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

Library of Congress Cataloging-in-Publication Data
Sedgewick, Robert, 1946-
Algorithms / Robert Sedgewick, Kevin Wayne.—4th ed.
p. cm.
Includes index.
ISBN 978-0-321-57351-3 (hardcover : alk. paper) 1. Computer algorithms—Textbooks.
I. Wayne, Kevin Daniel, 1971- II. Title.
QA76.9.A43S429 2011
005.1—dc22
2011000707

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.
Second printing, January 2012

To Adam, Andrew, Brett, Robbie
and especially Linda

To Jackie and Alex

CONTENTS

(*continued on next page*)

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

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-driven 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.

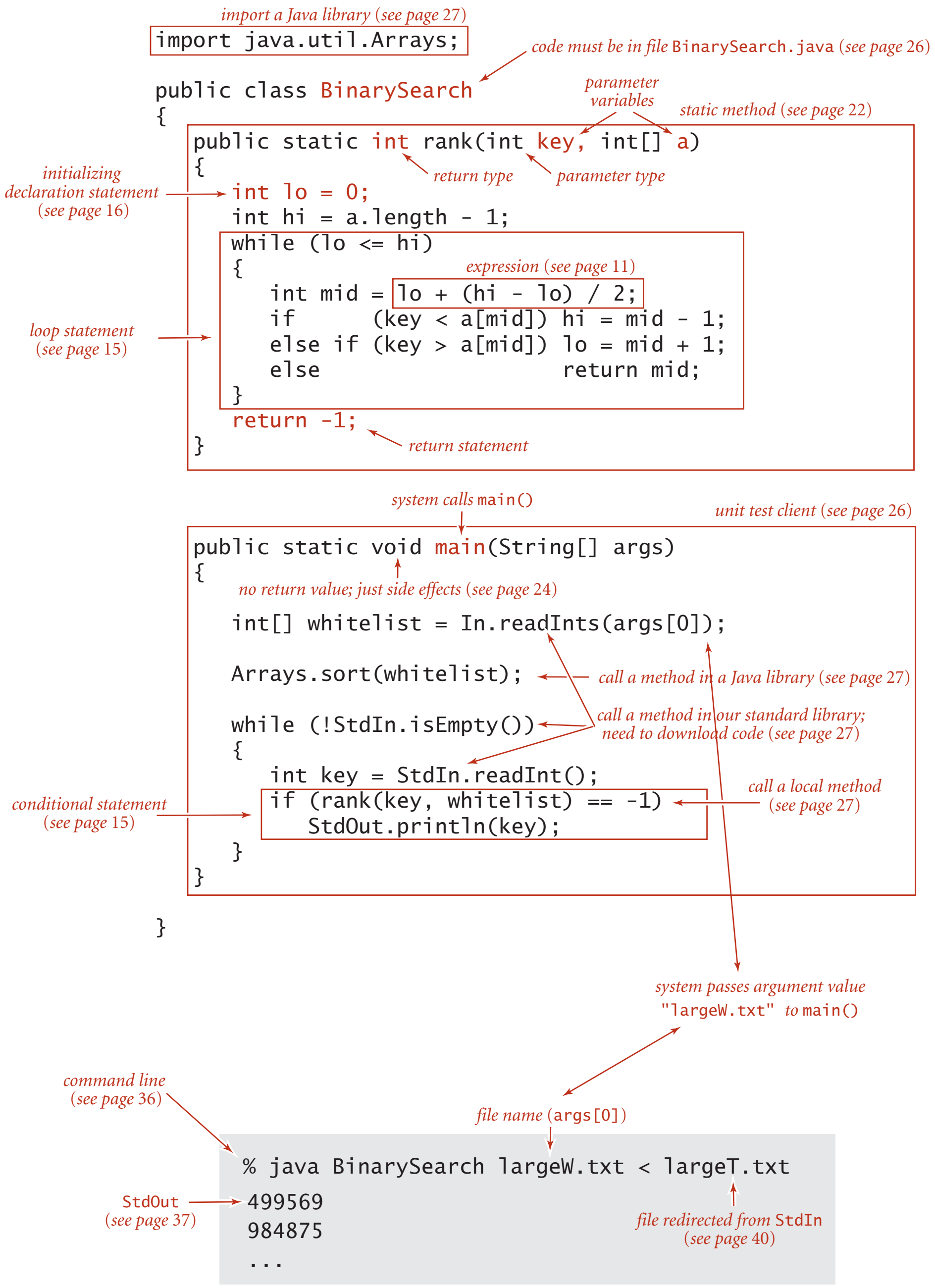
These are important and significant advantages over the alternatives of working with English-language descriptions of algorithms.

