Talking to your Computer About Music: A Gibber Help Guide

Anthony T. Marasco - University of Texas Rio Grande Valley - 2023

Building Instruments

When performing with Gibber, we start by asking our computer to build some instruments for us to use. We can assign a personalized name to those newly-built instruments that we use to will refer to them by in the future. The act of building a specific instrument out of a group of potential instruments is called *initialization*. A personalized placeholder word, phrase, or letter that we assign the instrument to is called a *variable*.

To generate instruments in Gibber, we run functions that create a single member of an instrument family. These families are known as *Classes*, a term used to define a group of items built from a shared blueprint. One individual member of a Class is referred to as an *Object*, and by assigning that Object to a variable, we can create many instruments that share sonic traits and performance abilities, yet still be able to perform them individually and customize certain aspects about them.

Example

Initialize/Build two instruments and assign them to variables/personalized placeholders:

```
melodySynth = Synth()
clicky = Clave()
```

Once we assign an instrument to a variable, we'll refer to it by that name from now. Make sure to use self-referential words, letters, or phrases as variables so you can easily identify your instruments in a performance!

Tip

Some instruments come with presets that you can use to customize the way they sound. We can chose a preset for an instrument at the moment we build it by adding the name of the preset (encased in single quotes) inside of the parenthesis following the instrument's name. Here's an example that calls for a synthesizer to be built with the "acidBass" preset:

```
bassSynth = Synth('acidBass')
```

Performing Instruments

To perform our instruments, we'll need to write code that 1) address them by name and 2) calls one of their built-in method functions. To call a method, add a period (`.`) directly after the instrument's name and follow it with the method function name and a pair of parenthesis. Most methods require additional information meant to clarify their action—known as *arguments*— to be passed in through the parenthesis when they are called.

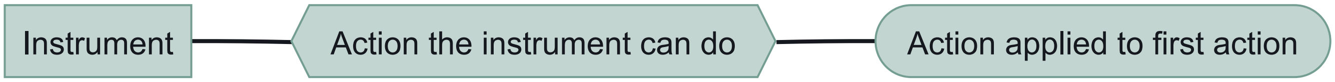
Example

Play a note with our bass synthesizer by calling its `.note()` method. Specify that the 4th note in the current scale should be played by passing in a number 4 inside of the parenthesis:

```
bassSynth.note(4)
```

Chaining Methods

It's possible for method functions to have their own set of actions to perform, and therefor, their own method functions. The diagram below displays this concept and how we can think about the order of operations neccesary to making this happen in our program: the instrument is addressed by name first, then we select an action for that instrument to perform, and finally, we apply a secondary action to modify how the first action is carried out.



Think outside the code

Let's apply this concept to an everyday scenario to better understand how it works. Imagine that you want to use your phone to generate and display a map of your surroundings and highlight any nearby locations that you've previously marked with the label "Favorite Spots." The first method called would ask your phone to use its GPS service to create and display a map. The second method would ask the map creating function to search through all of your tagged locations and highlight any that you marked as being your favorite.

Each method is a feature of the method or object address or called before it: the phone has the ability to build a map, but only the map has an ability to highlight specific locations on itself. Any example line of code in a program designed to carry out this task might look like this:

```
my_phone.makeMap.highlightLocationsByLabel("Favorite Spots")
```

In Gibber, we can use chained methods to repeat an instrument's actions repeatedly in a rhythmic sequence, allowing us to build looping melody lines, chord patterns, and percussion grooves.

Sequencing Patterns

We can generate looping patterns by using the `.seq()` method that is built into most performance-based methods of our instruments. `.seq()` requires two arguments to be passed in that answer the following questions:

1. What is the pattern to play? (an ordered list of numbers that represent scale degrees/notes in a melody or progressive volumes of percussive "hits")
2. How long does the sequence stay on the current value in the pattern before moving on to the next value?

Tip

Gibber handles time in a sequence by using whole numbers and fractions to represent **meaures** or **divisions of a measure**. The table below is a helpful collection of common timing values to use in sequences:

Value	Musical Time Value
`1` (or multiples - `2`, `4`, `8`, etc.)	entire measure(s)
`1/2`	half of a measure /half note
`1/4`	quarter of a measure/quarter note
`1/8`	eight of a measure/eight note
`1/16`	sixteenth or a measure/sixteenth note
`1/32`	thirty-secondth of a measure/thrity-secondth note

Inside of the parenthesis of `.seq()`, we can create lists of pitches and time values to define the notes of our pattern and their matching durations. Known as an *array*, these lists will represent the musical information the sequencer method needs to know in order to generate looping melodies, drum patterns, bass lines, and chord progressions. Arrays will always be encased inside of a pair of square brackets and each item in the array will be seperated by a comma.

