

## Algorithms, 4th edition

### 1. Fundamentals

- [1.1 Programming Model](#)
- [1.2 Data Abstraction](#)
- [1.3 Stacks and Queues](#)
- [1.4 Analysis of Algorithms](#)
- [1.5 Case Study: Union-Find](#)

### 2. Sorting

- [2.1 Elementary Sorts](#)
- [2.2 Mergesort](#)
- [2.3 Quicksort](#)
- [2.4 Priority Queues](#)
- [2.5 Sorting Applications](#)

### 3. Searching

- [3.1 Symbol Tables](#)
- [3.2 Binary Search Trees](#)
- [3.3 Balanced Search Trees](#)
- [3.4 Hash Tables](#)
- [3.5 Searching Applications](#)

### 4. Graphs

- [4.1 Undirected Graphs](#)
- [4.2 Directed Graphs](#)
- [4.3 Minimum Spanning Trees](#)
- [4.4 Shortest Paths](#)

### 5. Strings

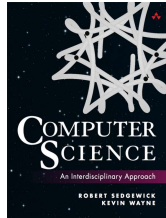
- [5.1 String Sorts](#)

- [5.2 Tries](#)
- [5.3 Substring Search](#)
- [5.4 Regular Expressions](#)
- [5.5 Data Compression](#)

## [6. Context](#)

- [6.1 Event-Driven Simulation](#)
- [6.2 B-trees](#)
- [6.3 Suffix Arrays](#)
- [6.4 Maxflow](#)
- [6.5 Reductions](#)
- [6.6 Intractability](#)

Related Booksites



Web Resources

[FAQ](#)

[Data](#)

[Code](#)

[Errata](#)

[Lectures](#)

[Cheatsheet](#)

[References](#)

[Online Course](#)

[Programming Assignments](#)

# 1.1 Programming Model

This section under major construction.

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.

## Primitive data types and expressions.

A *data type* is a set of values and a set of operations on those values. The following four primitive data types 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`)

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. We use *expressions* to apply the operations associated with each type.

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

The following table summarizes the set of values and most common operations on those values for Java's int, double, boolean, and char data types.

| type    | set of values  | operators                            | typical expressions |           |
|---------|--|--------------------------------------|---------------------|-----------|
|         |  |                                      | expression          | value     |
| int     | integers between $-2^{31}$ and $+2^{31}-1$ (32-bit two's complement) | +                                    | 5 + 3               | 8         |
|         |  | -                                    | 5 - 3               | 2         |
|         |  | *                                    | 5 * 3               | 15        |
|         |  | /                                    | 5 / 3               | 1         |
|         |  | %                                    | 5 % 3               | 2         |
| double  | double-precision real numbers (64-bit IEEE 754 standard)             | +                                    | 3.141 - .03         | 3.111     |
|         |  | -                                    | 2.0 - 2.0e-7        | 1.9999998 |
|         |  | *                                    | 100 * .015          | 1.5       |
|         |  | /                                    | 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] |                     |           |

- *Expressions.* Typical expressions are *infix*. When an expression contains more than one operator, the *precedence order* specifies the order in which they are applied: The operators \* and / (and %) have higher precedence than (are applied before) the + and - operators; among logical operators, ! is the highest precedence, followed by && and and then ||. Generally, operators of the same precedence are *left associative* (applied left to right). You can use parentheses to override these rules.
- *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 directive to convert a value of one type into a value of another type. For example (int) 3.7 is 3. Casting a double to an int truncates toward zero.
- *Comparisons.* The following *mixed-type* operators compare two values of the same type and produce a boolean value:
  - *equal* (==)
  - *not equal* (!=)
  - *less than* (<)

- *less than or equal* (<=)
- *greater than* (>)
- *greater than or equal* (>=)
- *Other primitive types.* Java's `int` has a 32-bit representation; Java's `double` type has a 64-bit representation. 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, with arithmetic operations (`float`)