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 "largeW.txt" to main()
command line (see page 36)
file name (args[0])
StdOut (see page 37)
file redirected 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 seven 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`
	`double` `2.0  1.0e-15  3.14`
	`boolean` `true  false`
	`char` `'a'  '+'  '9'  '\n'`
		source-code representation of a value

expression	`int` `lo + (hi - lo)/2`
	`double` `1.0e-15 * t`
	`boolean` `lo <= hi`
		a literal, a variable, or a sequence of operations on literals and/or variables that produces a value

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 provided 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`; both have 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 the value *before* the increment/decrement, not after.
- Other compound operators: 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)

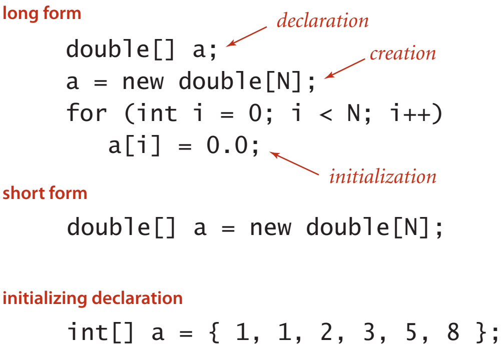
Java statements

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.



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

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

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

Declaring, creating, and initializing an array

Default array initialization. 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`.

Initializing declaration. 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];`

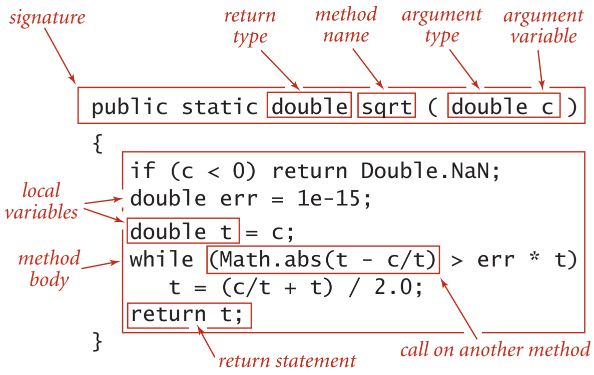
reverse the elements within an array	`int N = a.length;`
	`for (int i = 0; i < N/2; i++)`
	`{`
	`   double temp = a[i];`
	`   a[i] = a[N-1-i];`
	`   a[N-i-1] = temp;`
	`}`

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 < 0.0) 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 from 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"`	+ (concatenate)	`"Hi, " + "Bob"`	`"Hi, Bob"`
		`"Hello"`		`"12" + "34"`	`"1234"`
		`"2.5"`		`"1" + "+" + "2"`	`"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 `""`.

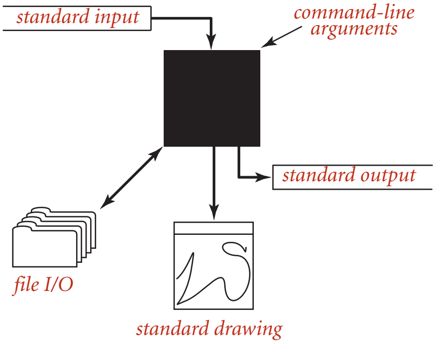
Command-line arguments. One important use of strings in Java programming is to enable a mechanism for passing information from the command line to the program. The mechanism is simple. When you type the `java` command followed by a library name followed by a sequence of strings, the Java system invokes 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.



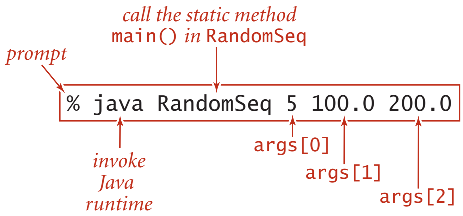
A bird's-eye view of a Java program

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).



```
% java RandomSeq 5 100.0 200.0
```

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
```