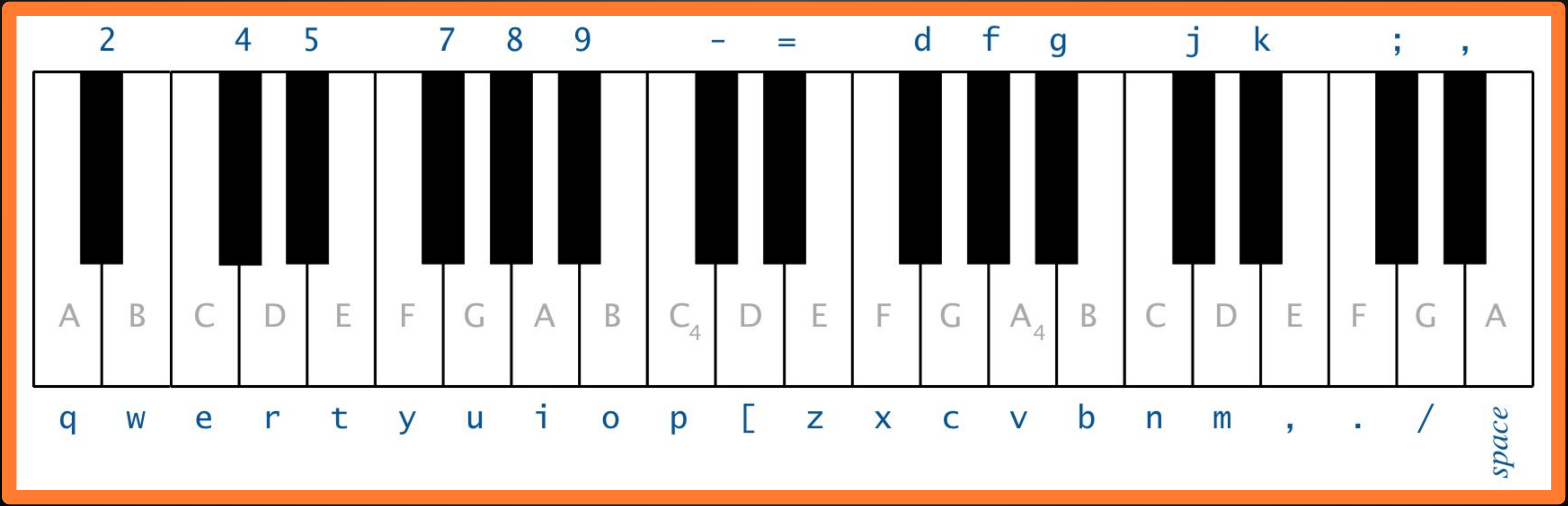
Guitar Hero

Tips & Tricks

Simulate the plucking of a guitar string using the Karplus–Strong algorithm, transforming your computer into a musical instrument



Administrative Info

- Partners allowed! Choose a partner whose skill level is close to your own
- See COS 126 website for guidelines

Overview

- This week, we're learning about performance analysis and getting a preview of **data structures**
- **GOALS:**
  - **Physically-modeled sound: compute sound waveform using a mathematical model of a musical instrument**
  - **Object-oriented programming: more practice with objects**
  - **Performance: efficient data structure that is crucial for this application**

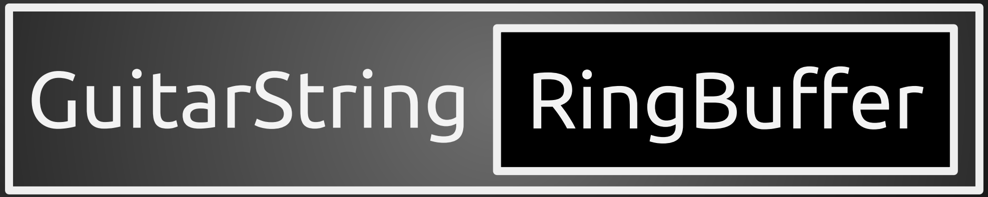
Overview

- This week, we're learning about performance analysis and getting a preview of **data structures**
- **RingBuffer** is your first classic data structure, a queue

RingBuffer

Overview

- This week, we're learning about performance analysis and getting a preview of data structures
- RingBuffer is your first classic data structure, a queue
- Each GuitarString uses 1 RingBuffer object

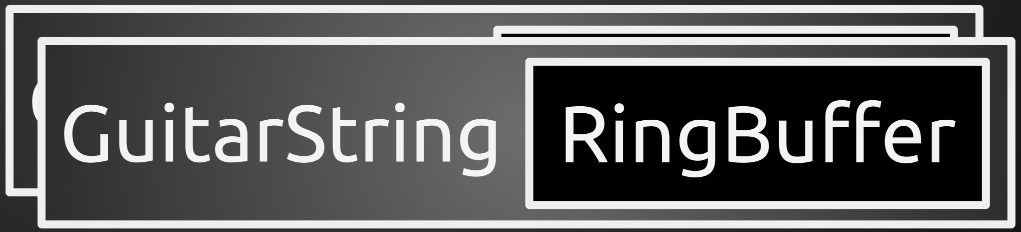


GuitarString

RingBuffer

Overview

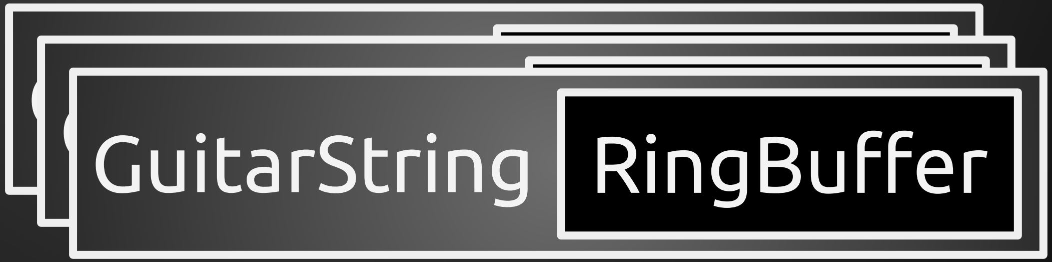
- This week, we're learning about performance analysis and getting a preview of **data structures**.
- **RingBuffer** is your first classic data structure, a queue.
- Each **GuitarString** uses 1 RingBuffer object
- **GuitarHero** uses 37 GuitarString objects



GuitarString RingBuffer

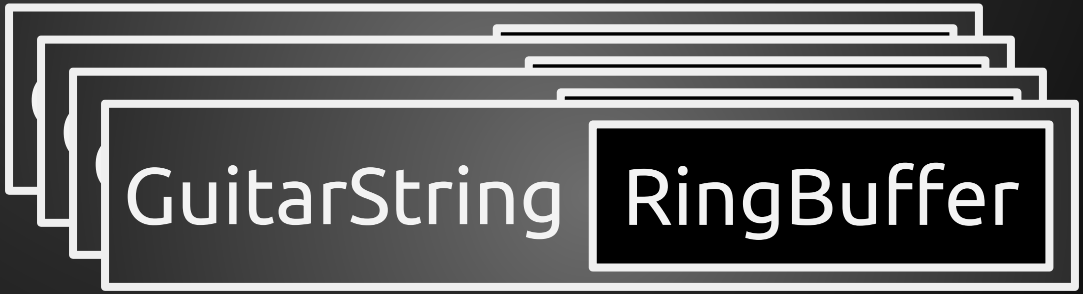
Overview

- This week, we're learning about performance analysis and getting a preview of **data structures**
- **RingBuffer** is your first classic data structure, a queue
- Each **GuitarString** uses 1 RingBuffer object
- **GuitarHero** uses 37 GuitarStrings objects



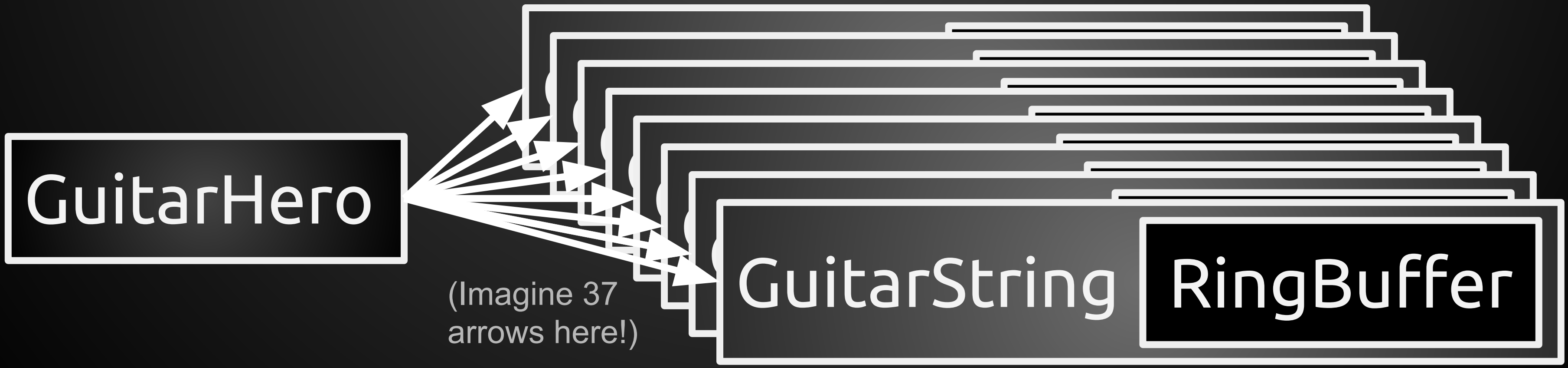
Overview

- This week, we're learning about performance analysis and getting a preview of **data structures**.
- **RingBuffer** is your first classic data structure, a queue
- Each **GuitarString** uses 1 RingBuffer object
- **GuitarHero** uses 37 GuitarStrings objects



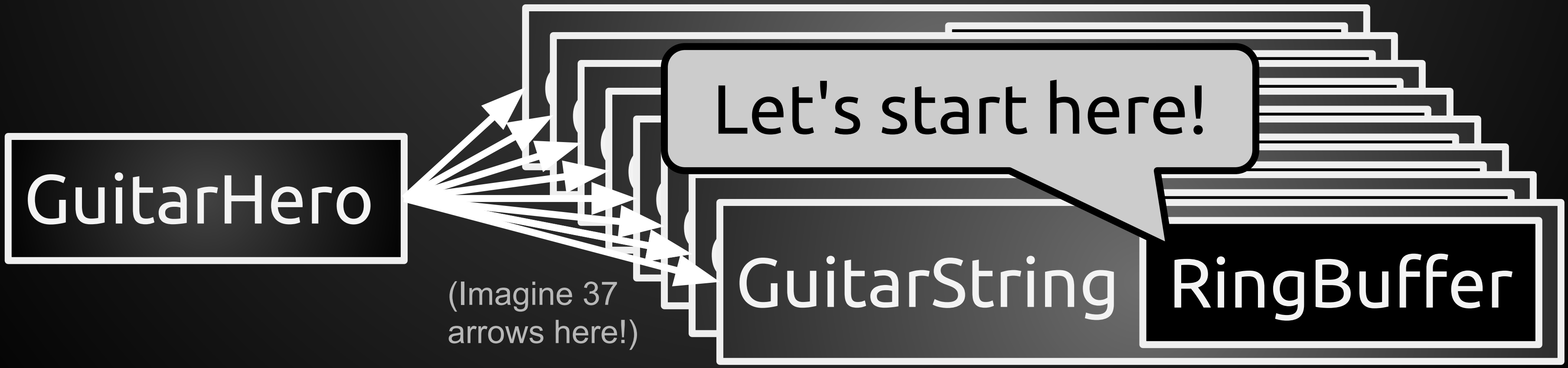
Overview

- This week, we're learning about performance analysis and getting a preview of **data structures**.
- **RingBuffer** is your first classic data structure, a queue.
- Each **GuitarString** uses 1 RingBuffer object
- **GuitarHero** uses 37 GuitarString objects

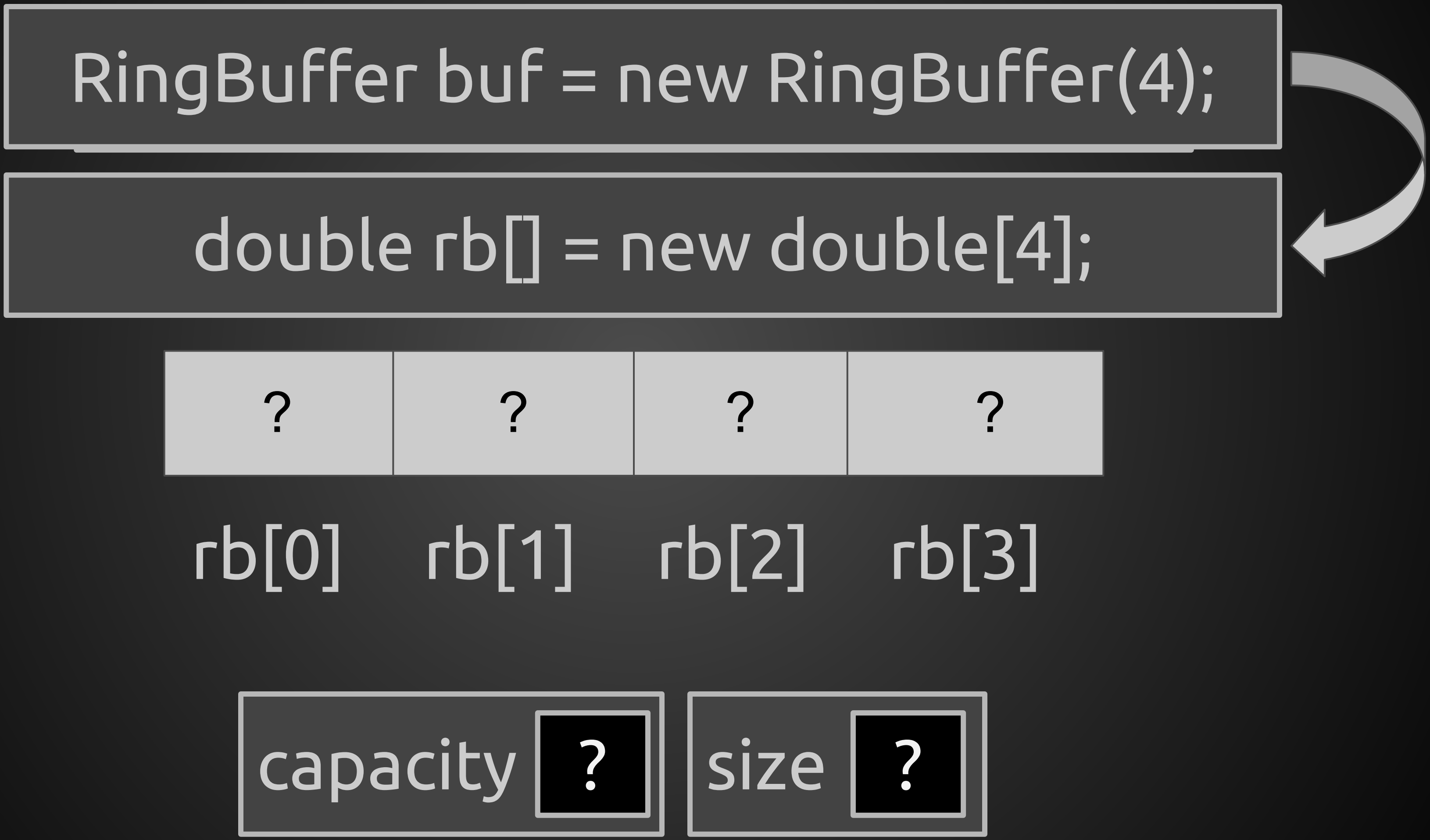


Overview

- This week, we're learning about performance analysis and getting a preview of **data structures**.
- **RingBuffer** is your first classic data structure, a queue.
- Each **GuitarString** uses 1 RingBuffer objects
- **GuitarHero** uses 37 GuitarStrings objects



RingBuffer



```
RingBuffer buf = new RingBuffer(4);
```

```
double rb[] = new double[4];
```

rb[0]	rb[1]	rb[2]	rb[3]
?	?	?	?

capacity ?

size ?