## 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.

- *Declarations* create variables of a specified type and name them with identifiers. Java is a *strongly typed* language because the Java compiler checks for consistency. The *scope* of a variable is the part of the program where it is defined.
- *Assignments* associate a data-type value (defined by an expression) with a variable.
- *Initializing declarations* combine a declaration with an assignment to initialize a variable at the same time it is declared.
- *Implicit assignments.* The following shortcuts are available when our purpose is to modify a variable's value relative to the current value:
  - Increment/decrement operators: the code `i++` is shorthand for `i = i + 1`. The code `++i` is the same except that the expression value is taken *after* the increment/decrement, not before.
  - Other compound operators: the code `i /= 2` is shorthand for `i = i/2`.
- *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. We refer to the statements in the block in a loop as the *body* of the loop.

- *Break and continue.* 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
- *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.
- *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.

The following table illustrates different kinds of Java statements.

| statement                       | examples   | definition   |
|---------------------------------|--|--|
| <i>declaration</i>              | <pre>int i; double c;</pre>  | create a variable of a specified type, named with a given identifier |
| <i>assignment</i>               | <pre>a = b + 3; discriminant = b*b - 4.0*c;</pre>  | assign a data-type value to a variable                               |
| <i>initializing declaration</i> | <pre>int i = 1; double c = 3.141592625;</pre>  | declaration that also assigns an initial value                       |
| <i>implicit assignment</i>      | <pre>i++; i += 1;</pre>  | $i = i + 1;$   |
| <i>conditional (if)</i>         | <pre>if (x &lt; 0) x = -x;</pre>   | execute a statement, depending on boolean expression                 |
| <i>conditional (if-else)</i>    | <pre>if (x &gt; y) max = x; else      max = y;</pre>   | execute one or the other statement, depending on boolean expression  |
| <i>loop (while)</i>             | <pre>int v = 0; while (v &lt;= N)     v = 2*v; double t = c; while (Math.abs(t - c/t) &gt; 1e-15*t)     t = (c/t + t) / 2.0;</pre> | execute statement until boolean expression is false                  |
| <i>loop (for)</i>               | <pre>for (int i = 1; i &lt;= N; i++)     sum += 1.0/i; for (int i = 0; i &lt;= N; i++)     StdOut.println(2*Math.PI*i/N);</pre>    | compact version of while statement                                   |
| <i>call</i>                     | <pre>int key = StdIn.readInt();</pre>  | invoke other methods (see page 22)                                   |
| <i>return</i>                   | <pre>return false;</pre>   | return from a method (see page 24)                                   |



# Arrays.

An *array* stores a sequence of values that are all of the same type. If we have  $N$  values, we can use the notation  $a[i]$  to refer to the  $i$ th value for any value of  $i$  from 0 to  $N-1$ .

- *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.
- *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. The default initial value is zero for numeric types and `false` for type `boolean`.
- *Initializing declaration.* We can specify the initialization values at compile time, by listing literal values between curly braces, separated by commas.

**long form**

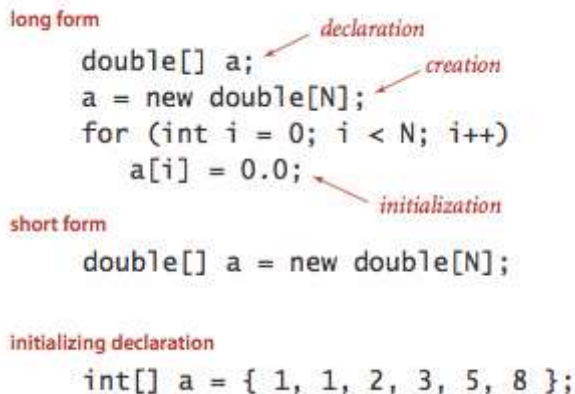
```
double[] a;
a = new double[N];
for (int i = 0; i < N; i++)
    a[i] = 0.0;
```

