Guitar Hero

**Purpose**

This lab was designed to demonstrate arrays of objects and to practice implementing classes.

**Program Shells**

* [StdDraw.java](https://www.cs.utexas.edu/~scottm/cs312/javaCode/A11GuitarHero/StdDraw.java) - This is a non-standard Java file similar to DrawingPanel. It allows the creation and display of a GUI. You won't make any new calls to StdDraw. If you want more info here is the [standard documentation](http://introcs.cs.princeton.edu/java/stdlib/javadoc/StdDraw.html). (Note StdDraw does not use pixel coordinates. Instead it takes x and y locations between 0 and 1 and plots them assuming the drawing area is a unit square.)
* [StdAudio.java](https://www.cs.utexas.edu/~scottm/cs312/javaCode/A11GuitarHero/StdAudio.java) - Used to play sounds. You should not have to change any of the calls to this class or call any other methods. If interested, here is [the documentation](http://introcs.cs.princeton.edu/java/stdlib/javadoc/StdAudio.html)
* [GuitarHeroLite.java](https://www.cs.utexas.edu/~scottm/cs312/javaCode/A11GuitarHero/GuitarHeroLite.java)- A class that creates a GUI and allows the user to "pluck" two guitar strings by pressing the 'a' or 'c' key on the keyboard. This class will not work until you complete the GuitarString and RingBuffer classes. Implement a GuitarHero.java class similar to GuitarHeroLite, but your class will support 37 strings, not just 2.
* [RingBufferTester.java](https://www.cs.utexas.edu/~scottm/cs312/javaCode/A11GuitarHero/RingBufferTester.java) - Simple tests for your RingBuffer class.

**Background:**

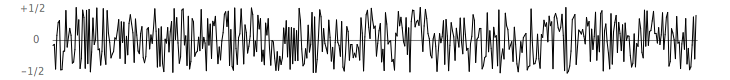
Write a program to simulate plucking a guitar string using the [*Karplus-Strong* algorithm](http://tinyurl.com/bvptl5l). This algorithm played a seminal role in the emergence of physically modeled sound synthesis (where a physical description of a musical instrument is used to synthesize sound electronically).

**Digital audio.** For more background on digital audio, review [Standard audio section of this web page](http://introcs.cs.princeton.edu/java/15inout/)and the [Superposition of sound waves](http://introcs.cs.princeton.edu/java/21function/) section of this web page.

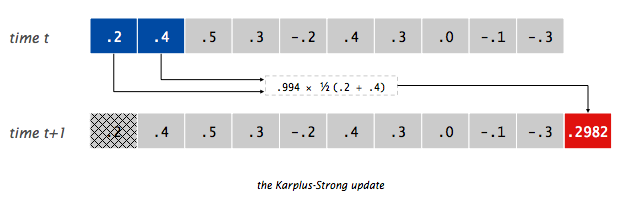
**Simulate the plucking of a guitar string.** When a guitar string is plucked, the string vibrates and creates sound. The length of the string determines its *fundamental frequency* of vibration. We model a guitar string by sampling its *displacement* (a real number between -1/2 and +1/2) at *N* equally spaced points (in time), where *N* equals the *sampling rate* (44,100) divided by the fundamental frequency (rounding the quotient up to the nearest integer).



* *Plucking the string.* The excitation of the string can contain energy at any frequency. We simulate the excitation with *white noise*: set each of the *N* displacements to a random real number between -1/2 and +1/2.



* *The resulting vibrations.* After the string is plucked, the string vibrates. The pluck causes a displacement which spreads wave-like over time. The Karplus-Strong algorithm simulates this vibration by maintaining a *ring buffer* of the *N* samples: the algorithm repeatedly deletes the first sample from the buffer and adds to the end of the buffer the average of the first two samples, scaled by an *energy decay factor* of 0.994.



**Why it works?** The two primary components that make the Karplus-Strong algorithm work are the ring buffer feedback mechanism and the averaging operation.

