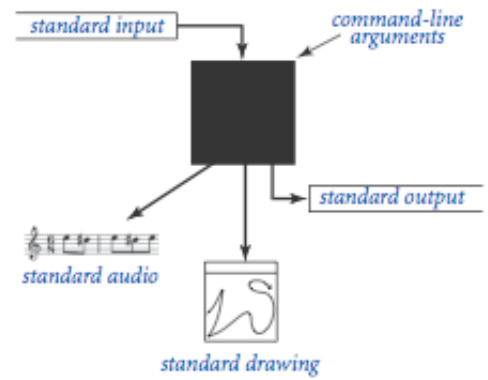


Bird's-eye view. A Java program takes input values from the command line and prints a string of characters as output. By default, both command-line input and standard output are associated with the application that takes commands (the one in which you have been typing the `java` and `javac` commands). We use the generic term *terminal window* to refer to this application. Here are some instructions for using the command line on your system. [[Windows](#) · [Mac](#) · [Linux](#)]

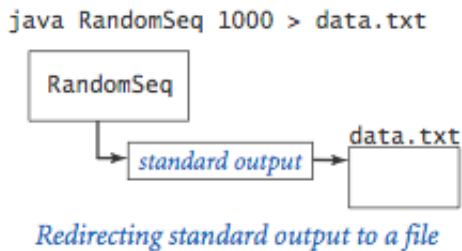


A bird's-eye view of a Java program (revisited)

- **Command-line input.** All classes have a `main()` method that takes a `Stringarray` as argument. That array is the sequence of command-line arguments that we type, provided to Java by the operating system. By convention, both Java and the operating system process the arguments as strings, so if we intend for an argument to be a number, we use a method such as `Integer.parseInt()` to convert it from `String` to the appropriate type.
- **Standard output.** To print output values in our programs, we have been using `System.out.println()`. Java sends the results to an abstract stream of characters known as standard output. By default, the operating system connects standard output to the terminal window - all of the output in our programs so far has been appearing in the terminal window.

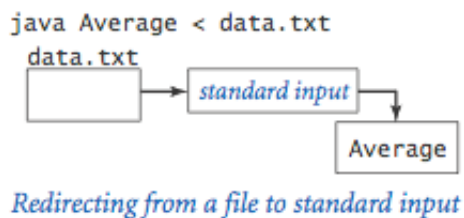
Redirection and piping. For many applications, typing input data as a standard input stream from the terminal window is untenable because doing so limits our program's processing power by the amount of data that we can type. Similarly, we often want to save the information printed on the standard output stream for later use. We can use operating system mechanisms to address both issues.

- *Redirecting standard output to a file.* By adding a simple directive to the command that invokes a program, we can *redirect* its standard output to a file, for permanent storage or for input to some other program at a later time. For example, the command



specifies that the standard output stream is not to be printed in the terminal window, but instead is to be written to a text file named `data.txt`. Each call to `StdOut.print()` or `StdOut.println()` appends text at the end of that file. In this example, the end result is a file that contains 1,000 random values. No output appears in the terminal window: it goes directly into the file named after the `>` symbol. Thus, we can save away information for later retrieval.

- *Redirecting standard output from a file.* Similarly, we can redirect standard input so that `StdIn` reads data from a file instead of the terminal application. For example, the command

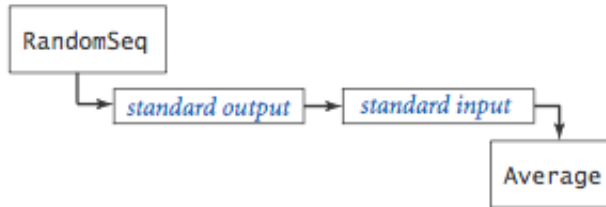


reads a sequence of numbers from the file `data.txt` and computes their average value.

Specifically, the `<` symbol is a directive to implement the standard input stream by reading from the file `data.txt` instead of by waiting for the user to type something into the terminal window. When the program calls `StdIn.readDouble()`, the operating system reads the value from the file. This facility to redirect standard input from a file enables us to process huge amounts of data from any source with our programs, limited only by the size of the files that we can store.

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

```
java RandomSeq 1000 | java Average
```



Piping the output of one program to the input of another

specifies that the standard output for `RandomSeq` and the standard input stream for `Average` are the *same* stream. That is, the result has the same effect as the following sequence of commands

```
% java RandomSeq 1000 > data.txt
% java Average < data.txt
```

but the file `data.txt` is not needed.

- **Filters.** For many common tasks, it is convenient to think of each program as a filter that converts a standard input stream to a standard output stream in some way, with piping as the command mechanism to connect programs together. For example, [MovingAverage.java](#) takes a command-line argument `N` and prints on standard output a stream of numbers where each number in the output stream is the average of the `N` numbers starting at the corresponding position in the standard input stream.

Your operating system also provides a number of filters. For example, the `sort` filter puts the lines on standard input in sorted order:

```
% java RandomSeq 5 | sort
0.035813305516568916
0.14306638757584322
0.348292877655532103
0.5761644592016527
0.9795908813988247
```

Another useful filter is `more`, which reads data from standard input and displays it in your terminal window one screenful at a time. For example, if you type

```
% java RandomSeq 1000 | more
```

you will see as many numbers as fit in your terminal window, but `more` will wait for you to hit the space bar before displaying each succeeding screenful.