**short form**

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

**initializing declaration**

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



- *Using an array.* 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`. Java does *automatic bounds checking*—if you access an array with an illegal index your program will terminate with an [ArrayIndexOutOfBoundsException](#).
- *Aliasing.* 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.

```

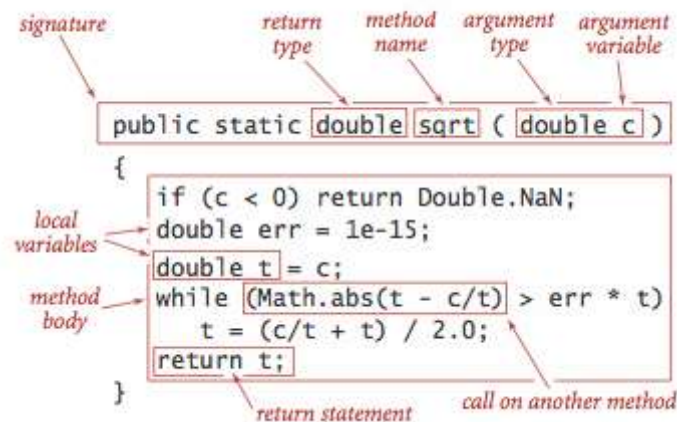
This situation is known as *aliasing* and can lead to subtle bugs.

- *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. To refer to the entry in row *i* and column *j* of a two-dimensional array `a[][]`, we use the notation `a[i][j]`.

## Static methods.

Static methods are called *functions* in many programming languages, since they can behave like mathematical functions. Each static method is a sequence of statements that are executed, one after the other, when the static method is *called*.

- *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 or causes a *side effect*. Each static method is composed of a *signature* and a *body*.



- *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 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.

- *Properties of methods.* Java methods have the following features:
  - *Arguments are passed by value.* When calling a function, the argument value is fully evaluated and the resulting value is *copied* into argument variable. This is known as *pass by value*. Array (and other object) references are also passed by value: the method cannot change the reference, but it can change the entries in the array (or value of the object).
  - *Method names can be overloaded.* Methods within a class can have the same name, provided they have different signatures. This features is known as *overloading*.
  - *A method has a single return value but may have multiple return statements.* A Java method can provide only one return value. Control goes back to the calling program as soon as the first return statement is reached.
  - *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 and produces side effects (consume input, produce output, change entries in an array, or otherwise change the state of the system).
- *Recursion.* A *recursive* method is a method that calls itself either directly or indirectly. There are three important rules of thumb in developing recursive programs:
  - The recursion has a *base case*.
  - Recursive calls must address subproblems that are *smaller* in some sense, so that recursive calls converge to the base case.
  - Recursive calls should not address subproblems that *overlap*.
- *Basic programming model.* A *library of static methods* is a set of static methods that are defined in a Java class. 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()`.
- *Modular programming.* Libraries of static methods enable *modular programming*, where static methods in one library can call static methods defined in other libraries. This approach has many important advantages.
  - Work with modules of reasonable size
  - Share and reuse code without having to reimplement it
  - Substitute improved implementations
  - Develop appropriate abstract models for addressing programming problems
  - Localize debugging
- *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.

