

HW2: Guitar Hero (due Thursday, October 10, 2019 11:30pm)

many thanks to Kevin Wayne for this nifty assignment

This programming assignment will give you practice with queues, interfaces, objects, and arrays of objects. Turn in the following files using the link on the course website:

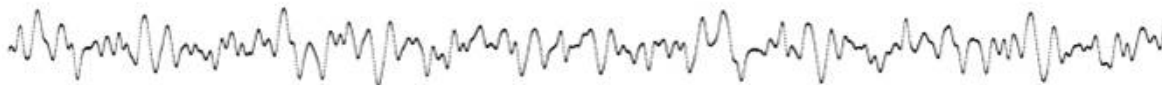
- `GuitarString.java` – A class that represents a single string of a guitar.
- `Guitar37.java` – A class that represents a Guitar with 37 strings.

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.

Guitars and Sound

This page describes how guitar string's 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. The next pages have details of how to implement this.

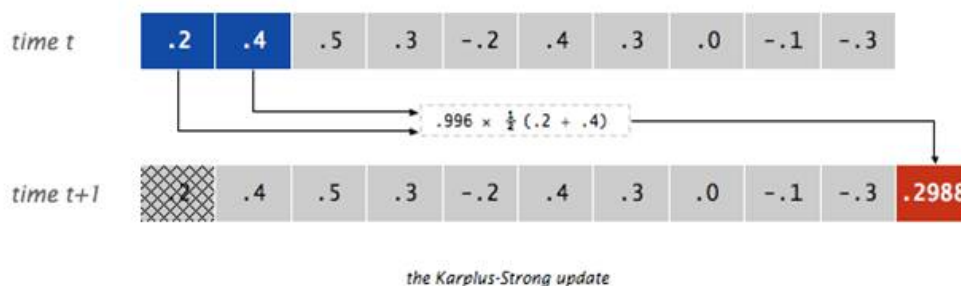
When a guitar string is plucked, the string vibrates and creates sound. The length of the string determines its fundamental frequency of vibration. We model a guitar string by sampling its displacement (a real number between $-1/2$ and $+1/2$) at N equally spaced points in time, where N equals the sampling 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.



The very end of the spec has an optional section that explains why this simulates sound.

Part 1: GuitarString class

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. You are limited to the `Queue<E>` methods in Table 14.2 on page 888 of the textbook (`add`, `remove`, `isEmpty`, `size`, and `peek`). You are not allowed to use other data structures or other `Queue<E>` methods to solve this problem.

GuitarString should have 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.

GuitarString should also implement the following 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 Details

- You will be provided with a testing program that you can use to verify that your class has the basic functionality required. The testing program will not check to make sure that you are using a queue, that you are checking for appropriate exceptions to throw, or that you're using the queue efficiently.
- Normally we would encourage you to write a single constructor and to use the `this(...)` notation to have one constructor call another. That won't be possible for the `GuitarString` class because the two constructors are completely different.
- Because you are using Java's queue structure to implement the `GuitarString`, you will need to include this import declaration at the beginning of the class: `import java.util.*;`

- 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! The guitars are explained in the next section.

Part 2: 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.

```

1 public interface Guitar {
2     public void playNote(int pitch);
3     public boolean hasString(char key);
4     public void pluck(char key);
5     public double sample();
6     public void tic();
7     public int time();
8 }

```

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 a method for determining the current time (the number of times `tic` has been called).

You are being provided with 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". Keep in mind that `GuitarLite` does not have a main method. There is a separate class called `GuitarHero` that has main (the initial version constructs a `GuitarLite` object).

Part 3: Guitar37 class

In this third part of the assignment, your task is to make a different implementation of the Guitar interface known as Guitar37. It will model a guitar with 37 different strings. Because it has so many strings, we will want to keep track of them in a data structure. Your Guitar37 objects should each keep track of an array of 37 GuitarString objects.

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-[=zxdcfvgnbjmk,.;/' "

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 Details

- In working on this third 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.



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

- 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 can't 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 should change `GuitarHero` to use `Guitar37` instead of `GuitarLite` so you can play the full instrument on your computer!

Style Guidelines and Grading

In terms of external correctness, your class must provide all of the functionality described above. In terms of style, we will be grading on your use of comments, good variable names, consistent indentation and good coding style to implement these operations. Additionally, your solution must satisfy all constraints outlined in this specification.

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.

Java Style Guidelines

Appropriately use control structures like loops and `if/else` statements. Avoid redundancy using techniques such as methods, loops, and factoring common code out of `if/else` statements. Properly use indentation, good variable names, and types. Do not have any lines of code longer than 100 characters. Please refer to the [Style Guide](#) and the [General Style Deductions](#).

(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.