CSE 143: Computer Programming II — Summer 2022

A2: Guitar Hero — *due July 7, 2022 11:59pm*

many thanks to Kevin Wayne for this nifty assignment

This assignment will assess your mastery of the following objectives:

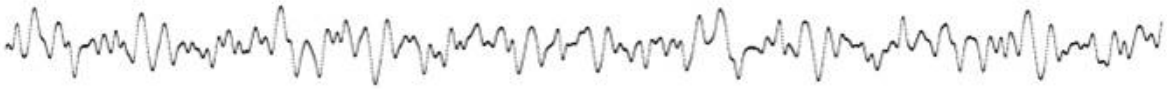
- Implement a well-designed Java class to meet a given specification.
- Use a queue (via the `Queue<E>` interface) to implement a provided algorithm.
- Write a class that implements an existing interface.
- Follow prescribed conventions for code quality, documentation, and readability.

There are many support files for this assignment that can be found on the course website. We will be using two utility classes known as StdAudio and StdDraw that are used in the Princeton intro CS course. You don't have to understand the details of these utility classes, but you can read about them here if you're interested.

Background: Guitars and Sound

This section describes how guitar strings make sound and how we will represent that in our code to simulate sound. This section will be confusing and you do not need to fully understand the technical details of why this works in order to do the assignment. Later sections have details of how to implement this.

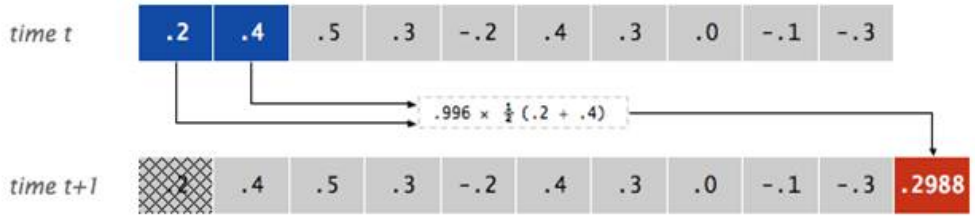
When a guitar string is plucked, the string vibrates and creates sound. The length of the string determines its fundamental frequency of vibration. We model a guitar string by sampling its displacement (a real number between -1/2 and +1/2) at N equally spaced points in time, where N equals the sampling rate (44,100) divided by the fundamental frequency of the string (rounded to the nearest integer). We store these displacement values in a structure that we will refer to as a **ring buffer**.



Plucking a string moves it and gives it energy. The excitation of the string can contain energy at any frequency. We simulate the excitation by filling the ring buffer with white noise. In other words, we set each of the N sample displacements to a random real number between -1/2 and +1/2.

Simulating Sound

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: for each step the algorithm deletes the first sample from the ring buffer and adds to the end of the ring buffer the average of the first two samples, scaled by an energy decay factor of 0.996. More details on why this simulates sound are at the end of the spec.



the Karplus-Strong update

Part 1: `GuitarString` class

You must not use any `Queue<E>` methods not listed here.

In the first part of the assignment, you will implement a class called `GuitarString` that models a vibrating guitar string of a given frequency. The `GuitarString` object will need to keep track of a ring buffer. You are to implement the ring buffer as a queue using the `Queue<E>` interface and the `LinkedList<E>` implementation. For this assignment, you are limited to the `Queue<E>` methods mentioned in class (`add`, `remove`, `isEmpty`, `size`, and `peek`). You are not allowed to use other data structures or other `Queue<E>` methods to solve this problem.

Your `GuitarString` class should include the following constructors:

`public` **`GuitarString`**`(double frequency)`
Constructs a `GuitarString` of the given frequency. It creates a ring buffer of the desired capacity N (sampling rate divided by frequency, rounded to the nearest integer), and initializes it to represent a guitar string at rest by enqueueing N zeros. The sampling rate is specified by the constant `StdAudio.SAMPLE_RATE`. If the frequency is less than or equal to 0 or if the resulting size of the ring buffer would be less than 2, your method should throw an `IllegalArgumentException`

`public` **`GuitarString`**`(double[] init)`
Constructs a `GuitarString` and initializes the contents of the ring buffer to the values in the array. If the array has fewer than two elements, your constructor should throw an `IllegalArgumentException`. This constructor is used only for testing purposes.

Your `GuitarString` class should also implement the following public methods:

`public void` **`pluck`**`()`
This method should replace the N elements in the ring buffer with N random values between -0.5 inclusive and +0.5 exclusive (i.e. $-0.5 \leq \text{value} < 0.5$).