- *External libraries.* We use static methods from three different kinds of libraries, each requiring (slightly) differing procedures for code reuse.
  - Standard system libraries in `java.lang`, including `java.lang.Math`, `java.lang.Integer`, and `java.lang.Double`. These libraries are always available in Java.
  - Imported system libraries such as `java.util.Arrays`. An import statement at the beginning of the program is needed to use such libraries.
  - Libraries in this book. Follow [these instructions](#) for adding `algs4.jar` to your Java classpath.

To invoke a method from another library, we prepend the library name to the method name for each call: `Math.sqrt()`, `Arrays.sort()`, `BinarySearch.rank()`, and `StdIn.readInt()`.

---

## APIs.

- *Java libraries.*
- *Our standard libraries.*
- *Your own libraries.*

## Strings.

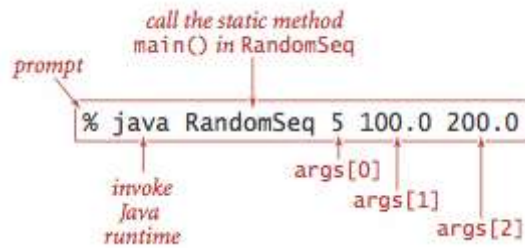
| type                | set of values       | typical literals         | operators          | typical expressions                              |                              |
|---------------------|---------------------|--------------------------|--------------------|--|------------------------------|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

- *Concatenation.*

- *Conversion.*
- *Automatic conversion.*
- *Command-line arguments.*

## Input and output.

- *Commands and arguments.*



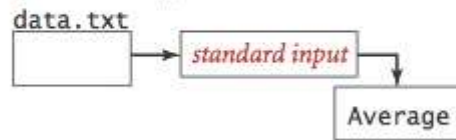
- *Standard output.*
- *Formatted output.*

| type   | code | typical literal    | sample format strings | converted string values for output |
|--------|------|--------------------|-----------------------|------------------------------------|
| int    | d    | 512                | "%14d"                | "512"                              |
|        |      |                    | "%-14d"               | "512"                              |
| double | f    | 1595.1680010754388 | "%14.2f"              | "1595.17"                          |
|        | e    |                    | "%.7f"                | "1595.1680011"                     |
|        | e    |                    | "%14.4e"              | "1.5952e+03"                       |
| String | s    | "Hello, World"     | "%14s"                | "Hello, World"                     |
|        |      |                    | "%-14s"               | "Hello, World"                     |
|        |      |                    | "%-14.5s"             | "Hello"                            |

- *Standard input.*
- *Redirection and piping.*

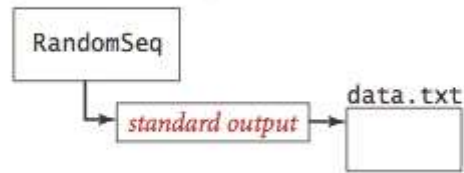
redirecting from a file to standard input

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



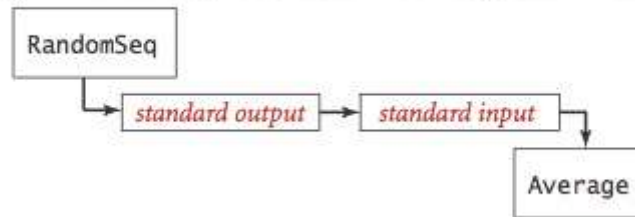
redirecting standard output to a file

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



piping the output of one program to the input of another

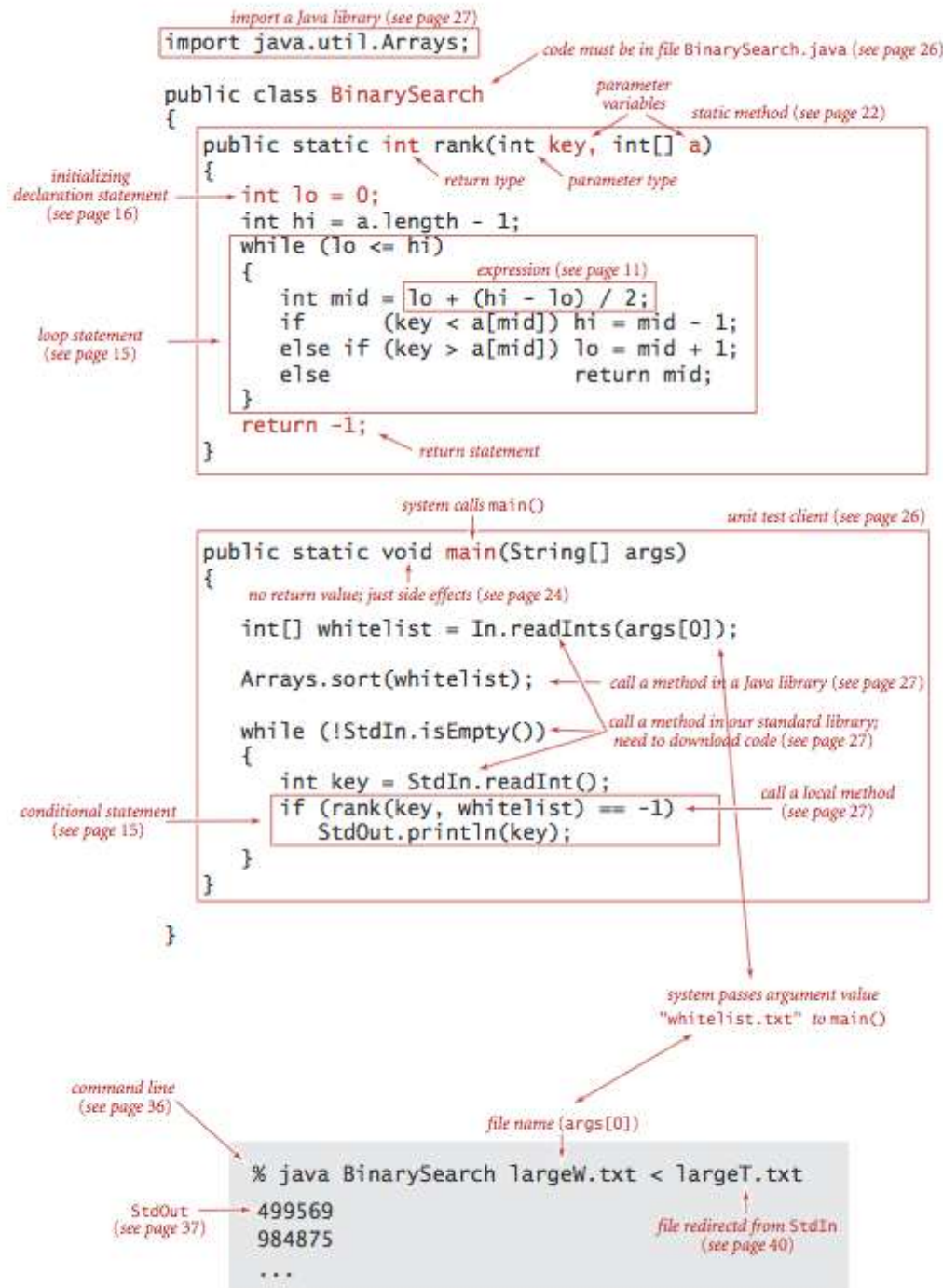
```
% java RandomSeq 1000 100.0 200.0 | java Average
```



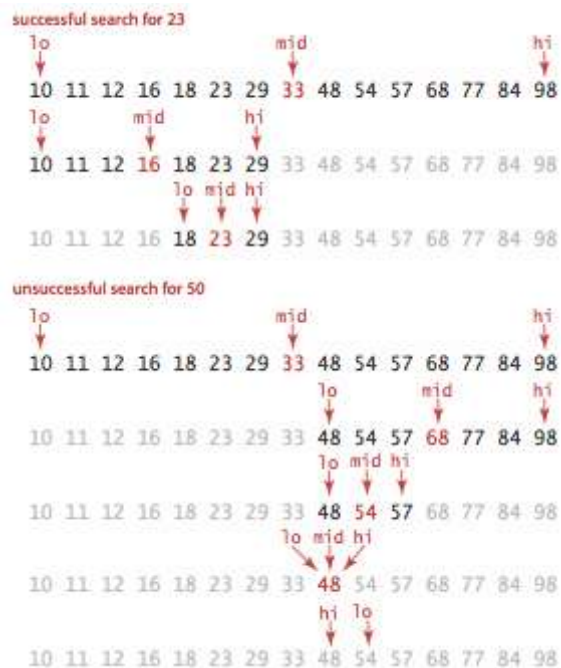
- *Input and output from a file.*
- *Standard drawing.*

## Binary search.

Below is a complete Java program [BinarySearch.java](#) that illustrates many of the basic features of our programming model. It implements a classic algorithm known as *binary search* and tests it for an application known as *allowlist filtering*.



