**Sound Lab**

The AP CS exam will have questions that challenge your ability to mentally track what is happening inside loops, often iterating over arrays; we'd better practice! If you need to review the syntax for loops and/or arrays, information will have to be provided and copied to our AP class folders. Please let your teacher know!

1. Create a Runner class with a public static void main(String[] args) method where you will test all of your code. Next, create a SoundLabProbs class (with a default constructor) that will contain all the methods below. Call the methods on a SoundLabProbs object in Runner to test them.
2. Complete the method public void triangle(int n), that *prints* the pattern below (nested for loops will be very helpful). Note this method has void return type, you can't print what it returns.

1

22

333

4444 //n = 4

1

22

333

4444

55555

666666

7777777 //n = 7

1. Complete the method public int lastIndexOf(int[] nums, int value), that returns the **last** index where the valueparameter occurs. The method should return -1 if value can't be found. Traversing the array in reverse would be wise.

lastIndexOf(new int[] {8, 2, 4, 6, 8}, 8) >>> 4

lastIndexOf(new int[] {2, 4, 6, 12}, 8) >>> -1

1. (Riddle) You have two buckets. One holds exactly five gallons and the other three gallons. How can you measure exactly four gallons of water into the five-gallon bucket? Assume you have an unlimited supply of water and that there are no measurement markings of any kind on the buckets.
2. Complete the method public int range(int[] nums), that will return the "range" of the values in the array. The range is defined as the difference between the smallest and largest elements in the array. This must be done without sorting the array.

range(new int[] {8, 3, 5, 7, 2, 4}) >>> 6

range(new int[] {15, 22, 8, 19, 31}) >>> 23

1. Complete the method public int minDifference(int[] nums) that returns the minimum difference (closest to zero) between two *neighboring* numbers in an array.

minDifference(new int[] {4, 8, 6, 1, 5, 9, 4}) >>> 2 //between 8 and 6

1. Complete the method public int priceIsRight(int[] bids, int price), that (similar to the famous game show) returns the element in bids that is closest to price *without going over*. If all elements in bids are larger than price, your method should return -1.

priceIsRight(new int[] {900, 885, 990, 1}, 800) >>> 1

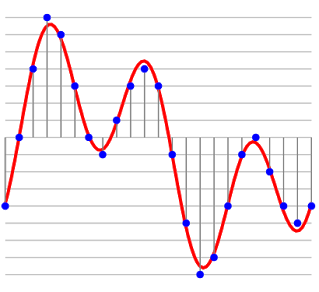
priceIsRight(new int[] {1500, 1600, 2000, 2500}, 1900) >>> 1600

1. Complete the method public int[] productExceptSelf(int[] nums), that will return a new array result such that result[i] is the product of all the elements in nums *except* nums[i]. Note this method returns an array; check the "Printing an array" powerpoint (CS 1 folder) for help.

productExceptSelf(new int[] {1, 2, 3, 4}) >>> {24, 12, 8, 6}

**Sound lab**

Computers represent sound as a sequence of numbers. Each number represents the intensity of a sound at specific moment. These numbers are called ***samples***.



1. Peruse the information in the **"Basic sound concepts"** document, in the lab folder. Watch the videos for a crash course on how sound works, then read on for a brief primer on how computers *represent* sound. Understanding sound and how computers represent sound will help immensely in making sense of this lab.
2. We will represent (sound) samples as doubles ranging from -1.0 to 1.0.  Import the starter code files; if you're using Eclipse now, [this](https://youtu.be/y5OD8_xqCK0) video will help. In a class with a main() method, execute the following code segment (if using BlueJ, note that the SoundTester class won't compile yet):

double[] clip = {0.5, 1, 0, -1};

Sound.show(clip); //static methods are called on the Sound class

* 1. You should see a window showing a graph of 0.0005 seconds of sound. Convince yourself that the image you see corresponds to the values in the array.
  2. Now try playing the sound by adding the following line; you will probably hear nothing more than a brief click. Check your volume if you don't hear anything.



Sound.play(clip);

* 1. Comment out the previous lines, then execute the following code segment:

double[] clip = Sound.pureTone(3.0, 1.0);

Sound.show(clip);

You should see a sound wave of a pure tone - the kind of sound produced by a tuning fork (above right).

The first argument (parameter) in the pureTone method is the frequency. This tells us how many times the wave should repeat each second. This wave repeats 3 times per second, so its frequency is 3 Hertz (Hz). The second number indicates that we want the tone to last for 1 second (pure tones are really sine waves).

Try listening (using the Sound.play() method seen previously) to tones with frequencies between about 200 and 1000 Hz. What do you notice about these sounds?