`public void` **`tic`**`()`
This method should apply the Karplus-Strong update once (performing one step). It should 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 (0.996). Your class should include a public constant for the energy decay factor.

`public double` **`sample`**`()`
This method should return the current sample (the value at the front of the ring buffer).

Implementation Guidelines

- You will need to use the `Math.round` method and cast the result to an `int` to find the size of the buffer in the constructor that takes a single `double` parameter. You can use the following expression:

```
(int)(Math.round(StdAudio.SAMPLE_RATE / frequency))
```

- It is difficult in commenting the `GuitarString` class to know what constitutes an implementation detail and what is okay to discuss in client comments. Assume that a client of the `GuitarString` class is familiar with the concept of a ring buffer. The fact that we are implementing it as a queue is an implementation detail. So don't mention how you implement the ring buffer. But you can

discuss the ring buffer itself and the changes that your methods make to the state of the ring buffer (e.g., moving values from the front to the back of the ring buffer). You may also assume that the client is familiar with the Karplus-Strong algorithm.

- At this point, you can also run `GuitarHero` using `GuitarLite` and you should hear sound on your computer! (You will not be able to play sound in Ed, but you can download the files and run them in jGRASP.) The guitars are explained in the next section.

Interlude: `Guitar` and `GuitarHero`

This part of the assignment does not involve writing any code! This part describes the important supporting files and how to run the program.

In the next part of the assignment, you are going to build on the `GuitarString` class to write a class that keeps track of a musical instrument with multiple strings. There could be many possible guitar objects with different kinds of strings. As a result, we introduce an interface known as `Guitar` that each guitar object implements.

`Guitar interface`

```
public interface Guitar {
    public void playNote(int pitch);
    public boolean hasString(char key);
    public void pluck(char key);
    public double sample();
    public void tic();
    public int time();
}
```

The interface allows a client to specify what to play in one of two ways. A client can specify exactly which note to play by calling the `playNote` method passing it a pitch. Pitch is specified as an integer where the value 0 represents concert-A and all other notes are specified relative to concert-A using what is known as a chromatic scale. Not every value of pitch can be played by any given guitar. If it can't be played, it is ignored.

Additionally, a client can also specify a character that indicates which note to play by calling the `pluck` method. Different guitar objects will have different mappings from characters to notes. The interface includes a method called `hasString` that is paired with pluck that lets a client verify that a particular character has a corresponding string for this guitar. The `pluck` method has a precondition that the key is legal for this guitar.

The `Guitar` interface also has methods for getting the current sound sample (the sum of all samples from the strings of the guitar), to advance the time forward one "tic", and determining the current time (the number of times `tic` has been called).

We provide a sample class called `GuitarLite` that implements the `Guitar` interface. Once you have verified that your `GuitarString` class passes the testing program, you can play the `GuitarLite` instrument. It has only two strings: "a" and "c".

To test your guitar, we provide a separate client class called `GuitarHero` that has a `main` method (the initial version constructs a `GuitarLite` object). `GuitarLite` does not have a main method.

Part 2: The `Guitar37` class

In this second part of the assignment, your task is to write a different implementation of the `Guitar` interface known as `Guitar37`. It will model a guitar with 37 different strings. Unlike `GuitarLite` which has a separate field for each of its strings, you will want to use a data structure, specifically an array, to keep track of the strings in `Guitar37`.

Keys

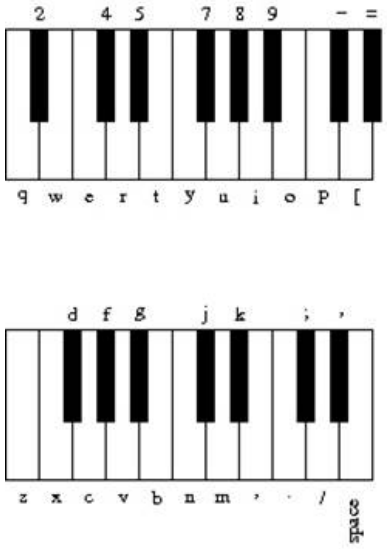
The `Guitar37` class has a total of 37 notes on the chromatic scale from 110Hz to 880Hz. We will use the following string to map keys typed by the user to positions in your array of strings. The i-th character of this string should correspond to the i-th character of your array:

```
"q2we4r5ty7u8i9op-[=zxdcfvgbnjmk,.;/' "
```

This use of keyboard characters imitates a piano keyboard, making playing songs a little easier for people used to 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, as in the drawing below.