The static method `rank()` takes an integer key and a *sorted* array of `int` values as arguments and returns the index of the key if it is present in the array, -1 otherwise. It accomplishes this task by maintaining variables `lo` and `hi` such that the key is in `a[lo..hi]` if it is in the array, then entering into a loop that tests the middle entry in the interval (at index `mid`). If the key is equal to `a[mid]`, the return value is `mid`; otherwise the method cuts the interval size about in half, looking at the left half if the key is less than `a[mid]` and at the right half if the key is greater than `a[mid]`. The process terminates when the key is found or the interval is empty.



- *Development client.*
- *Allowlisting.* For testing, we use the sample files [tinyAllowlist.txt](#), [tinyText.txt](#), [largeAllowlist.txt](#), and [largeText.txt](#).
- *Performance.*



## Input and output libraries.

Here is a list of the input and output libraries that we use throughout the textbook and beyond.

| PROGRAM                                  | DESCRIPTION / JAVADOC  |
|--|--|
| <a href="#"><u>StdIn.java</u></a>        | <a href="#"><u>read numbers and text from standard input</u></a> |
| <a href="#"><u>StdOut.java</u></a>       | <a href="#"><u>write numbers and text to standard output</u></a> |
| <a href="#"><u>StdDraw.java</u></a>      | <a href="#"><u>draw geometric shapes in a window</u></a>         |
| <a href="#"><u>StdAudio.java</u></a>     | <a href="#"><u>create, play, and manipulate sound</u></a>        |
| <a href="#"><u>StdRandom.java</u></a>    | <a href="#"><u>generate random numbers</u></a>                   |
| <a href="#"><u>StdStats.java</u></a>     | <a href="#"><u>compute statistics</u></a>                        |
| <a href="#"><u>StdArrayIO.java</u></a>   | <a href="#"><u>read and write 1D and 2D arrays</u></a>           |
| <a href="#"><u>In.java</u></a>           | <a href="#"><u>read numbers and text from files and URLs</u></a> |
| <a href="#"><u>Out.java</u></a>          | <a href="#"><u>write numbers and text to files</u></a>           |
| <a href="#"><u>Draw.java</u></a>         | <a href="#"><u>draw geometric shapes</u></a>                     |
| <a href="#"><u>DrawListener.java</u></a> | <a href="#"><u>support keyboard and mouse events in Draw</u></a> |
| <a href="#"><u>Picture.java</u></a>      | <a href="#"><u>process digital images</u></a>                    |
| <a href="#"><u>Stopwatch.java</u></a>    | <a href="#"><u>measure running time</u></a>                      |
| <a href="#"><u>BinaryStdIn.java</u></a>  | <a href="#"><u>read bits from standard input</u></a>             |
| <a href="#"><u>BinaryStdOut.java</u></a> | <a href="#"><u>write bits to standard output</u></a>             |
| <a href="#"><u>BinaryIn.java</u></a>     | <a href="#"><u>read bits from files and URLs</u></a>             |
| <a href="#"><u>BinaryOut.java</u></a>    | <a href="#"><u>write bits to files</u></a>                       |

We briefly describe the input and output libraries and include a sample client.

## Standard input and standard output.

[StdIn.java](#) and [StdOut.java](#) are libraries for reading in numbers and text from standard input and printing out numbers and text to standard output. Our versions have a simpler interface than the corresponding Java ones (and provide a few technical improvements). [RandomSeq.java](#) generates random numbers in a given range. [Average.java](#) reads in a sequence of real numbers from standard input and prints their average on standard output.

