1.5 Input and Output

In this section we extend the set of simple abstractions (command-line input and standard output) that we have been using as the interface between our Java programs and the outside world to include standard input, standard drawing, and standard audio. Standard input makes it convenient for us to write programs that process arbitrary amounts of input and to interact with our programs; standard drawing makes it possible for us to work with graphical representations of images, freeing us from having to encode everything as text; and standard audio adds sound. These extensions are easy to use, and you will find that they bring you to yet another new world of programming.

The abbreviation I/O is universally understood to mean input/output, a collective term that refers to the mechanisms by which programs communicate with the outside world. Your computer’s operating system controls the physical devices that are connected to your computer. To implement the standard I/O abstractions, we use libraries of methods that interface to the operating system.

You have already been accepting argument values from the command line and printing strings in a terminal window; the purpose of this section is to provide you with a much richer set of tools for processing and presenting data. Like the System.out.println() and System.out.print() methods that you have been using, these methods do not implement mathematical functions—their purpose is to cause some side effect, either on an input device or an output device. Our prime concern is using such devices to get information into and out of our programs.

An essential feature of standard I/O mechanisms is that there is no limit on the amount of input or output data, from the point of view of the program. Your programs can consume input or produce output indefinitely.

One use of standard I/O mechanisms is to connect your programs to files on your computer’s disk. It is easy to connect standard input, standard output, standard drawing, and standard audio to files. Such connections make it easy to have your Java programs save or load results to files for archival purposes or for later reference by other programs or applications.
Program 1.5.1 Generating a random sequence

```java
public class RandomSeq {
    public static void main(String[] args)
    { // Print a random sequence of N real values in [0, 1)
        int N = Integer.parseInt(args[0]);
        for (int i = 0; i < N; i++)
            System.out.println(Math.random());
    }
}
```

This program illustrates the conventional model that we have been using so far for Java programming. It takes a command-line argument N and prints N random numbers between 0 and 1. From the program's point of view, there is no limit on the length of the output sequence.

Standard input. Our class StdIn is a library that implements a standard input abstraction to complement the standard output abstraction. Just as you can print a value to standard output at any time during the execution of your program, you can read a value from a standard input stream at any time.

Standard drawing. Our class StdDraw allows you to create drawings with your programs. It uses a simple graphics model that allows you to create drawings consisting of points and lines in a window on your computer. StdDraw also includes facilities for text, color, and animation.

Standard audio. Our class StdAudio allows you to create sound with your programs. It uses a standard format to convert arrays of numbers into sound.

To use both command-line input and standard output, you have been using built-in Java facilities. Java also has built-in facilities that support abstractions like standard input, standard draw, and standard audio, but they are somewhat more complicated to use, so we have developed a simpler interface to them in our StdIn, StdDraw, and StdAudio libraries. To logically complete our programming model, we also include a StdOut library. To use these libraries, download StdIn.java, StdOut.java, StdDraw.java, and StdAudio.java and place them in the same directory as your program (or use one of the other mechanisms for sharing libraries described on the booksite).

The standard input and standard output abstractions date back to the development of the Unix operating system in the 1970s and are found in some form on all modern systems. Although they are primitive by comparison to various mechanisms developed since, modern programmers still depend on them as a reliable way to connect data to programs. We have developed for this book standard draw and standard audio in the same spirit as these earlier abstractions to provide you with an easy way to produce visual and aural output.

**Standard output** Java's `System.out.println()` and `System.out.print()` methods implement the basic standard output abstraction that we need. Nevertheless, to treat standard input and standard output in a uniform manner (and to provide a few technical improvements), starting in this section and continuing through the rest of the book, we use similar methods that are defined in our `StdOut` library. `StdOut.print()` and `StdOut.println()` are nearly the same as the Java methods that you have been using (see the booksite for a discussion of the differences, which need not concern you now). The `StdOut.printF0` method is a main topic of this section and will be of interest to you now because it gives you more control over the appearance of the output. It was a feature of the C language of the early 1970s that still survives in modern languages because it is so useful.

Since the first time that we printed double values, we have been distracted by excessive precision in the printed output. For example, when we use `System.out.println(Math.PI)` we get the output 3.141592653589793, even though we might
public class StdOut

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

API for our library of static methods for standard output

prefer to see 3.14 or 3.14159. The print() and println() methods present each number to 15 decimal places even when we would be happy with just a few digits of precision. The printf() method is more flexible: it allows us to specify the number of digits and the precision when converting data type values to strings for output. With printf(), we can write Stdout.printf("%7.5f", Math.PI) to get 3.14159, and we can replace System.out.print(t) with

Stdout.printf("The square root of %.1f is %.6f", c, t);

in Newton (Program 1.3.6) to get output like

The square root of 2.0 is 1.414214

Next, we describe the meaning and operation of these statements, along with extensions to handle the other built-in types of data.

Formatted printing basics. In its simplest form, printf() takes two arguments. The first argument is a format string that describes how to convert the second argument into 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 that specifies the field width of the converted value (the number of characters in the converted output string). By default, blanks are added on the left to make the length of the converted output equal to the field width; if we want the blanks on the right, we can insert a minus sign before the field width. (If the converted output string is larger than the field width, the field width is ignored.) Following the width, we have the option of including a period followed by the number of digits to put after the decimal point (the precision) for a double value or the number of characters to take from the beginning of the string for a String value. The most important thing to remember about using printf() is that the conversion code in the format and the type of the corresponding argument must match. That is, Java must be able to convert from the type of the argument to the type required by the conversion code. Every type of data can be converted to String, but if you write Stdout.printf("%1d", Math.PI) or Stdout.printf("%.4f", 512), you will get an IllegalArgumentException run-time error.

Format string. The first argument of printf() is a String that may contain characters other than a format string. Any part of the argument that is not part of a format string passes through to the output, with the format string replaced by the argument value (converted to a string as specified). For example, the statement

Stdout.printf("PI is approximately %.2f\n", Math.PI);

prints the line

PI is approximately 3.14

Note that we need to explicitly include the newline character \n in the argument in order to print a new line with printf().
Multiple arguments. The `printf()` function can take more than two arguments. In this case, the format string will have a format specifier for each additional argument, perhaps separated by other characters to pass through to the output. For example, if you were making payments on a loan, you might use code whose inner loop contains the statements

```java
String formats = "%3s %16.2f %17.2f %5.2f\n";
StdOut.printf(formats, month[1], pay, balance, interest);
```

to print the second and subsequent lines in a table like this (see EXERCISE 1.5.14):

<table>
<thead>
<tr>
<th>month</th>
<th>payment</th>
<th>balance</th>
<th>interest</th>
</tr>
</thead>
<tbody>
<tr>
<td>Jan</td>
<td>$299.00</td>
<td>$9742.67</td>
<td>$41.67</td>
</tr>
<tr>
<td>Feb</td>
<td>$299.00</td>
<td>$9484.26</td>
<td>$40.59</td>
</tr>
<tr>
<td>Mar</td>
<td>$299.00</td>
<td>$9224.78</td>
<td>$39.52</td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Formatted printing is convenient because this sort of code is much more compact than the string-concatenation code that we have been using.