Note that for high frequencies, the graph produced by Sound.show() will look like nothing more than vertical lines – the sine wave is too highly compressed for the wave to be visible.

The Sound utility class (included) is responsible for producing sound, and can do the following (more information on these methods in the class' comments):

**class Sound.java**

----------------------------------------------------------------------

static void  **show**(double[] clip)

static void  **play**(double[] clip)

static double[] **pureTone**(double frequency, double seconds)

static int  **toNumSamples**(double seconds)

1. Create a class **SoundClip.java**. The SoundClip class is a bundle of the actual sound clip you'll hear playing and some methods for sound manipulation. You will write all the following code in this file.
   1. Add an instance variable double[] clip that stores the samples of this sound clip.
   2. Add a default (no-parameter) constructor that initializes clip to null.
   3. Add a constructor that initializes clip to the value of the parameter.
2. We can change the volume of a sound clip by multiplying each sample by a specified number. For example, multiplying by 0.5 will halve the volume; multiplying by 1.5 will increase volume by 50%.

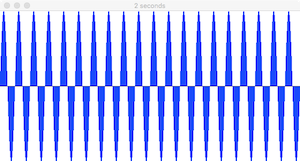
In SoundClip, complete the void adjustVolume(double factor) method, which adjusts the volume of clip by multiplying each sample by the given factor.

* 1. In the **SoundTester.java** file, uncomment the call to the corresponding method that will run a test of your adjustVolume method. If using BlueJ, you'll need to comment out the calls to methods you haven't added yet. You will do this for all the following methods.

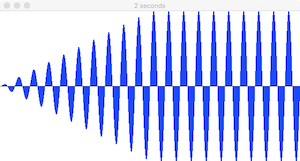
1. Complete the void mix(double[] clip1, double[] clip2) method, which takes in two sound clips and *sets* clip to the sum of the given clips. For example, mixing [-0.2, 0.1, -0.9] with [-0.3, 0.0, 0.9, 0.4, 0.8], clip should store [-0.5, 0.1, 0.0, 0.4, 0.8].
   1. Note that this method is *setting* (not adjusting) the value of clip. You can't assume that clip has already been initialized (and is long enough to store the mixture of clip1 and clip2). You'd better initialize it! Check the **FAQ** for more info.
2. Complete the void append(String fileName) method, which appends the contents (sound samples) of the text file specified by fileName to clip. The file contains a single integer representing the number of samples, followed by samples number of space-separated doubles.
   1. Arrays are objects, and ***immutable*** - you can't add or remove elements after creation. Create a new array result that is clip.length + samples long. When you're finishing copying / appending values, the instance variable clip should store a reference to the result array (clip = result).

**Note:** Refer to the powerpoints and [this](https://youtu.be/y5OD8_xqCK0) video for help reading from a text file. Make sure the text file is the project folder (NOT the ***src*** folder if using Eclipse). Also, make sure you add throws IOException to EVERY method header that does file reading (or calls another method that does).

1. Complete the void fadeIn(double seconds) method, which takes a length of time in seconds, and modifies clip so that it begins at a volume of 0 and steadily increases to its original volume at the end of the given length of time. For example, if clip is 2 seconds long and looks like look like this:



then fadeIn(1.0) should modify clip as follows:



**Note:**  Use Sound's toNumSamples method to convert the given length of time into the corresponding number of samples (array elements). For example, if seconds is 2.0, then Sound.toNumSamples(seconds) might return something like 16,000, indicating that the first 16,000 values in the array must be adjusted.

**Hint:**  Suppose we wish to fade in over the course of the first 600 samples in the array. Then we should multiply the first sample by 0.0, the 300th sample by 0.5, and the 600th sample by 1.0. In general, the nth sample should be multiplied by n/600 (in this case). Be careful when dividing an integer by an integer!

1. Complete the void fadeOut(double seconds) method, which takes a length of time and modifies clip such that the sound fades out for the specified number of seconds. For example, if clip refers to a 3-second sound clip, then fadeOut(1.0) will leave the first 2 seconds of the clip unchanged and modify the last second of the clip so that it begins gradually decreases to an intensity of 0 by the end.
2. Complete the void reverse() method that will reverse clip.
3. Complete the void speedUp(double factor) method that speeds up clip by a given factor. For example, speeding up a 10-second clip by a factor of 2.0 would make the clip play twice as fast (in 5 seconds), and therefore will sound higher pitched. This method does not slow down; for the curious, check the **(Advanced)** section for why.