RingBuffer

```
RingBuffer buf = new RingBuffer(4);
```

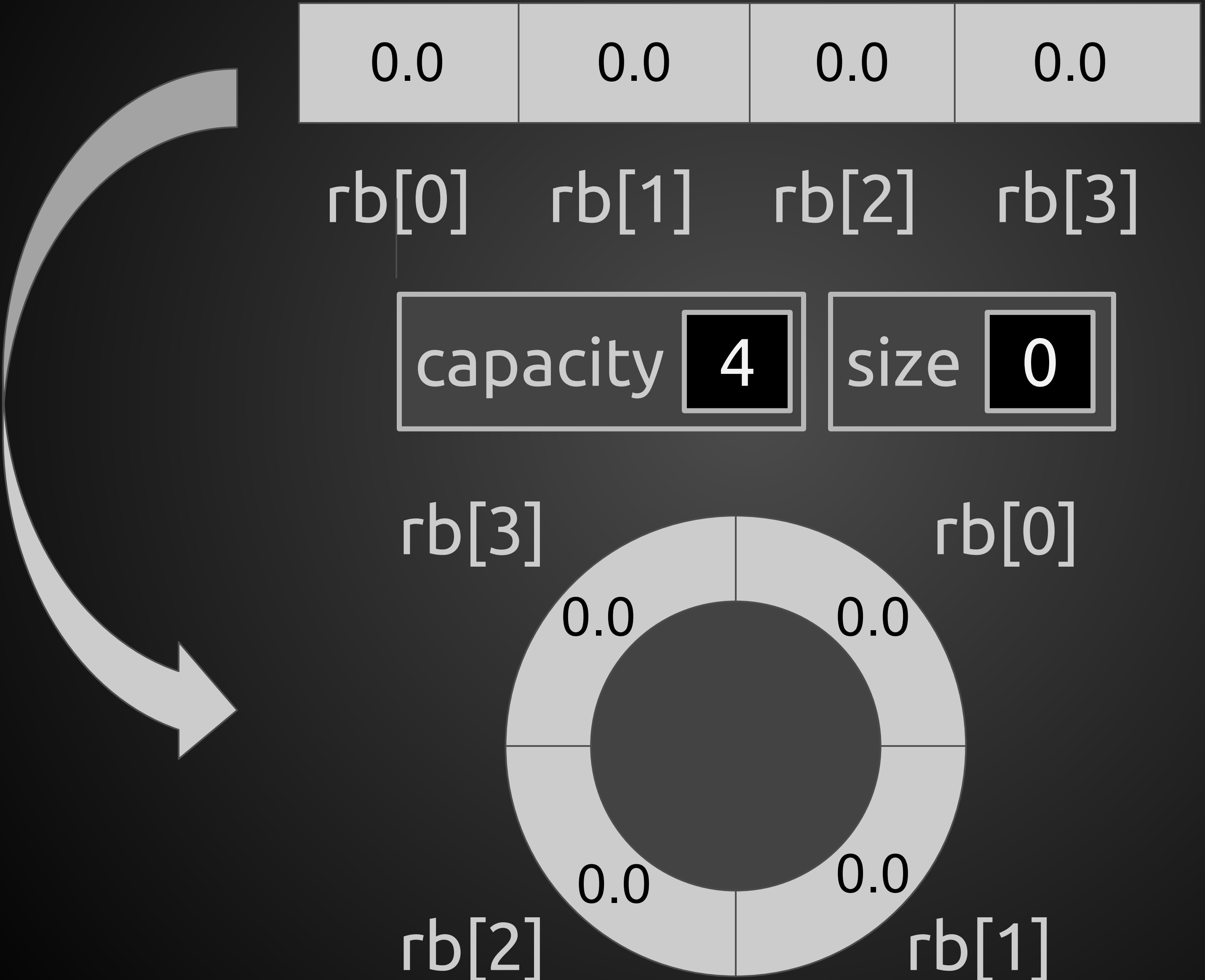
```
double rb[] = new double[4];
```

0.0	0.0	0.0	0.0
rb[0]	rb[1]	rb[2]	rb[3]

capacity 4

size 0

RingBuffer

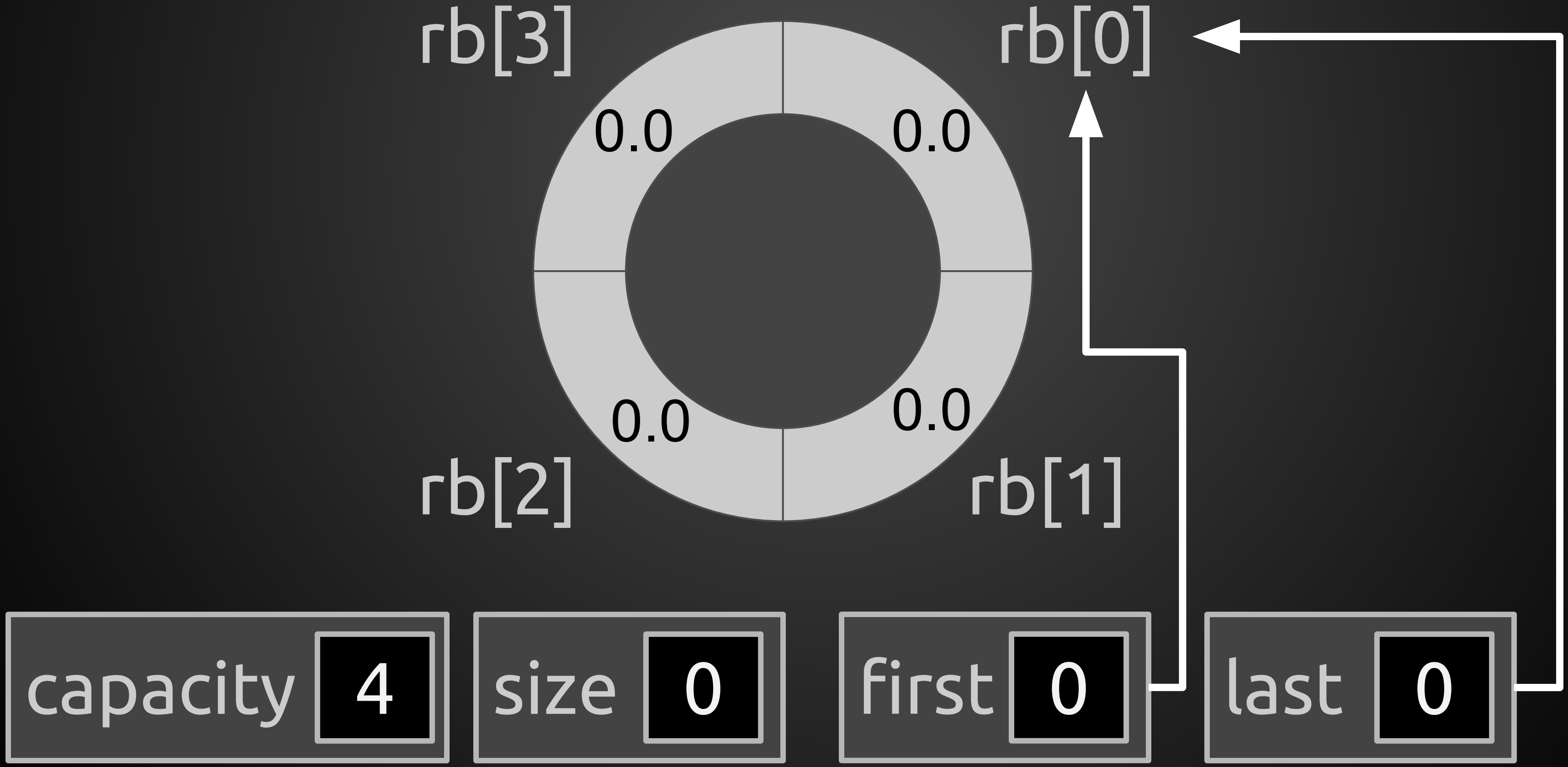


0.0	0.0	0.0	0.0
rb[0]	rb[1]	rb[2]	rb[3]

capacity 4

size 0

RingBuffer



rb[3]

rb[0]

0.0

0.0

0.0

0.0

rb[2]

rb[1]

