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.

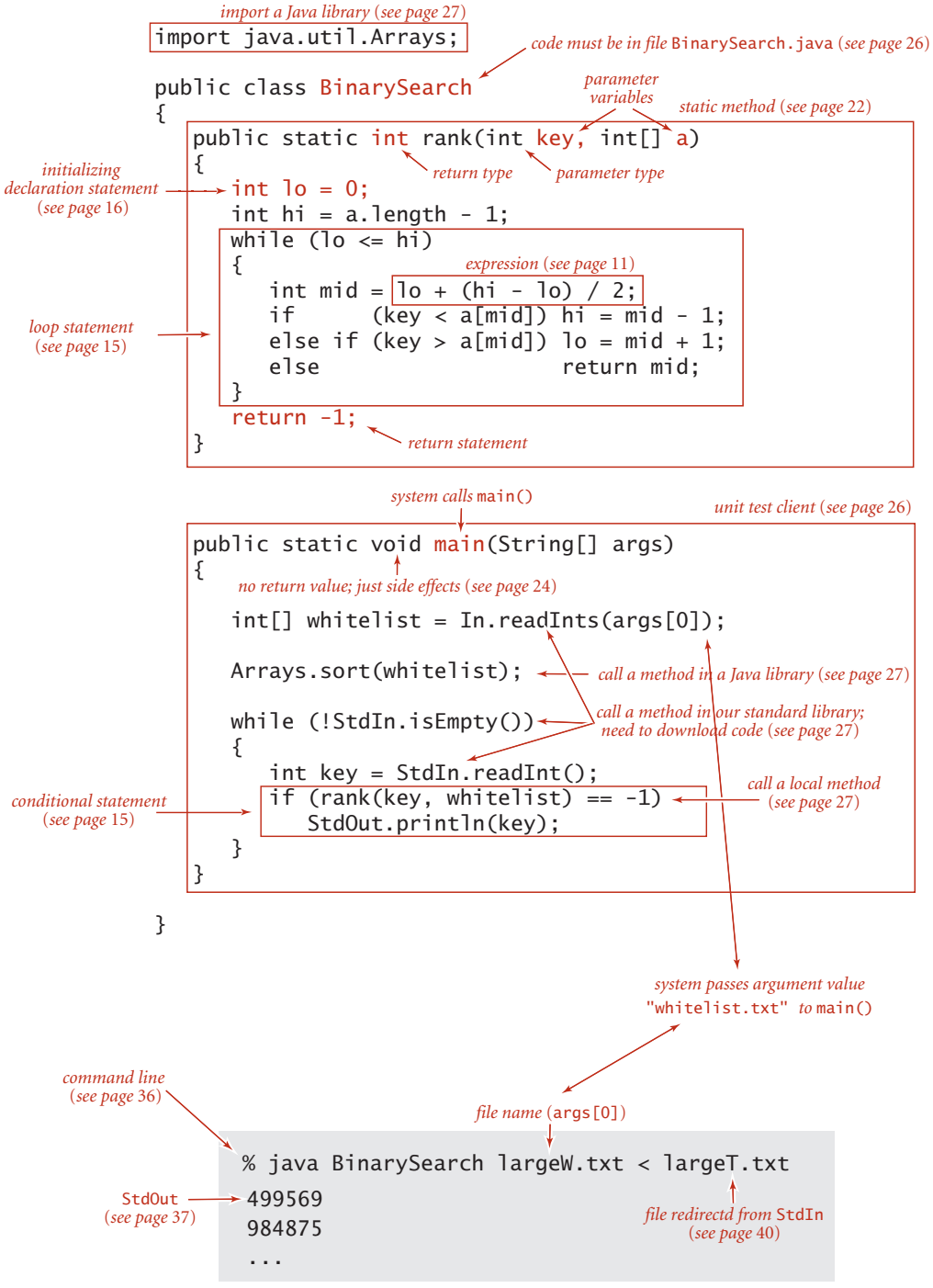
These are important and significant advantages over the alternatives of working with English-language descriptions of algorithms.

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.



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

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)
		`-` (subtract)
		`*` (multiply)
		`/` (divide)
		`%` (remainder)
			`5 + 3`
			`5 - 3`
			`5 * 3`
			`5 / 3`
			`5 % 3`
				`8`
				`2`
				`15`
				`1`
				`2`

`double`	double-precision real numbers (64-bit IEEE 754 standard)	`+` (add)
		`-` (subtract)
		`*` (multiply)
		`/` (divide)
			`3.141 - .03`
			`2.0 - 2.0e-7`
			`100 * .015`
			`6.02e23 / 2.0`
				`3.111`
				`1.9999998`
				`1.5`
				`3.01e23`

`boolean`	`true` or `false`	`&&` (and)
		`||` (or)
		`!` (not)
		`^` (xor)
			`true && false`
			`false || true`
			`!false`
			`true ^ true`
				`false`
				`true`
				`true`
				`false`

`char`	characters (16-bit)	[*arithmetic operations, rarely used*]		

Primitive data types in Java

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)
		`-` (subtract)
		`*` (multiply)
		`/` (divide)
		`%` (remainder)
			`5 + 3`
			`5 - 3`
			`5 * 3`
			`5 / 3`
			`5 % 3`
				`8`
				`2`
				`15`
				`1`
				`2`

`double`	double-precision real numbers (64-bit IEEE 754 standard)	`+` (add)
		`-` (subtract)
		`*` (multiply)
		`/` (divide)
			`3.141 - .03`
			`2.0 - 2.0e-7`
			`100 * .015`
			`6.02e23 / 2.0`
				`3.111`
				`1.9999998`
				`1.5`
				`3.01e23`

