as `1 + 2 * 3`, where we often rely on precedence rules instead of parentheses. The same basic mechanisms that we consider can handle precedence rules, but we avoid that complication. For specificity, we support the familiar binary operators `*`, `+`, `-`, and `/`, as well as a square-root operator `sqrt` that takes just one argument. We could easily allow more operators and more kinds of operators to embrace a large class of familiar mathematical expressions, involving trigonometric, exponential, and logarithmic functions. Our focus is on understanding how to interpret the string of parentheses, operators, and numbers to enable performing in the proper order the low-level arithmetic operations that are available on any computer. Precisely how can we convert an arithmetic expression—a string of characters—to the value that it represents? A remarkably simple algorithm that was developed by E. W. Dijkstra in the 1960s uses two stacks (one for operands and one for operators) to do this job. An expression consists of parentheses, operators, and operands (numbers). Proceeding from left to right and taking these entities one at a time, we manipulate the stacks according to four possible cases, as follows:

- Push *operands* onto the operand stack.
- Push *operators* onto the operator stack.
- Ignore *left* parentheses.
- On encountering a *right* parenthesis, pop an operator, pop the requisite number of operands, and push onto the operand stack the result of applying that operator to those operands.

After the final right parenthesis has been processed, there is one value on the stack, which is the value of the expression. This method may seem mysterious at first, but it

Dijkstra’s Two-Stack Algorithm for Expression Evaluation

```
public class Evaluate
{
   public static void main(String[] args)
   {
      Stack<String> ops  = new Stack<String>();
      Stack<Double> vals = new Stack<Double>();
      while (!StdIn.isEmpty())
      {  // Read token, push if operator.
         String s = StdIn.readString();
         if      (s.equals("("))               ;
         else if (s.equals("+"))    ops.push(s);
         else if (s.equals("-"))    ops.push(s);
         else if (s.equals("*"))    ops.push(s);
         else if (s.equals("/"))    ops.push(s);
         else if (s.equals("sqrt")) ops.push(s);
         else if (s.equals(")"))
         {  // Pop, evaluate, and push result if token is ")".
            String op = ops.pop();
            double v = vals.pop();
            if      (op.equals("+"))    v = vals.pop() + v;
            else if (op.equals("-"))    v = vals.pop() - v;
            else if (op.equals("*"))    v = vals.pop() * v;
            else if (op.equals("/"))    v = vals.pop() / v;
            else if (op.equals("sqrt")) v = Math.sqrt(v);
            vals.push(v);
         }  // Token not operator or paren: push double value.
         else vals.push(Double.parseDouble(s));
      }
      StdOut.println(vals.pop());
   }
}
```

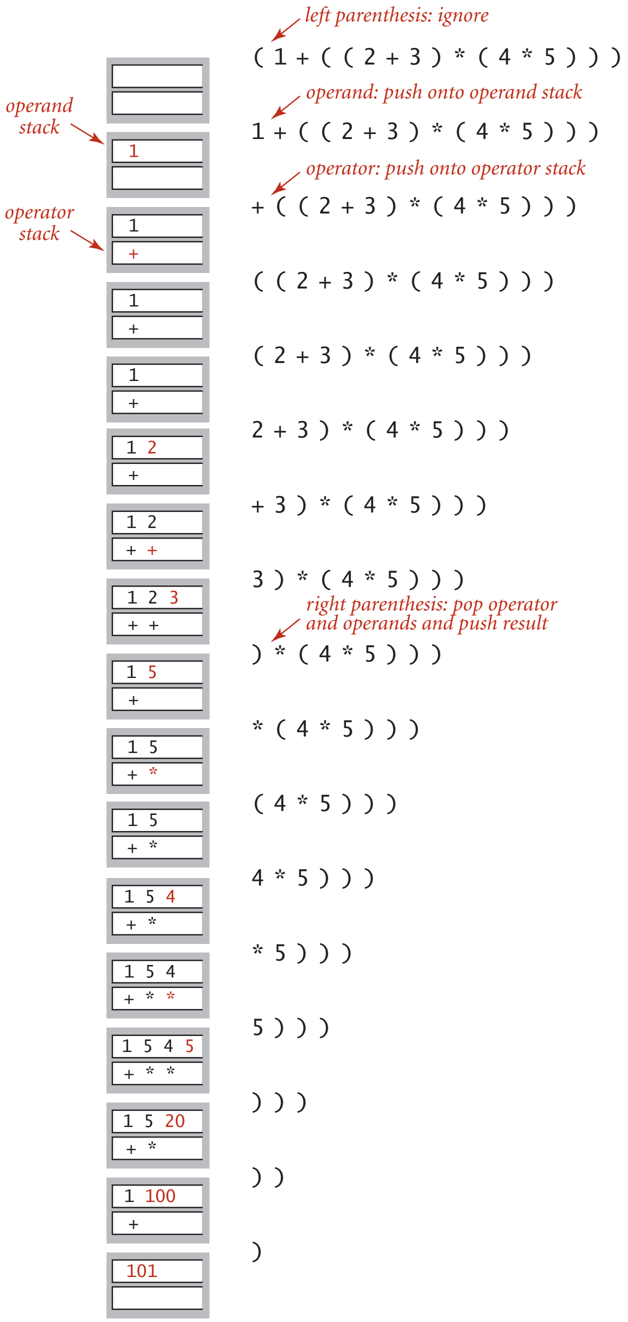
This Stack client uses two stacks to evaluate arithmetic expressions, illustrating an essential computational process: interpreting a string as a program and executing that program to compute the desired result. With generics, we can use the code in a single Stack implementation to implement one stack of String values and another stack of Double values. For simplicity, this code assumes that the expression is fully parenthesized, with numbers and characters separated by whitespace.

```
% java Evaluate
( 1 + ( ( 2 + 3 ) * ( 4 * 5 ) ) )
101.0

% java Evaluate
( ( 1 + sqrt ( 5.0 ) ) / 2.0 )
1.618033988749895
```

is easy to convince yourself that it computes the proper value: any time the algorithm encounters a subexpression consisting of two operands separated by an operator, all surrounded by parentheses, it leaves the result of performing that operation on those operands on the operand stack. The result is the same as if that value had appeared in the input instead of the subexpression, so we can think of replacing the subexpression by the value to get an expression that would yield the same result. We can apply this argument again and again until we get a single value. For example, the algorithm computes the same value for all of these expressions:

```
( 1 + ( ( 2 + 3 ) * ( 4 * 5 ) ) )
( 1 + ( 5 * ( 4 * 5 ) ) )
( 1 + ( 5 * 20 ) )
( 1 + 100 )
101
```

`Evaluate` on the previous page is an implementation of this algorithm. This code is a simple example of an *interpreter*: a program that interprets the computation specified by a given string and performs the computation to arrive at the result.



Trace of Dijkstra's two-stack arithmetic expression-evaluation algorithm

Implementing collections To address the issue of implementing Bag, Stack and Queue, we begin with a simple classic implementation, then address improvements that lead us to implementations of the APIs articulated on page 121.

Fixed-capacity stack. As a strawman, we consider an abstract data type for a fixed-capacity stack of strings, shown on the opposite page. The API differs from our Stack API: it works only for String values, it requires the client to specify a capacity, and it does not support iteration. The primary choice in developing an API implementation is to *choose a representation for the data*. For FixedCapacityStackOfStrings, an obvious choice is to use an array of String values. Pursuing this choice leads to the implementation shown at the bottom on the opposite page, which could hardly be simpler (each method is a one-liner). The instance variables are an array a[] that holds 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 N is 0.
- The top of the stack (if it is nonempty) is at a[N-1].

As usual, thinking in terms of invariants of this sort is the easiest way to verify that an implementation operates as intended. *Be sure that you fully understand this implementation.* The best way to do so is to examine a trace of the stack contents for a sequence of operations, as illustrated at left for the test client, which reads strings from standard input and pushes each string onto a stack, unless it is "-", when it pops the stack and prints the result. The primary performance characteristic of this implementation is that the *push and pop operations take time independent of the stack size*. For many applications, it is the method of choice because of its simplicity. But it has several drawbacks that limit its potential applicability as a general-purpose tool, which we now address. With a moderate amount of effort (and some help from Java language mechanisms), we can develop an implementation that is broadly useful. This effort is worthwhile because the implementations that we develop serve as a model for implementations of other, more powerful, abstract data types throughout the book.