**Standard input** Our `StdIn` library takes data from a standard input stream that may be empty or may contain a sequence of values separated by whitespace (spaces, tabs, newline characters, and the like). Each value is a `String` or a value from one of Java's primitive types. One of the key features of the standard input stream is that your program consumes values when it reads them. Once your program has read a value, it cannot back up and read it again. This assumption is restrictive, but it reflects physical characteristics of some input devices and simplifies implementing the abstraction. The library consists of the nine methods: `isEmpty()`, `readInt()`, `readDouble()`, `readLong()`, `readBoolean()`, `readChar()`, `readString()`, `readLine()`, and `readAll()`. Within the input stream model, these methods are largely self-documenting (the names describe their effect), but their precise operation is worthy of careful consideration, so we will consider several examples in detail.

**Typing input.** When you use the `java` command to invoke a Java program from the command line, you actually are doing three things: issuing a command to start executing your program, specifying the values of the command line arguments, and beginning to define the standard input stream. The string of characters that you type in the terminal window after the command line is the standard input stream. When you type characters, you are interacting with your program. The program waits for you to create the standard input stream. For example, consider the following program, which takes a command-line argument `N`, then reads `N` numbers from standard input and adds them:

```java
public class AddInts {
    public static void main(String[] args) {
        int N = Integer.parseInt(args[0]);
        int sum = 0;
        for (int i = 0; i < N; i++)
            { int value = StdIn.readInt();
                sum += value;
            }
        StdOut.println("Sum is " + sum);
    }
}
```

When you type `java AddInts 4`, after the program takes the command-line argument, it calls the method `StdIn.readInt()` and waits for you to type an integer. Suppose that you want 144 to be the first value. As you type 1, then 4, and then 4, nothing happens, because `StdIn` does not know that you are doing typing the integer. But when you then type `<return>` to signify the end of your integer, `StdIn.readInt()` immediately returns the value 144, which your program adds to `sum`.
and then calls StdIn.readInt() again. Again, nothing happens until you type the second value: if you type 2, then 3, then 3, and then return to end the number, StdIn.readInt() returns the value 233, which your program again adds to sum. After you have typed four numbers in this way, AddInts expects no more input and prints out the sum, as desired.

**Input format.** If you type abc or 12.2 or true when StdIn.readInt() is expecting an int, it will respond with a NumberFormatException. The format for each type is the same as you have been using for literal values within Java programs. For convenience, StdIn treats strings of consecutive whitespace characters as identical to one space and allows you to delimit your numbers with such strings. It does not matter how many spaces you put between numbers, or whether you enter numbers on one line or separate them with tab characters or spread them out over several lines, (except that your terminal application processes standard input one line at a time, so it will wait until you type return before sending all of the numbers on that line to standard input). You can mix values of different types in an input stream, but whenever the program expects a value of a particular type, the input stream must have a value of that type.

**Interactive user input.** TwentyQuestions (Program 1.5.2) is a simple example of a program that interacts with its user. The program generates a random integer and then gives clues to a user trying to guess the number. (As a side note: by using binary search, you can always get to the answer in at most twenty questions. See Section 4.2.) The fundamental difference between this program and others that we have written is that the user has the ability to change the control flow while the program is executing. This capability was very important in early applications of computing, but we rarely write such programs nowadays because modern applications typically take such input through the graphical user interface, as discussed in Chapter 3. Even a simple program like TwentyQuestions illustrates that writing programs that support user interaction is potentially very difficult because you have to plan for all possible user inputs.

**Program 1.5.2 Interactive user input**

```java
public class TwentyQuestions {
    public static void main(String[] args) {
        int N = (int) (Math.random() * 1000000);
        StdOut.print("I'm thinking of a number 
");
        StdOut.println("between 1 and 1,000,000");
        int m = 0;
        while (m != N) {
            StdOut.print("What's your guess? ");
            m = StdIn.readInt();
            if (m == N) StdOut.println("You win!");
            if (m < N) StdOut.println("Too low");
            if (m > N) StdOut.println("Too high");
        }
    }
}
```

This program plays a simple guessing game. You type numbers, each of which is an implicit question ("Is this the number?") and the program tells you whether your guess is too high or too low. You can always get it to print You win! with less than twenty questions. To use this program, you need to first download StdIn.java and StdOut.java in the same directory as this code (which is in a file named TwentyQuestions.java).
1.5 Input and Output

Program 1.5.3 Averaging a stream of numbers

```java
public class Average {
    public static void main(String[] args) {
        // Average the numbers on the input stream.
        double sum = 0.0;
        int cnt = 0;
        while (!StdIn.isEmpty()) {
            // Read a number and cumulate the sum.
            double value = StdIn.readDouble();
            sum += value;
            cnt++;
        }
        double average = sum / cnt;
        StdOut.println("Average is "+average);
    }
}
```

This program reads in a sequence of real numbers from standard input and prints their average on standard output (provided that the sum does not overflow). From its point of view, there is no limit on the size of the input stream. The commands on the right below use redirection and piping (discussed in the next subsection) to provide 100,000 numbers to average.

Processing an arbitrary-size input stream. Typically, input streams are finite: your program marches through the input stream, consuming values until the stream is empty. But there is no restriction of the size of the input stream, and some programs simply process all the input presented to them. Average (Program 1.5.3) is an example that reads in a sequence of real numbers from standard input and prints their average. It illustrates a key property of using an input stream: the length of the stream is not known to the program. We type all the numbers that we have, and then the program averages them. Before reading each number, the program uses the method StdIn.isEmpty() to check whether there are any more numbers in the input stream. How do we signal that we have no more data to type? By convention, we type a special sequence of characters known as the end-of-file sequence. Unfortunately, the terminal applications that we typically encounter on modern operating systems use different conventions for this critically important sequence. In this book, we use <ctrl-d> (many systems require <ctrl-d> to be on a line by itself); the other widely used convention is <ctrl-z> on a line by itself. Average is a simple program, but it represents a profound new capability in programming: with standard input, we can write programs that process an unlimited amount of data. As you will see, writing such programs is an effective approach for numerous data-processing applications.

Standard input is a substantial step up from the command-line input model that we have been using, for two reasons, as illustrated by TwentyQuestions and Average. First, we can interact with our program—with command-line arguments, we can only provide data to the program before it begins execution. Second, we can read in large amounts of data—with command-line arguments, we can only enter values that fit on the command line. Indeed, as illustrated by Average, the amount of data can be potentially unlimited, and many programs are made simpler by that assumption. A third raison d'être for standard input is that your operating system makes it possible to change the source of standard input, so that you do not have to type all the input. Next, we consider the mechanisms that enable this possibility.