Example

Pass in an array of scale degree values and an array of corresponding time values to create a looping sequence of six notes played by our melody synthesizer. Each note in the sequence has its own duration, found in the matching position of the second array (e.g scale degree `4` (the first value in the first array) is the first note played in the sequence, and it has a time duration of `1/4`, (the first value in second array)):

```
melodySynth.note.seq([4,3,0,6,5,3],[1/4,1/8,1/8,1/4,1/8,1/8])
```

Apply the same technique to create a pattern of clave hits played at varying volumes and durations:

```
clicky.trigger.seq([0.75, 1, 0.5, 1,0.25,0.25], [1/4,1/16,1/8,1/16,1/4,1/4])
```

Pattern Transformations

We can tranform the pitches and time durations in a pattern as we perform. To do this, we need to assign our pattern information to a variable when we create it. From that point on, we can address our pattern in the same way we address our instruments: first by their name, then by the method function or property we'd like to run or modify using our Dot Notation (`.`) format.

Some common transformations that we can apply to pattern data include:

- `transpose()` = transposes the pitches in our pattern by shifting them to higher or lower scale degrees. Pass in a single argument to specify the number of scale degrees all pitches in your pattern should be shifted by. Use positive numbers to move pitches up the scale and negative numbers to move pitches down the scale.
- `rotate()` = changes the order of pitches or time durations in your pattern by shifting them forward or backward through the array/list. Pass in a single argument to specify the number of spots all pitches/durations in your pattern should be shifted. Use positive numbers to move items forward in order and negative numbers to move items backward.

⚠ Warning

Pitches/durations never disappear from a sequence when they are rotated. The item at the end of the array will be rotated around to become the first item in the sequence if rotating forward, or the last item of the sequence if rotating backwards.

Example

Build a sequence and assign the pattern of pitches to a unique variable:

```
melodySynth.note.seq(myPattern = [0,2,_,6], 1/8)
```

Address the pattern by name to run its `.rotate()` method. Rotate the order of pitches forward in the sequence by two positions:

```
myPattern.rotate(2)
//new pattern order will now be [_,6,0,2]
```

Transpose the pitches in the pattern up by three scale degrees:

```
myPattern.transpose(3)
//scale degrees in pattern will now be [_,9,3,5]
```

The `.rotate()` and `.transpose()` methods can be sequenced using their internal `.seq()` method. This allows us to build repeating patterns of pitch transposition that repeat through a performance by passing in an array of rotation and transposition instructions.

Example

Create and start a sequence of pitch transpositions that occurs every four measures for a twelve measure phrase. The first value in the array of transposition instructions will cause the pattern to not transpose (using `0` as an argument to indicate no pitch change) for the first four measures. The second group of four measures will see the pitches transpose up by 4 scale degrees, and the third group of four measure will bring the pitches back down by four scale degrees to their original values:

```
bassSynth.note.seq(myPattern2 = [4,0], 1/2)
myPattern2.transpose.seq([0,4,-4], 4)
```

Audio Effects and Busses

By default, instruments in Gibber send their audio signal directly to the primary output. We can add audio effects into the signal chain by first building one or more effects and connecting our instruments to their inputs.

Gibber has a handful of audio effects to choose from, all of which have properties that can be modified when the effect is built or after the fact during a performance. To build an effect, create a new variable, then assign the result of a function that creates that effect to the variable. To add the effect into the signal chain between your instrument and the primary output, we first call the instrument's .fx() method, and then call that method's internal .add()function. In the parenthesis of the .add() method function, we need to add the variable name(s) of the effect(s) that we want our instrument's output signal to pass through.

Tip

Effects don't need to be assigned to a variable name in order to work, but it is reccomended practice if you want to easily address that effect by name and update it's properties during a perofrmance. Effect also don't need to be built in a seperate line of code before our instruments connect to them. It's possible to consolidate both steps in a single line of code. Read through the Example block below to see multiple approaches to achieving the same goal of creating and connecting to audio effects.

We can also modify the properties of an effect as soon as we build it, allowing us to set up the ideal amount and intensity of audio processing for our from the start. The example block below demonstrates the many different approaches to building, modifying, and connecting to audio effects in a performance.

Example

Create two audio effects. Assign them to variables in order to address them directly in the future. Add both effects in a chain to our melody synth:

```
delay = Delay()
verb =  Reverb()

melody_synth.fx.add(delay, verb)
//melody_synth's audio output will now be passed into the delay first, then into the reverb
```

Modify the delay's `time` property and reverb's `roomSize` property during the performance:

```
delay.time = 1/16
verb.roomSize = 0.48
```

Build a third effect for a kick drum and customize its properties at the same time. Notice the use of `{ }` brackets to encase the property definitions and `:` to seperate the property names from their new values:

```
crusher = BitCrusher({bitDepth: 0.3, sampleRate: 0.7})
//creates a new object from the BitCrusher class and sets its two properties to custom values

kick.fx.add(crusher)
```

Build a fourth effect for our bass synth. Do not assign it to a variable and build the effect inside of the same line of code we use to connect our bass and effect together:

```
bass.fx.add(Flanger())
```