StdIn	StdOut	N	a[]				
(push)	*(pop)*		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
-	to	4	to	be	or	not	to
be		5	to	be	or	not	be
-	be	4	to	be	or	not	be
-	not	3	to	be	or	not	be
that		4	to	be	or	that	be
-	that	3	to	be	or	that	be
-	or	2	to	be	or	that	be
-	be	1	to	be	or	that	be
is		2	to	is	or	not	to

Trace of FixedCapacityStackOfStrings test client

API `public class FixedCapacityStackOfStrings`

	`FixedCapacityStackOfStrings(int cap)`	*create an empty stack of capacity* `cap`
`void`	`push(String item)`	*add a string*
`String`	`pop()`	*remove the most recently added string*
`boolean`	`isEmpty()`	*is the stack empty?*
`int`	`size()`	*number of strings on the stack*

test client

```
public static void main(String[] args)
{
   FixedCapacityStackOfStrings s;
   s = new FixedCapacityStackOfStrings(100);
   while (!StdIn.isEmpty())
   {
      String item = StdIn.readString();
      if (!item.equals("-"))
           s.push(item);
      else if (!s.isEmpty()) StdOut.print(s.pop() + " ");
   }

   StdOut.println("(" + s.size() + " left on stack)");
}
```

application

```
% more tobe.txt
to be or not to - be - - that - - - is

% java FixedCapacityStackOfStrings < tobe.txt
to be not that or be (2 left on stack)
```

implementation

```
public class FixedCapacityStackOfStrings
{
   private String[] a; // stack entries
   private int N;      // size

   public FixedCapacityStackOfStrings(int cap)
   {  a = new String[cap];  }

   public boolean isEmpty() {  return N == 0; }
   public int size()        {  return N; }

   public void push(String item)
   {  a[N++] = item; }

   public String pop()
   {  return a[--N]; }

}
```

An abstract data type for a fixed-capacity stack of strings

Generics. The first drawback of `FixedCapacityStackOfStrings` is that it works only for `String` objects. If we want a stack of `double` values, we would need to develop another class with similar code, essentially replacing `String` with `double` everywhere. This is easy enough but becomes burdensome when we consider building a stack of `Transaction` values or a queue of `Date` values, and so forth. As discussed on page 122, Java's parameterized types (generics) are specifically designed to address this situation, and we saw several examples of client code (on pages 125, 126, 127, and 129). But how do we *implement* a generic stack? The code on the facing page shows the details. It implements a class `FixedCapacityStack` that differs from `FixedCapacityStackOfStrings` only in the code highlighted in red—we replace every occurrence of `String` with `Item` (with one exception, discussed below) and declare the class with the following first line of code:

```
public class FixedCapacityStack<Item>
```

The name `Item` is a *type parameter*, a symbolic placeholder for some concrete type to be used by the client. You can read `FixedCapacityStack<Item>` as *stack of items*, which is precisely what we want. When implementing `FixedCapacityStack`, we do not know the actual type of `Item`, but a client can use our stack for any type of data by providing a concrete type when the stack is created. Concrete types must be reference types, but clients can depend on autoboxing to convert primitive types to their corresponding wrapper types. Java uses the type parameter `Item` to check for type mismatch errors—even though no concrete type is yet known, variables of type `Item` must be assigned values of type `Item`, and so forth. But there is one significant hitch in this story: We would like to implement the constructor in `FixedCapacityStack` with the code

```
a = new Item[cap];
```

which calls for creation of a generic array. For historical and technical reasons beyond our scope, *generic array creation is disallowed in Java*. Instead, we need to use a cast:

```
a = (Item[]) new Object[cap];
```

This code produces the desired effect (though the Java compiler gives a warning, which we can safely ignore), and we use this idiom throughout the book (the Java system library implementations of similar abstract data types use the same idiom).

API

public class FixedCapacityStack<Item>		
	FixedCapacityStack(int cap)	*create an empty stack of capacity* cap
void	push(Item item)	*add an item*
Item	pop()	*remove the most recently added item*
boolean	isEmpty()	*is the stack empty?*
int	size()	*number of items on the stack*

test client

```
public static void main(String[] args)
{
   FixedCapacityStack<String> s;
   s = new FixedCapacityStack<String>(100);
   while (!StdIn.isEmpty())
   {
      String item = StdIn.readString();
      if (!item.equals("-"))
            s.push(item);
      else if (!s.isEmpty()) StdOut.print(s.pop() + " ");
   }

   StdOut.println("(" + s.size() + " left on stack)");
}
```

application

```
% more tobe.txt
to be or not to - be - - that - - - is

% java FixedCapacityStack < tobe.txt
to be not that or be (2 left on stack)
```

implementation

```
public class FixedCapacityStack<Item>
{
   private Item[] a;    // stack entries
   private int N;       // size

   public FixedCapacityStack(int cap)
   {  a = (Item[]) new Object[cap];   }

   public boolean isEmpty() {  return N == 0; }
   public int size()        {  return N; }

   public void push(Item item)
   {  a[N++] = item; }

   public Item pop()
   {  return a[--N]; }

}
```

An abstract data type for a fixed-capacity generic stack

Array resizing. Choosing an array to represent the stack contents implies that clients must estimate the maximum size of the stack ahead of time. In Java, we cannot change the size of an array once created, so the stack always uses space proportional to that maximum. A client that chooses a large capacity risks wasting a large amount of memory at times when the collection is empty or nearly empty. For example, a transaction system might involve billions of items and thousands of collections of them. Such a client would have to allow for the possibility that each of those collections could hold all of those items, even though a typical constraint in such systems is that each item can appear in only one collection. Moreover, every client risks *overflow* if the collection grows larger than the array. For this reason, `push()` needs code to test for a full stack ,and we should have an `isFull()` method in the API to allow clients to test for that condition. We omit that code, because our desire is to relieve the client from having to deal with the concept of a full stack, as articulated in our original `Stack` API. Instead, we modify the array implementation to dynamically adjust the size of the array `a[]` so that it is both sufficiently large to hold all of the items and not so large as to waste an excessive amount of space. Achieving these goals turns out to be remarkably easy. First, we implement a method that moves a stack into an array of a different size:

```
private void resize(int max)
{  // Move stack of size N <= max to a new array of size max.
   Item[] temp = (Item[]) new Object[max];
   for (int i = 0; i < N; i++)
      temp[i] = a[i];
   a = temp;
}
```

Now, 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 there is no room, we *double* the size of the array. Then we simply insert the new item with the code `a[N++] = item`, as before:

```
public void push(String item)
{  // Add item to top of stack.
   if (N == a.length) resize(2*a.length);
   a[N++] = item;
}
```

Similarly, in `pop()`, we begin by deleting the item, then we *halve* the array size if it is too large. If you think a bit about the situation, you will see that the appropriate test is whether the stack size is less than *one-fourth* the array size. After the array is halved, it will be about half full and can accommodate a substantial number of `push()` and `pop()` operations before having to change the size of the array again.

```
public String pop()
{  // Remove item from top of stack.
   String item = a[--N];
   a[N] = null;  // Avoid loitering (see text).
   if (N > 0 && N == a.length/4) resize(a.length/2);
   return item;
}
```

With this implementation, the stack never overflows and never becomes less than one-quarter full (unless the stack is empty, when the array size is 1). We will address the performance analysis of this approach in more detail in SECTION 1.4.

Loitering. Java's garbage collection policy is to reclaim the memory associated with any objects that can no longer be accessed. In our `pop()` implementations, the reference to the popped item remains in the array. The item is effectively an *orphan*—it will be never be accessed again—but the Java garbage collector has no way to know this until it is overwritten. Even when the client is done with the item, the reference in the array may keep it alive. This condition (holding a reference to an item that is no longer needed) is known as *loitering*. In this case, loitering is easy to avoid, by setting the array entry corresponding to the popped item to `null`, thus overwriting the unused reference and making it possible for the system to reclaim the memory associated with the popped item when the client is finished with it.

push()	pop()	N	a.length	a[]							
				0	1	2	3	4	5	6	7
		0	1	*null*							
to		1	1	to							
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*	*null*
-	to	4	8	to	be	or	not	*null*	*null*	*null*	*null*
be		5	8	to	be	or	not	be	*null*	*null*	*null*
-	be	4	8	to	be	or	not	*null*	*null*	*null*	*null*
-	not	3	8	to	be	or	*null*	*null*	*null*	*null*	*null*
that		4	8	to	be	or	that	*null*	*null*	*null*	*null*
-	that	3	8	to	be	or	*null*	*null*	*null*	*null*	*null*
-	or	2	4	to	be	*null*	*null*				
-	be	1	2	to	*null*						
is		2	2	to	is						

Trace of array resizing during a sequence of `push()` and `pop()` operations

Iteration. As mentioned earlier in this section, one of the fundamental operations on collections is to process each item by *iterating* through the collection using Java's *foreach* statement. This paradigm leads to clear and compact code that is free from dependence on the details of a collection's implementation. To consider the task of implementing iteration, we start with a snippet of client code that prints all of the items in a collection of strings, one per line:

```
Stack<String> collection = new Stack<String>();
...
for (String s : collection)
   StdOut.println(s);
...
```

Now, this *foreach* statement is shorthand for a `while` construct (just like the `for` statement itself). It is essentially equivalent to the following `while` statement:

```
Iterator<String> i = collection.iterator();
while (i.hasNext())
{
   String s = i.next();
   StdOut.println(s);
}
```

This code exposes the ingredients that we need to implement in any iterable collection:

- The collection must implement an `iterator()` method that returns an `Iterator` object.
- The `Iterator` class must include two methods: `hasNext()` (which returns a `boolean` value) and `next()` (which returns a generic item from the collection).