```
% java Average
10.0 5.0 6.0 3.0 7.0 32.0
3.14 6.67 17.71
<Ctrl-d>
Average is 10.0577777777778
```

[In.java](#) and [Out.java](#) are object-oriented versions that support multiple input and output streams, including reading from a file or URL and writing to a file.

## Standard drawing.

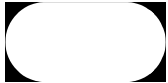
[StdDraw.java](#) is an easy-to-use library for drawing geometric shapes, such as points, lines, and circles. [RightTriangle.java](#) draws a right triangle and a circumscribing circle.

[Draw.java](#) is an object-oriented versions that support drawing in multiple windows.

## Standard audio.

[StdAudio.java](#) is an easy-to-use library for synthesizing sound. [Tone.java](#) reads in a frequency and duration from the command line, and it sonifies a sine wave of the given frequency for the given duration.

```
% java Tone 440.0 3.0
```



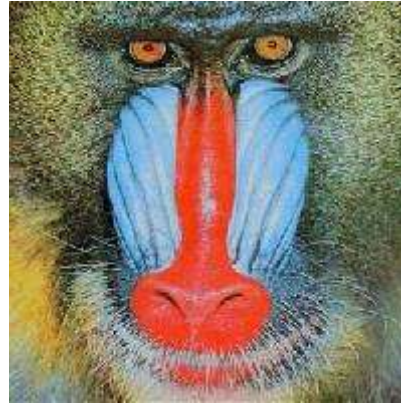
## Image processing.

[Picture.java](#) is an easy-to-use library for image processing. [Scale.java](#) takes the name of a picture file and two integers (width w and height h) as command-line arguments and scales the image to w-by-h.