* *The ring buffer feedback mechanism.* The ring buffer models the medium (a string tied down at both ends) in which the energy travels back and forth. The length of the ring buffer determines the fundamental frequency of the resulting sound. Sonically, the feedback mechanism reinforces only the fundamental frequency and its harmonics (frequencies at integer multiples of the fundamental). The energy decay factor (.994 in this case) models the slight dissipation in energy as the wave makes a roundtrip through the string.

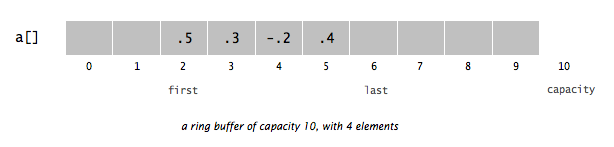
* *The averaging operation.* The averaging operation serves as a gentle *low-pass filter* (which removes higher frequencies while allowing lower frequencies to pass, hence the name). Because it is in the path of the feedback, this has the effect of gradually attenuating the higher harmonics while keeping the lower ones, which corresponds closely with how a plucked guitar string sounds.

From a mathematical physics viewpoint, the Karplus-Strong algorithm approximately solves the [1D wave equation](http://en.wikipedia.org/wiki/Wave_equation), which describes the transverse motion of the string as a function of time.

**Classes to Implement:**

**RingBuffer:**A RingBuffer (also known as a buffered queue) is a data structure that allows access to the item that has been in the data structure the longest amount of time. A queue is similar to a line.

Your first task is to create a class to model a ring buffer.



Write a class named RingBuffer that implements the following methods: (All methods are public!)

public class RingBuffer

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

RingBuffer(int capacity) // create an empty ring buffer, with given capacity

int size() // return number of items currently in the buffer

boolean isEmpty() // is the buffer empty (size equals zero)?

boolean isFull() // is the buffer full (size equals capacity)?

void enqueue(double x) // add item x to the end (if not full)

double dequeue() // delete and return item from the front (if not empty)

double peek() // return item from the front of the buffer

String toString() //form [front, next, …, next, last]

RingBuffer shall throw a NoSuchElementException if the client attempts to dequeue() from or peek() at an empty buffer and a IllegalStateException if  enqueue(double val) is called when the buffer is full.

Be clear on the difference between the RingBuffer's capacity and its size. The capacity is fixed and doesn't change. It's like spots we could store values. For example, the capacity of this egg carton is 12, but its size (number of eggs present is 0):

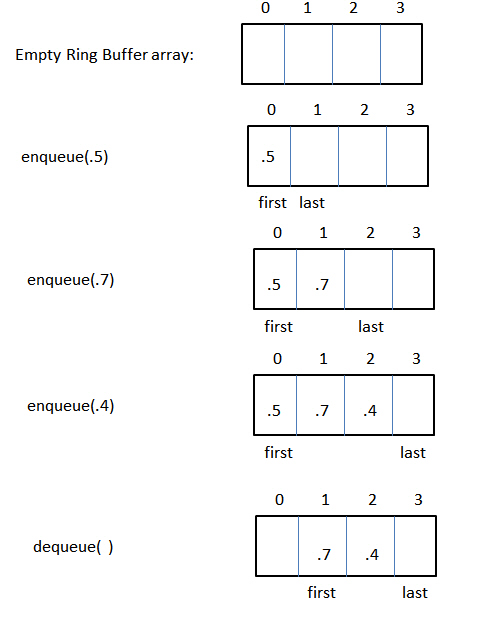
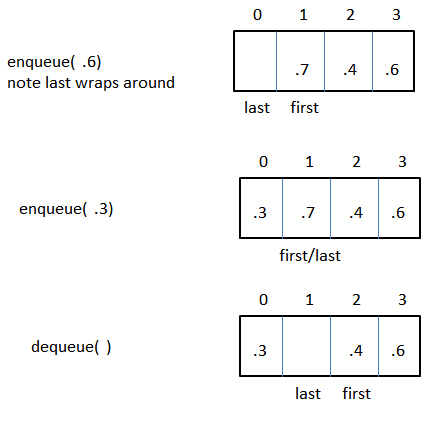