In Java, we use the `interface` mechanism to express the idea that a class implements a specific method (see page 100). For iterable collections, the necessary interfaces are already defined for us in Java. To make a class iterable, the first step is to add the phrase `implements Iterable<Item>` to its declaration, matching the interface

```
public interface Iterable<Item>
{
   Iterator<Item> iterator();
}
```

(which is in `java.lang.Iterable`), and to add a method `iterator()` to the class that returns an `Iterator<Item>`. Iterators are generic, so we can use our parameterized type `Item` to allow clients to iterate through objects of whatever type is provided by our client. For the array representation that we have been using, we need to iterate through an array in reverse order, so we name the iterator ReverseArrayIterator and add this method:

```
public Iterator<Item> iterator()
{  return new ReverseArrayIterator();  }
```

What is an iterator? An object from a class that implements the methods hasNext() and next(), as defined in the following interface (which is in java.util.Iterator):

```
public interface Iterator<Item>
{
    boolean hasNext();
    Item next();
    void remove();
}
```

Although the interface specifies a remove() method, we always use an empty method for remove() in this book, because interleaving iteration with operations that modify the data structure is best avoided. For ReverseArrayIterator, these methods are all one-liners, implemented in a nested class within our stack class:

```
private class ReverseArrayIterator implements Iterator<Item>
{
   private int i = N;

   public boolean hasNext() {  return i > 0;   }
   public Item next()       {  return a[--i];  }
   public void remove()     {                  }
}
```

Note that this nested class can access the instance variables of the enclosing class, in this case a[] and N (this ability is the main reason we use nested classes for iterators). Technically, to conform to the Iterator specification, we should throw exceptions in two cases: an UnsupportedOperationException if a client calls remove() and a NoSuchElementException if a client calls next() when i is 0. Since we only use iterators in the *foreach* construction where these conditions do not arise, we omit this code. One crucial detail remains: we have to include

```
import java.util.Iterator;
```

at the beginning of the program because (for historical reasons) Iterator is not part of java.lang (even though Iterable *is* part of java.lang). Now a client using the *foreach* statement for this class will get behavior equivalent to the common for loop for arrays, but does not need to be aware of the array representation (an implementation detail). This arrangement is of critical importance for implementations of fundamental data types like the collections that we consider in this book and those included in Java libraries. For example, it frees us to switch to a totally different representation *without having to change any client code*. More important, taking the client's point of view, it allows clients to use iteration *without having to know any details of the class implementation*.

ALGORITHM 1.1 is an implementation of our `Stack` API that resizes the array, allows clients to make stacks for any type of data, and supports client use of *foreach* to iterate through the stack items in LIFO order. This implementation is based on Java language nuances involving `Iterator` and `Iterable`, but there is no need to study those nuances in detail, as the code itself is not complicated and can be used as a template for other collection implementations.

For example, we can implement the `Queue` API by maintaining two indices as instance variables, a variable `head` for the beginning of the queue and a variable `tail` for the end of the queue. To remove an item, use `head` to access it and then increment `head`; to insert an item, use `tail` to store it, and then increment `tail`. If incrementing an index brings it past the end of the array, reset it to 0. Developing the details of checking when the queue is empty and when the array is full and needs resizing is an interesting and worthwhile programming exercise (see EXERCISE 1.3.14).

StdIn	StdOut	N	head	tail	a[]							
(enqueue)	*(dequeue)*				0	1	2	3	4	5	6	7
		5	0	5	to	be	or	not	to			
-	to	4	1	5	to	be	or	not	to			
be		5	1	6	to	be	or	not	to	be		
-	be	4	2	6	to	be	or	not	to	be		
-	or	3	3	6	to	be	or	that	to	be		

Trace of `ResizingArrayQueue` test client

In the context of the study of algorithms, ALGORITHM 1.1 is significant because it almost (but not quite) achieves optimum performance goals for any collection implementation:

- Each operation should require time independent of the collection size.
- The space used should always be within a constant factor of the collection size.

The flaw in `ResizingArrayStack` is that some *push* and *pop* operations require resizing: this takes time proportional to the size of the stack. Next, we consider a way to correct this flaw, using a fundamentally different way to structure data.

ALGORITHM 1.1 Pushdown (LIFO) stack (resizing array implementation)

```
import java.util.Iterator;
public class ResizingArrayStack<Item> implements Iterable<Item>
{
   private Item[] a = (Item[]) new Object[1];  // stack items
   private int N = 0;                          // number of items

   public boolean isEmpty()  {  return N == 0; }
   public int size()         {  return N;      }

   private void resize(int max)
   {  // Move stack to a new array of size max.
      Item[] temp = (Item[]) new Object[max];
      for (int i = 0; i < N; i++)
         temp[i] = a[i];
      a = temp;
   }

   public void push(Item item)
   {  // Add item to top of stack.
      if (N == a.length) resize(2*a.length);
      a[N++] = item;
   }

   public Item pop()
   {  // Remove item from top of stack.
      Item item = a[--N];
      a[N] = null;  // Avoid loitering (see text).
      if (N > 0 && N == a.length/4) resize(a.length/2);
      return item;
   }

   public Iterator<Item> iterator()
    {  return new ReverseArrayIterator();  }

   private class ReverseArrayIterator implements Iterator<Item>
   {  // Support LIFO iteration.
      private int i = N;
      public boolean hasNext() {  return i > 0;    }
      public    Item next()    {  return a[--i];   }
      public    void remove()  {                   }
   }
}
```

This generic, iterable implementation of our Stack API is a model for collection ADTs that keep items in an array. It resizes the array to keep the array size within a constant factor of the stack size.

Linked lists Now we consider the use of a fundamental data structure that is an appropriate choice for representing the data in a collection ADT implementation. This is our first example of building a data structure that is not directly supported by the Java language. Our implementation serves as a model for the code that we use for building more complex data structures throughout the book, so you should read this section carefully, even if you have experience working with linked lists.

> **Definition.** A *linked list* is a recursive data structure that is either empty (*null*) or a reference to a *node* having a generic item and 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. As with a recursive program, the concept of a recursive data structure can be a bit mindbending at first, but is of great value because of its simplicity.

Node record. With object-oriented programming, implementing linked lists is not difficult. We start with a *nested class* that defines the node abstraction:

```
private class Node
{
   Item item;
   Node next;
}
```

A `Node` has two instance variables: an `Item` (a parameterized type) and a `Node`. We define `Node` within the class where we want to use it, and make it `private` because it is not for use by clients. As with any data type, we create an object of type `Node` by invoking the (no-argument) constructor with `new Node()`. The result is a reference to a `Node` object whose instance variables are both initialized to the value `null`. The `Item` is a placeholder for any data that we might want to structure with a linked list (we will use Java's generic mechanism so that it can represent any reference type); the instance variable of type `Node` characterizes the linked nature of the data structure. To emphasize that we are just using the `Node` class to structure the data, we define no methods and we refer directly to the instance variables in code: if `first` is a variable associated with an object of type `Node`, we can refer to the instance variables with the code `first.item` and `first.next`. Classes of this kind are sometimes called *records*. They do not implement abstract data types because we refer directly to instance variables. However, `Node` and its client code are in the same class in all of our implementations and not accessible by clients of that class, so we still enjoy the benefits of data abstraction.