Redirection and piping. For many applications, typing input data as a standard input stream from the terminal window is untenable because our program's processing power is then limited by the amount of data that we can type (and our typing speed). Similarly, we often want to save the information printed on the standard output stream for later use. To address such limitations, we next focus on the idea that standard input is an abstraction—the program just expects its input and has no dependence on the source of the input stream. Standard output is a similar abstraction. The power of these abstractions derives from our ability (through the operating system) to specify various other sources for standard input and standard output, such as a file, the network, or another program. All modern operating systems implement these mechanisms.
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, either for permanent storage or for input to another program at a later time. For example,

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

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 `System.out.println()` or `System.out.print()` 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. Note that we do not have to change `RandomSeq` (Program 1.2.1) in any way for this mechanism to work—it is using the standard output abstraction and is unaffected by our use of a different implementation of that abstraction. You can use this mechanism to save output from any program that you write. Once you have expended a significant amount of effort to obtain a result, we often want to save the result for later reference. In a modern system, you can save some information by using cut-and-paste or some similar mechanism that is provided by the operating system, but cut-and-paste is inconvenient for large amounts of data. By contrast, redirection is specifically designed to make it easy to handle large amounts of data.

Redirecting from a file to standard input. Similarly, we can redirect standard input so that `StdIn` reads data from a file instead of the terminal application:

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

This command reads a sequence of numbers from the file `data.txt` and computes their average value. Specifically, the `<` symbol is a directive that tells the operating system to implement the standard input stream by reading from the text file `data.txt` instead of waiting for the user to type something into the terminal window. When the program calls `StdIn.readDouble()`, the operating system reads the value from the file. The file `data.txt` could have been created by any application, not just a Java program—virtually every application on your computer can create text files. This facility to redirect from a file to standard input enables us to create data-driven code where we can change the data processed by a program without having to change the program at all. Instead, we keep data in files and write programs that read from standard input.

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 command

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

specifies that the standard output for `RandomSeq` and the standard input stream for `Average` are the same stream. The effect is as if `RandomSeq` were typing the numbers it generates into the terminal window while `Average` is running. This example also has the same effect as the following sequence of commands:

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

In this case, the file `data.txt` is not created. This difference is profound, because it removes another limitation on the size of the input and output streams that we can process. For example, we could replace 1000 in our example with 1000000000, even though we might not have the space to save a billion numbers on our computer (we do need the time to process them, however). When `RandomSeq` calls `System.out.println()`, a string is added to the end of the stream; when `Average` calls `StdIn.readDouble()`, a string is removed from the beginning of the stream. The timing of precisely what happens is up to the operating system: it might run `RandomSeq` until it produces some output, and then run `Average` to consume that output, or it might run `Average` until it needs some output, and then run `RandomSeq` until it produces the needed output. The end result is the same, but our programs are freed from worrying about such details because they work solely with the standard input and standard output abstractions.

Piping the output of one program to the input of another.
1.5 Input and Output

mechanism to connect programs together. For example, RangeFilter (PROGRAM 1.5.4) takes two command-line arguments and prints on standard output those numbers from standard input that fall within the specified range. You might imagine standard input to be measurement data from some instrument, with the filter being used to throw away data outside the range of interest for the experiment at hand. Several standard filters that were designed for Unix still survive (sometimes with different names) as commands in modern operating systems. For example, the sort filter puts the lines on standard input in sorted order:

```
% java RandomSeq 6 | sort
0.038133055166694
0.143066875758432
0.34829287765532103
0.57616649201682
0.72345072733392226
0.979508013988247
```

We discuss sorting in SECTION 4.2. A second useful filter is grep, which prints the lines from standard input that match a given pattern. For example, if you type

```
% grep lo < RangeFilter.java
```

you get the result

```
// Filter out numbers not between lo and hi.
int lo = Integer.parseInt(args[0]);
if (t >= lo && t <= hi) StdOut.print(t + " ");
```

Programmers often use tools such as grep to get a quick reminder of variable names or language usage details. A third 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. The term filter is perhaps misleading; it was meant to describe programs like RangeFilter that write some subsequence of standard input to standard output, but it is now often used to describe any program that reads from standard input and writes to standard output.

**Filter**s. Piping, a core feature of the original Unix system of the early 1970s, still survives in modern systems because it is a simple abstraction for communicating among disparate programs. Testimony to the power of this abstraction is that many Unix programs are still being used today to process files that are thousands or millions of times larger than imagined by the programs' authors. We can communicate with other Java programs via calls on methods, but standard input and standard output allow us to communicate with programs that were written at another time and, perhaps, in another language. With standard input and standard output, we are agreeing on a simple interface to the outside world. 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
Elements of Programming

Multiple streams. For many common tasks, we want to write programs that take input from multiple sources and/or produce output intended for multiple destinations. In Section 3.1 we discuss our out and in libraries, which generalize StdOut and StdIn to allow for multiple input and output streams. These libraries include provisions not just for redirecting these streams to and from files, but also from arbitrary web pages.

Processing large amounts of information plays an essential role in many applications of computing. A scientist may need to analyze data collected from a series of experiments, a stock trader may wish to analyze information about recent financial transactions, or a student may wish to maintain collections of music and movies. In these and countless other applications, data-driven programs are the norm. Standard output, standard input, redirection, and piping provides us with the capability to address such applications with our Java programs. We can collect data into files on our computer through the web or any of the standard devices and use redirection and piping to connect data to our programs. Many (if not most) of the programming examples that we consider throughout this book have this ability.

Standard drawing. Up to this point, our input/output abstractions have focused exclusively on text strings. Now we introduce an abstraction for producing drawings as output. This library is easy to use and allows us to take advantage of a visual medium to cope with far more information than possible with just text. As with standard input, our standard drawing abstraction is implemented in a library that you need to download from the book site, StdDraw. Java. Standard drawing is very simple: we imagine an abstract drawing device capable of drawing lines and points on a two-dimensional canvas. The device is capable of responding to the commands that our programs issue in the form of calls to methods in StdDraw such as the following:

```java
public class StdDraw { // basic drawing commands
    public void line(double x0, double y0, double x1, double y1)
    public void point(double x, double y)
    
    Like the methods for standard input and standard output, these methods are nearly self-documenting: StdDraw. line() draws a straight line segment connecting the
```

point \((x_0, y_0)\) with the point \((x_1, y_1)\) whose coordinates are given as arguments. StdDraw. point() draws a spot centered on the point \((x, y)\) whose coordinates are given as arguments. The default scale is the unit square (all coordinates between 0 and 1). The standard implementation displays the canvas in a window on your computer's screen, with black lines and points on a white background. The window includes a menu option to save your drawing to a file, in a format suitable for publishing on paper or on the web.