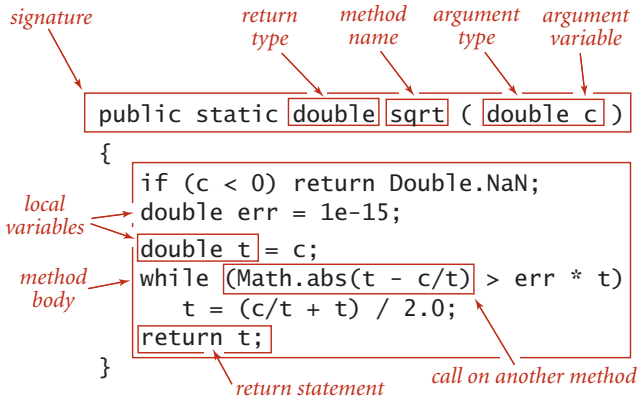
`boolean`	`true` or `false`	`&&` (and)
		`||` (or)
		`!` (not)
		`^` (xor)
			`true && false`
			`false || true`
			`!false`
			`true ^ true`
				`false`
				`true`
				`true`
				`false`

`char`	characters (16-bit)	[*arithmetic operations, rarely used*]		

Primitive data types in Java

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.



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).

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

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)

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

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.

standard Java system types in java.lang

`Integer`	`int` *wrapper*
`Double`	`double` *wrapper*
`String`	*indexed* `char`*s*
`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)*

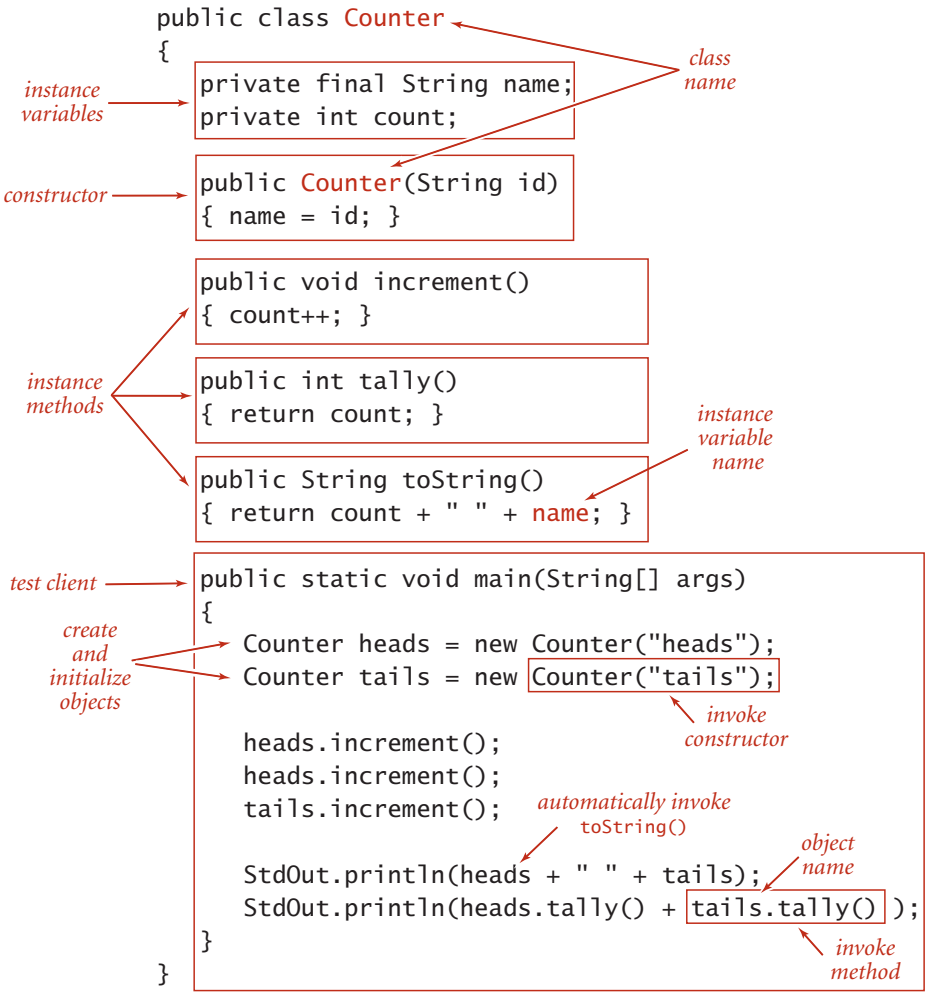
data-oriented graph types

`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



Anatomy of a class that defines a data type

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; }

}
```

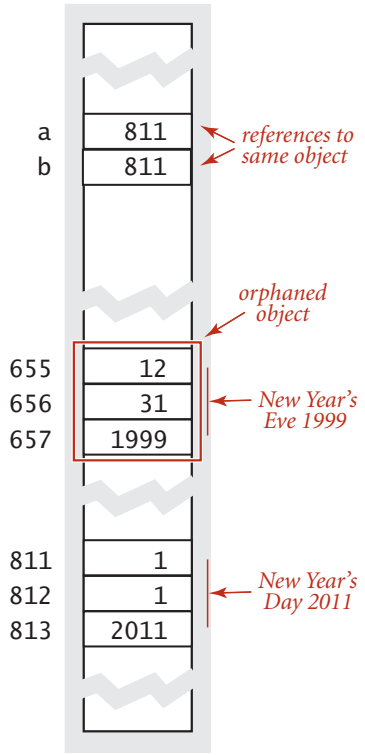
application

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

An abstract data type for a simple counter

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

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