Building a linked list. Now, from the recursive definition, we can represent a linked list with a variable of type `Node` simply by ensuring that its value is either *null* or a reference to a `Node` whose `next` field is a reference to a linked list. 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();
```

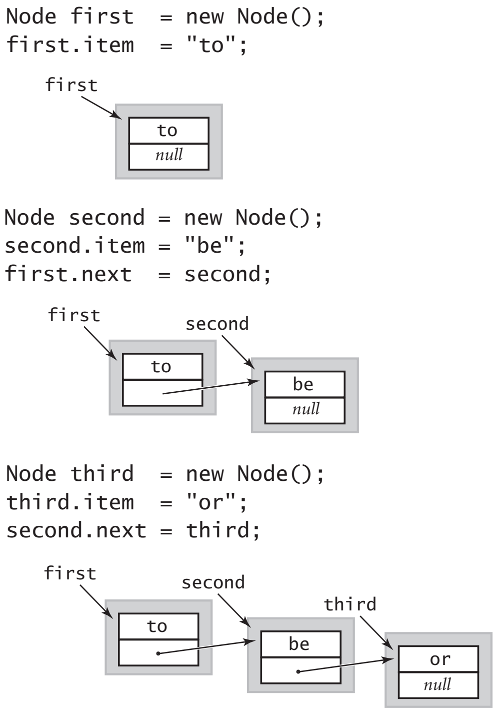
and set the item field in each of the nodes to the desired value (for simplicity, these examples assume that `Item` is `String`):

```
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;
```

(Note that `third.next` remains `null`, the value it was initialized to at the time of creation.)As a result, `third` is a linked list (it is a reference to a node that has a reference to `null`, which is the null reference to an empty linked list), and `second` is a linked list (it is a reference to a node that has a reference to `third`, which is a linked list), and `first` is a linked list (it is a reference to a node that has a reference to `second`, which is a linked list). The code that we will examine does these assignment statements in a different order, depicted in the diagram on this page.



```
Node first  = new Node();
first.item  = "to";
```

```
Node second = new Node();
second.item = "be";
first.next  = second;
```

```
Node third  = new Node();
third.item  = "or";
second.next = third;
```

Linking together a list

A LINKED LIST REPRESENTS A SEQUENCE of items. In the example just considered, `first` represents the sequence `to be or`. We can also use an array to represent a sequence of items. For example, we could use

```
String[] s = { "to", "be", "or" };
```

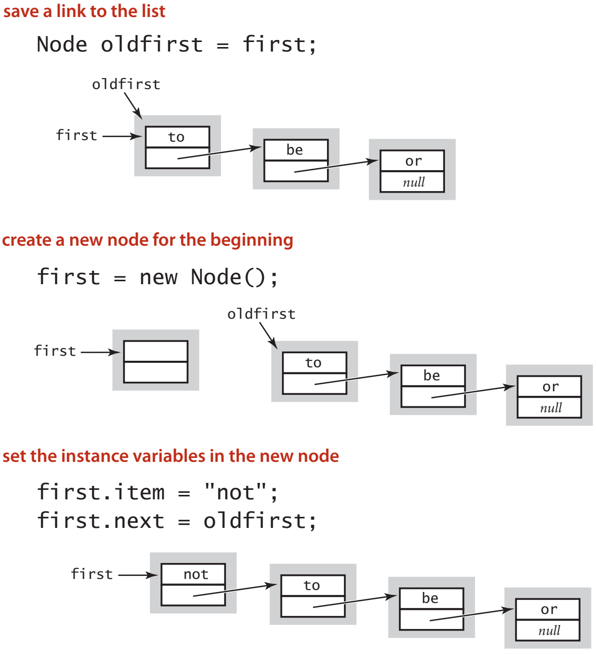
to represent the same sequence of strings. The difference is that it is easier to insert items into the sequence and to remove items from the sequence with linked lists. Next, we consider code to accomplish these tasks.

When tracing code that uses linked lists and other linked structures, we use a visual representation where

- We draw a rectangle to represent each object
- We put the values of instance variables within the rectangle
- We use arrows that point to the referenced objects to depict references

This visual representation captures the essential characteristic of linked lists. For economy, we use the term *links* to refer to node references. For simplicity, when item values are strings (as in our examples), we put the string within the object rectangle rather than the more accurate rendition depicting the string object and the character array that we discussed in SECTION 1.2. This visual representation allows us to focus on the links.

Insert at the beginning. First, 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`, and assign its `item` field to `not` and its `next` field to `oldfirst`. This code for inserting a node at the beginning of a linked list involves just a few assignment statements, so the amount of time that it takes is independent of the length of the list.



save a link to the list

```
Node oldfirst = first;
```

create a new node for the beginning

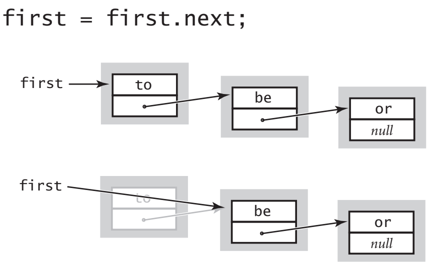
```
first = new Node();
```

set the instance variables in the new node

```
first.item = "not";
first.next = oldfirst;
```

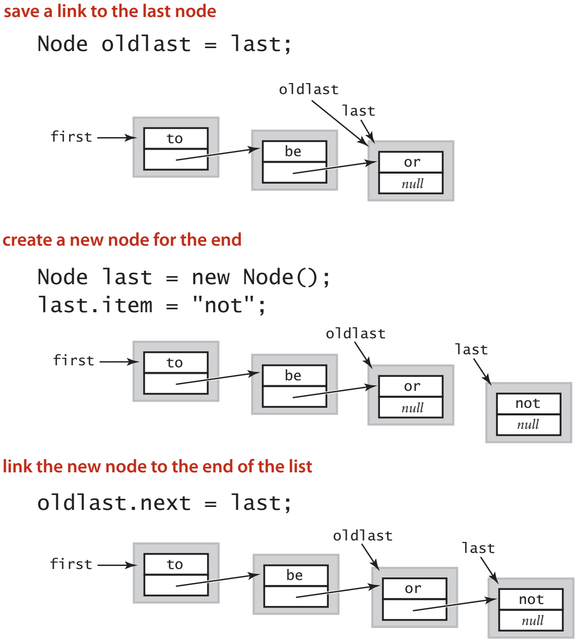
Inserting a new node at the beginning of a linked list

Remove from the beginning. Next, 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 variable of type `Item`) 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 Java memory management system eventually reclaims the memory it occupies. Again, this operation just involves one assignment statement, so its running time is independent of the length of the list.


```
first = first.next;
```

Removing the first node in a linked list

Insert at the end. How do we add a node to the *end* of a linked list? To do so, we need a link to the last node in the list, because that node's link has to be changed to reference a new node containing the item to be inserted. Maintaining an extra link is not something that should be taken lightly in linked-list code, because every method that modifies the list needs code to check whether that variable needs to be modified (and to make the necessary modifications). For example, the code that we just examined for removing the first node in the list might involve changing the reference to the last node in the list, since when there is only one node in the list, it is both the first one and the last one! Also, this code does not work (it follows a null link) in the case that the list is empty. Details like these make linked-list code notoriously difficult to debug.


save a link to the last node
```
Node oldlast = last;
```
create a new node for the end
```
Node last = new Node();
last.item = "not";
```
link the new node to the end of the list
```
oldlast.next = last;
```

Inserting a new node at the end of a linked list

Insert/remove at other positions. In summary, we have shown that we can implement the following operations on linked lists with just a few instructions, provided that we have access to both a link `first` to the first element in the list and a link `last` to the last element in the list:

- Insert at the beginning.
- Remove from the beginning.
- Insert at the end.

Other operations, such as the following, are not so easily handled:

- Remove a given node.
- Insert a new node before a given node.

For example, how can we remove the last node from a list? The link `last` is no help, because we need to set the link in the previous node in the list (the one with the same value as `last`) to `null`. In the absence of any other information, the only solution is to traverse the entire list looking for the node that links to `last` (see below and EXERCISE 1.3.19). Such a solution is undesirable because it takes time proportional to the length of the list. The standard solution to enable arbitrary insertions and deletions is to use a *doubly-linked list*, where each node has two links, one in each direction. We leave the code for these operations as an exercise (see EXERCISE 1.3.31). We do not need doubly linked lists for any of our implementations.

Traversal. To examine every item in an array, we use familiar code like the following loop for processing the items in an array `a[]`:

```
for (int i = 0; i < N; i++)
{
   // Process a[i].
}
```

There is a corresponding idiom for examining the items in a linked list: We initialize a loop index variable `x` to reference the first `Node` of the linked list. Then we find 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` and 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 code like the following loop for processing the items in a linked list whose first item is associated with the variable `first`:

```
for (Node x = first; x != null; x = x.next)
{
   // Process x.item.
}
```

This idiom is as natural as the standard idiom for iterating through the items in an array. In our implementations, we use it as the basis for iterators for providing client code the capability of iterating through the items, without having to know the details of the linked-list implementation.

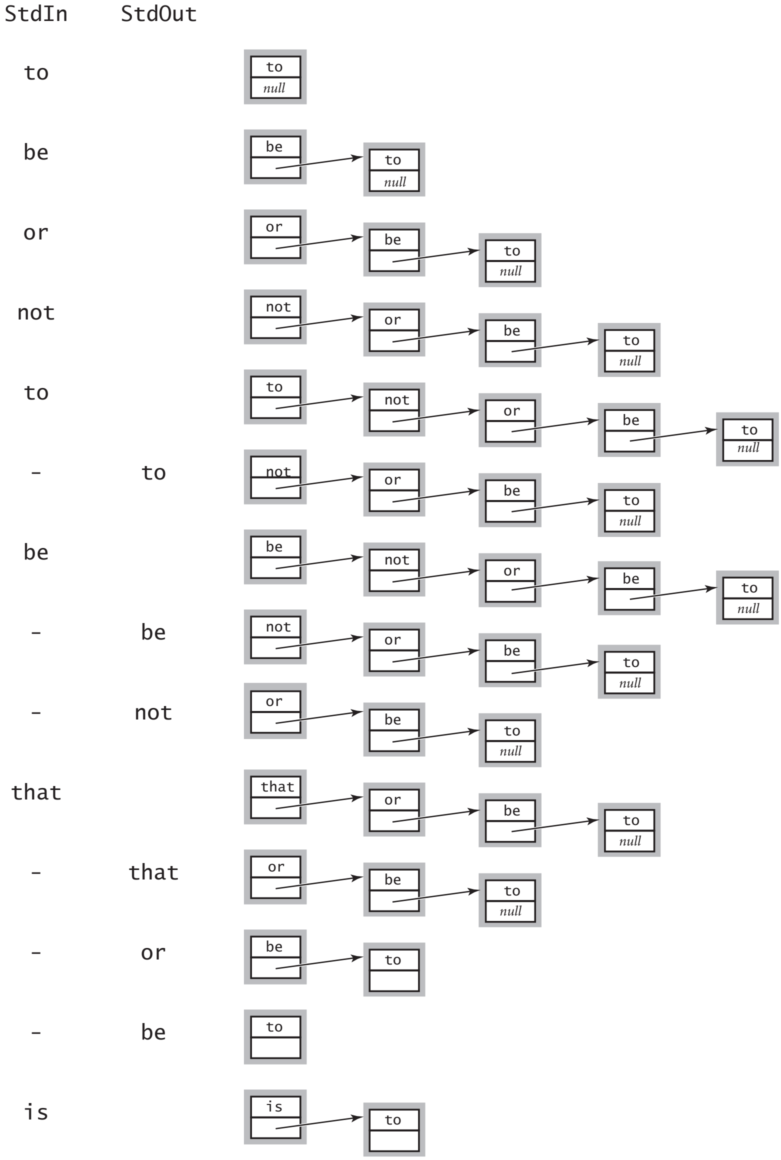
Stack implementation. Given these preliminaries, developing an implementation for our Stack API is straightforward, as shown in ALGORITHM 1.2 on page 149. It maintains the stack as a linked list, with the top of the stack at the beginning, referenced by an instance variable first. Thus, to push() an item, we add it to the beginning of the list, using the code discussed on page 144 and to pop() an item, we remove it from the beginning of the list, using the code discussed on page 145. To implement size(), we keep track of the number of items in an instance variable N, incrementing N when we push and decrementing N when we pop. To implement isEmpty() we check whether first is null (alternatively, we could check whether N is 0). The implementation uses the generic type Item—you can think of the code <Item> after the class name as meaning that any occurrence of Item in the implementation will be replaced by a client-supplied data-type name (see page 134). For now, we omit the code to support iteration, which we consider on page 155. A trace for the test client that we have been using is shown on the next page. This use of linked lists achieves our optimum design goals:

- It can be used for any type of data.
- The space required is always proportional to the size of the collection.
- The time per operation is always independent of the size of the collection.

This implementation is a prototype for many *algorithm* implementations that we consider. It defines the linked-list *data structure* and implements the client methods push() and pop() that achieve the specified effect with just a few lines of code. The algorithms and data structure go hand in hand. In this case, the code for the algorithm implementations is quite simple, but the properties of the data structure are not at all elementary, requiring explanations on the past several pages. This interaction between data structure definition and algorithm implementation is typical and is our focus in ADT implementations throughout this book.

```
public static void main(String[] args)
{  // Create a stack and push/pop strings as directed on StdIn.

   Stack<String> s = new Stack<String>();

   while (!StdIn.isEmpty())
   {
      String item = StdIn.readString();
      if (!item.equals("-"))
           s.push(item);
      else if (!s.isEmpty()) StdOut.print(s.pop() + " ");
   }

   StdOut.println("(" + s.size() + " left on stack)");
}
```

Test client for Stack



Trace of Stack development client

ALGORITHM 1.2 Pushdown stack (linked-list implementation)

```
public class Stack<Item> implements Iterable<Item>
{
   private Node first; // top of stack (most recently added node)
   private int N;      // number of items

   private class Node
   {  // nested class to define nodes
      Item item;
      Node next;
   }

   public boolean isEmpty() {  return first == null; }  // Or: N == 0.
   public int size()        {  return N; }

   public void push(Item item)
   {  // Add item to top of stack.
      Node oldfirst = first;
      first = new Node();
      first.item = item;
      first.next = oldfirst;
      N++;
   }

   public Item pop()
   {  // Remove item from top of stack.
      Item item = first.item;
      first = first.next;
      N--;
      return item;
   }

   // See page 155 for iterator() implementation.

   // See page 147 for test client main().

}
```

This generic Stack implementation is based on a linked-list data structure. It can be used to create stacks containing any type of data. To support iteration, add the highlighted code described for Bag on page 155.

```
% more tobe.txt
to be or not to - be - - that - - - is

% java Stack < tobe.txt
to be not that or be (2 left on stack)
```

Queue implementation. An implementation of our Queue API based on the linked-list data structure is also straightforward, as shown in ALGORITHM 1.3 on the facing page. It maintains the queue as a linked list in order from least recently to most recently added items, with the beginning of the queue referenced by an instance variable first and the end of the queue referenced by an instance variable last. Thus, to enqueue() an item, we add it to the end of the list (using the code discussed on page 145, augmented to set both first and last to refer to the new node when the list is empty) and to dequeue() an item, we remove it from the beginning of the list (using the same code as for pop() in Stack, augmented to update last when the list becomes empty). The implementations of size() and isEmpty() are the same as for Stack. As with Stack the implementation uses the generic type parameter Item, and we omit the code to support iteration, which we consider in our Bag implementation on page 155. A development client similar to the one we used for Stack is shown below, and the trace for this client is shown on the following page. This implementation uses the same *data structure* as does Stack—a linked list—but it implements different *algorithms* for adding and removing items, which make the difference between LIFO and FIFO for the client. Again, the use of linked lists achieves our optimum design goals: it can be used for any type of data, the space required is proportional to the number of items in the collection, and the time required per operation is always independent of the size of the collection.

```
public static void main(String[] args)
{  // Create a queue and enqueue/dequeue strings.
   Queue<String> q = new Queue<String>();

   while (!StdIn.isEmpty())
   {
      String item = StdIn.readString();
      if (!item.equals("-"))
           q.enqueue(item);
      else if (!q.isEmpty()) StdOut.print(q.dequeue() + " ");
   }