Your first drawing. The HelloWorld equivalent for graphics programming with StdDraw is to draw a triangle with a point inside. To form the triangle, we draw three lines: one from the point \((0, 0)\) at the lower left corner to the point \((1, 0)\), one from that point to the third point at \((1/2, \sqrt{3}/2)\), and one from that point back to \((0, 0)\). As a final flourish, we draw a spot in the middle. Once you have successfully downloaded StdDraw. java and then compiled and run Triangle, you are off and running to write your own programs that draw figures comprised of lines and points. This ability literally adds a new dimension to the output that you can produce.

When you use a computer to create drawings, you get immediate feedback (the drawing) so that you can refine and improve your program quickly. With a computer program, you can create drawings that you could not contemplate making by hand. In particular, instead of viewing our data as just numbers, we can use pictures, which are far more expressive. We will consider other graphics examples after we discuss a few other drawing commands.
Control commands. The default coordinate system for standard drawing is the unit square, but we often want to draw plots at different scales. For example, a typical situation is to use coordinates in some range for the x-coordinate, or the y-coordinate, or both. Also, we often want to draw lines of different thickness and points of different size than the standard. To accommodate these needs, StdDraw has the following methods:

```java
public class StdDraw (basic control commands)
    void setXscale(double x0, double x1) reset x range to (x0, x1)
    void setYscale(double y0, double y1) reset y range to (y0, y1)
    void setPenRadius(double r) set pen radius to r
```

Note: Methods with the same names but no arguments reset to default values.

For example, when we issue the command `StdDraw.setXscale(0, N)`, we are telling the drawing device that we will be using x-coordinates between 0 and N. Note that the two-call sequence

```java
    StdDraw.setXscale(x0, x1);
    StdDraw.setYscale(y0, y1);
```

sets the drawing coordinates to be within a bounding box whose lower left corner is at (x0, y0) and whose upper right corner is at (x1, y1). If you use integer coordinates, Java casts them to double, as expected. Scaling is the simplest of the transformations commonly used in graphics. In the applications that we consider in this chapter, we use it in a straightforward way to match our drawings to our data.

The pen is circular, so that lines have rounded ends, and when you set the pen radius to r and draw a point, you get a circle of radius r. The default pen radius is .005 and is not affected by coordinate scaling. This default is about 1/200 the width of the default window, so that if you draw 50 points equally spaced along a horizontal or vertical line, you will

```java
    int N = 50;
    StdDraw.setXscale(0, N);
    StdDraw.setYscale(0, N);
    for (int i = 0; i < N; i++)
        StdDraw.line(0, i, N, i);
```

1.5 Input and Output

Program 1.5.5 Input-to-drawing filter

```java
public class PlotFilter
{
    public static void main(String[] args)
    { // Plot points in standard input.
        double x0 = StdIn.readInt();
        double y0 = StdIn.readInt();
        double x1 = StdIn.readInt();
        double y1 = StdIn.readInt();
        StdDraw.setXscale(x0, x1);
        StdDraw.setYscale(y0, y1);
        // Read and plot the rest of the points.
        while (!StdIn.isEmpty())
            { // Read and plot a point.
                double x = StdIn.readInt();
                double y = StdIn.readInt();
                StdDraw.point(x, y);
            }
    }
}
```

Some data is inherently visual. The file USA.txt on the booksite has the coordinates of the US states with populations over 500 (by convention, the first four numbers are the minimum and maximum x and y values).
be able to see individual circles, but if you draw 100 such points, the result will look like a line. When you issue the command StdDraw.setPenRadius(.01), you are saying that you want the thickness of the lines and the size of the points to be two times the .005 standard.

Filtering data to a standard drawing. One of the simplest applications of standard draw is to plot data, by filtering it from standard input to the drawing. PlotFilter (Program 1.5.5) is a such a filter: it reads a sequence of points defined by \((x, y)\) coordinates and draws a spot at each point. It adopts the convention that the first four numbers on standard input specify the bounding box, so that it can scale the plot without having to make an extra pass through all the points to determine the scale (this kind of convention is typical with such data files). The graphical representation of points plotted in this way is far more expressive (and far more compact) than the numbers themselves or anything that we could create with the standard output representation that our programs have been limited to until now. The plotted image that is produced by Program 1.5.5 makes it far easier for us to infer properties of the cities (such as, for example, clustering of population centers) than does a list of the coordinates. Whenever we are processing data that represents the physical world, a visual image is likely to be one of the most meaningful ways that we can use to display output. PlotFilter illustrates just how easily you can create such an image.

Plotting a function graph. Another important use of StdDraw is to plot experimental data or the values of a mathematical function. For example, suppose that we want to plot values of the function \(y = \sin(4x) + \sin(20x)\) in the interval \((0, \pi)\). Completing this task is a prototypical example of sampling: there are an infinite number of points in the interval, so we have to make do with evaluating the function at a finite number of points within the interval. We sample the function by choosing a set of \(x\)-values, then computing \(y\)-values by evaluating the function at each \(x\)-value. Plotting the function by connecting successive points with lines produces what is known as a piecewise linear approximation. The simplest way to proceed is to regularly space the \(x\) values: we decide ahead of time on a sample size, then space the \(x\)-coordinates by the interval size divided by the sample size. To make sure that the values we plot fall in the visible canvas, we scale the \(x\)-axis corresponding to the sample size and the \(y\)-axis corresponding to the maximum and minimum values of the function within the interval. The smoothness of the curve depends on properties of the function and the size of the sample. If the sample size is too small, the rendition of the function may not be at all accurate (it might not be very smooth, and it might miss major fluctuations); if the sample is too large, producing the plot may be time-consuming, since some functions are time-consuming to compute. (In Section 2.4, we will look at a method for plotting a smooth curve without using an excessive number of points.) You can use this same technique to plot the function graph of any function you choose: decide on an \(x\)-interval where you want to plot the function, compute function values evenly spaced throughout that interval and store them in an array, determine and set the \(y\)-scale, and draw the line segments.