```
% java Scale mandrill.jpg 298 298
```



```
% java Scale mandrill.jpg 200 200
```



```
% java Scale mandrill.jpg 200 400
```



**Q + A**

**Q.** How important is it to use a good shuffling algorithm?

**A.** Here's an [amusing anecdote](#) of what happens when you don't do it correctly (and your business is online poker!). If you're running an online casino, here's the recommended approach for shuffling a deck of cards: (i) get a cryptographically secure pseudo-random number generator, (ii) assign a random 64-bit number to each card, (iii) sort the cards according to their number.

**Creative Problems**

27. **Binomial distribution.** Estimate the number of recursive calls that would be used by the method call `binomial1(100, 50, .25)` in [Binomial.java](#). Develop a better implementation that is based on saving computed values in an array.

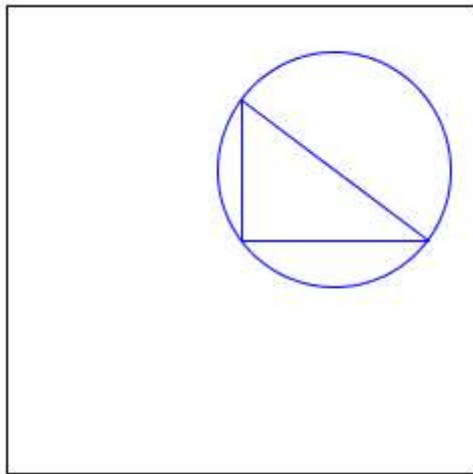
## Web Exercises

1. **Sattolo's algorithm.** Write a program [Sattolo.java](#) that generates a uniformly distributed cycle of length N using [Sattolo's algorithm](#).
2. **Wget.** Write a program [Wget.java](#) that reads in data from the URL specified on the command line and saves it in a file with the same name.

```
% java Wget http://introcs.cs.princeton.edu/data/codes.csv
% more codes.csv
United States,USA,00
Alabama,AL,01
Alaska,AK,02
...
```

3. **Right triangle.** Write a client [RightTriangle.java](#) that draws a right triangle and a circumscribing circle.

```
% java RightTriangle
```



4. **Bouncing ball.** Write a program [BouncingBall.java](#) that illustrates the motion of a bouncing ball.

0:00 / 1:16



*Last modified on July 08, 2022.*

Copyright © 2000–2019 [Robert Sedgewick](#) and [Kevin Wayne](#). All rights reserved.