   StdOut.println("(" + q.size() + " left on queue)");
}
```

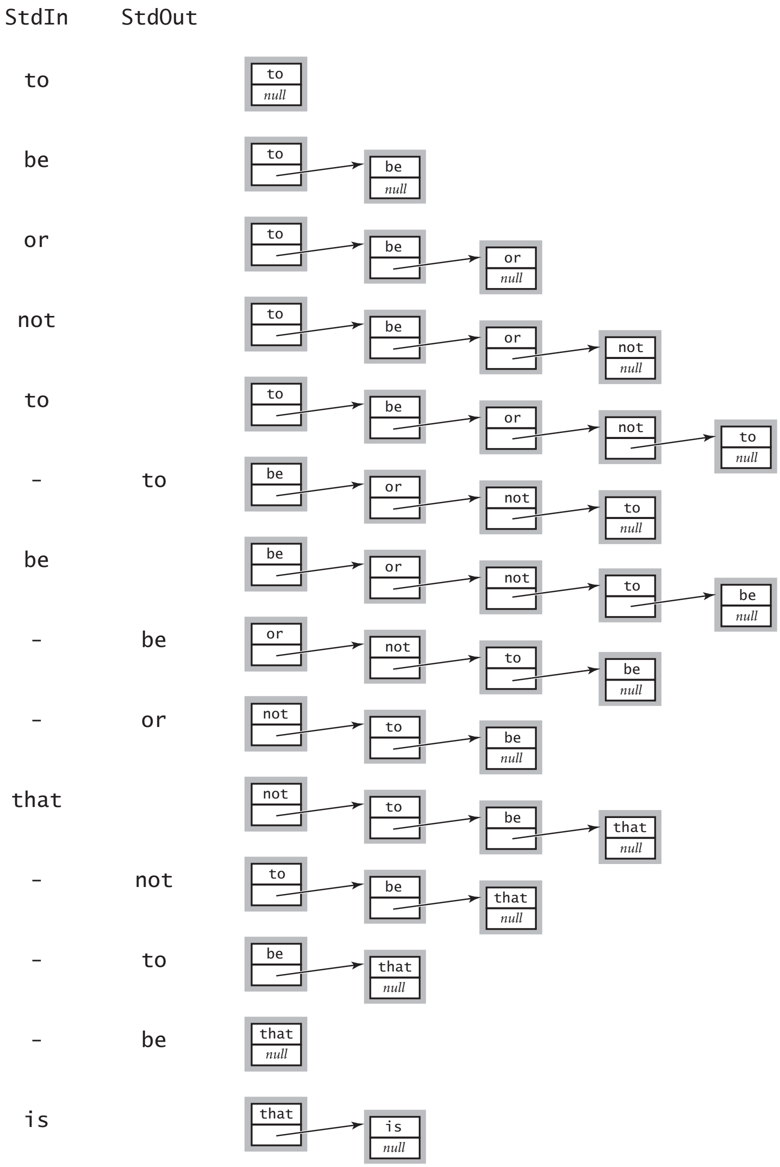
Test client for Queue

```
% more tobe.txt
to be or not to - be - - that - - - is

% java Queue < tobe.txt
to be or not to be (2 left on queue)
```

ALGORITHM 1.3 FIFO queue

```
public class Queue<Item> implements Iterable<Item>
{
   private Node first; // link to least recently added node
   private Node last;  // link to most recently added node
   private int N;      // number of items on the queue

   private class Node
   {  // nested class to define nodes
      Item item;
      Node next;
   }

   public boolean isEmpty() {  return first == null;  }  // Or: N == 0.
   public int size()        {  return N;  }

   public void enqueue(Item item)
   {  // Add item to the end of the list.
      Node oldlast = last;
      last = new Node();
      last.item = item;
      last.next = null;
      if (isEmpty()) first = last;
      else           oldlast.next = last;
      N++;
   }

   public Item dequeue()
   {  // Remove item from the beginning of the list.
      Item item = first.item;
      first = first.next;
      if (isEmpty()) last = null;
      N--;
      return item;
   }

   // See page 155 for iterator() implementation.

   // See page 150 for test client main().

}
```

This generic `Queue` implementation is based on a linked-list data structure. It can be used to create queues containing any type of data. To support iteration, add the highlighted code described for `Bag` on page 155.



Trace of Queue development client

LINKED LISTS ARE A FUNDAMENTAL ALTERNATIVE to arrays for structuring a collection of data. From a historical perspective, this alternative has been available to programmers for many decades. Indeed, a landmark in the history of programming languages was the development of LISP by John McCarthy in the 1950s, where linked lists are the primary structure for programs and data. Programming with linked lists presents all sorts of challenges and is notoriously difficult to debug, as you can see in the exercises. In modern code, the use of safe pointers, automatic garbage collection (see page 111), and ADTs allows us to encapsulate list-processing code in just a few classes such as the ones presented here.

Bag implementation. Implementing our `Bag` API using a linked-list data structure is simply a matter of changing the name of `push()` in `Stack` to `add()` and removing the implementation of `pop()`, as shown in ALGORITHM 1.4 on the facing page (doing the same for `Queue` would also be effective but requires a bit more code). This implementation also highlights the code needed to make `Stack`, `Queue`, and `Bag` all iterable, by traversing the list. For `Stack` the list is in LIFO order; for `Queue` it is in FIFO order; and for `Bag` it happens to be in LIFO order, but the order is not relevant. As detailed in the highlighted code in ALGORITHM 1.4, to implement iteration in a collection, the first step is to include

```
import java.util.Iterator;
```

so that our code can refer to Java's `Iterator` interface. The second step is to add

```
implements Iterable<Item>
```

to the class declaration, a promise to provide an `iterator()` method. The `iterator()` method itself simply returns an object from a class that implements the `Iterator` interface:

```
public Iterator<Item> iterator()
{  return new ListIterator();  }
```

This code is a promise to implement a class that implements the `hasNext()`, `next()`, and `remove()` methods that are called when a client uses the *foreach* construct. To implement these methods, the nested class `ListIterator` in ALGORITHM 1.4 maintains an instance variable `current` that keeps track of the current node on the list. Then the `hasNext()` method tests if `current` is `null`, and the `next()` method saves a reference to the current item, updates `current` to refer to the next node on the list, and returns the saved reference.

ALGORITHM 1.4 Bag

```
import java.util.Iterator;

public class Bag<Item> implements Iterable<Item>
{
   private Node first;  // first node in list

   private class Node
   {
       Item item;
       Node next;
   }

   public void add(Item item)
   {  // same as push() in Stack
      Node oldfirst = first;
      first = new Node();
      first.item = item;
      first.next = oldfirst;
   }

   public Iterator<Item> iterator()
   {  return new ListIterator();  }

   private class ListIterator implements Iterator<Item>
   {
       private Node current = first;

       public boolean hasNext()
       {  return current != null;  }

       public void remove() { }

       public Item next()
       {
           Item item = current.item;
           current = current.next;
           return item;
       }
   }
}
```

This Bag implementation maintains a linked list of the items provided in calls to add(). Code for isEmpty() and size() is the same as in Stack and is omitted. The iterator traverses the list, maintaining the current node in current. We can make Stack and Queue iterable by adding the code highlighted in red to ALGORITHMS 1.1 and 1.2, because they use the same underlying data structure and Stack and Queue maintain the list in LIFO and FIFO order, respectively.

Overview The implementations of bags, queues, and stacks that support generics and iteration that we have considered in this section provide a level of abstraction that allows us to write compact client programs that manipulate collections of objects. Detailed understanding of these ADTs is important as an introduction to the study of algorithms and data structures for three reasons. First, we use these data types as building blocks in higher-level data structures throughout this book. Second, they illustrate the interplay between data structures and algorithms and the challenge of simultaneously achieving natural performance goals that may conflict. Third, the focus of several of our implementations is on ADTs that support more powerful operations on collections of objects, and we use the implementations here as starting points.

Data structures. We now have two ways to represent collections of objects, arrays and linked lists. Arrays are built in to Java; linked lists are easy to build with standard Java records. These two alternatives, often referred to as *sequential allocation* and *linked allocation*, are fundamental. Later in the book, we develop ADT implementations that combine and extend these basic structures in numerous ways. One important extension is to data structures with multiple links. For example, our focus in SECTIONS 3.2 and 3.3 is on data structures known as *binary trees* that are built from nodes that each have *two* links. Another important extension is to *compose* data structures: we can have a bag of stacks, a queue of arrays, and so forth. For example, our focus in CHAPTER 4 is on graphs, which we represent as arrays of bags. It is very easy to define data structures of arbitrary complexity in this way: one important reason for our focus on abstract data types is an attempt to control such complexity.

data structure	advantage	disadvantage
array	index provides immediate access to any item	need to know size on initialization
linked list	uses space proportional to size	need reference to access an item

Fundamental data structures

Our treatment of bags, queues, and stacks in this section is a prototypical example of the approach that we use throughout this book to describe data structures and algorithms. In approaching a new applications domain, we identify computational challenges and use data abstraction to address them, proceeding as follows:

- Specify an API.
- Develop client code with reference to specific applications.
- Describe a data structure (representation of the set of values) that can serve as the basis for the *instance variables* in a class that will implement an ADT that meets the specification in the API.
- Describe algorithms (approaches to implementing the set of operations) that can serve as the basis for implementing the *instance methods* in the class.
- Analyze the performance characteristics of the algorithms.

In the next section, we consider this last step in detail, as it often dictates which algorithms and implementations can be most useful in addressing real-world applications.

data structure	section	ADT	representation
parent-link tree	1.5	`UnionFind`	array of integers
binary search tree	3.2, 3.3	`BST`	two links per node
string	5.1	`String`	array, offset, and length
binary heap	2.4	`PQ`	array of objects
hash table (separate chaining)	3.4	`SeparateChainingHashST`	arrays of linked lists
hash table (linear probing)	3.4	`LinearProbingHashST`	two arrays of objects
graph adjacency lists	4.1, 4.2	`Graph`	array of `Bag` objects
trie	5.2	`TrieST`	node with array of links
ternary search trie	5.3	`TST`	three links per node

Examples of data structures developed in this book

Q&A

Q. Not all programming languages have generics, even early versions of Java. What are the alternatives?

A. One alternative is to maintain a different implementation for each type of data, as mentioned in the text. Another is to build a stack of `Object` values, then cast to the desired type in client code for `pop()`. The problem with this approach is that type mismatch errors cannot be detected until run time. But with generics, if you write code to push an object of the wrong type on the stack, like this:

```
Stack<Apple> stack = new Stack<Apple>();
Apple  a = new Apple();
...
Orange b = new Orange();
...
stack.push(a);
...
stack.push(b);     // compile-time error
```

you will get a compile-time error:

```
push(Apple) in Stack<Apple> cannot be applied to (Orange)
```

This ability to discover such errors at compile time is reason enough to use generics.

Q. Why does Java disallow generic arrays?

A. Experts still debate this point. You might need to become one to understand it! For starters, learn about *covariant arrays* and *type erasure*.

Q. How do I create an array of stacks of strings?

A. Use a cast, such as the following:

```
Stack<String>[] a = (Stack<String>[]) new Stack[N];
```

Warning: This cast, in client code, is different from the one described on page 134. You might have expected to use `Object` instead of `Stack`. When using generics, Java checks for type safety at compile time, but throws away that information at run time, so it is left with `Stack<Object>[]` or just `Stack[]`, for short, which we must cast to `Stack<String>[]`.

Q. What happens if my program calls `pop()` for an empty stack?

A. It depends on the implementation. For our implementation on page 149, you will get a `NullPointerException`. In our implementations on the booksite, we throw a runtime exception to help users pinpoint the error. Generally, including as many such checks as possible is wise in code that is likely to be used by many people.

Q. Why do we care about resizing arrays, when we have linked lists?

A. We will see several examples of ADT implementations that need to use arrays to perform other operations that are not easily supported with linked lists. `ResizingArrayStack` is a model for keeping their memory usage under control.

Q. Why declare `Node` as a nested class? Why `private`?

A. By declaring the nested class `Node` to be `private`, we restrict access to methods and instance variables 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 the instance variables `public` or `private`. *Note for experts*: 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. When I type `javac Stack.java` to run ALGORITHM 1.2 and similar programs, I find `Stack.class` *and* a file `Stack$Node.class.` What is the purpose of that second one?

A. That file is for the inner class `Node`. Java's naming convention is to use `$` to separate the name of the outer class from the inner class.

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 just as libraries of all the operations we can imagine, but also as a mechanism 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

Q & A *(continued)*

want. The `java.util.Stack` API is an example of a *wide interface*, which we generally strive to avoid.

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. What should the `Stack` iterator do if the client calls `push()` or `pop()` during iterator?

A. Throw a `java.util.ConcurrentModificationException` to make it a *fail-fast iterator*. See 1.3.50.

Q. Can I use a *foreach* loop with arrays?

A. Yes (even though arrays do not implement the `Iterable` interface). The following one-liner prints out the command-line arguments:

```
public static void main(String[] args)
{  for (String s : args) StdOut.println(s);  }
```

Q. Can I use a *foreach* loop with strings?

A. No. `String` does not implement `Iterable`.

Q. Why not have a single `Collection` data type that implements methods to add items, remove the most recently inserted, remove the least recently inserted, remove random, iterate, return the number of items in the collection, and whatever other operations we might desire? Then we could get them all implemented in a single class that could be used by many clients.

A. Again, this is an example of a *wide interface*. Java has such implementations in its `java.util.ArrayList` and `java.util.LinkedList` classes. One reason to avoid them is that it there is no assurance that all operations are implemented efficiently. Throughout this book, we use APIs as starting points for designing efficient algorithms and data structures, which is certainly easier to do for interfaces with just a few operations as opposed to an interface with many operations. Another reason to insist on narrow interfaces is that they enforce a certain discipline on client programs, which makes client code much easier to understand. If one client uses `Stack<String>` and another uses `Queue<Transaction>`, we have a good idea that the LIFO discipline is important to the first and the FIFO discipline is important to the second.

EXERCISES

1.3.1 Add a method `isFull()` to `FixedCapacityStackOfStrings`.

1.3.2 Give the output printed by `java Stack` for the input

```
it was - the best - of times - - - it  was - the - -
```

1.3.3 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 onto the stack; the pop operations print out the return values. 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`

1.3.4 Write a stack client `Parentheses` that reads in a text stream from standard input and uses a stack to determine whether its parentheses are properly balanced. For example, your program should print `true` for `[()]{}{[()()]()}` and `false` for `[(])`.

1.3.5 What does the following code fragment print when `N` is `50`? Give a high-level description of what it does when presented with a positive integer `N`.

```
Stack<Integer> stack = new Stack<Integer>();
while (N > 0)
{
   stack.push(N % 2);
   N = N / 2;
}
for (int d : stack) StdOut.print(d);
StdOut.println();
```

Answer: Prints the binary representation of `N` (`110010` when `N` is `50`).

EXERCISES *(continued)*

1.3.6 What does the following code fragment do to the queue q?

```
Stack<String> stack = new Stack<String>();
while (!q.isEmpty())
   stack.push(q.dequeue());
while (!stack.isEmpty())
   q.enqueue(stack.pop());
```

1.3.7 Add a method peek() to Stack that returns the most recently inserted item on the stack (without popping it).

1.3.8 Give the contents and size of the array for DoublingStackOfStrings with the input

```
it was - the best - of times - - - it was - the - -
```

1.3.9 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 ) )
```

1.3.10 Write a filter InfixToPostfix that converts an arithmetic expression from infix to postfix.

1.3.11 Write a program EvaluatePostfix 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.

1.3.12 Write an iterable Stack *client* that has a static method 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.

1.3.13 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 onto 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`

1.3.14 Develop a class `ResizingArrayQueueOfStrings` that implements the queue abstraction with a fixed-size array, and then extend your implementation to use array resizing to remove the size restriction.

1.3.15 Write a `Queue` client that takes a command-line argument `k` and prints the `k`th from the last string found on standard input (assuming that standard input has `k` or more strings).

1.3.16 Using `readInts()` on page 126 as a model, write a static method `readDates()` for `Date` that reads dates from standard input in the format specified in the table on page 119 and returns an array containing them.

1.3.17 Do EXERCISE 1.3.16 for `Transaction`.

LINKED-LIST EXERCISES

This list of exercises is intended to give you experience in working with linked lists. Suggestion: make drawings using the visual representation described in the text.

1.3.18 Suppose `x` is a linked-list node and not the last node on the list. What is the effect of the following code fragment?

```
x.next = x.next.next;
```

Answer: Deletes from the list the node immediately following `x`.

1.3.19 Give a code fragment that removes the last node in a linked list whose first node is `first`.

1.3.20 Write a method `delete()` that takes an `int` argument `k` and deletes the `k`th element in a linked list, if it exists.

1.3.21 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.

1.3.22 Suppose that `x` is a linked list `Node`. What does the following code fragment do?

```
t.next = x.next;
x.next = t;
```

Answer: Inserts node `t` immediately after node `x`.

1.3.23 Why does the following code fragment not do the same thing 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!

1.3.24 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).

1.3.25 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).

1.3.26 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.

1.3.27 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.

1.3.28 Develop a recursive solution to the previous question.

1.3.29 Write a `Queue` implementation that uses a *circular* linked list, which is the same as a linked list except that no links are *null* and the value of `last.next` is `first` whenever the list is not empty. Keep only one `Node` instance variable (`last`).

1.3.30 Write a function that takes the first `Node` in a linked list as argument and (destructively) reverses the list, returning the first `Node` in the result.

Iterative solution: To accomplish this task, 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 Node reverse(Node x)
{
   Node first   = x;
   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

LINKED-LIST EXERCISES *(continued)*

nodes) and the boundary cases (dealing with the first or last items). This is usually much trickier than handling the normal cases.