Outline and filled shapes. StdDraw also includes methods to draw circles, rectangles, and arbitrary polygons. Each shape defines an outline. When the method name is just the shape name, that outline is traced by the drawing pen. When the name begins with filled, the named shape is instead filled solid, not traced. As usual, we summarize the available methods in an API:

```java
public class StdDraw (shapes) {
    void circle(double x, double y, double r)
    void filledCircle(double x, double y, double r)
    void square(double x, double y, double r)
    void filledSquare(double x, double y, double r)
    void polygon(double[] x, double[] y)
    void filledPolygon(double[] x, double[] y)
}
```

The arguments for circle() and filledCircle() define a circle of radius \(r\) centered at \((x, y)\); the arguments for square() and filledSquare() define a square
1.5 Input and Output

Of side length 2r centered on \((x, y)\); and the arguments for `polygon()` and `filled-Polygon()` define a sequence of points that we connect by lines, including one from the last point to the first point. If you want to define shapes other than squares or circles, use one of these methods. For example,

\[
\begin{align*}
\text{double[]} \, xd &= \{ x-r, x, x+r, x \}; \\
\text{double[]} \, yd &= \{ y, y+r, y, y-r \}; \\
\text{StdDraw} \cdot \text{polygon}(xd, yd);
\end{align*}
\]

plots a diamond (a rotated 2r-by-2r square) centered on the point \((x, y)\).

**Text and color.** Occasionally, you may wish to annotate or highlight various elements in your drawings. StdDraw has a method for drawing text, another for setting parameters associated with text, and another for changing the color of the ink in the pen. We make scant use of these features in this book, but they can be very useful, particularly for drawings on your computer screen. You will find many examples of their use on the booksite.

A class StdDraw (advanced control commands)

```java
public class StdDraw
{
    // Advanced control commands
    void text(double x, double y, String s); // Draw text at (x, y) with string s
    void setFont(Font f); // Set the font
    void setPenColor(Color c); // Set the pen color

    // Methods to clear the canvas
    void clear(); // Clear the entire canvas
    void clear(Color c); // Clear to a specific color

    // Methods to control animation
    void show(int dt); // Show for dt milliseconds
    void show(); // Show immediately
}
```

In this code, Font and Color are non-primitive types that you will learn about in Section 3.1. Until then, we leave the details to StdDraw. The available pen colors are BLACK, BLUE, CYAN, DARK GRAY, GRAY, GREEN, LIGHT GRAY, MAGENTA, ORANGE, PINK, RED, WHITE, and YELLOW, defined as constants within StdDraw. For example, the call `StdDraw.setPenColor(StdDraw.GRAY)` changes to gray ink. The default ink color is BLACK. The default font in StdDraw suffices for most of the drawings that you need (and you can find information on using other fonts on the booksite). For example, you might wish to use these methods to annotate function plots to highlight relevant values, and you might find it useful to develop similar methods to annotate other parts of your drawings.

Shapes, color, and text are basic tools that you can use to produce a dizzying variety of images, but you should use them sparingly. Use of such artifacts usually presents a design challenge, and our StdDraw commands are crude by the standards of modern graphics libraries, so that you are likely to need an extensive number of calls to them to produce the beautiful images that you may imagine. On the other hand, using color or labels to help focus on important information in drawings is often worthwhile, as is using color to represent data values.

**Animation.** The StdDraw library supplies additional methods that provide limitless opportunities for creating interesting effects.

The default canvas size is 512-by-512 pixels; if you want to change it, call `setCanvasSize()` before any drawing commands. The `clear()` and `show()` methods support dynamic changes in the images on the computer screen. Such effects can provide compelling visualizations. We give an example next that also works for the printed page. There are more examples in the booksite that are likely to capture your imagination.
Bouncing ball. The **HelloWorld** of animation is to produce a black ball that appears to move around on the canvas. Suppose that the ball is at position \((r_x, r_y)\) and we want to create the impression of moving it to a new position nearby, such as, for example, \((r_x + 0.01, r_y + 0.02)\). We do so in two steps:

- Erase the drawing.
- Draw a black ball at the new position.

To create the illusion of movement, we iterate these steps for a whole sequence of positions (one that will form a straight line, in this case). But these two steps do not suffice, because the computer is so quick at drawing that the image of the ball will rapidly flicker between black and white instead of creating an animated image. Accordingly, StdDraw has a show() method that allows us to control when the results of drawing actions are actually shown on the display. You can think of it as collecting all of the lines, points, shapes, and text that we tell it to draw, and then immediately drawing them all when we issue the show() command. To control the apparent speed, show() takes an argument \(dt\) that tells StdDraw to wait \(dt\) milliseconds after doing the drawing. By default, StdDraw issues a show() after each line(), point(), or other drawing command; we turn that option off when we call StdDraw.show(t) and turn it back on when we call StdDraw.show() with no arguments. With these commands, we can create the illusion of motion with the following steps:

- Erase the drawing (but do not show the result).
- Draw a black ball at the new position.
- Show the result of both commands, and wait for a brief time.

**BouncingBall** (Program 1.5.6) implements these steps to create the illusion of a ball moving in the 2-by-2 box centered on the origin. The current position of the ball is \((r_x, r_y)\), and we compute the new position at each step by adding \(v_x\) to \(r_x\) and \(v_y\) to \(r_y\). Since \((v_x, v_y)\) is the fixed distance that the ball moves in each time unit, it represents the velocity. To keep the ball in the drawing, we simulate the effect of the ball bouncing off the walls according to the laws of elastic collision. This effect is easy to implement: when the ball hits a vertical wall, we just change the velocity in the x-direction from \(v_x\) to \(-v_x\), and when the ball hits a horizontal wall, we change the velocity in the y-direction from \(v_y\) to \(-v_y\). Of course, you have to download the code from the booksite and run it on your computer to see motion. To make the image clearer on the printed page, we modified BouncingBall to use a gray background that also shows the track of the ball as it moves (see Exercise 1.5.34).
Standard drawing completes our programming model by adding a "picture is worth a thousand words" component. It is a natural abstraction that you can use to better open up your programs to the outside world. With it, you can easily produce the function plots and visual representations of data that are commonly used in science and engineering. We will put it to such uses frequently throughout this book. Any time that you spend now working with the sample programs on the last few pages will be well worth the investment. You can find many useful examples on the booksite and in the exercises, and you are certain to find some outlet for your creativity by using StdDraw to meet various challenges. Can you draw an N-pointed star? Can you make our bouncing ball actually bounce (add gravity)? You may be surprised at how easily you can accomplish these and other tasks.

```java
public class StdDraw

void line(double x0, double y0, double x1, double y1)
void point(double x, double y)
void text(double x, double y, String s)
void circle(double x, double y, double r)
void filledCircle(double x, double y, double r)
void square(double x, double y, double r)
void filledSquare(double x, double y, double r)
void polygon(double[] x, double[] y)
void filledPolygon(double[] x, double[] y)
void setXscale(double x0, double x1)
void setYscale(double y0, double y1)
void setPenRadius(double r)
void setPenColor(Color c)
void setCanvasSize(int w, int h)
void clear(Color c)
void show(int dt)
void save(String filename)
```

Note: Methods with the same names but no arguments reset to default values.