capacity 4

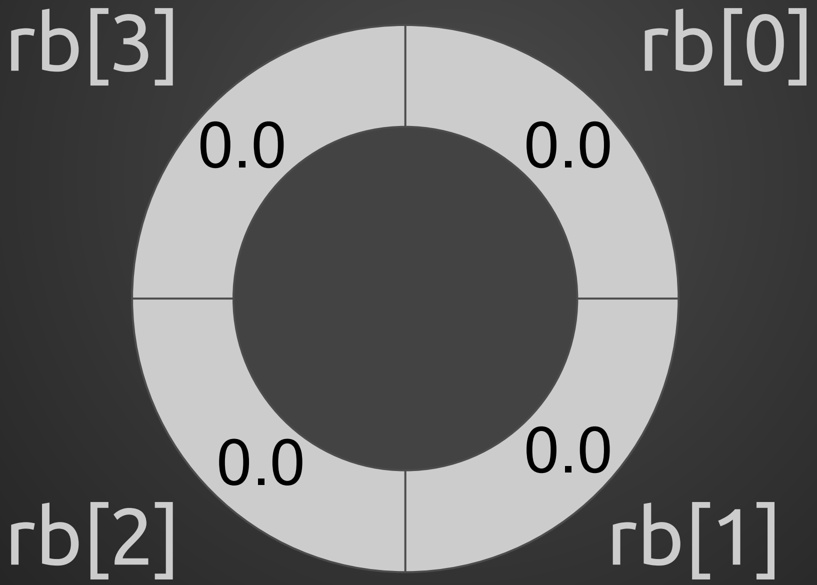
size 0

first 0

last 0

RingBuffer

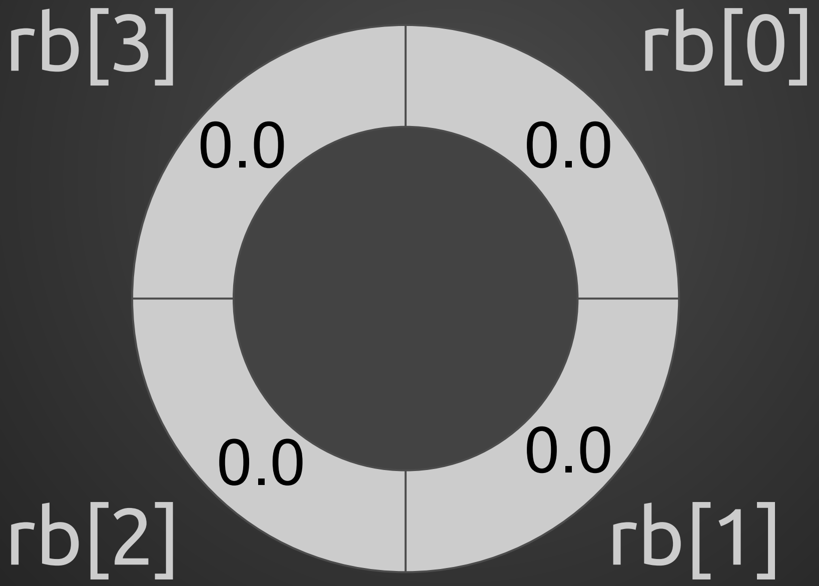
```
buf.enqueue(2.1);
```



capacity 4 | size 0 | first 0 | last 0

RingBuffer

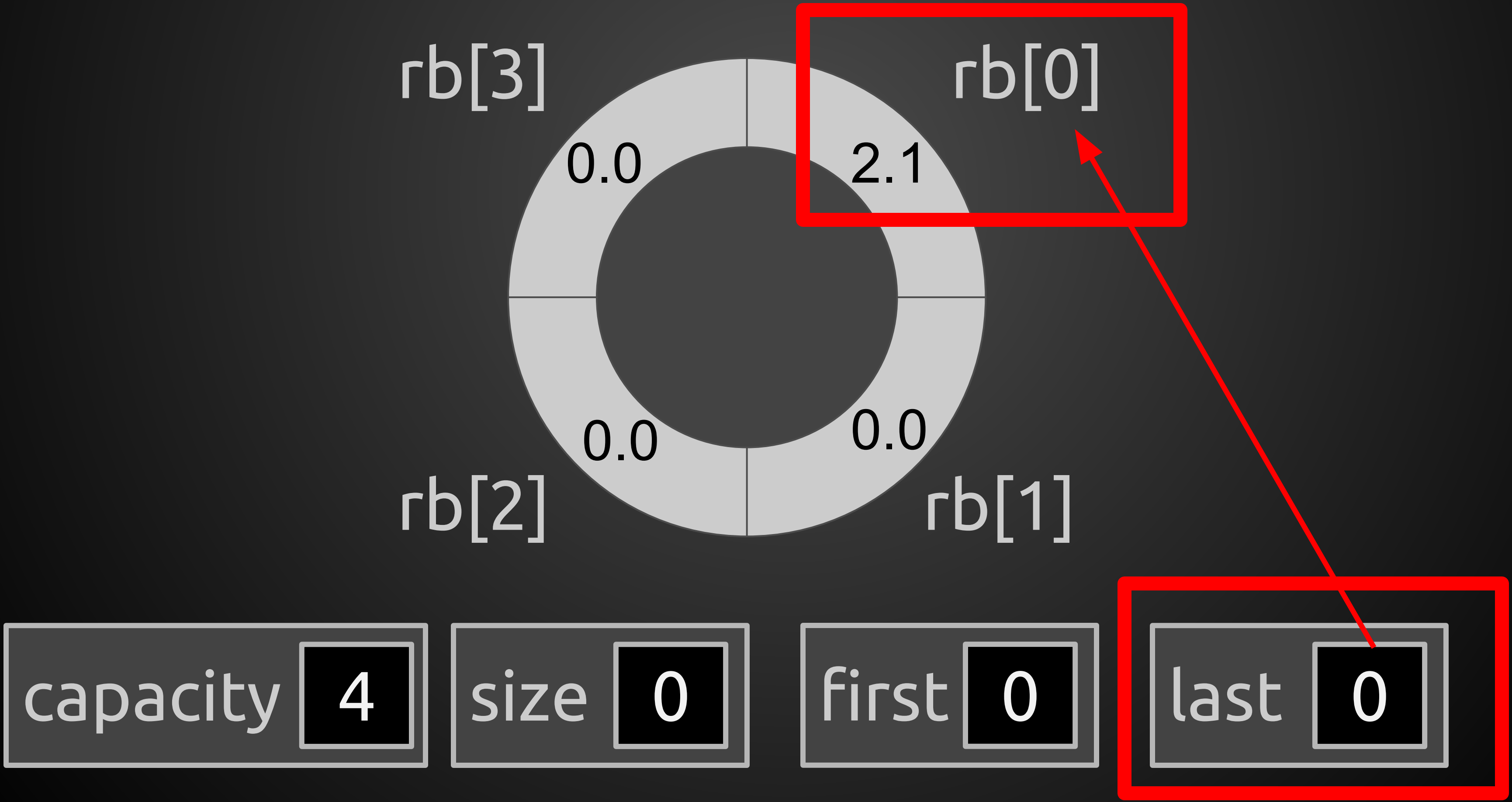
```
buf.enqueue(2.1);
```



capacity	4	size	0	first	0	last	0

RingBuffer

```
buf.enqueue(2.1);
```



capacity 4

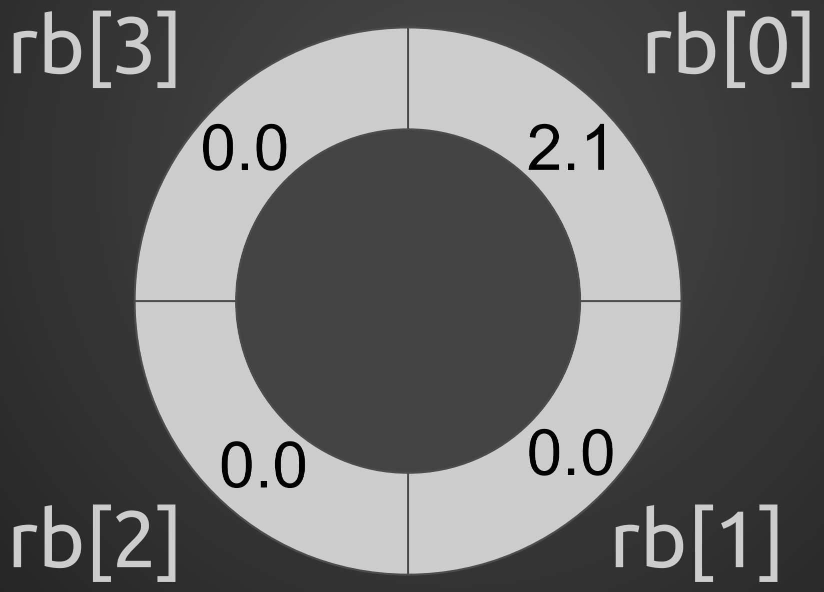
size 0

first 0

last 0

RingBuffer

```
buf.enqueue(2.1);
```



capacity 4

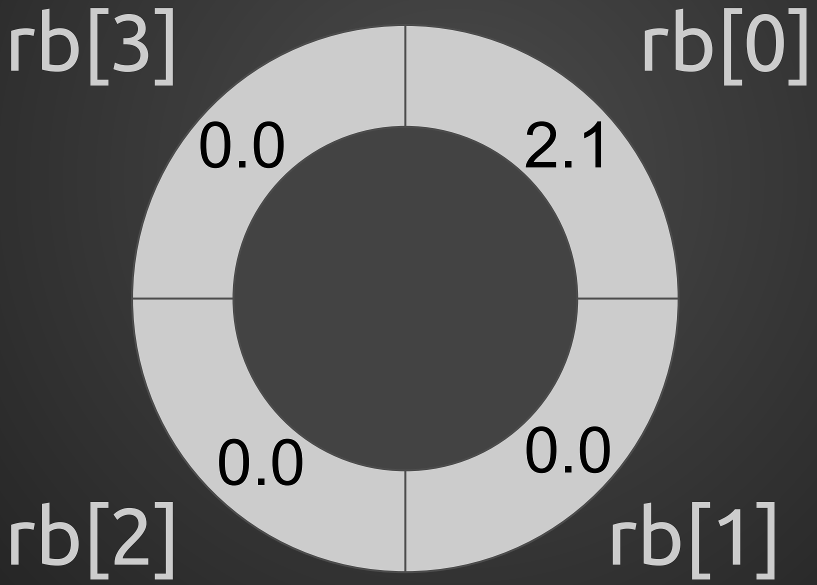
size 1

first 0

last 1

RingBuffer

```
buf.enqueue(1.7);
```



capacity 4

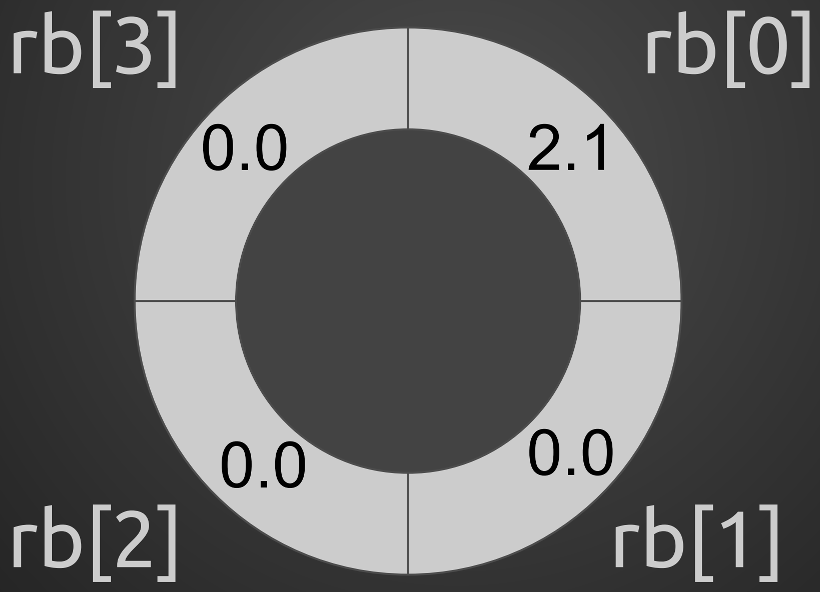
size 1

first 0

last 1

RingBuffer

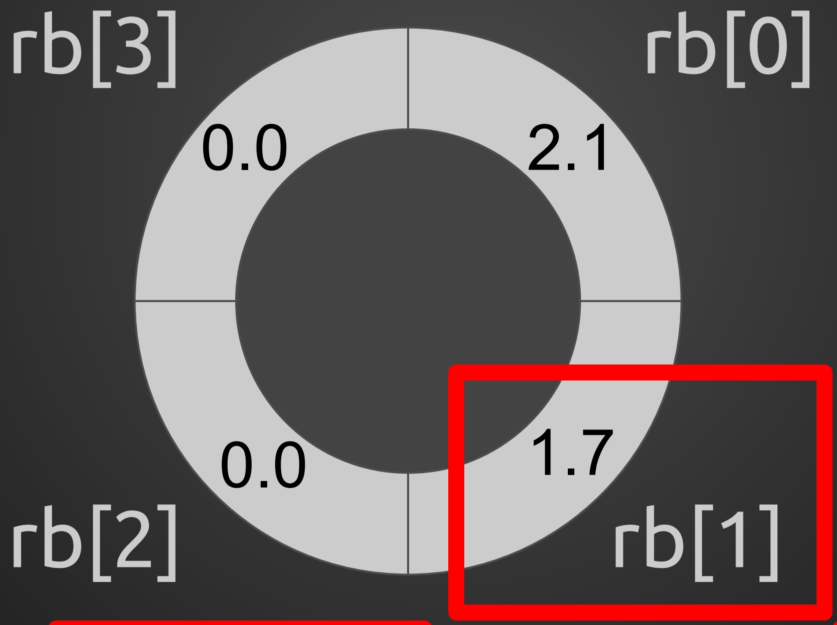
```
buf.enqueue(1.7);
```



capacity 4 | size 1 | first 0 | last 1

RingBuffer

```
buf.enqueue(1.7);
```



capacity 4

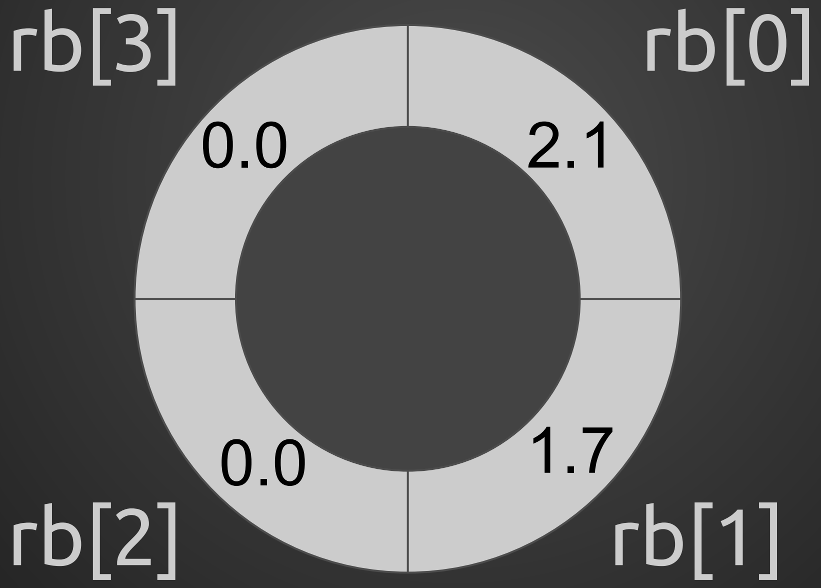
size 2

first 0

last 2

RingBuffer

```
double val = buf.dequeue();
```



capacity 4

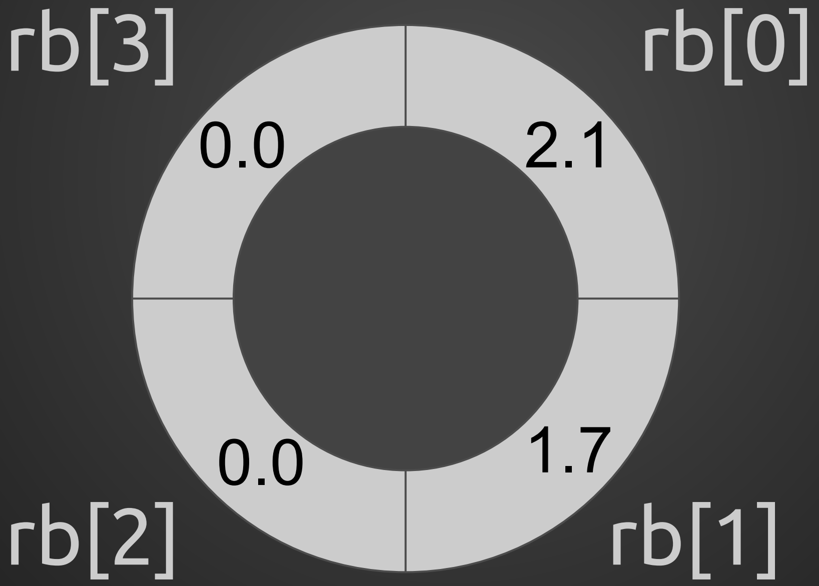
size 2

first 0

last 2

RingBuffer

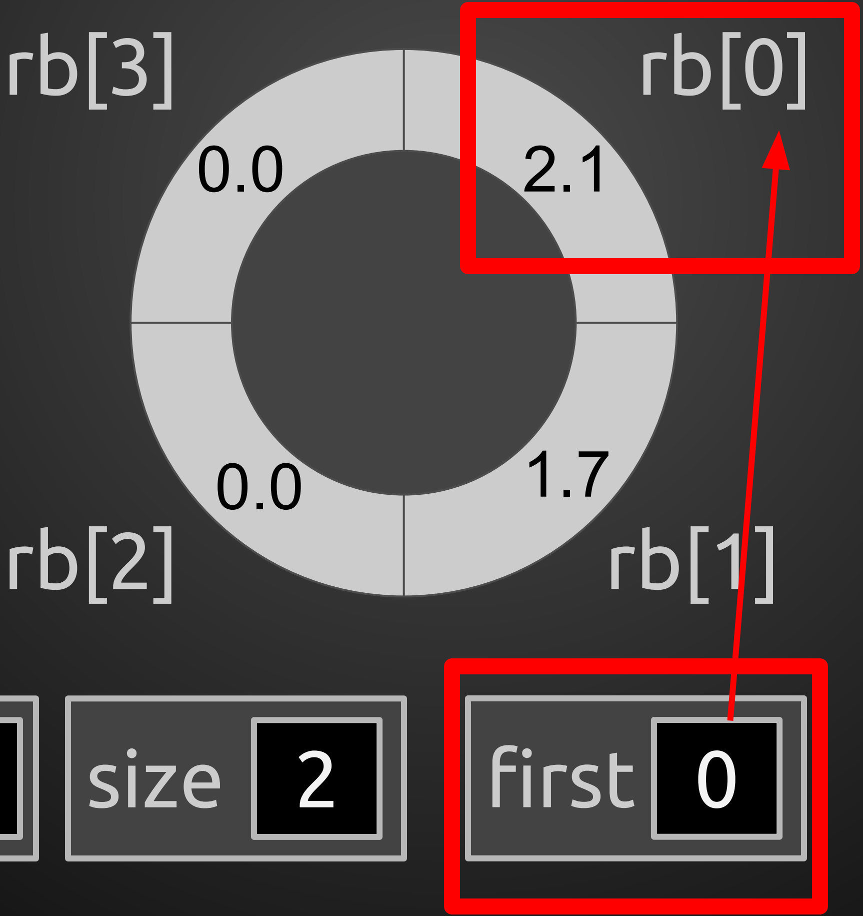
```
double val = buf.dequeue();
```



capacity 4 | size 2 | first 0 | last 2

RingBuffer

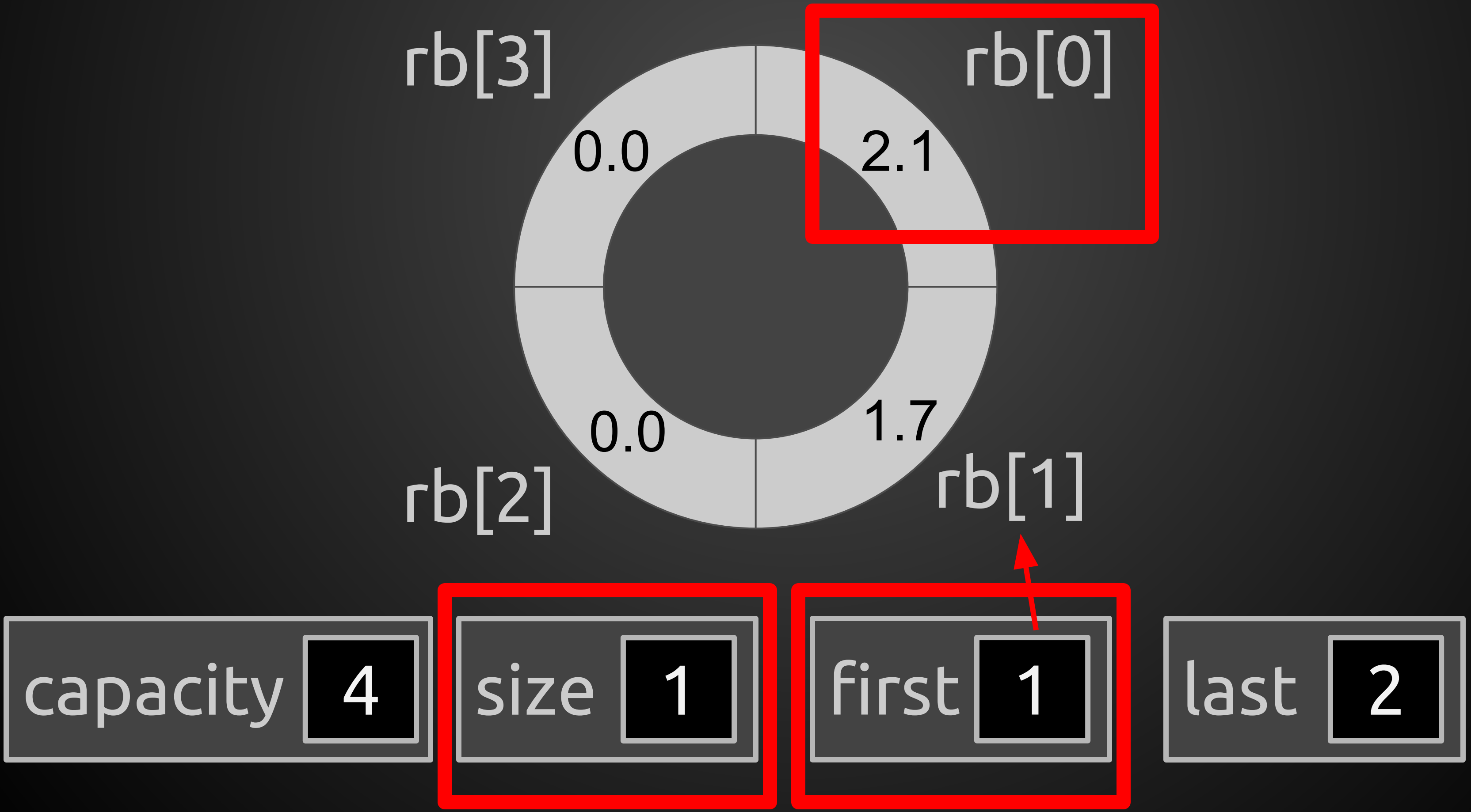
```
double val = buf.dequeue();
val = ?
```



capacity	size	first	last
4	2	0	2

RingBuffer

double val = buf.dequeue();
val = 2.1



capacity 4

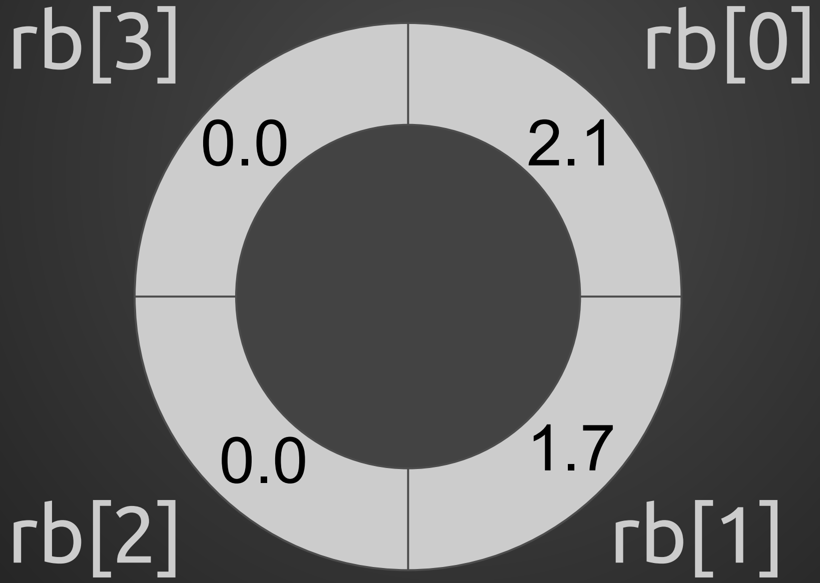
size 1

first 1

last 2

RingBuffer

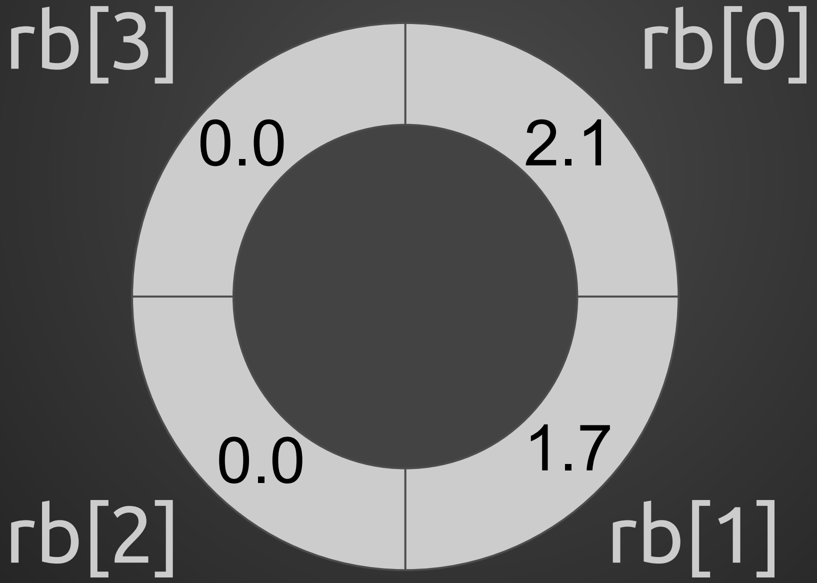
```
val = buf.dequeue();
val = ?
```



capacity	4	size	1	first	1	last	2

RingBuffer

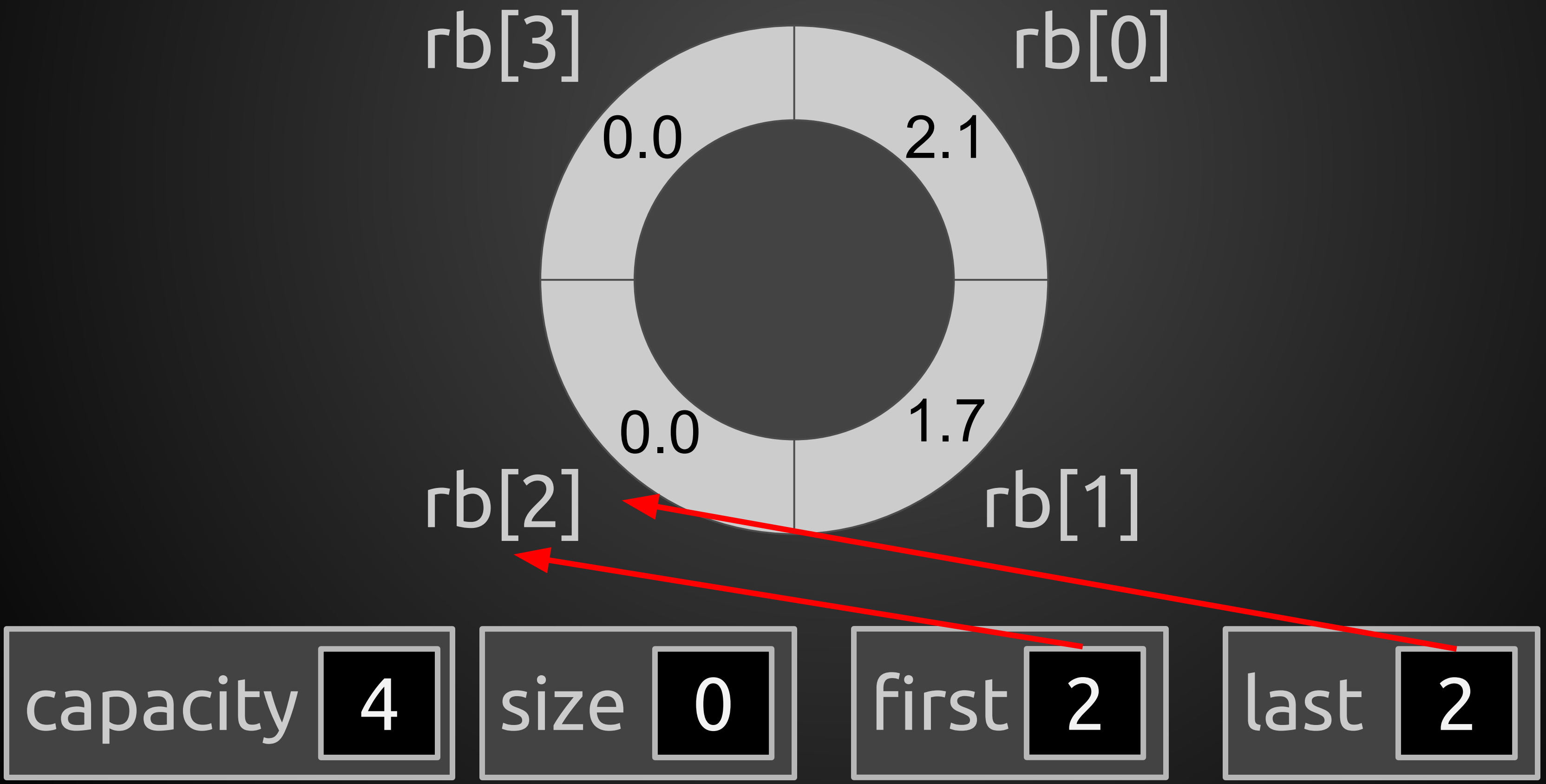
val = buf.dequeue();
val = 1.7



capacity 4	size 0	first 2	last 2

RingBuffer

```
val = buf.dequeue();
val = 1.7
```



rb[3]

rb[0]