Recursive solution: Assuming the linked list has N nodes, we recursively reverse the last $N-1$ nodes, and then carefully append the first node to the end.

```
public Node reverse(Node first)
{
   if (first == null) return null;
   if (first.next == null) return first;
   Node second = first.next;
   Node rest = reverse(second);
   second.next = first;
   first.next  = null;
   return rest;
}
```

1.3.31 Implement a nested class `DoubleNode` for building doubly-linked lists, where each node contains a reference to the item preceding it and the item following it in the list (`null` if there is no such item). Then implement static methods for the following tasks: insert at the beginning, insert at the end, remove from the beginning, remove from the end, insert before a given node, insert after a given node, and remove a given node.

CREATIVE PROBLEMS

1.3.32 *Steque.* A stack-ended queue or *steque* is a data type that supports *push*, *pop*, and *enqueue*. Articulate an API for this ADT. Develop a linked-list-based implementation.

1.3.33 *Deque.* A double-ended queue or *deque* (pronounced "deck") is like a stack or a queue but supports adding and removing items at both ends. A deque stores a collection of items and supports the following API:

`public class Deque<Item> implements Iterable<Item>`

	`Deque()`	*create an empty deque*
`boolean`	`isEmpty()`	*is the deque empty?*
`int`	`size()`	*number of items in the deque*
`void`	`pushLeft(Item item)`	*add an item to the left end*
`void`	`pushRight(Item item)`	*add an item to the right end*
`Item`	`popLeft()`	*remove an item from the left end*
`Item`	`popRight()`	*remove an item from the right end*

API for a generic double-ended queue

Write a class `Deque` that uses a doubly-linked list to implement this API and a class `ResizingArrayDeque` that uses a resizing array.

1.3.34 *Random bag.* A *random bag* stores a collection of items and supports the following API:

`public class RandomBag<Item> implements Iterable<Item>`

	`RandomBag()`	*create an empty random bag*
`boolean`	`isEmpty()`	*is the bag empty?*
`int`	`size()`	*number of items in the bag*
`void`	`add(Item item)`	*add an item*

API for a generic random bag

Write a class `RandomBag` that implements this API. Note that this API is the same as for `Bag`, except for the adjective *random*, which indicates that the iteration should provide

CREATIVE PROBLEMS *(continued)*

the items in *random* order (all *N*! permutations equally likely, for each iterator). *Hint*: Put the items in an array and randomize their order in the iterator's constructor.

1.3.35 *Random queue.* A *random queue* stores a collection of items and supports the following API:

`public class RandomQueue<Item>`

	`RandomQueue()`	*create an empty random queue*
`boolean`	`isEmpty()`	*is the queue empty?*
`void`	`enqueue(Item item)`	*add an item*
`Item`	`dequeue()`	*remove and return a random item (sample without replacement)*
`Item`	`sample()`	*return a random item, but do not remove (sample with replacement)*

API for a generic random queue

Write a class `RandomQueue` that implements this API. *Hint*: Use an array representation (with resizing). To remove an item, swap one at a random position (indexed `0` through `N-1`) with the one at the last position (index `N-1`). Then delete and return the last object, as in `ResizingArrayStack`. Write a client that deals bridge hands (13 cards each) using `RandomQueue<Card>`.

1.3.36 *Random iterator.* Write an iterator for `RandomQueue<Item>` from the previous exercise that returns the items in random order.

1.3.37 *Josephus problem.* In the Josephus problem from antiquity, *N* people are in dire straits and agree to the following strategy to reduce the population. They arrange themselves in a circle (at positions numbered from 0 to *N*–1) and proceed around the circle, eliminating every *M*th person until only one person is left. Legend has it that Josephus figured out where to sit to avoid being eliminated. Write a `Queue` client `Josephus` that takes *N* and *M* from the command line and prints out the order in which people are eliminated (and thus would show Josephus where to sit in the circle).

```
% java Josephus 7 2
1 3 5 0 4 2 6
```

1.3.38 *Delete* k*th element.* Implement a class that supports the following API:

`public class GeneralizedQueue<Item>`

	`GeneralizedQueue()`	*create an empty queue*
`boolean`	`isEmpty()`	*is the queue empty?*
`void`	`insert(Item x)`	*add an item*
`Item`	`delete(int k)`	*delete and return the* `k`*th least recently inserted item*

API for a generic generalized queue

First, develop an implementation that uses an array implementation, and then develop one that uses a linked-list implementation. *Note*: the algorithms and data structures that we introduce in Chapter 3 make it possible to develop an implementation that can guarantee that both `insert()` and `delete()` take time prortional to the logarithm of the number of items in the queue—see Exercise 3.5.27.

1.3.39 *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 for 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. Develop an API for a `RingBuffer` and an implementation that uses an array representation (with circular wrap-around).

1.3.40 *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 reinsert it at the beginning. Name your program `MoveToFront`: it implements the well-known *move-to-front* strategy, which is useful for caching, data compression, and many other applications where items that have been recently accessed are more likely to be reaccessed.

1.3.41 *Copy a queue.* Create a new constructor so that

```
Queue<Item> r = new Queue<Item>(q);
```

makes `r` a reference to a new and independent copy of the queue `q`. You should be able to push and pop from either `q` or `r` without influencing the other. *Hint*: Delete all of the elements from `q` and add these elements to both `q` and `r`.

CREATIVE PROBLEMS *(continued)*

1.3.42 *Copy a stack.* Create a new constructor for the linked-list implementation of `Stack` so that

```
Stack<Item> t = new Stack<Item>(s);
```

makes `t` a reference to a new and independent copy of the stack `s`.

1.3.43 *Listing files.* A folder is a list of files and folders. Write a program that takes the name of a folder as a command-line argument and prints out all of the files contained in that folder, with the contents of each folder recursively listed (indented) under that folder's name. *Hint*: Use a queue, and see `java.io.File`.

1.3.44 *Text editor buffer.* Develop a data type for a buffer in a text editor that implements the following API:

`public class Buffer`

	`Buffer()`	*create an empty buffer*
`void`	`insert(char c)`	*insert c at the cursor position*
`char`	`delete()`	*delete and return the character at the cursor*
`void`	`left(int k)`	*move the cursor k positions to the left*
`void`	`right(int k)`	*move the cursor k positions to the right*
`int`	`size()`	*number of characters in the buffer*

API for a text buffer

Hint: Use two stacks.

1.3.45 *Stack generability.* Suppose that we have a sequence of intermixed *push* and *pop* operations as with our test stack client, where the integers `0`, `1`, ..., `N-1` in that order (*push* directives) are intermixed with N minus signs (*pop* directives). Devise an algorithm that determines whether the intermixed sequence causes the stack to underflow. (You may use only an amount of space independent of N—you cannot store the integers in a data structure.) Devise a linear-time algorithm that determines whether a given permutation can be generated as output by our test client (depending on where the *pop* directives occur).

Solution: The stack does not overflow unless there exists an integer k such that the first k pop operations occur before the first k push operations. If a given permutation can be generated, it is uniquely generated as follows: if the next integer in the output permutation is in the top of the stack, pop it; otherwise, push it onto the stack.

1.3.46 *Forbidden triple for stack generability.* Prove that a permutation can be generated by a stack (as in the previous question) if and only if it has no forbidden triple (a, b, c) such that $a < b < c$ with c first, a second, and b third (possibly with other intervening integers between c and a and between a and b).

Partial solution: Suppose that there is a forbidden triple (a, b, c). Item c is popped before a and b, but a and b are pushed before c. Thus, when c is pushed, both a and b are on the stack. Therefore, a cannot be popped before b.

1.3.47 *Catenable queues, stacks, or steques.* Add an extra operation *catenation* that (destructively) concatenates two queues, stacks, or steques (see EXERCISE 1.3.32). *Hint*: Use a circular linked list, maintaining a pointer to the last item.

1.3.48 *Two stacks with a deque.* Implement two stacks with a single deque so that each operation takes a constant number of deque operations (see EXERCISE 1.3.33).

1.3.49 *Queue with three stacks.* Implement a queue with three stacks so that each queue operation takes a constant (worst-case) number of stack operations. *Warning*: high degree of difficulty.

1.3.50 *Fail-fast iterator.* Modify the iterator code in `Stack` to immediately throw a `java.util.ConcurrentModificationException` if the client modifies the collection (via `push()` or `pop()`) during iteration? b).

Solution: Maintain a counter that counts the number of `push()` and `pop()` operations. When creating an iterator, store this value as an `Iterator` instance variable. Before each call to `hasNext()` and `next()`, check that this value has not changed since construction of the iterator; if it has, throw the exception.