API for our library of static methods for standard drawing

1.5 Input and Output

Standard audio As a final example of a basic abstraction for output, we consider StdAudio, a library that you can use to play, manipulate, and synthesize sound files. You probably have used your computer to process music. Now you can write programs to do so. At the same time, you will learn some concepts behind a venerable and important area of computer science and scientific computing: digital signal processing. We will only scratch the surface of this fascinating subject, but you may be surprised at the simplicity of the underlying concepts.

Concert A. Sound is the perception of the vibration of molecules, in particular, the vibration of our eardrums. Therefore, oscillation is the key to understanding sound. Perhaps the simplest place to start is to consider the musical note A above middle C, which is known as concert A. This note is nothing more than a sine wave, scaled to oscillate at a frequency of 440 times per second. The function \( \sin(\theta) \) repeats itself once every \( 2\pi \) units on the \( x \)-axis, so if we measure \( t \) in seconds and plot the function \( \sin(2\pi \times 440) \), we get a curve that oscillates 440 times per second.

When you play an A by plucking a guitar string, pushing air through a trumpet, or causing a small cone to vibrate in a speaker, this sine wave is the prominent part of the sound that you hear and recognize as concert A. We measure frequency in hertz (cycles per second). When you double or halve the frequency, you move up or down one octave on the scale. For example, 880 hertz is one octave above concert A and 110 hertz is two octaves below concert A. For reference, the frequency range of human hearing is about 20 to 20,000 hertz. The amplitude (y-value) of a sound corresponds to the volume. We plot our curves between \(-1\) and \(+1\) and assume that any devices that record and play sound will scale as appropriate, with further scaling controlled by you when you turn the volume knob.

<table>
<thead>
<tr>
<th>note</th>
<th>frequency</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>440.00</td>
</tr>
<tr>
<td>A#</td>
<td>466.20</td>
</tr>
<tr>
<td>B</td>
<td>493.88</td>
</tr>
<tr>
<td>C</td>
<td>523.25</td>
</tr>
<tr>
<td>C#</td>
<td>554.37</td>
</tr>
<tr>
<td>D</td>
<td>587.33</td>
</tr>
<tr>
<td>D#</td>
<td>622.25</td>
</tr>
<tr>
<td>E</td>
<td>659.26</td>
</tr>
<tr>
<td>F</td>
<td>698.46</td>
</tr>
<tr>
<td>F#</td>
<td>739.99</td>
</tr>
<tr>
<td>G</td>
<td>783.99</td>
</tr>
<tr>
<td>G#</td>
<td>830.61</td>
</tr>
<tr>
<td>A</td>
<td>880.00</td>
</tr>
</tbody>
</table>

Notes, numbers, and waves
Other notes. A simple mathematical formula characterizes the other notes on the chromatic scale. There are twelve notes on the chromatic scale, divided equally on a logarithmic (base 2) scale. We get the $i$th note above a given note by multiplying its frequency by the $(i/12)$th power of 2. In other words, the frequency of each note in the chromatic scale is precisely the frequency of the previous note in the scale multiplied by the twelfth root of two (about 1.06). This information suffices to create music! For example, to play the tune *Faire Jacques*, we just need to play each of the notes $A B C A$ by producing sine waves of the appropriate frequency for about half a second and then repeat the pattern. The primary method in the StdAudio library, `StdAudio.play()`, allows you to do just that.

**Sampling.** For digital sound, we represent a curve by sampling it at regular intervals, in precisely the same manner as when we plot function graphs. We sample sufficiently often that we have an accurate representation of the curve—a widely used sampling rate for digital sound is 44,100 samples per second. For concert $A$, that rate corresponds to plotting each cycle of the sine wave by sampling it at about 100 points. Since we sample at regular intervals, we only need to compute the $y$-coordinates of the sample points. It is that simple: we represent sound as an array of numbers (double values that are between $-1$ and $+1$). Our standard sound library method `StdAudio.play()` takes an array as its argument and plays the sound represented by that array on your computer. For example, suppose that you want to play concert $A$ for 10 seconds. At 44,100 samples per second, you need an array of 441,000 double values. To fill the array, use a for loop that samples the function $\sin(2\pi t \times 440)$ at $t = 0/44100$.

1/44100, 2/44100, 3/44100, ... 4410000/44100. Once we fill the array with these values, we are ready for `StdAudio.play()`, as in the following code:

```java
int sps = 44100; // samples per second
int Hz = 440; // concert A
double duration = 10.0; // ten seconds
int N = (int) (sps * duration); // total number of samples
double[] a = new double[N+1];
for (int i = 0; i <= N; i++)
a[i] = Math.sin(2*Math.PI * i * Hz / sps);
StdAudio.play(a);
```

This code is the HelloWorld of digital audio. Once you use it to get your computer to play this note, you can write code to play other notes and make music! The difference between creating sound and plotting an oscillating curve is nothing more than the output device. Indeed, it is instructive and entertaining to send the same numbers to both standard draw and standard audio (see Exercise 1.5.23).

**Saving to a file.** Music can take up a lot of space on your computer. At 44,100 samples per second, a four-minute song corresponds to $4 \times 60 \times 44100 = 10,584,000$ numbers. Therefore, it is common to represent the numbers corresponding to a song in a binary format that uses less space than the string-of-digits representation that we use for standard input and output. Many such formats have been developed in recent years—`StdAudio` uses the .wav format. You can find some information about the .wav format on the booksite, but you do not need to know the details, because `StdAudio` takes care of the conversions for you. Our standard library for audio allows you to play .wav files, to write programs to create and manipulate arrays of double values, and to read and write them as .wav files.

```java
public class StdAudio
{
    public void play(String file) // play the given .wav file
    public void play(double[] A) // play the given sound wave
    public void play(double s) // play sample for 1/44100 second
    public void save(String file, double[] a) // save to a .wav file
    public double[] read(String file) // read from a .wav file
}
```

API for our library of static methods for standard audio.
Program 1.5.7  Digital signal processing

```java
public class PlayThatTune {
    public static void main(String[] args) {
        // Read a tune from StdIn and play it.
        int sps = 44100;
        while (!StdIn.isEmpty()) {
            // Read and play one note.
            int pitch = StdIn.readInt();
            double duration = StdIn.readDouble();
            double Hz = 440 * Math.pow(2, pitch / 12.0);
            int N = (int) (sps * duration);
            double[] a = new double[N + 1];
            for (int i = 0; i <= N; ++i) {
                a[i] = Math.sin(2 * Math.PI * i * Hz / sps);
            }
            StdAudio.play(a);
        }
    }
}
```

This is a data-driven program that plays pure tones from the notes on the chromatic scale, specified on standard input as a pitch (distance from concert A) and a duration (in seconds). The test client reads the notes from standard input, creates an array by sampling a sine wave of the specified frequency and duration at 44100 samples per second, and then plays each note by calling StdAudio.play().