0.0

2.1

0.0

1.7

rb[2]

rb[1]

capacity 4

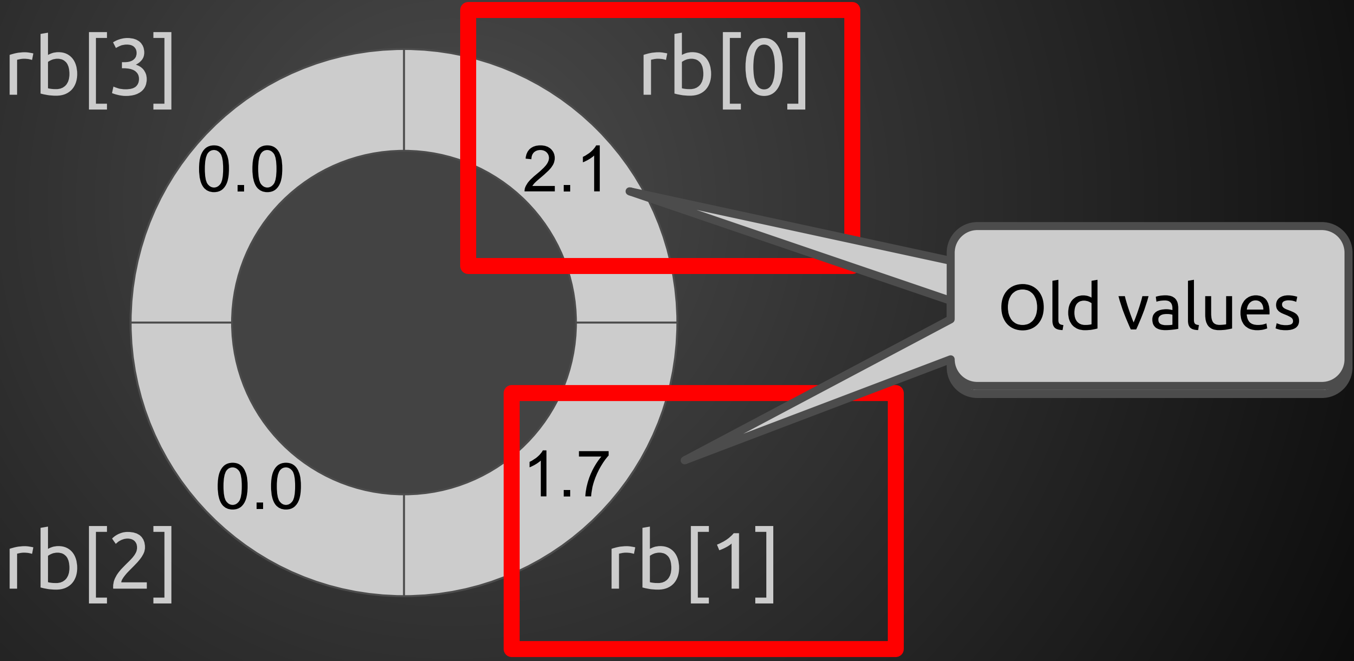
size 0

first 2

last 2

RingBuffer

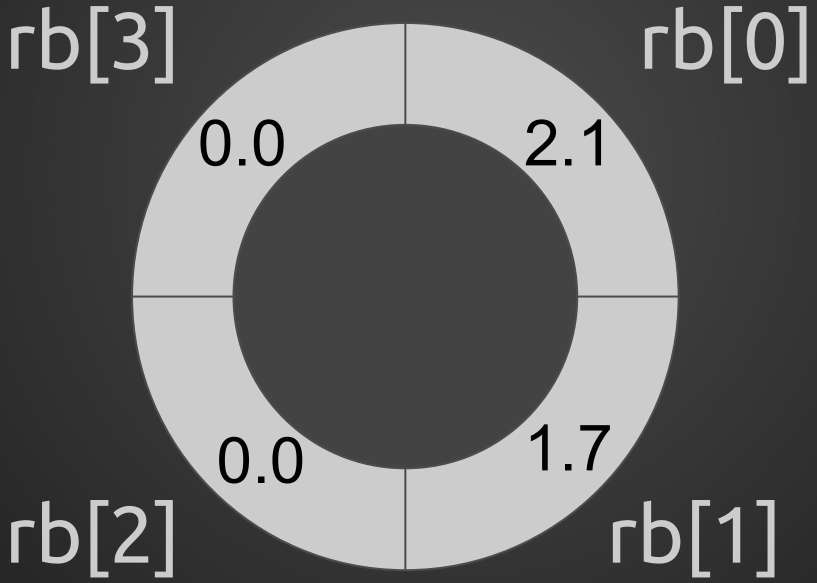
```
val = buf.dequeue();
val = 1.7
```



capacity	4	size	0	first	2	last	2

RingBuffer

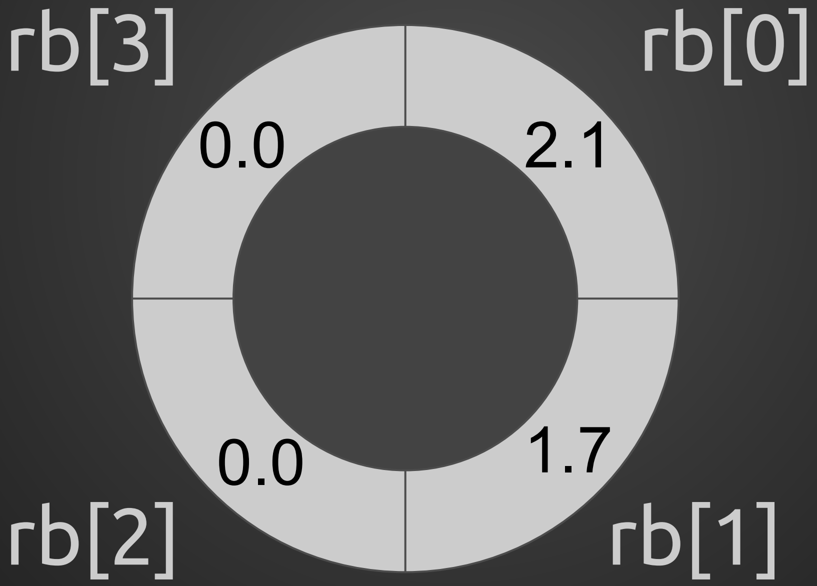
val = buf.dequeue();
val = ?



capacity 4	size 0	first 2	last 2

RingBuffer

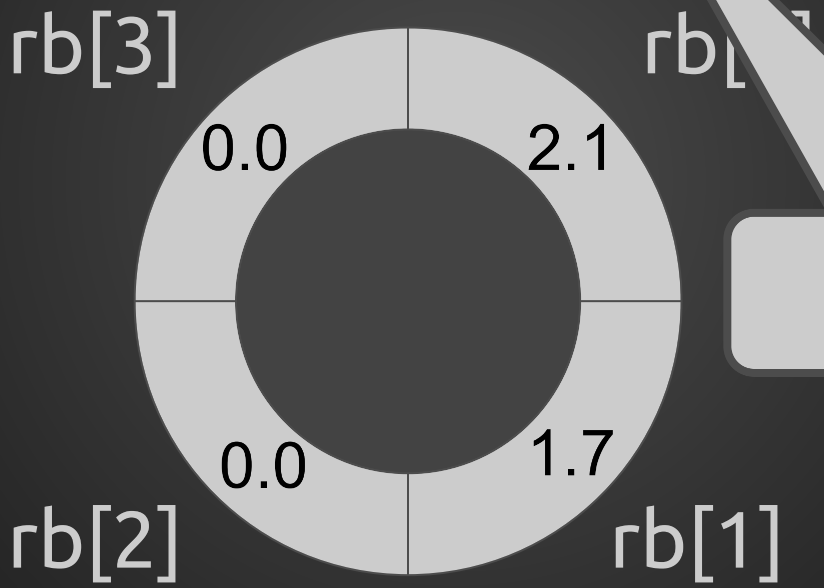
```
val = buf.dequeue();
val = ?
```



capacity	4	size	0	first	2	last	2

RingBuffer

```
val = buf.dequeue();
val = ?
```



EXCEPTION!

capacity 4

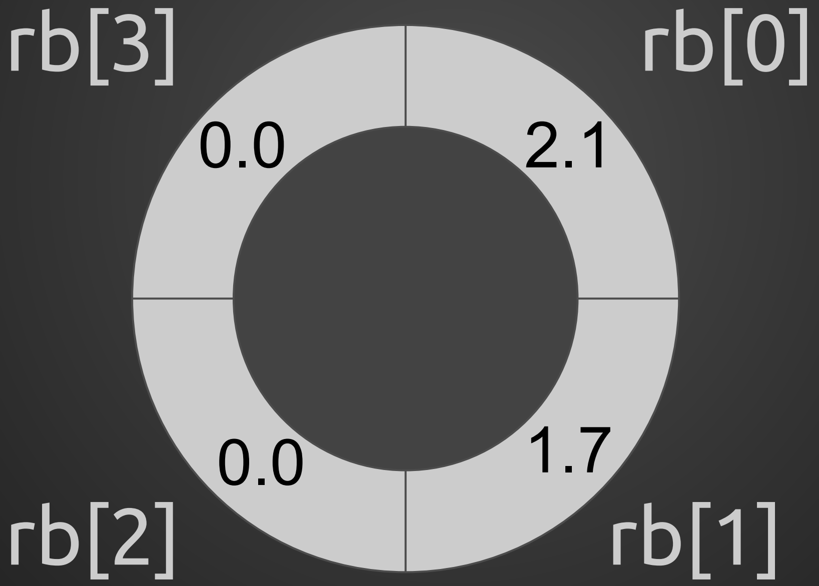
size 0

first 2

last 2

RingBuffer

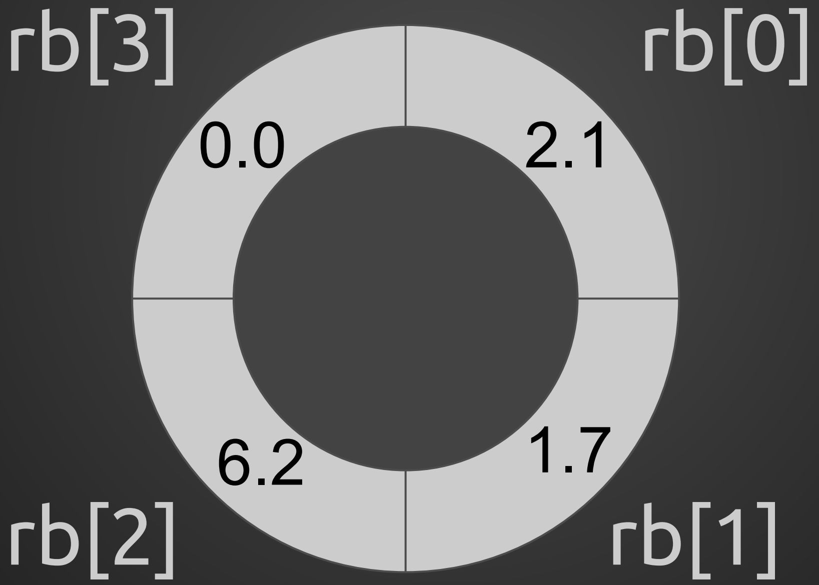
```
buf.enqueue(6.2);
```



capacity 4 | size 0 | first 2 | last 2

RingBuffer

```
buf.enqueue(6.2);
```



capacity 4

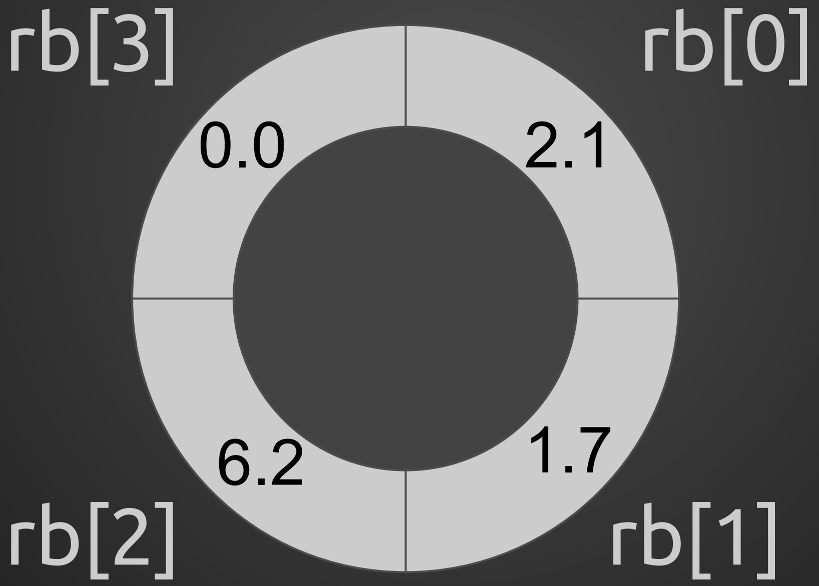
size 1

first 2

last 3

RingBuffer

```
buf.enqueue(3.7);
```



capacity 4

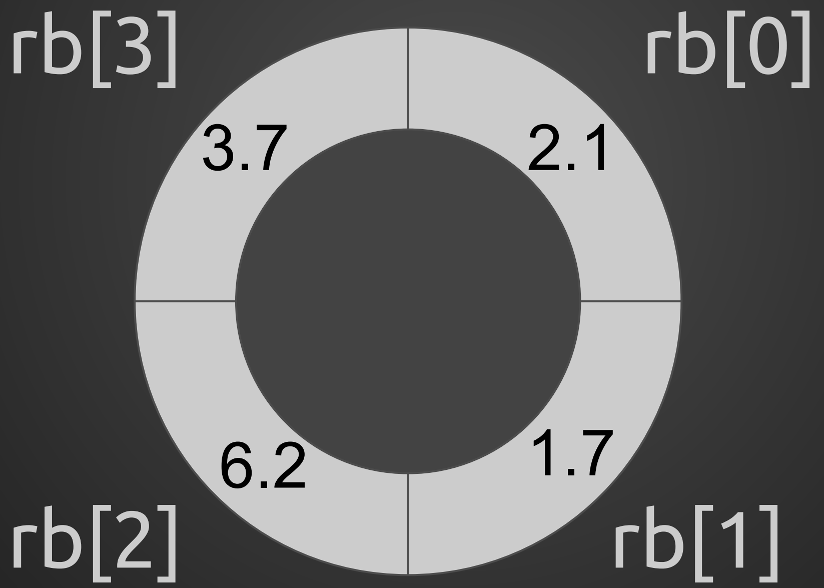
size 1

first 2

last 3

RingBuffer

```
buf.enqueue(3.7);
```



capacity 4

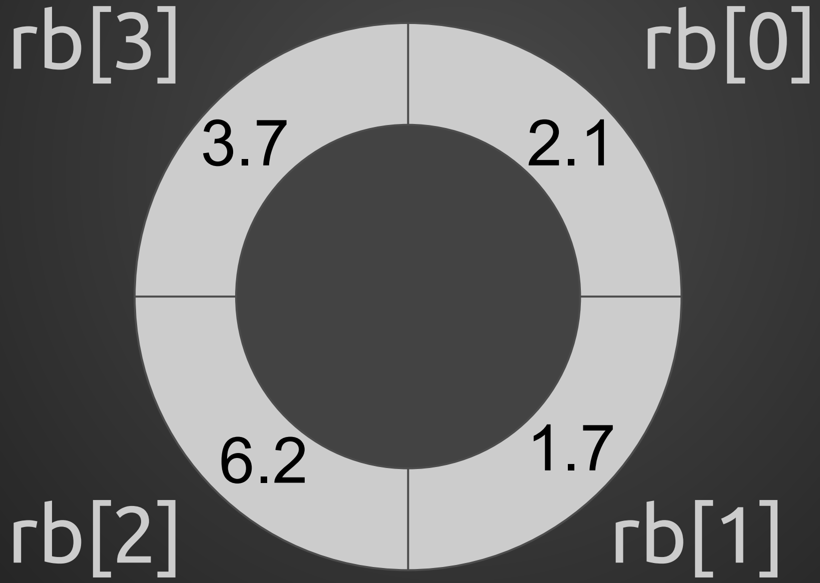
size 2

first 2

last 4

RingBuffer

```
buf.enqueue(3.7);
```



capacity 4

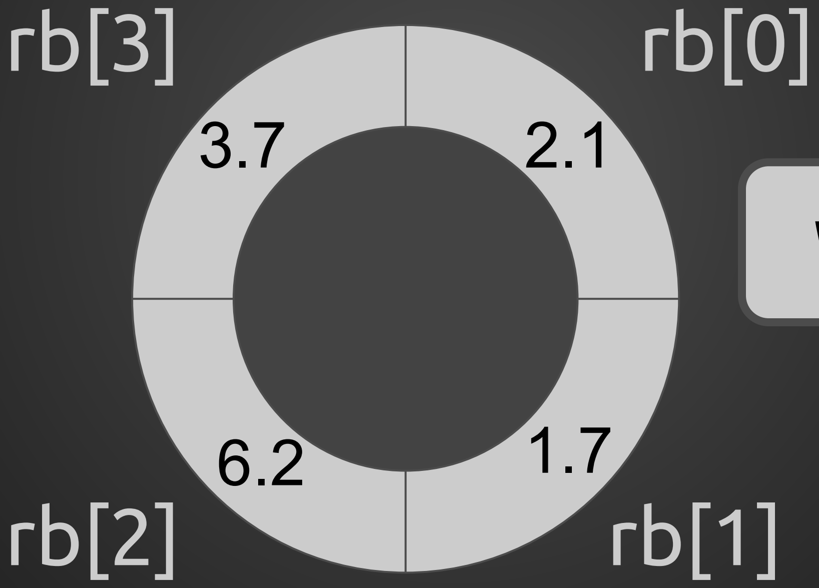
size 2

first 2

last ?

RingBuffer

```
buf.enqueue(3.7);
```



WRAP AROUND!

capacity 4

size 2

first 2

last 0

Discussion

- RingBuffer - similar to LFSR, except you don't shift all the elements down each time you insert a new value
- What is the order of growth of LFSR's step() method?

Discussion

- RingBuffer - similar to LFSR, except you don't shift all the elements down each time you insert a new value
- What is the order of growth of LFSR's step() method?
  - ANSWER - **linear** (shift elements of array)

Discussion

- RingBuffer - similar to LFSR, except you don't shift all the elements down each time you insert a new value
- What is the order of growth of LFSR's step() method?
  - ANSWER - **linear** (shift elements of array)
- What is the order of growth of RingBuffer’s enqueue() and dequeue() methods?

Discussion

- RingBuffer - similar to LFSR, except you don't shift all the elements down each time you insert a new value
- What is the order of growth of LFSR's step() method?
  - ANSWER - **linear** (shift elements of array)
- What is the order of growth of RingBuffer’s enqueue() and dequeue() methods?
  - ANSWER - **constant** (shift elements of array)
  - Updating the RingBuffer’s 44100 times per second!