This egg carton also has a capacity of 12, but its size is 3 because it currently contains 3 eggs:



Since the ring buffer has a known maximum capacity, implement it using an array of doubles of that length. For efficiency, use *cyclic wrap-around*: Maintain one integer instance variable first that stores the index of the item that has been in the RingBuffer the longest. (The front of the line.)  Maintain a second integer instance variable last that stores the index one beyond the most recently inserted item. To insert an item, put it at index last and increment last. To remove an item, take it from index first and increment first. When either index equals capacity, make it wrap-around by changing the index to 0.

Consider this series of operations on a RingBuffer of capacity 4.

You can use the RingBufferTester class to see if your RingBuffer works. Do this before going on to the next class! Note, you may pass all the tests in RingBufferTester, but still have errors in your RingBuffer. (Recall failing tests proves the presence of a bug but passing all tests does NOT prove the absence of bugs unless you test every possible condition, which is normally impractical or even impossible.)

**GuitarString:**

Next, create a data type to model a vibrating guitar string. Write a class named GuitarString that implements the following API: (All methods are public!)

public class GuitarString

public ringBuffer; // public for grading purposes only!

// MUST BE CALLED ringBuffer for grading!

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

GuitarString(double frequency) // use a sampling rate of 44,100

GuitarString(double[] init) // size and values are given by the array

void pluck() // set the buffer to white noise

void tic() // advance the simulation one time step

double sample() // return the current sample

int time() // return number of tics

* *Constructors.* There are two ways to create a GuitarString object.   
  + The first constructor creates a RingBuffer of the desired capacity *N* (sampling rate 44,100 divided by *frequency*, rounded up to the nearest integer), and initializes it to represent a guitar string at rest by enqueuing *N* zeros.
  + The second constructor creates a RingBuffer of capacity equal to the size of the array and initializes the contents of the buffer to the values in the array. On this assignment, its main purpose is for debugging and grading.

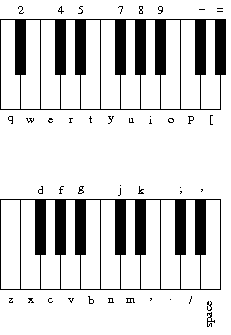
* *Pluck.* Replace the *N* items in the ring buffer with *N* random values between -0.5 and +0.5.
* *Tic.* Apply the Karplus-Strong update: delete the sample at the front of the ring buffer and add to the end of the ring buffer the average of the first two samples, multiplied by the energy decay factor.
* *Sample.* Return the value of the item at the front of the ring buffer.
* *Time.* Return the total number of times tic() was called.

**Interactive guitar player.** [GuitarHeroLite.java](ftp://ftp.cs.princeton.edu/pub/cs126/guitar/GuitarHeroLite.java) is a sample GuitarString client that plays the guitar in real-time, using the keyboard to input notes. When the user types the lowercase letter 'a' or 'c', the program plucks the corresponding string. Since the combined result of several sound waves is the superposition of the individual sound waves, we play the sum of all string samples.

Write a program GuitarHero that is similar to GuitarHeroLite but supports a total of 37 notes on the chromatic scale from 110Hz to 880Hz. In general, make the *i*th character of the string below play the *i* note.

String keyboard = "q2we4r5ty7u8i9op-[=zxdcfvgbnjmk,.;/' "; // String ends with a space

This keyboard arrangement imitates a piano keyboard: The "white keys" are on the qwerty and zxcv rows and the "black keys" on the 12345 and asdf rows of the keyboard.



Note, you DO NOT have to draw the keyboard on the StdDraw. Simply alter the displayed text to something more meaningful.

The *i*th character of the string corresponds to a frequency of 440 × 1.05956(*i*- 24), so that the character 'q' is approximately 110Hz, 'i' is close to 220Hz, 'v' is close to 440Hz, and ' ' is close to 880Hz.