PlayThatTune (Program 1.5.7) is an example that shows how easily we can create music with StdAudio. It takes notes from standard input, indexed on the chromatic scale from concert A, and plays them on standard audio. You can imagine all sorts of extensions on this basic scheme, some of which are addressed in the exercises. We include StdAudio in our basic arsenal of programming tools because sound processing is one important application of scientific computing that is certainly familiar to you. Not only has the commercial application of digital signal processing had a phenomenal impact on modern society, but the science and engineering behind it combines physics and computer science in interesting ways. We will study more components of digital signal processing in some detail later in the book. (For example, you will learn in Section 2.1 how to create sounds that are more musical than the pure sounds produced by PlayThatTune.)

I/O is a particularly convincing example of the power of abstraction because standard input, standard output, standard draw, and standard audio can be tied to different physical devices at different times without making any changes to programs. Although devices may differ dramatically, we can write programs that do I/O without depending on the properties of specific devices. From this point forward, we will use methods from StdOut, StdIn, StdDraw, and/or StdAudio in nearly every program in this book, and you will use them in nearly all of your programs, so make sure to download copies of these libraries. For economy, we collectively refer to these libraries as Std*. One important advantage of using such libraries is that you can switch to new devices that are faster, cheaper, or hold more data without changing your program at all. In such a situation, the details of the connection are a matter to be resolved between your operating system and the Std* implementations. On modern systems, new devices are typically supplied with software that resolves such details automatically for both the operating system and for Java.

Conceptually, one of the most significant features of the standard input, standard output, standard draw, and standard audio data streams is that they are infinite. From the point of view of your program, there is no limit on their length. This point of view not only leads to programs that have a long useful life (because they are less sensitive to changes in technology than programs with built-in limits). It also is related to the Turing machine, an abstract device used by theoretical computer scientists to help us understand fundamental limitations on the capabilities of real computers. One of the essential properties of the model is the idea of a finite discrete device that works with an unlimited amount of input and output.
Q&A

Q. Why are we not using the standard Java libraries for input, graphics, and sound?

A. We are using them, but we prefer to work with simpler abstract models. The Java libraries behind StdIn, StdDraw, and StdAudio are built for production programming, and the libraries and their APIs are a bit unwieldy. To get an idea of what they are like, look at the code in StdIn.java, StdDraw.java, and StdAudio.java.

Q. So, let me get this straight. If I use the format %2.4f for a double value, I get two digits before the decimal point and four digits after, right?

A. No, that specifies just four digits after the decimal point. The first value is the width of the whole field. You want to use the format %7.2f to specify seven characters in total, four before the decimal point, the decimal point itself, and two digits after the decimal point.

Q. What other conversion codes are there for printf()?

A. For integer values, there is o for octal and x for hexadecimal. There are also numerous formats for dates and times. See the book site for more information.

Q. Can my program re-read data from standard input?

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

Q. What happens if my program attempts to read data from standard input after it is exhausted?

A. You will get an error. StdIn.isEmpty() allows you to avoid such an error by checking whether there is more input available.

Q. What does the error message Exception in thread "main" java.lang.NoClassDefFoundError: StdIn mean?

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

Q. I have a different working directory for each project that I am working on, so I have copies of StdOut.java, StdIn.java, StdDraw.java, and StdAudio.java in each of them. Is there some better way?

A. Yes. You can put them all in one directory and use the "classpath" mechanism to tell Java where to find them. This mechanism is operating-system dependent—you can find instructions on how to use it on the booksite.

Q. My terminal window hangs at the end of a program using StdAudio. How can I avoid having to use <ctrl-c> to get a command prompt?

A. Add a call to System.exit(0) as the last line in main(). Don't ask why.

Q. So I use negative integers to go below concert A when making input files for PlayThatTune?

A. Right. Actually, our choice to put concert A at 0 is arbitrary. A popular standard is to start numbering at the C five octaves below concert A. By that convention, concert A is 69 and you do not need to use negative numbers.
1.5.1 Write a program that reads in integers (as many as the user enters) from standard input and prints out the maximum and minimum values.

1.5.2 Modify your program from the previous exercise to insist that the integers must be positive (by prompting the user to enter positive integers whenever the value entered is not positive).

1.5.3 Write a program that takes an integer $N$ from the command line, reads $N$ double values from standard input, and prints their mean (average value) and standard deviation (square root of the sum of the squares of their differences from the average).

1.5.4 Extend your program from the previous exercise to create a filter that prints all the values that are further than $1.5$ standard deviations from the mean. Use an array.

1.5.5 Write a program that reads in a sequence of integers and prints out both the integer that appears in a longest consecutive run and the length of the run. For example, if the input is $1 2 2 2 1 5 1 1 7 7 7 7 1 1$, then your program should print Longest run: 4 consecutive 7s.

1.5.6 Write a filter that reads in a sequence of integers and prints back out the integers, removing repeated values that appear consecutively. For example, if the input is $1 2 2 2 1 5 1 1 7 7 7 7 1 1 1 1 1 1 1 1 1$, your program should print $1 2 2 2 1 5 1 7 1$.

1.5.7 Write a program that takes a command-line argument $N$, reads in $N-1$ distinct integers between 1 and $N$, and determines the missing value.

1.5.8 Write a program that reads in positive real numbers from standard input and prints out their geometric and harmonic means. The geometric mean of $N$ positive numbers $x_1, x_2, \ldots, x_N$ is $(x_1 \times x_2 \times \ldots \times x_N)^{1/N}$. The harmonic mean is $(1/x_1 + 1/x_2 + \ldots + 1/x_N)/(1/N)$. Hint: For the geometric mean, consider taking logs to avoid overflow.

1.5.9 Suppose that the file input.txt contains the two strings F and F. What does the following command do (see Exercise 1.2.35)?

```java
java Dragon < input.txt | java Dragon | java Dragon
```

```java
public class Dragon {
    public static void main(String[] args) {
        String dragon = StdIn.readString();
        String nogard = StdIn.readString();
        StdOut.println(dragon + "L" + nogard);
        StdOut.println("R");
        StdOut.println(dragon + "R" + nogard);
        StdOut.println();
    }
}
```

1.5.10 Write a filter TenPerLine that takes a sequence of integers between 0 and 99 and prints 10 integers per line, with columns aligned. Then write a program RandomIntSeq that takes two command-line arguments $M$ and $N$ and outputs $N$ random integers between 0 and $M-1$. Test your programs with the command java RandomIntSeq 200 100 | java TenPerLine.

1.5.11 Write a program that reads in text from standard input and prints out the number of words in the text. For the purpose of this exercise, a word is a sequence of non-whitespace characters that is surrounded by whitespace.

1.5.12 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.

1.5.13 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? For each, the input comes from standard input and consists of $N$ real numbers between 0 and 1.
1.5.14 Write a program that prints a table of the monthly payments, remaining principal, and interest paid for a loan, taking three numbers as command-line arguments: the number of years, the principal, and the interest rate (see Exercise 1.2.24).