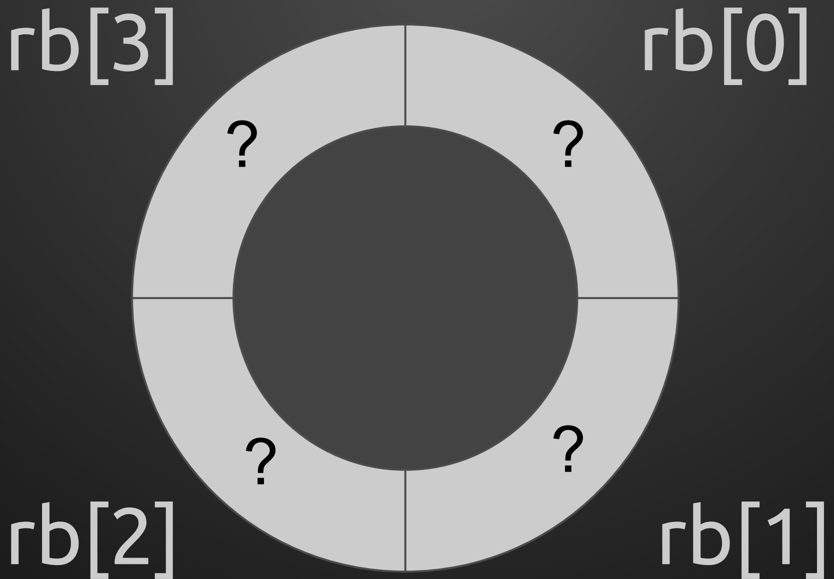
RingBuffer Testing/Debugging

What does the following code do:

```
double value = 0.0;
RingBuffer buf = new RingBuffer(4);
for (int i = 0; i < 4; i++) buf.enqueue(i/10.0);
for (int i = 0; i < 3; i++) value = buf.dequeue();
StdOut.println(value);
```

RingBuffer Testing/Debugging

```
double value = 0.0;
RingBuffer buf = new RingBuffer(4);
for (int i = 0; i < 4; i++) buf.enqueue(i/10.0);
for (int i = 0; i < 3; i++) value = buf.dequeue();
StdOut.println(value);
```



capacity □

size □

first □

last □

GuitarString

Each GuitarString has one RingBuffer

GuitarString

Each GuitarString has one RingBuffer.

GuitarString has two constructors

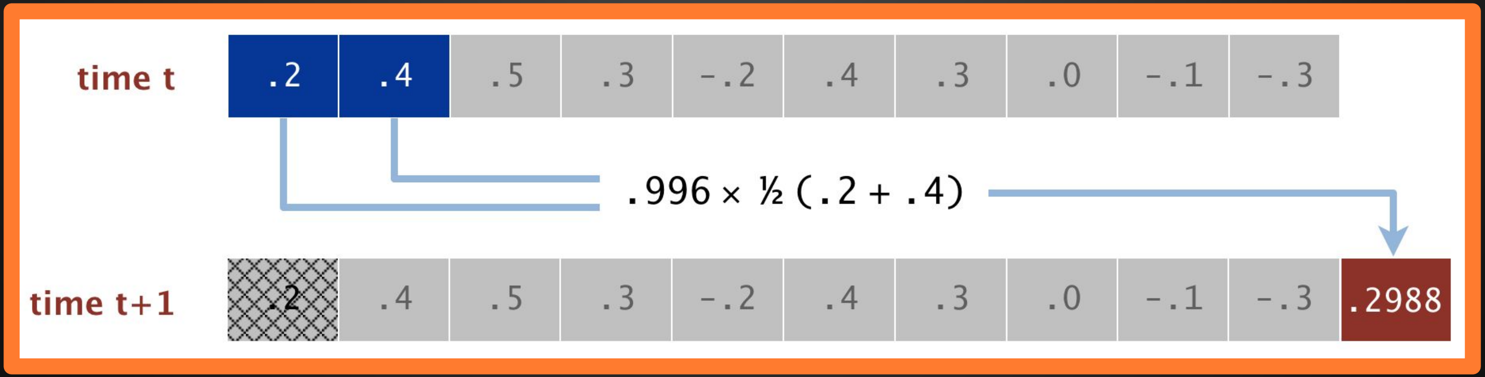
The job of ***every*** constructor is to initialize ***all*** instance variables!

GuitarString

Each GuitarString has one RingBuffer

GuitarString has two constructors. The job of ***every*** constructor is to initialize ***all*** instance variables!

Implement Karplus-Strong algorithm



time t	.2	.4	.5	.3	-.2	.4	.3	.0	-.1	-.3	
time t+1	.2	.4	.5	.3	-.2	.4	.3	.0	-.1	-.3	.2988

GuitarString

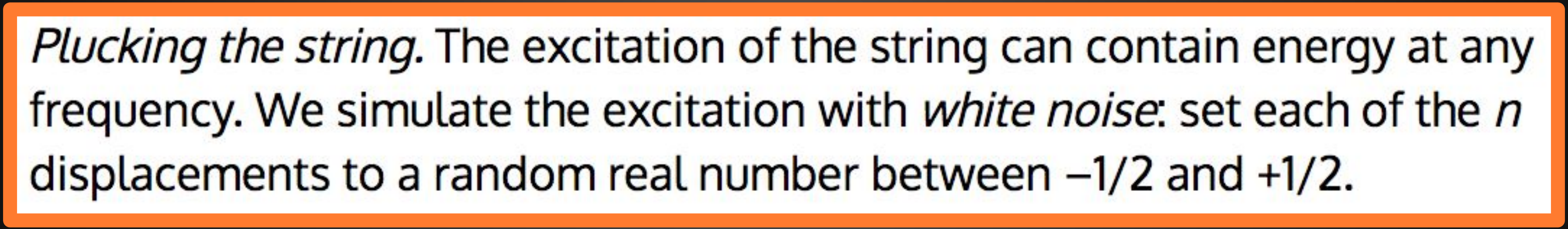
Implement Karplus-Strong algorithm.

Takes random numbers and turns them into music!

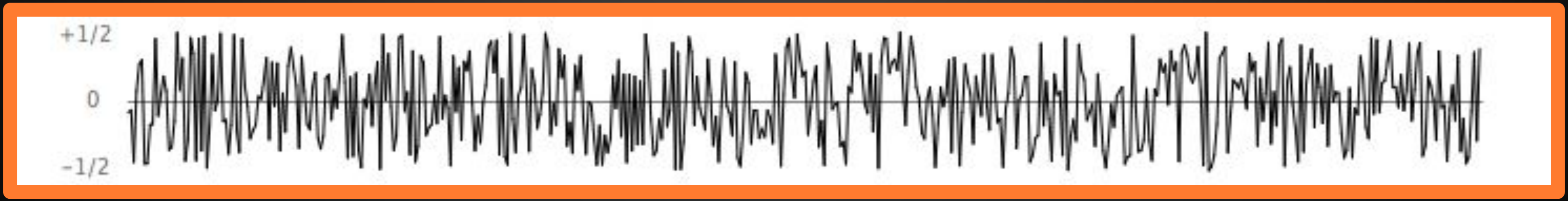
GuitarString

Implement Karplus-Strong algorithm.

Takes random numbers and turns them into music!



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.



GuitarString

Two constructors:

1. GuitarString(double frequency)
2. GuitarString(double[] init)

GuitarString

Two constructors:

1. **GuitarString(double frequency)**
"The first constructor creates a RingBuffer of the desired capacity n (the sampling rate 44,100 divided by the frequency, rounded up to the nearest integer), and initializes it to represent a guitar string at rest by **enqueuing n zeros**"

2. GuitarString(double[] init)

GuitarString

Two constructors:

1. GuitarString(double frequency)

2. **GuitarString(double[] init)**

"The second constructor creates a RingBuffer of capacity equal to the length n of the array, and initializes the contents of the ring buffer to the corresponding values in the array. In this assignment, this constructor's main purpose is to facilitate testing and debugging"

GuitarString

Two constructors:

1. GuitarString(double frequency)
2. GuitarString(double[] init)

Did you initialize **all** your instance variables in both constructors?

GuitarString

pluck() replaces all n items in a RingBuffer with n random values between -0.5 and +0.5

GuitarString

pluck() replaces all n items in a RingBuffer with n random values between -0.5 and +0.5

How many elements will be in your RingBuffer
… before calling pluck()?
… after calling pluck()?

GuitarString

pluck() replaces all n items in the ring buffer with n random values between -0.5 and +0.5

How many elements will be in your RingBuffer

… before calling pluck()?

… after calling pluck()?

Always n

GuitarString

pluck() replaces all n items in a RingBuffer with n random values between -0.5 and +0.5

How many elements will be in your RingBuffer
… before calling pluck()?
… after calling pluck()?

How to **replace** n elements in a RingBuffer?

GuitarString

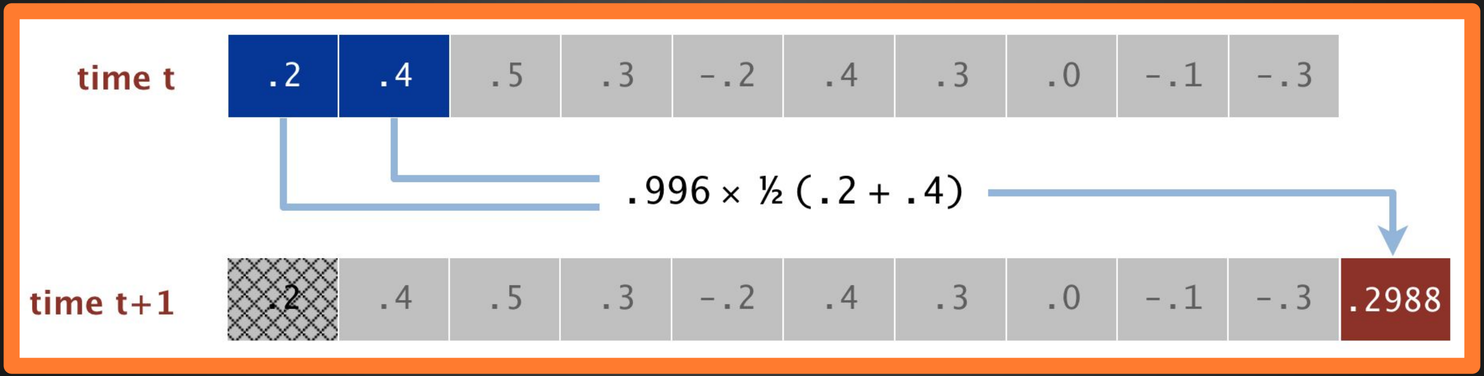
pluck() replaces all n items in a RingBuffer with n random values between -0.5 and +0.5

How many elements will be in your RingBuffer
… before calling pluck()?
… after calling pluck()?

How to **replace** n elements in a RingBuffer?

GuitarString

tic() "**delete** the first sample from RingBuffer and **adds to the end** of the RingBuffer the average of the deleted sample and the **first sample**, scaled by an energy decay factor of 0.996"

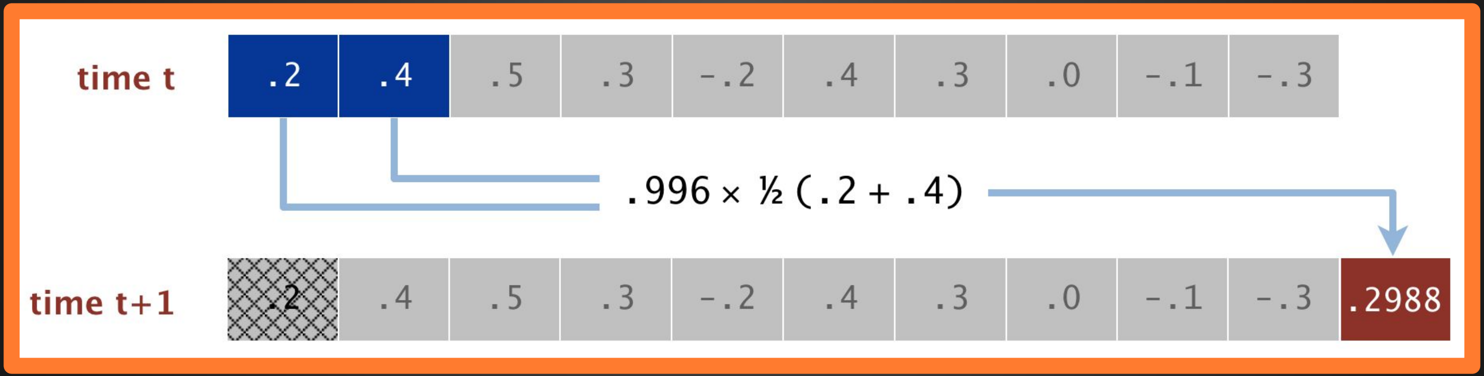


time t	.2	.4	.5	.3	-.2	.4	.3	.0	-.1	-.3	
time t+1	.2	.4	.5	.3	-.2	.4	.3	.0	-.1	-.3	.2988

.996 × ½ (.2 + .4)

GuitarString

tic() "**delete** the first sample from RingBuffer and **adds to the end** of the RingBuffer the average of the deleted sample and the **first sample**, scaled by an energy decay factor of 0.996"



time t	.2	.4	.5	.3	-.2	.4	.3	.0	-.1	-.3	
time t+1	.2	.4	.5	.3	-.2	.4	.3	.0	-.1	-.3	.2988

$.996 \times \frac{1}{2}(.2 + .4)$

sample() "return the value of the **item at the front** of the RingBuffer"

GuitarString

main() write your own tests here. must call every method and, if the method has a return value, should use that value for something, like printing

The test cases you write in main() will improve your understanding!

GuitarHero

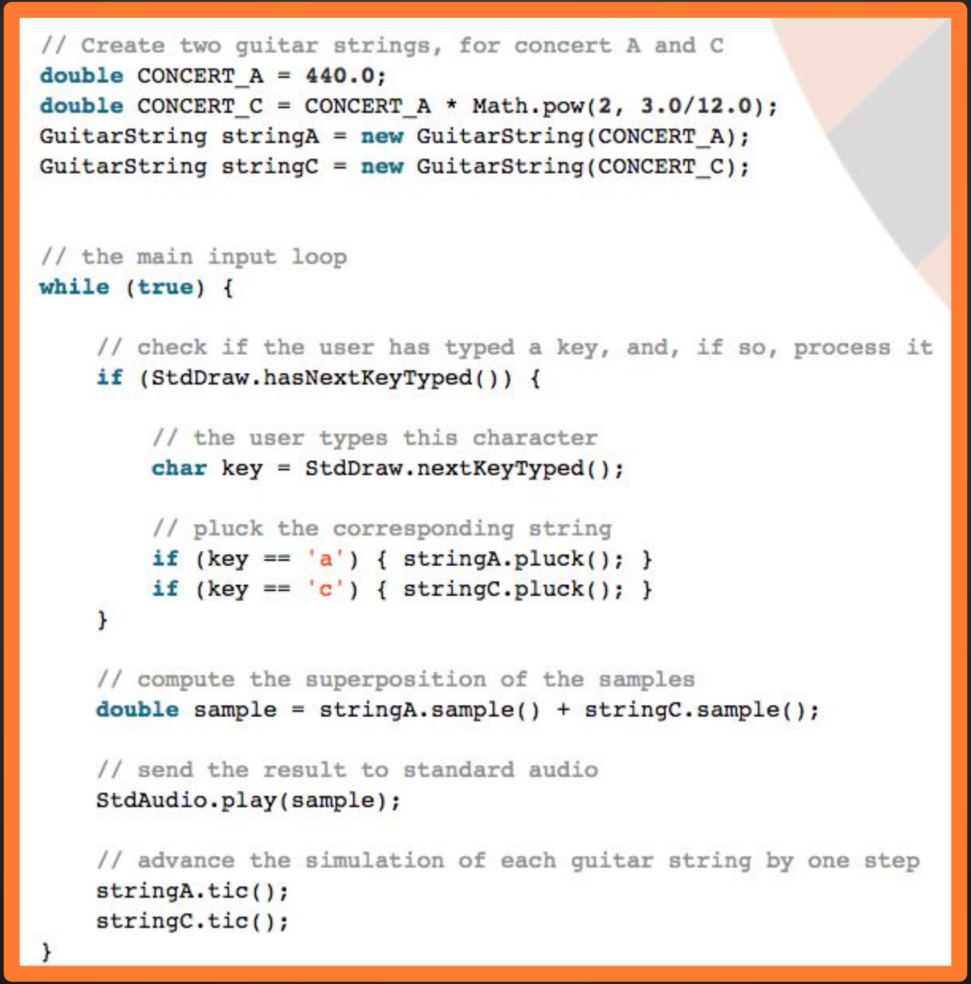
- Model many simultaneously vibrating guitar strings
- Classic guitar has 6 strings and 19 frets
- Our digital guitar has 37 strings
- Create an array of GuitarString objects
- Apply law of superposition

string i has frequency

$440 \times 2^{(i-24)/12}$

GuitarHero

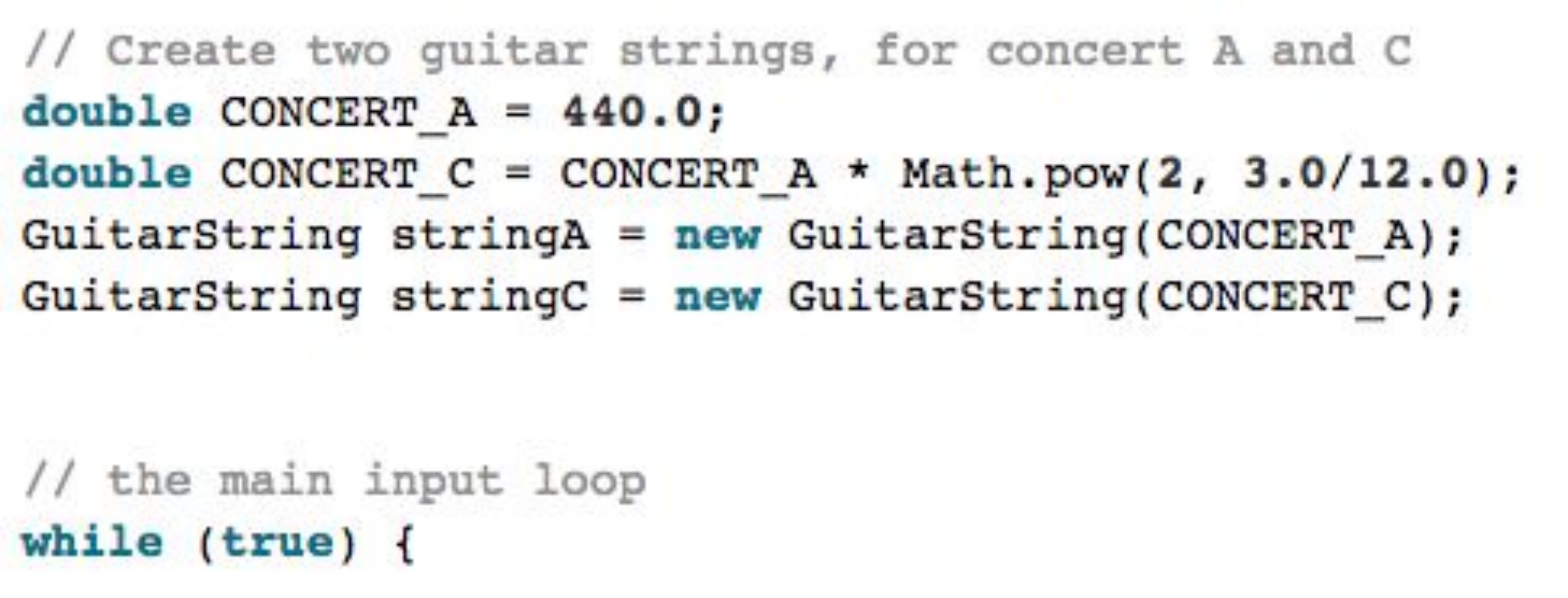
Take GuitarHeroLite and add 35 GuitarStrings to it!



```java
// Create two guitar strings, for concert A and C
double CONCERT_A = 440.0;
double CONCERT_C = CONCERT_A * Math.pow(2, 3.0/12.0);
GuitarString stringA = new GuitarString(CONCERT_A);
GuitarString stringC = new GuitarString(CONCERT_C);


// the main input loop
while (true) {

    // check if the user has typed a key, and, if so, process it
    if (StdDraw.hasNextKeyTyped()) {

        // the user types this character
        char key = StdDraw.nextKeyTyped();

        // pluck the corresponding string
        if (key == 'a') { stringA.pluck(); }
        if (key == 'c') { stringC.pluck(); }
    }

    // compute the superposition of the samples
    double sample = stringA.sample() + stringC.sample();

    // send the result to standard audio
    StdAudio.play(sample);

    // advance the simulation of each guitar string by one step
    stringA.tic();
    stringC.tic();
}
```

GuitarHero

Starts like this...

