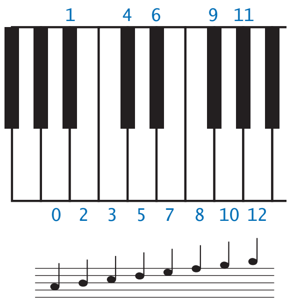
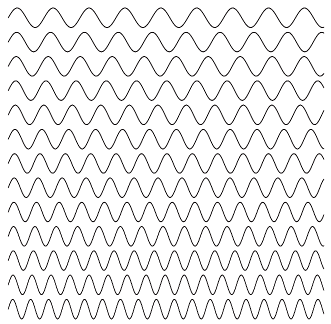
Standard audio As a final example of a basic abstraction for output, we consider StdAudio, a library that you can use to play, manipulate, and synthesize sound. You probably have used your computer to process music. Now you can write programs to do so. At the same time, you will learn some concepts behind a venerable and important area of computer science and scientific computing: *digital signal processing*. We will merely 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 $\sin(t)$ repeats itself once every 2π units, so if we measure t in seconds and plot the function $\sin(2\pi t \times 440)$, 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.



note	*i*	*frequency*
A	0	440.00
A♯ *or* B♭	1	466.16
B	2	493.88
C	3	523.25
C♯ *or* D♭	4	554.37
D	5	587.33
D♯ *or* E♭	6	622.25
E	7	659.26
F	8	698.46
F♯ *or* G♭	9	739.99
G	10	783.99
G♯ *or* A♭	11	830.61
A	12	880.00

$440 \times 2^{i/12}$



Notes, numbers, and waves

Other notes. A simple mathematical formula characterizes the other notes on the chromatic scale. There are 12 notes on the chromatic scale, evenly spaced 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 2 (about 1.06). This information suffices to create music! For example, to play the tune *Frère Jacques,* play each of the notes *A B C# A* by producing sine waves of the appropriate frequency for about half a second each, and then repeat the pattern. The primary method in the `StdAudio` library, `StdAudio.play()`, allows you to do exactly this.

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 real numbers* (between -1 and $+1$). The 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 a `double` array of length 441,001. To fill in the array, use a `for` loop that samples the function $\sin(2\pi t \times 440)$ at $t = 0/44{,}100$, 1/44,100, 2/44,100, 3/44,100, …, 441,000/44,100. Once we fill the array with these values, we are ready for `StdAudio.play()`, as in the following code:

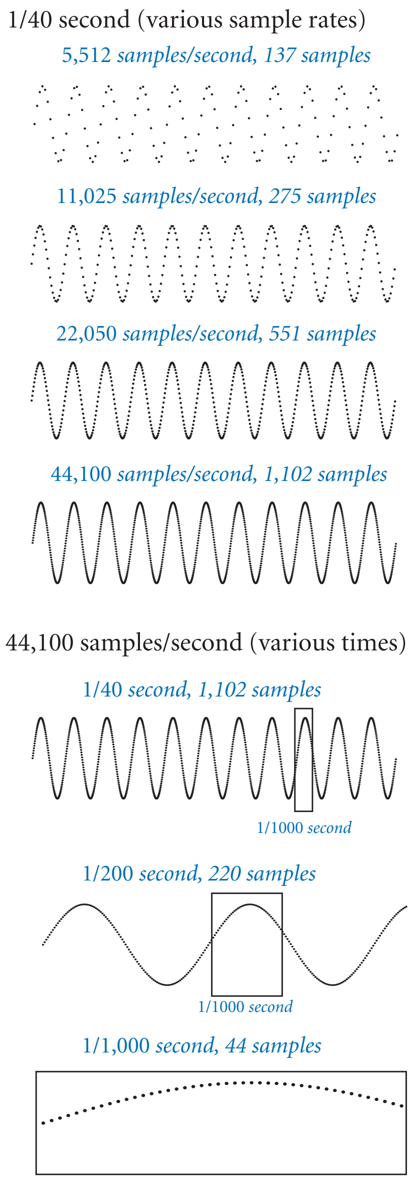
```
int SAMPLING_RATE = 44100;       // samples per second
int hz = 440;                    // concert A
double duration = 10.0;          // ten seconds
int n = (int) (SAMPLING_RATE * duration);
double[] a = new double[n+1];
for (int i = 0; i <= n; i++)
   a[i] = Math.sin(2 * Math.PI * i * hz / SAMPLING_RATE);
StdAudio.play(a);
```

This code is the “Hello, World” 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 drawing and standard audio (see Exercise 1.5.27).

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 × 60 × 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 read `.wav` files, write `.wav` files, and convert `.wav` files to arrays of `double` values for processing.

`PlayThatTune` (Program 1.5.7) is an example that shows how you can use `StdAudio` to turn your computer into a musical instrument. 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 standard audio 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 combine 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`.)



Sampling a sine wave

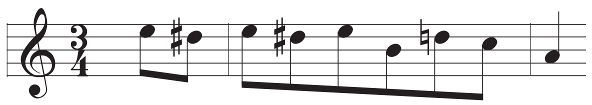
Program 1.5.7 Digital signal processing

```
public class PlayThatTune
{
   public static void main(String[] args)
   {  // Read a tune from StdIn and play it.
      int SAMPLING_RATE = 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) (SAMPLING_RATE * duration);
         double[] a = new double[n+1];
         for (int i = 0; i <= n; i++)
            a[i] = Math.sin(2*Math.PI * i * hz / SAMPLING_RATE);
         StdAudio.play(a);
      }
   }
}
```

`pitch`	*distance from A*
`duration`	*note play time*
`hz`	*frequency*
`n`	*number of samples*
`a[]`	*sampled sine wave*

This data-driven program turns your computer into a musical instrument. It reads notes and durations from standard input and plays a pure tone corresponding to each note for the specified duration on standard audio. Each note is specified as a pitch (distance from concert A). After reading each note and duration, the program creates an array by sampling a sine wave of the specified frequency and duration at 44,100 samples per second, and plays it using `StdAudio.play()`.

```
% more elise.txt
7 0.25
6 0.25
7 0.25
6 0.25
7 0.25
2 0.25
5 0.25
3 0.25
0 0.50
```



```
% java PlayThatTune < elise.txt
```

The API table below summarizes the methods in `StdAudio`:

`public class StdAudio`		
`void`	`play(String filename)`	*play the given `.wav` file*
`void`	`play(double[] a)`	*play the given sound wave*
`void`	`play(double x)`	*play sample for 1/44,100 second*
`void`	`save(String filename, double[] a)`	*save to a `.wav` file*
`double[]`	`read(String filename)`	*read from a `.wav` file*

API for our library of static methods for standard audio

Summary I/O is a compelling example of the power of abstraction because standard input, standard output, standard drawing, 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 can 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. 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, are 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 both for the operating system and for Java.