1.5.15 Write a program that takes three command-line arguments x, y, and z, reads from standard input a sequence of point coordinates (x, y, z), and prints the coordinates of the point closest to (x, y, z). Recall that the square of the distance between (x, y, z) and (x<sub>i</sub>, y<sub>i</sub>, z<sub>i</sub>) is \((x - x<sub>i</sub>)^2 + (y - y<sub>i</sub>)^2 + (z - z<sub>i</sub>)^2\). For efficiency, do not use Math.sqrt() or Math.pow().

1.5.16 Given the positions and masses of a sequence of objects, write a program to compute their center-of-mass, or centroid. The centroid is the average position of the N objects, weighted by mass. If the positions and masses are given by \((x<sub>i</sub>, y<sub>i</sub>, m<sub>i</sub>)\), then the centroid \((x, y, m)\) is given by:

\[
\begin{align*}
m &= m_1 + m_2 + \ldots + m_N \\
x &= (m_1 x_1 + m_2 x_2 + \ldots + m_N x_N) / m \\
y &= (m_1 y_1 + m_2 y_2 + \ldots + m_N y_N) / m
\end{align*}
\]

1.5.17 Write a program that reads in a sequence of real numbers between -1 and +1 and prints out their average magnitude, average power, and the number of zero crossings. The average magnitude is the average of the absolute values of the data values. The average power is the average of the squares of the data values. The number of zero crossings is the number of times a data value transitions from a strictly negative number to a strictly positive number, or vice versa. These three statistics are widely used to analyze digital signals.

1.5.18 Write a program that takes a command-line argument \(N\) and plots an \(N\)-by-\(N\) checkerboard with red and black squares. Color the lower left square red.

1.5.19 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 points of size on the circumference of a circle, and then, with probability \(p\) for each pair of points, draws a gray line connecting them.

1.5.20 Write code to draw hearts, spades, clubs, and diamonds. To draw a heart, draw a diamond, then attach two semicircles to the upper left and upper right sides.

1.5.21 Write a program that takes a command-line argument \(N\) and plots a rose with \(N\) petals (if \(N\) is odd) or \(2N\) petals (if \(N\) is even), by plotting the polar coordinates \((r, \theta)\) of the function \(f(\theta) = \sin(N\theta)\) for \(\theta\) ranging from 0 to \(2\pi\) radians.

1.5.22 Write a program that takes a string \(s\) from the command line and displays it in banner style on the screen, moving from left to right and wrapping back to the beginning of the string as the end is reached. Add a second command-line argument to control the speed.
1.5.23 Modify PlayThatTune to take additional command-line arguments that control the volume (multiply each sample value by the volume) and the tempo (multiply each note's duration by the tempo).

1.5.24 Write a program that takes the name of a .wav file and a playback rate \( r \) as command-line arguments and plays the file at the given rate. First, use `StdAudio`'s `read()` to read the file into an array \( a[] \). If \( r = 1 \), just play \( a[] \); otherwise create a new array \( b[] \) of approximate size \( r \times \) length. If \( r < 1 \), populate \( b[] \) by sampling from the original; if \( r > 1 \), populate \( b[] \) by interpolating from the original. Then play \( b[] \).

1.5.25 Write programs that use `StdDraw` to create each of the following designs.

1.5.26 Write a program `Circles` that draws filled circles of random size at random positions in the unit square, producing images like those below. Your program should take four command-line arguments: the number of circles, the probability that each circle is black, the minimum radius, and the maximum radius.

1.5.27 Visualizing audio. Modify `PlayThatTune` to send the values played to standard drawing, so that you can watch the sound waves as they are played. You will have to experiment with plotting multiple curves in the drawing canvas to synchronize the sound and the picture.

1.5.28 Statistical polling. When collecting statistical data for certain political polls, it is very important to obtain an unbiased sample of registered voters. Assume that you have a file with \( N \) registered voters, one per line. Write a filter that prints out a random sample of size \( M \) (see `Program` 1.4.1).

1.5.29 Terrain analysis. Suppose that a terrain is represented by a two-dimensional grid of elevation values (in meters). A peak is a grid point whose four neighboring cells are strictly lower. Write a program `Peaks` that reads a terrain from standard input and then computes and prints the number of peaks in the terrain.

1.5.30 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.

1.5.31 Spirographs. Write a program that takes three parameters \( R, r, \) and \( a \) from the command line and draws the resulting `spirograph`. A spirograph (technically, an epicycloid) is a curve formed by rolling a circle of radius \( r \) around a larger fixed circle of radius \( R \). If the pen offset from the center of the rolling circle is \((r + a)\), then the equation of the resulting curve at time \( t \) is given by

\[
\begin{align*}
x(t) &= (R + r) \cos(t) - (r + a) \cos ((R + r)t/r) \\
y(t) &= (R + r) \sin(t) - (r + a) \sin ((R + r)t/r)
\end{align*}
\]

Such curves were popularized by a best-selling toy that contains discs with gear teeth on the edges and small holes that you could put a pen in to trace spirographs.
1.5.32 Clock. Write a program that displays an animation of the second, minute, and hour hands of an analog clock. Use the method StdDraw.show(1000) to update the display roughly once per second.

1.5.33 Oscilloscope. Write a program to simulate the output of an oscilloscope and produce Lissajous patterns. These patterns are named after the French physicist, Jules A. Lissajous, who studied the patterns that arise when two mutually perpendicular periodic disturbances occur simultaneously. Assume that the inputs are sinusoidal, so that the following parametric equations describe the curve:

\[
x(t) = A_x \sin (w_x t + \theta_x) \\
y(t) = A_y \sin (w_y t + \theta_y)
\]

Take the six parameters \(A_x, w_x, \theta_x, A_y, w_y, \text{ and } \theta_y\) from the command line.

1.5.34 Bouncing ball with tracks. Modify BouncingBall to produce images like the ones shown in the text, which show the track of the ball on a gray background.

1.5.35 Bouncing ball with gravity. Modify BouncingBall to incorporate gravity in the vertical direction. Add calls to StdAudio.play() to add one sound effect when the ball hits a wall and a different one when it hits the floor.

1.5.36 Random tunes. Write a program that uses StdAudio to play random tunes. Experiment with keeping in key, assigning high probabilities to whole steps, repetition, and other rules to produce reasonable melodies.

1.5.37 Tile patterns. Using your solution to Exercise 1.5.25, write a program TilePattern that takes a command-line argument N and draws an N-by-N pattern, using the tile of your choice. Add a second command-line argument that adds a checkerboard option. Add a third command-line argument for color selection. Using the patterns on the facing page as a starting point, design a tile floor. Be creative! Note: These are all designs from antiquity that you can find in many ancient (and modern) buildings.