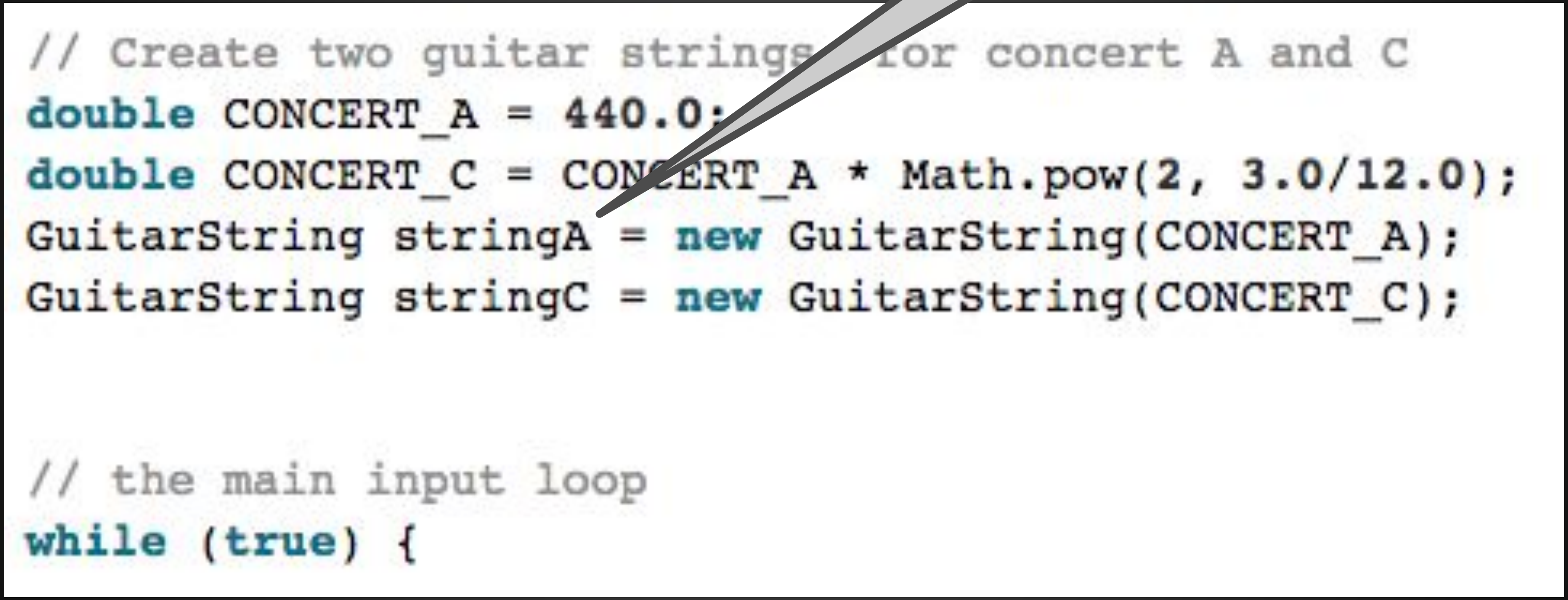


```
// Create two guitar strings, for concert A and C
double CONCERT_A = 440.0;
double CONCERT_C = CONCERT_A * Math.pow(2, 3.0/12.0);
GuitarString stringA = new GuitarString(CONCERT_A);
GuitarString stringC = new GuitarString(CONCERT_C);


// the main input loop
while (true) {
```

GuitarHero

Starts like this…

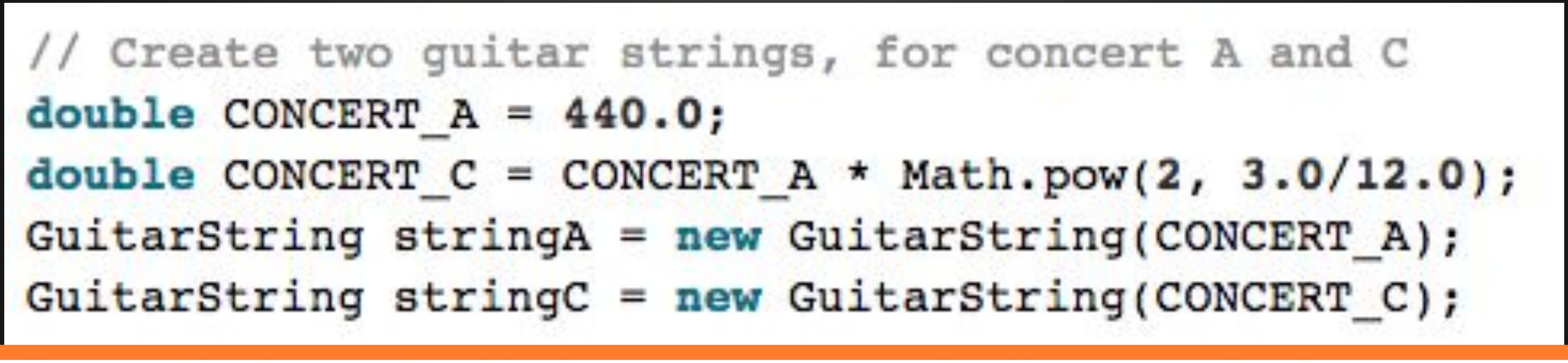
Do not make 37 GuitarString variables! Use an array



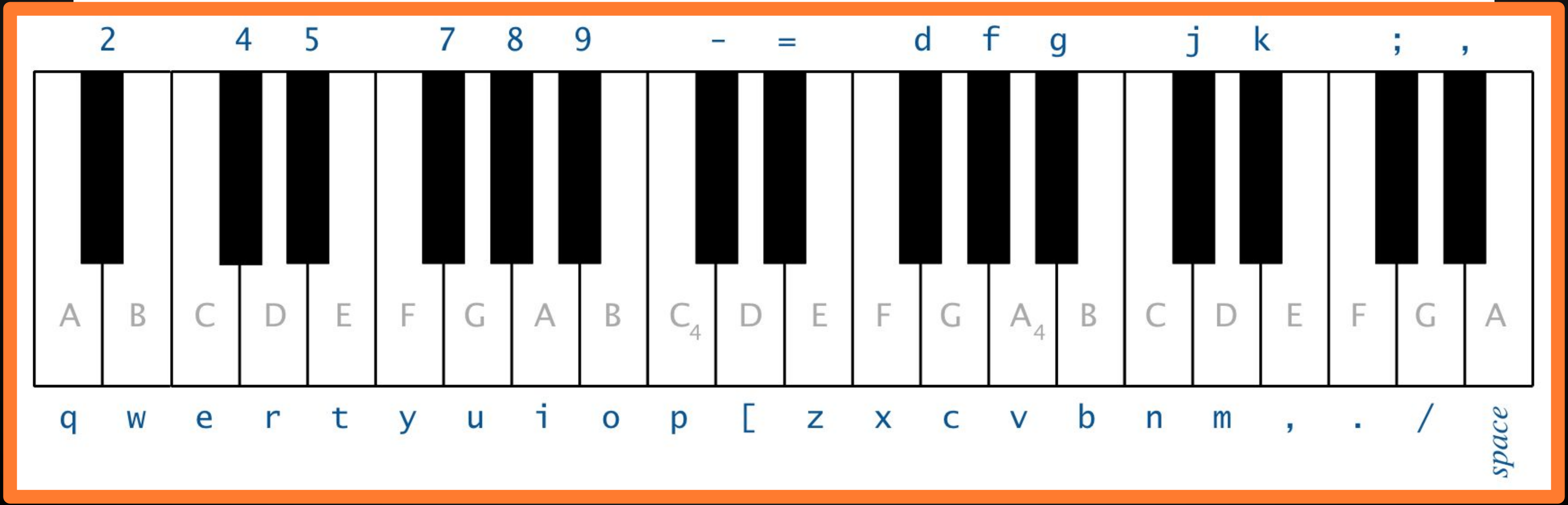
```
// Create two guitar strings, for concert A and C
double CONCERT_A = 440.0;
double CONCERT_C = CONCERT_A * Math.pow(2, 3.0/12.0);
GuitarString stringA = new GuitarString(CONCERT_A);
GuitarString stringC = new GuitarString(CONCERT_C);


// the main input loop
while (true) {
```

GuitarHero

Starts like this...



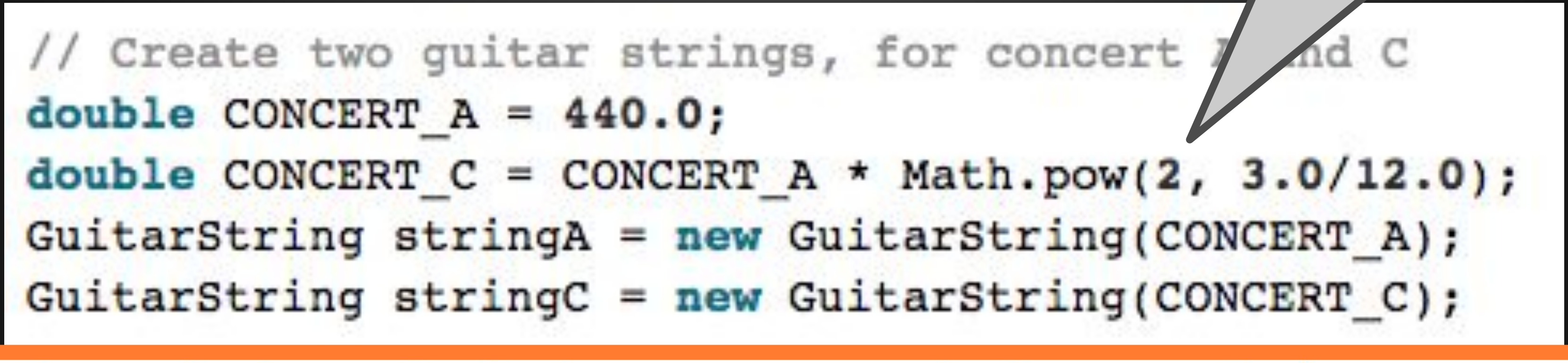
```
// Create two guitar strings, for concert A and C
double CONCERT_A = 440.0;
double CONCERT_C = CONCERT_A * Math.pow(2, 3.0/12.0);
GuitarString stringA = new GuitarString(CONCERT_A);
GuitarString stringC = new GuitarString(CONCERT_C);
```



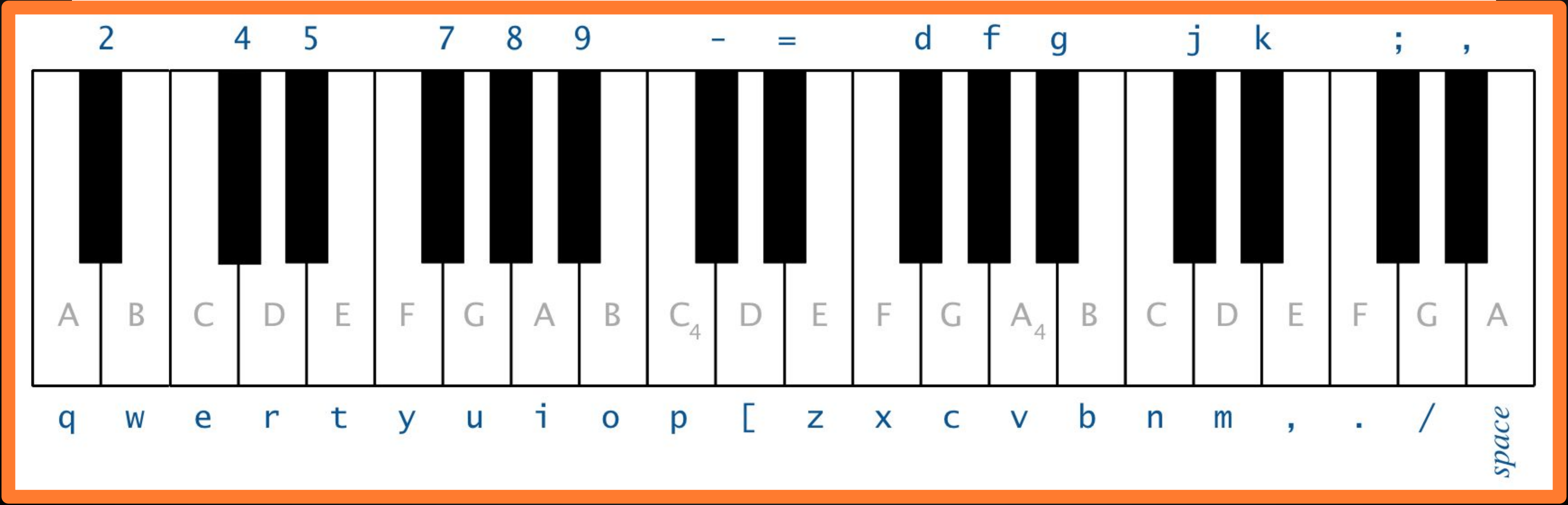
GuitarHero

Starts like this…

The formula for this mapping is is similar to this - Be careful of integer division!

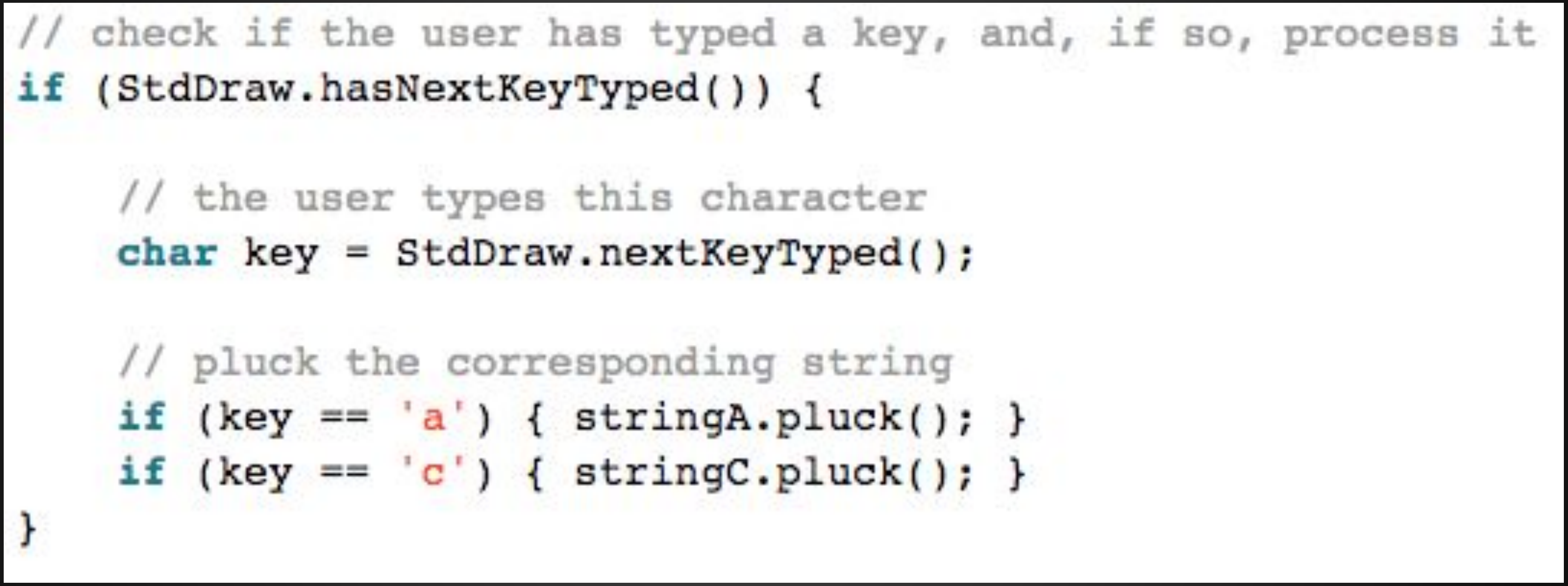


```
// Create two guitar strings, for concert A and C
double CONCERT_A = 440.0;
double CONCERT_C = CONCERT_A * Math.pow(2, 3.0/12.0);
GuitarString stringA = new GuitarString(CONCERT_A);
GuitarString stringC = new GuitarString(CONCERT_C);
```


2 4 5 7 8 9 - = d f g j k ; '
A B C D E F G A B C₄ D E F G A₄ B C D E F G A
q w e r t y u i o p [z x c v b n m , . / space

GuitarHero

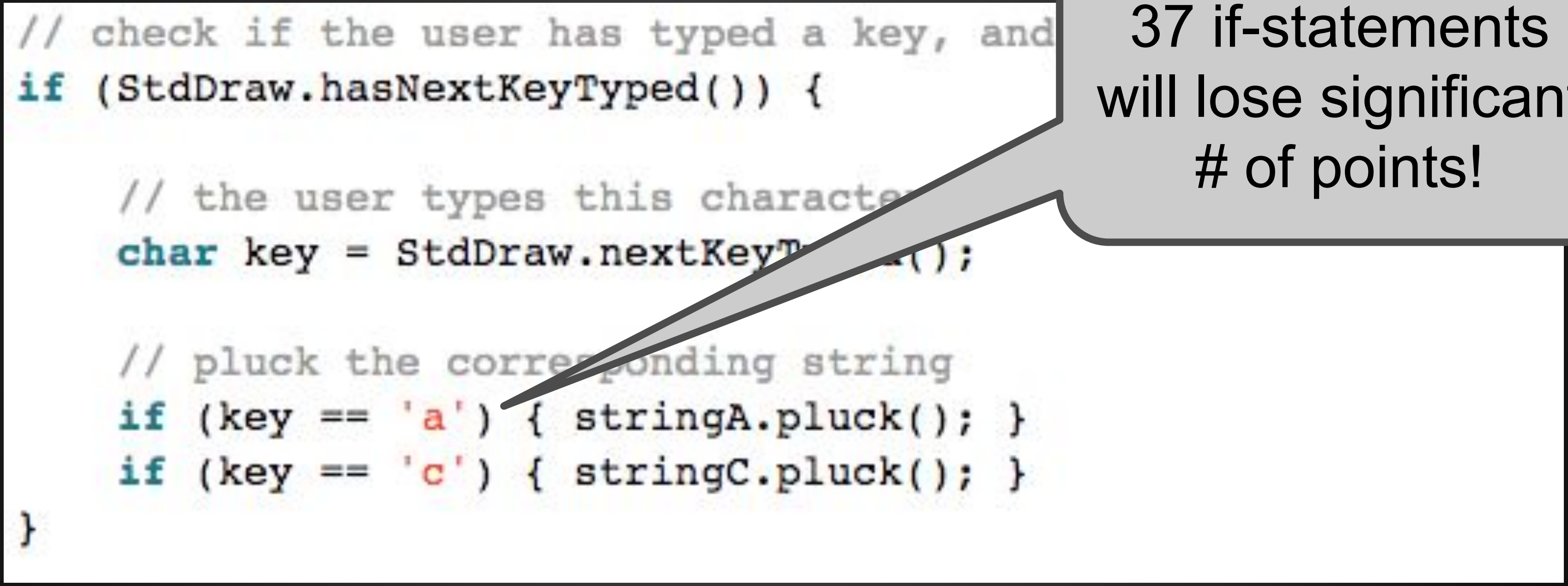
Now, the first part of the loop...



```
// check if the user has typed a key, and, if so, process it
if (StdDraw.hasNextKeyTyped()) {

    // the user types this character
    char key = StdDraw.nextKeyTyped();

    // pluck the corresponding string
    if (key == 'a') { stringA.pluck(); }
    if (key == 'c') { stringC.pluck(); }
}
```

GuitarHero

Now, the first part of the loop...