**(Advanced) More sound manipulation techniques**

Should you have the time / inclination, try these advanced sound manipulation techniques:

**HarmonicTone**

The tones produced by real musical instruments are not "pure" tones. For example, if a guitar plays an "A" note, we hear the string vibrating at 220 Hz (a pure tone), but we also hear the string vibrating at 2 \* 220 = 440 Hz, and at 3 \* 220 = 660 Hz, and at 4 \* 220 = 880 Hz, and so on. The lowest frequency is called the *fundamental frequency*, and the higher frequencies are called *overtones*.

Typically, the lower frequencies are loudest (have the highest amplitudes), and the higher overtones are much quieter (have lower amplitudes). The resulting sound is much richer than a pure tone. Different musical instruments produce sounds whose overtones have different relative volumes.

1. Add a static utility method that will adjust then return the volume of the supplied array (NOT SoundClip's clip instance variable):

public static double[] adjustVolume(double[] newClip, double factor)

This method will operate on the *local* variable newClip (NOT the *instance* variable clip), though you can copy/paste more of your code from the instance method. We will discuss what the static keyword means later this year, but you can check the powerpoint now if you're curious.

1. Complete the void harmonicTone(double frequency, double seconds, double[] amplitudes) method that sets clip to a new sound clip of the given length that is a *mix* of pure tones of different frequencies and amplitudes. The method takes a frequency, a duration in seconds, and an array containing amplitudes (volumes) to be used. Suppose the frequency is 300, the duration is 2.0 seconds, and the amplitudes are [1.0, 0.5, 0.2, 0.5, 0.1]. The resulting sound clip will be a mix of the following tones:

* 300 \* 1 = 300 Hz for 2.0 seconds with amplitude 1.0 (the default volume used by pureTone)
* 300 \* 2 = 600 Hz for 2.0 seconds with amplitude 0.5 (half of the volume used by pureTone)
* 300 \* 3 = 900 Hz for 2.0 seconds with amplitude 0.2
* 300 \* 4 = 1200 Hz for 2.0 seconds with amplitude 0.5
* 300 \* 5 = 1500 Hz for 2.0 seconds with amplitude 0.1

Use the *static* adjustVolume() method to scale the overtones' volumes (amplitudes) like this:

SoundClip.adjustVolume() //static methods called on the class, not an object

**PrettyTone**

Experiment with different inputs for harmonicTone's amplitudes. Also experiment with different amounts of fade-in and fade-out. Some musical instruments produce sounds that fade out slowly (e.g. guitar), while others fade out more quickly (e.g. flute). Some fade in quickly (guitar) while others fade in more slowly (flute). Complete the double[] prettyTone(double frequency, double seconds) method to produce a musical tone you like, given the supplied (fundamental) frequency and duration in seconds.

**Note:** This method sets clip to the resulting (pretty) tone, but also returns the current state of clip (necessary for the melody() method).

**Melody**

Given a musical note of some frequency, we can produce the next higher note (a half step up) by multiplying the frequency by the twelfth root of 2 (which is very close to 1.06). Likewise, we can produce a slightly lower note by dividing by 1.06. For example, starting from 200 Hz, we can keep multiplying by 1.06 to produce the following frequencies:

200, 212, 224, 238, 252, 267, 283, 300, 317, 336, 356, 378, 400

Playing a sequence of tones created in this manner will generally sound musical. For example, we might play the sequence 200, 200, 300, 300, 336, 336, 300.

Complete the void melody() method, which should *join* together several musical notes produced by prettyTone to produce a very decent melody.

**AutoVolume**

public void autoVolume() - Modifies the given clip to be louder or quieter so that its loudest sample is 1.0 or -1.0.

**Echo**

public void echo(int numCopies, double delayInSeconds) - Create sounds that echo, by combining a given clip with copies at lower and lower volumes. Ideally, the second copy would start playing at the same time the first copy is still finishing. This is very useful in producing musical effects like reverb and digital delay.

**Random**

Experiment with random. What does it sound like if you create a sound clip consisting of random doubles in the range from -1.0 to 1.0? Can you write a program to produce random pleasing melodies?

**Vibrato**

Write a method that will add a vibrato to a tone. Vibrato is the repeated small fluctuation of pitch on a sustained note. Human voices can add vibrato and tends to sound nice for a variety of reasons.

**SlowDown**

Changing the speed by a factor of 0.5 would make a 10-second clip play half as fast (in 20 seconds) and will therefore sound lower pitched. However, this can sound terrible! Think about it - you are doubling the duration, without adding any more information. Research and implement a way to slow down a clip while maintaining decent quality.

*Sound Lab adapted from the* ***Sound*** *project by Dave Feinberg*

*https://sites.google.com/site/feinbergcompsci*