You are being provided a skeleton version of the `Guitar37` class that includes this string defined as a constant called `KEYBOARD`. The i-th character of the string corresponds to a frequency of $440 \times 2^{(i-24)/12}$, so that the character "q" is 110Hz, "i" is 220Hz, "v" is 440Hz, and " " (space) is 880Hz.

As noted above, a pitch of 0 is supposed to correspond to concert-A, which will be at index 24 for the `Guitar37` object (corresponding to the character "v"). Thus, you can convert from a pitch value to an index in your string by adding 24 to the pitch value. The table below shows some examples of this conversion.



Key	Pitch
"q"	-24
"2"	-23
"w"	-22
"e"	-21
...	...
"v"	0
...	...
"/"	10
","	11
" "	12

Implementation Guidelines

- In working on this second part of the assignment, you are generalizing the code that you will find in `GuitarLite`. Because that instrument has just two strings, it uses two separate fields. Your instrument has 37 strings, so it uses an array of strings. Each of the operations defined in the interface needs to be generalized from using two specific strings to using an array of strings. For example, the sample method returns the sum of the current samples. `GuitarLite` does this by adding together two numbers. Your version will have to use a loop to find the sum of all 37 samples.
- The `GuitarLite` class is not well documented, it does not handle illegal keys, and it does not correctly implement the `time` method. Your `Guitar37` class should include complete comments.
- The `pluck` method should throw an `IllegalArgumentException` if the key is not one of the 37 keys it is designed to play (as noted above, this differs from the `playNote` method that simply ignores notes it cant play).
- You will be given a testing program for `Guitar37` as well called `Test37`. **This testing code should be stored in a separate directory from your solution because it includes a custom version of the `GuitarString` class and you don't want to accidentally overwrite your version of the class.** You should copy your `Guitar37` class to this folder, run it, and then compare against the sample output produced using the output comparison tool.
- Once you are done, you can change `GuitarHero` to use `Guitar37` instead of `GuitarLite` so you can play the full instrument on your computer! (Ed cannot produce sound, so you'll need to run your code in jGRASP to try this.)

Recall that Strings have an `indexOf` method that you might find helpful!

Code Quality Guidelines

In addition to producing the behavior described above, your code should be well-written and meet all expectations described in the General Style Deductions, Style Guide, and Commenting Guide. For this assessment, pay particular attention to the following elements:

Generic Structures and Interfaces

You should always use generic structures. If you make a mistake in specifying type parameters, the Java compiler may warn you that you have "unchecked or unsafe operations" in your program. You should also declare fields and variables using the appropriate interfaces when possible. When using `Queue`'s in 143, you should only use the methods described in class.

Data Fields

Properly encapsulate your objects by making data your fields `private`. Avoid unnecessary fields; use fields to store important data of your objects but not to store temporary values only used in one place. Fields should always be initialized inside a constructor or method, never at declaration.

Exceptions

The specified exceptions must be thrown correctly in the specified cases. Exceptions should be thrown as soon as possible, and no unnecessary work should be done when an exception is thrown. Exceptions should be documented in comments, including the type of exception thrown and under what conditions.

Commenting

Each method should have a header comment including all necessary information as described in the Commenting Guide. Comments should be written in your own words (i.e. not copied and pasted from this spec) and should not include implemenation details.

Testing and Submitting

While developing your code, you should work locally on your computer in jGRASP. To test your code, you should run the provided test files and copy the printed logs into the Output Comparison Tool to check against the expected output.

If you believe your behavior is correct, you can submit your work by uploading your files and clicking the "Mark" button in the Ed assessment. You will see the results of some automated tests along with tentative grades. **These grades are not final until you have received feedback from your TA**.

You should treat Ed as a final check for your correctness, not as a debugging tool. If you aren't passing the tests, you should go back to jGRASP to debug your work.

You may submit your work as often as you like until the deadline; we will always grade your most recent submission. If you submit a version that you later decide you do not want to have graded, **you must warn your TA not to grade that version and to wait for a later submission from you.**

Getting Help

If you find you are struggling with this assessment, make use of all the course resources that are available to you, such as:

- Reviewing relevant examples from class
- Reading the textbook
- Visiting the IPL or Taylor's office hours

- Posting a question on the Ed discussion board

Collaboration Policy

Remember that, while you are encouraged to use all resources at your disposal, including your classmates, **all work you submit must be entirely your own**. In particular, you should **NEVER** look at a solution to this assessment from another source (a classmate, a former student, an online repository, etc.). Please review the full policy in the syllabus for more details and ask the course staff if you are unclear on whether or not a resource is OK to use.

(Optional) Why This Assignment 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 (.996 in this case) models the slight dissipation in energy as the wave makes a round trip 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 actually plucked strings sound.