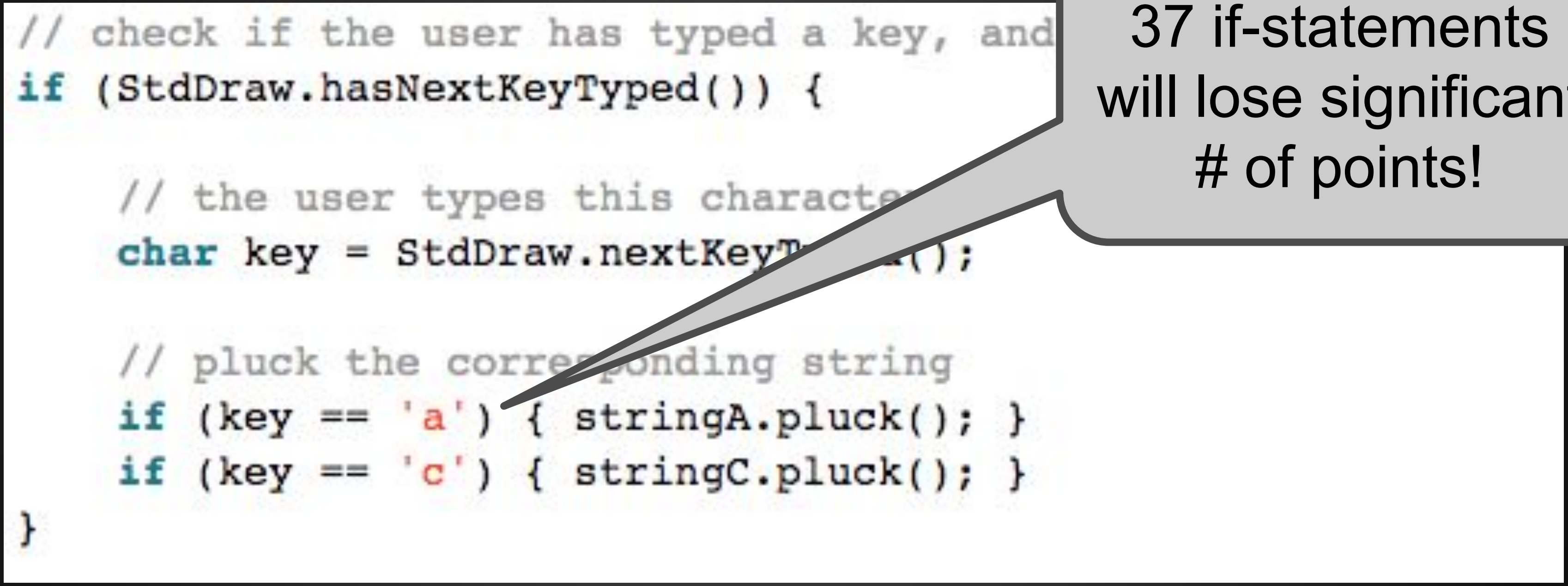


```
// check if the user has typed a key, and
if (StdDraw.hasNextKeyTyped()) {

    // the user types this characte
    char key = StdDraw.nextKeyT();

    // pluck the corresponding string
    if (key == 'a') { stringA.pluck(); }
    if (key == 'c') { stringC.pluck(); }
}
```

37 if-statements will lose significant # of points!

GuitarHero

Now, the first part of the loop...



```
// check if the user has typed a key, and
if (StdDraw.hasNextKeyTyped()) {

    // the user types this character
    char key = StdDraw.nextKeyTyped();

    // pluck the corresponding string
    if (key == 'a') { stringA.pluck(); }
    if (key == 'c') { stringC.pluck(); }
}
```

37 if-statements will lose significant # of points!

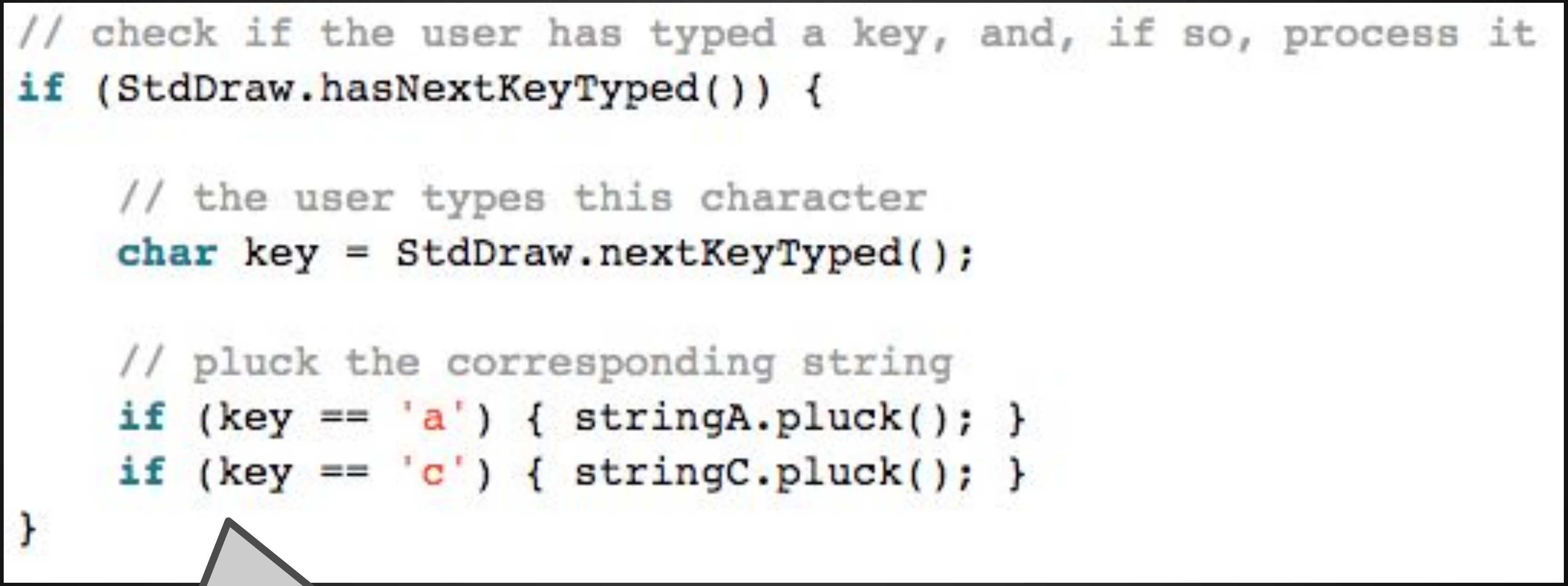
Instead, use **keyboard.indexOf()**

GuitarHero

```
String keyboard =
      "q2we4r5ty7u8i9op-[=zxdcfvgbnjmk,.;/' ";
...
keyboard.length();        // don’t hardwire 37!
keyboard.indexOf('q');    //  0
keyboard.indexOf('r');    //  5
keyboard.indexOf('+');    // -1
```

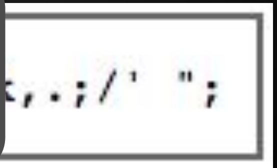
GuitarHero

Now, the first part of the loop...



```
// check if the user has typed a key, and, if so, process it
if (StdDraw.hasNextKeyTyped()) {

    // the user types this character
    char key = StdDraw.nextKeyTyped();

    // pluck the corresponding string
    if (key == 'a') { stringA.pluck(); }
    if (key == 'c') { stringC.pluck(); }
}
```

tead, use **keyboard.indexOf()**

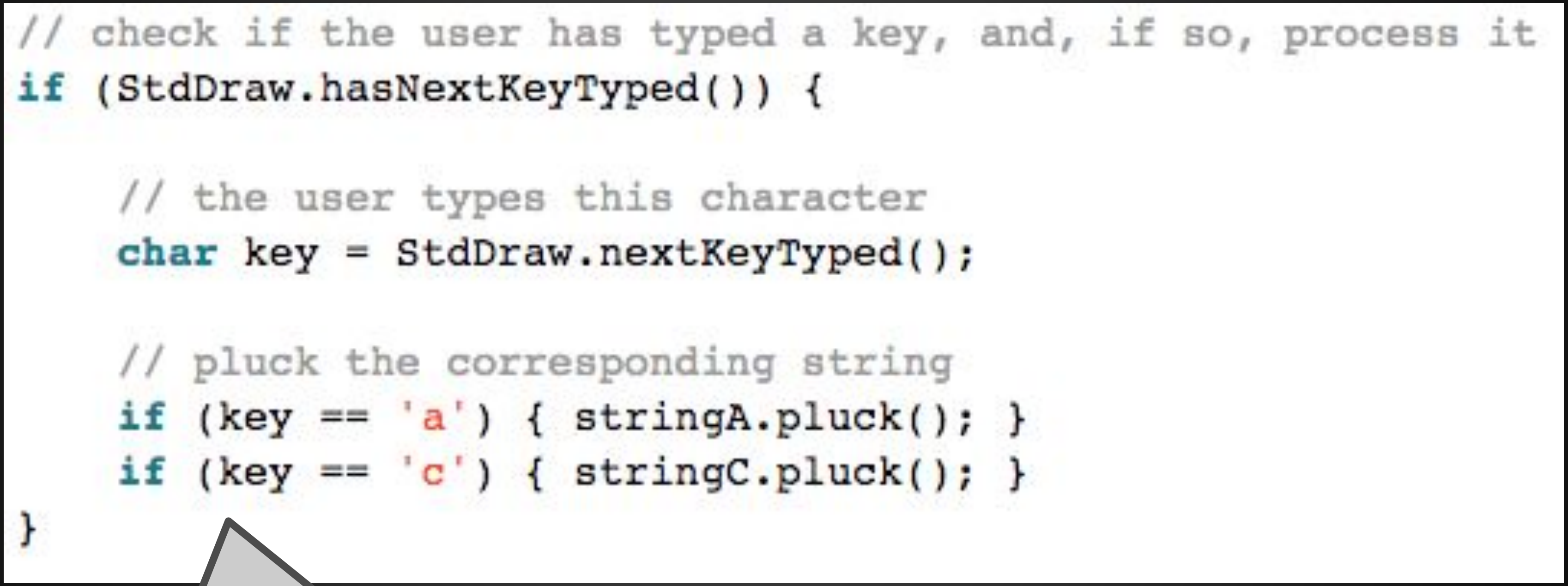


```
<,.;/' ";
```

What should you do if the user presses a key that is not on the keyboard?

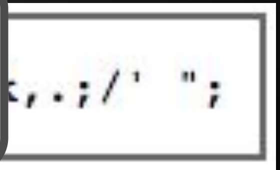
GuitarHero

Now, the first part of the loop...



```
// check if the user has typed a key, and, if so, process it
if (StdDraw.hasNextKeyTyped()) {

    // the user types this character
    char key = StdDraw.nextKeyTyped();

    // pluck the corresponding string
    if (key == 'a') { stringA.pluck(); }
    if (key == 'c') { stringC.pluck(); }
}
```

tead, use **keyboard.indexOf()**

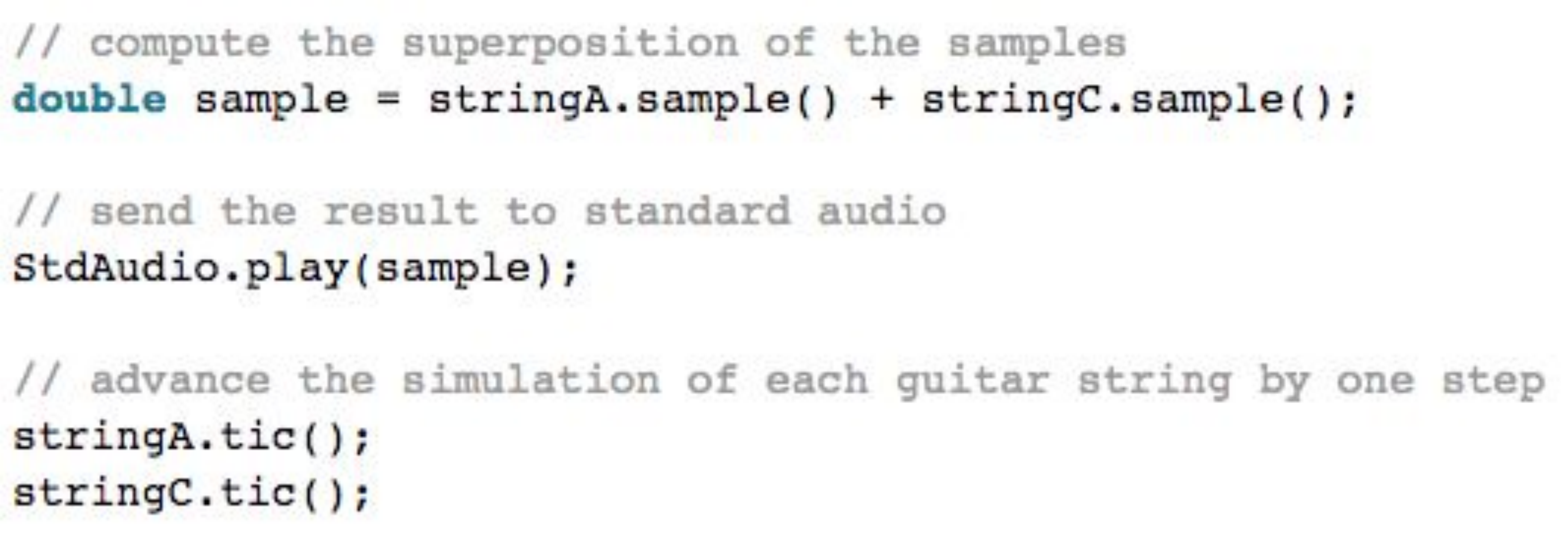


```
k,.;/'  ";
```

What should you do if the user presses a key that is not on the keyboard? **Ignore it**

GuitarHero

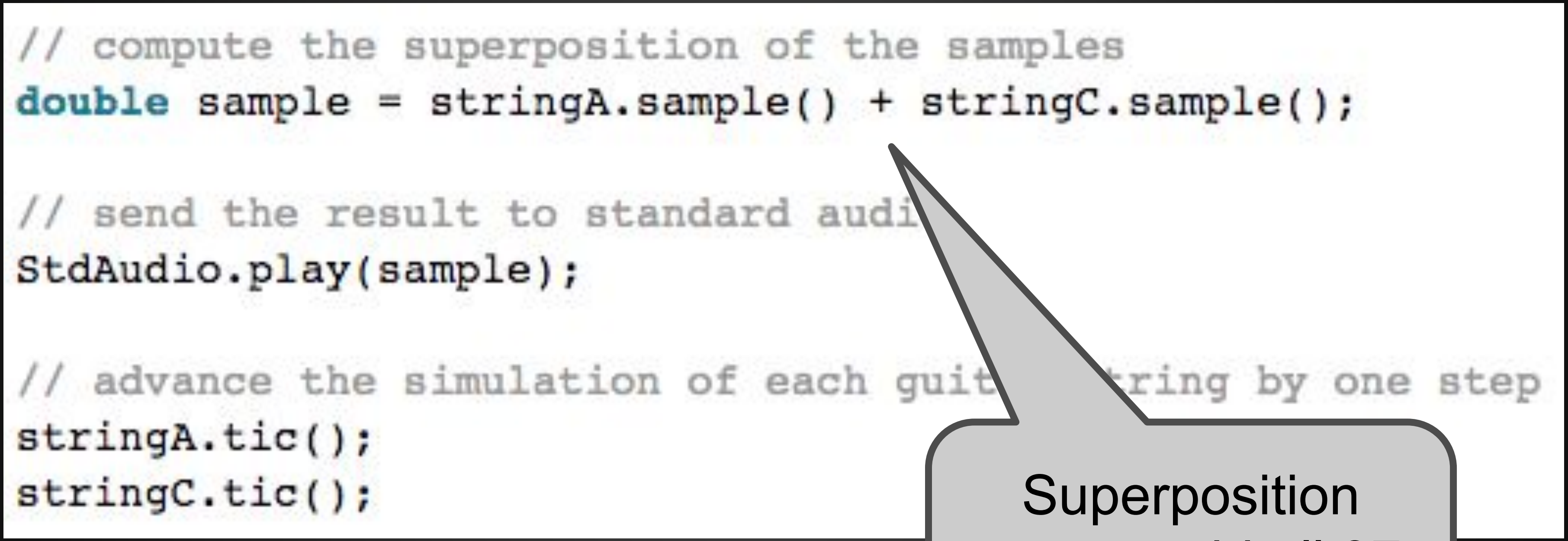
Last, handle the superposition correctly.



```
// compute the superposition of the samples
double sample = stringA.sample() + stringC.sample();

// send the result to standard audio
StdAudio.play(sample);

// advance the simulation of each guitar string by one step
stringA.tic();
stringC.tic();
```

GuitarHero

Last, handle the superposition correctly.