You can test your GuitarHero program by typing the following into your guitar to get the beginning of Led Zeppelin's Stairway to Heaven. Multiple notes in a column are dyads and chords.

w q q

8 u 7 y o p p

i p z v b z p b n z p n d [ i d z p i p z p i u i i

What is this familiar melody?

nn//SS/ ..,,mmn //..,,m //..,,m nn//SS/ ..,,mmn (S = space)

**Style:**As always, use good style in your classes and methods. If you have more than 3 GuitarStrings would you even THINK about using 3 separate GuitarString variables? Use the String indexOf method to map from the keyboard String to your array of GuitarString variables.

**Extra credit.** There are two optional extra credit parts. You can do either one if you want or both. If working with a partner, you can do them together or alone — but decide *before* you start it.

**Extra credit 1.** Write a program GuitarHeroVisualizer.java (by modifying GuitarHero.java) that plots the sound wave in real-time, as the user is playing the keyboard guitar. The output should look something like this:

To get credit, your program must (1) draw something for every sample, (2) run smoothly in real-time with no effects on the audio, and (3) run at a minimum of 24 frames per second. Depending on your computer, you might be able to do 60 FPS or more, but this is not required. Part of what is necessary to accomplish this is the usage of animation mode as described in the [StdDraw documentation](http://introcs.cs.princeton.edu/java/stdlib/javadoc/StdDraw.html), since deferring the on-screen display until an entire frame is complete helps make the StdDraw calls fast enough to not distort the audio. You can add extra embellishments if you like so long as your program satisfies all three aformentioned requirements.

**Extra credit 2.** Write a program AutoGuitar.java that will *automatically* play a song with GuitarStrings. *The song should be between 10 seconds and 2 minutes in length.*

It will make calls to StdAudio.play() but it **must not** make any other calls to the Std\* libraries. In particular, make sure there are no calls to StdDraw, and that it works with no user input. Your main() must terminate; it cannot loop infinitely. To get the timing to work, either use the Stopwatch class, or use the fact that 1/44100 of a second elapses between calls to StdAudio.play(). This will make it have consistent timing on different machines.

Use arrays, loops, conditionals, and randomization as you see fit. If you are interested, you can add chords, repetition, or different instruments (see below). You may submit additional classes, but don't modify your original classes (such as GuitarString).

If you find it helpful, you may submit a .txt file to be read with the [In](http://introcs.cs.princeton.edu/java/stdlib/javadoc/In.html) class, and you may hard-code the filename in your program.

**Challenge for the bored.** Here are suggestions on other effects and instruments to synthesize. Some come from the [paper](http://www.jstor.org/stable/10.2307/3680062) of Karplus and Strong.

* Harp strings: Flipping the sign of the new value before enqueueing it in tic() will change the sound from guitar-like to harp-like. You may want to play with the decay factors to improve the realism, and adjust the buffer sizes by a factor of two since the natural resonance frequency is cut in half by the tic() change.
* Drums: Flipping the sign of a new value with probability 0.5 before enqueueing it in tic() will produce a drum sound. A decay factor of 1.0 (no decay) will yield a better sound, and you will need to adjust the set of frequencies used.
* Guitars play each note on one of 6 physical strings. To simulate this you can divide your GuitarString instances into 6 groups, and when a string is plucked, zero out all other strings in that group.
* Pianos come with a damper pedal which can be used to make the strings stationary. You can implement this by, on iterations where a certain key (such as Shift) is [held down](http://introcs.cs.princeton.edu/java/stdlib/javadoc/StdDraw.html#isKeyPressed(int)), changing the decay factor.
* While we have used [equal temperament,](http://en.wikipedia.org/wiki/Equal_temperament) the ear finds it more pleasing when musical intervals follow the small fractions in the [just intonation](http://en.wikipedia.org/wiki/Just_intonation) system. For example, when a musician uses a brass instrument to play a [perfect fifth](http://en.wikipedia.org/wiki/Perfect_fifth) harmonically, the ratio of frequencies is 3/2 = 1.5 rather than 27/12 ∼ 1.498. Write a program where each successive pair of notes has just intonation.