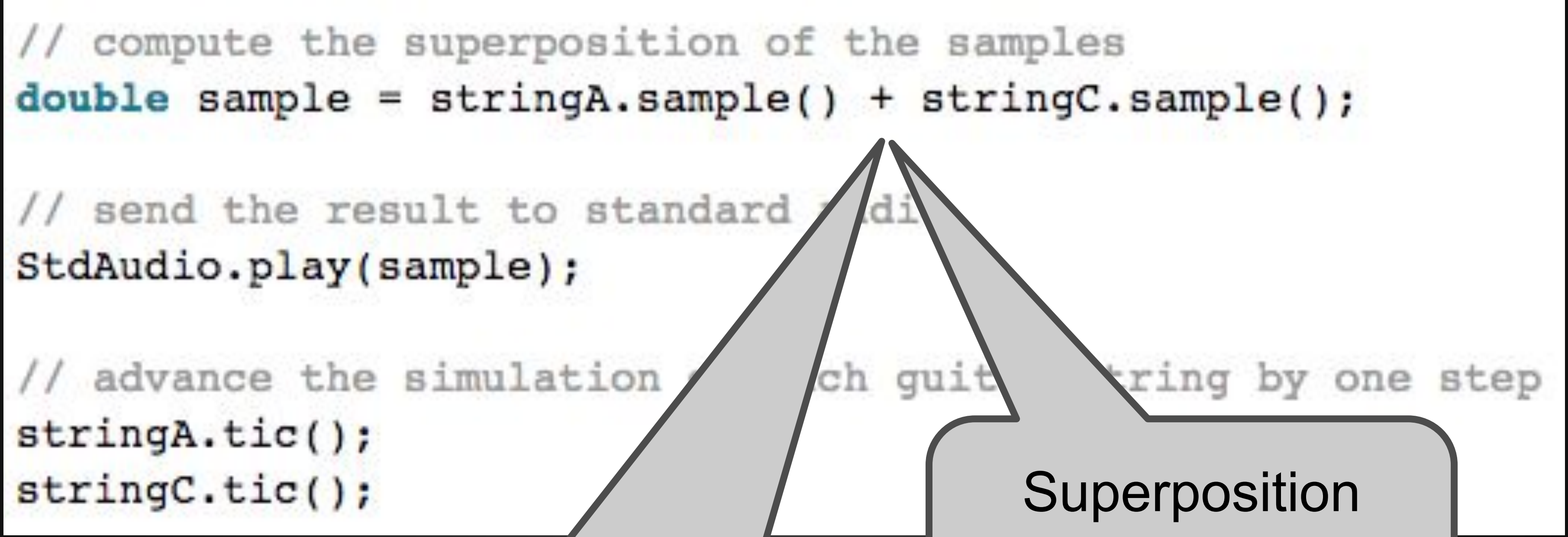


```
// compute the superposition of the samples
double sample = stringA.sample() + stringC.sample();

// send the result to standard audi

StdAudio.play(sample);

// advance the simulation of each guit      tring by one step
stringA.tic();
stringC.tic();
```

Superposition means add all 37 samples together

GuitarHero

Last, handle the superposition correctly.



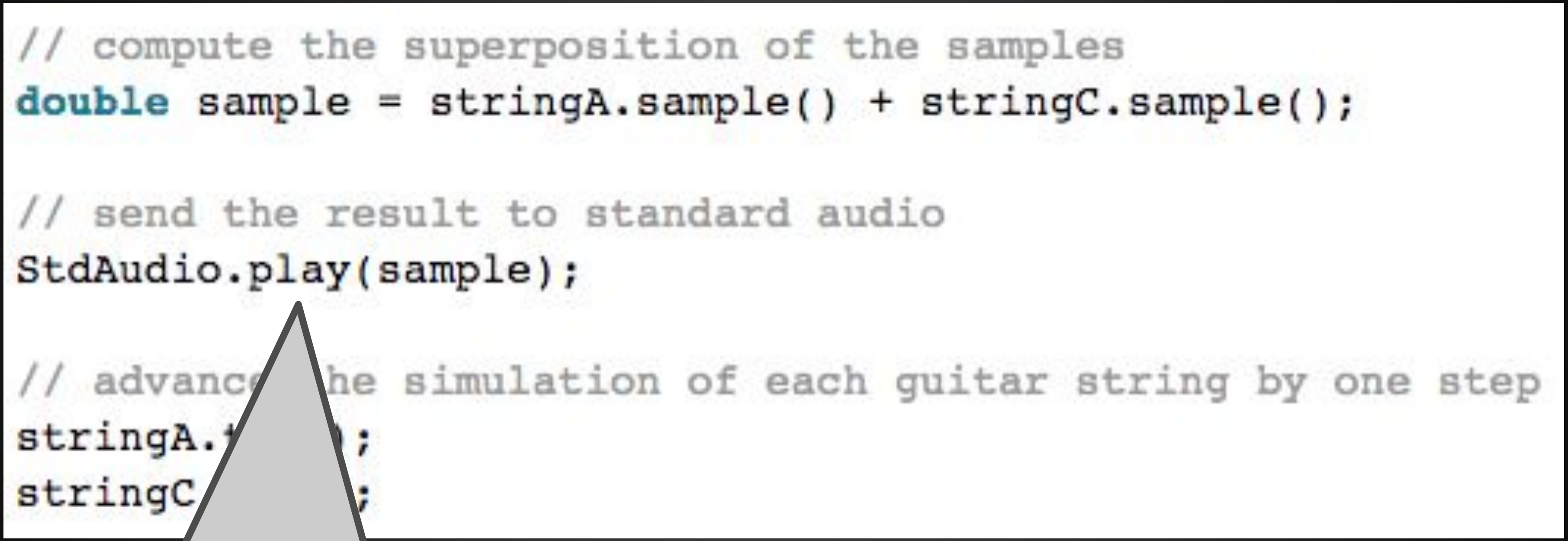
```
// compute the superposition of the samples
double sample = stringA.sample() + stringC.sample();

// send the result to standard a di

StdAudio.play(sample);

// advance the simulation o   ch guit    tring by one step
stringA.tic();
stringC.tic();
```

When you calculate this sum in a loop, don't forget to reset the sum to 0 between iterations!

Superposition means add all 37 samples together

GuitarHero

Last, handle the superposition correctly.

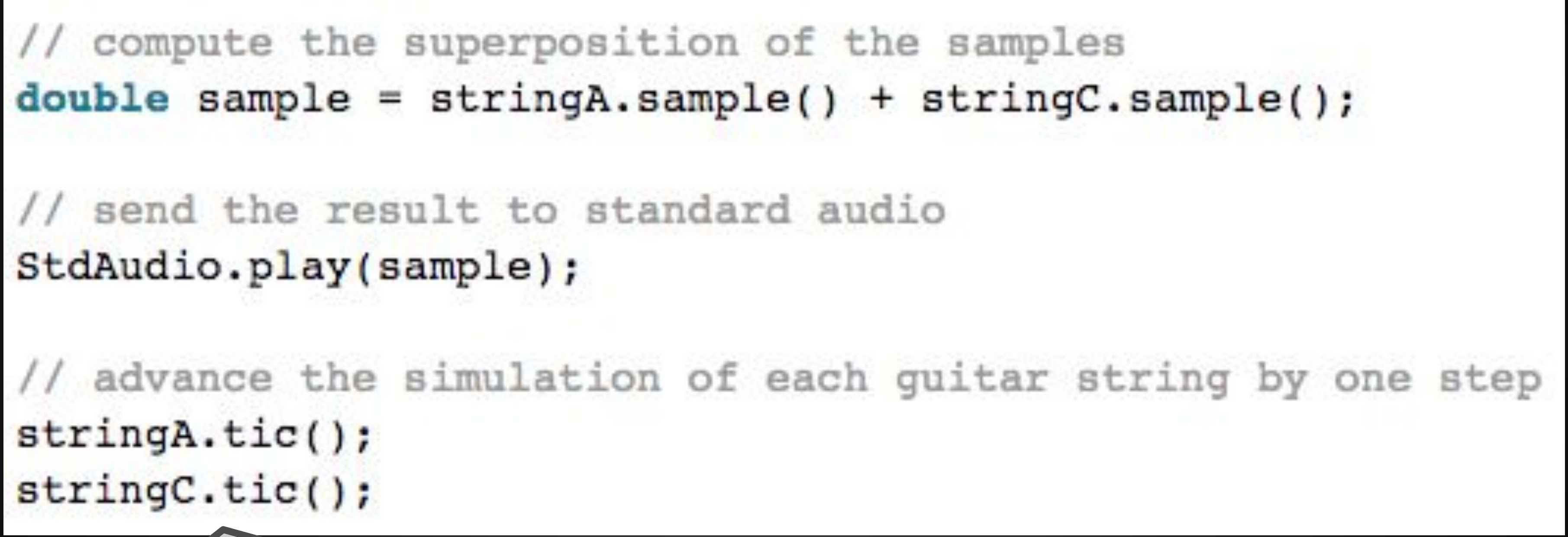


```
// compute the superposition of the samples
double sample = stringA.sample() + stringC.sample();

// send the result to standard audio
StdAudio.play(sample);

// advance the simulation of each guitar string by one step
stringA.tic();
stringC.tic();
```

Notice that, we play only once after summing all the samples

GuitarHero

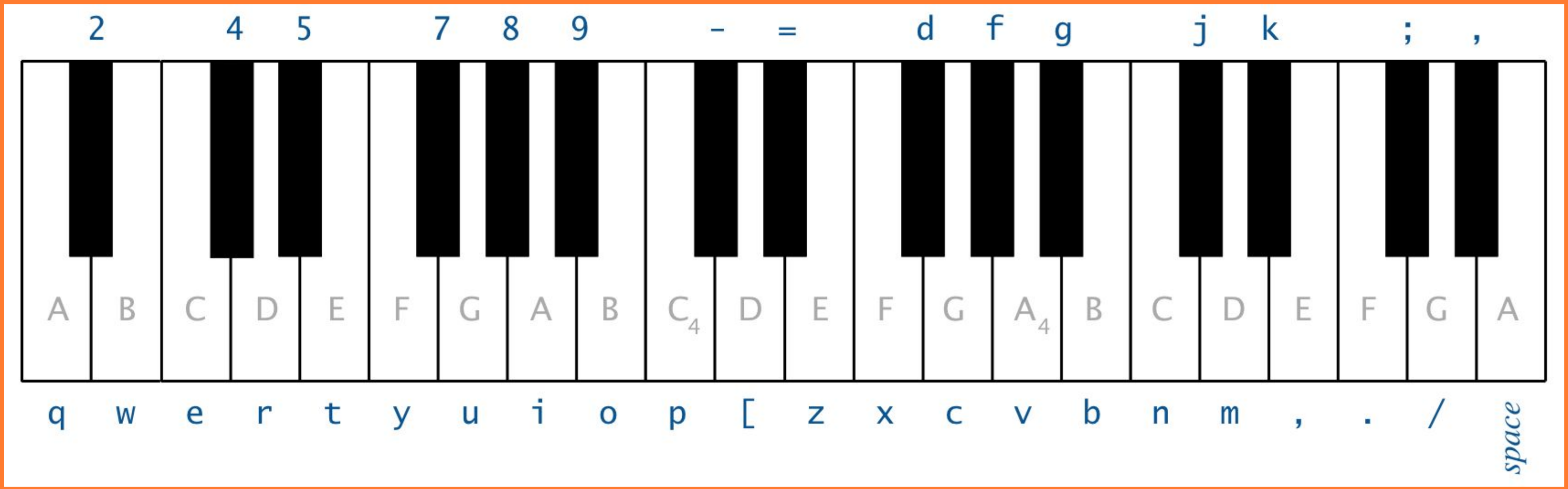
Last, handle the superposition correctly.

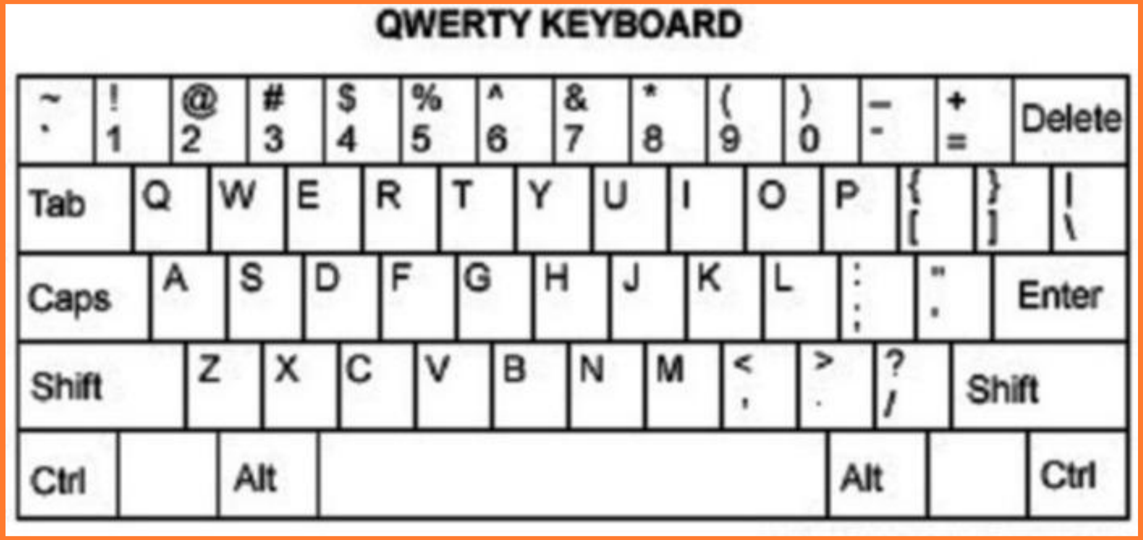


```
// compute the superposition of the samples
double sample = stringA.sample() + stringC.sample();

// send the result to standard audio
StdAudio.play(sample);

// advance the simulation of each guitar string by one step
stringA.tic();
stringC.tic();
```

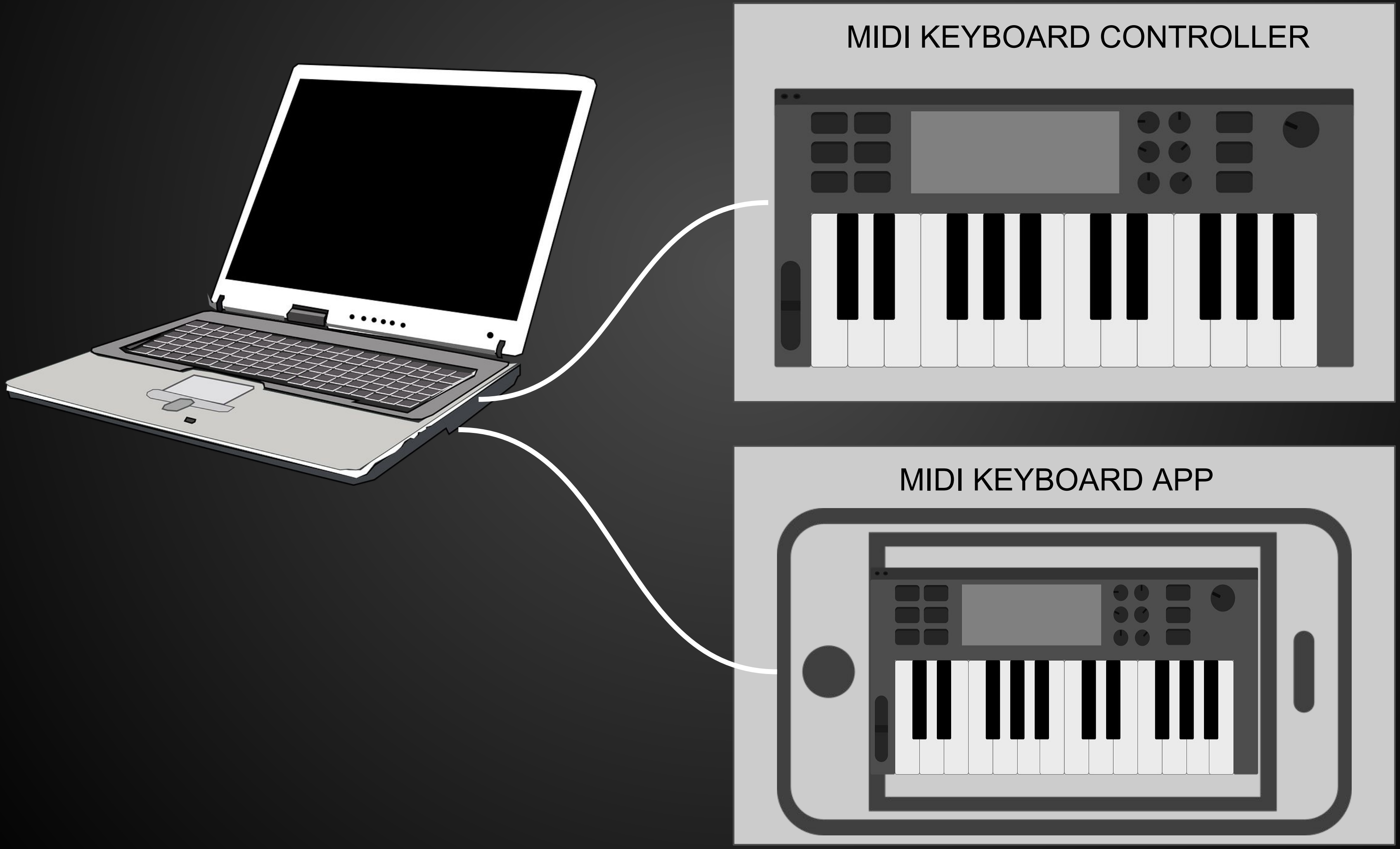
After we sampled each string, we call tic() on each GuitarString to get ready for next iteration

GuitarHero
User Interface


2 4 5 7 8 9 - = d f g j k ; ,
A B C D E F G A B C4 D E F G A4 B C D E F G A
q w e r t y u i o p [z x c v b n m , . / space


QWERTY KEYBOARD
~ ` ! 1 @ 2 # 3 $ 4 % 5 ^ 6 & 7 * 8 (9) 0 _ - + = Delete
Tab Q W E R T Y U I O P { [}] | \
Caps A S D F G H J K L : ; " ' Enter
Shift Z X C V B N M < , > . ? / Shift
Ctrl Alt Alt Ctrl

GuitarHero
MIDI - Checklist



GuitarHero