A Beginner's Introduction to SuperCollider: Creating Your First “Instrument”

Table of Contents

Introduction:

In this tutorial, I am going to introduce you to SuperCollider, and teach you how to build a synthesized instrument. SuperCollider is an amazing and rich tool, that allows you to create music and new sounds with a programming language. Unfortunately, like many powerful tools, SuperCollider has an immensely steep learning curve. To really understand the depth and power of SuperCollider takes practice and dedication, just like any other programming language.

But don't worry! Instead of walking you through the 'Hello World!' of SuperCollider, I am going to show you how to build something a little more complex, and give you the opportunity to play around with the tools as a way of getting to know them. As a result, I am not going to walk you through a lot of syntax rules or show you the ins and outs of writing code. Some experience with coding and reading programming documentation is necessary for this tutorial.

If you would like a more thorough and methodical introduction to SuperCollider, I would point you in the direction of two sources that I have learned a great deal from:

1) A pdf (roughly 100 pages long) entitled “A Gentle Introduction to SuperCollider” by Bruno Ruvario, which can be found here.

2) The Youtube channel of one Eli Fieldsetel. Eli has a large number of lectures, videos, and performances utilizing SuperCollider. His knowledge is very in depth, and you can learn a lot from his videos, found here.

What is SuperCollider?

SuperCollider, as the documentation/download page tells us (here), “is a platform for audio synthesis and algorithmic composition.” SuperCollider has three components, all of which come neatly packaged together. The three components are 1) the programming language itself, called “sclang”, 2) a real-time audio server which allows you to create sound on the fly, called “scsynth”, and 3) an IDE for writing your code and interacting with the server, called “scide”.

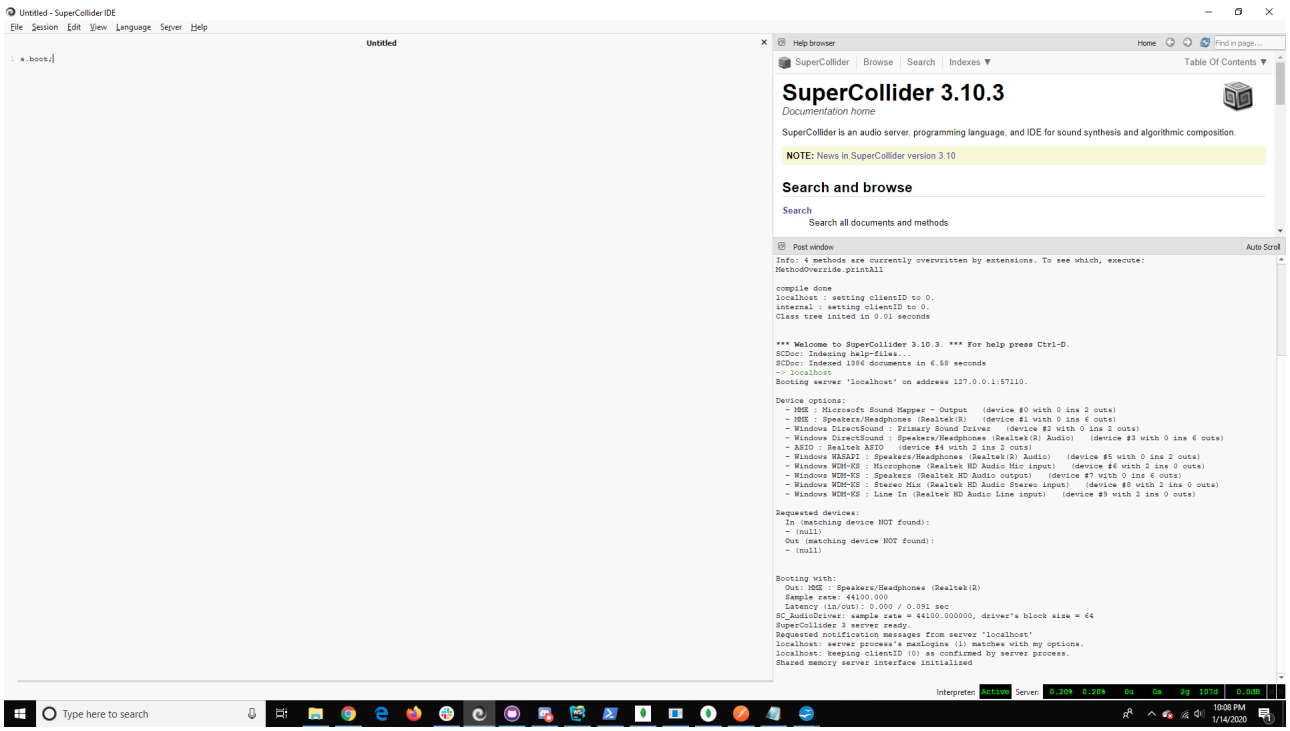
Windows Installation:

1) From the documentation/download page linked to above, select the version appropriate for your computer, and download the .exe file.
2) Open it to launch the Windows Setup Wizard

3) Click next, agree to the license, and select a folder to install SuperCollider.
4) Find SuperCollider on your computer and open it.

Mac Installation:

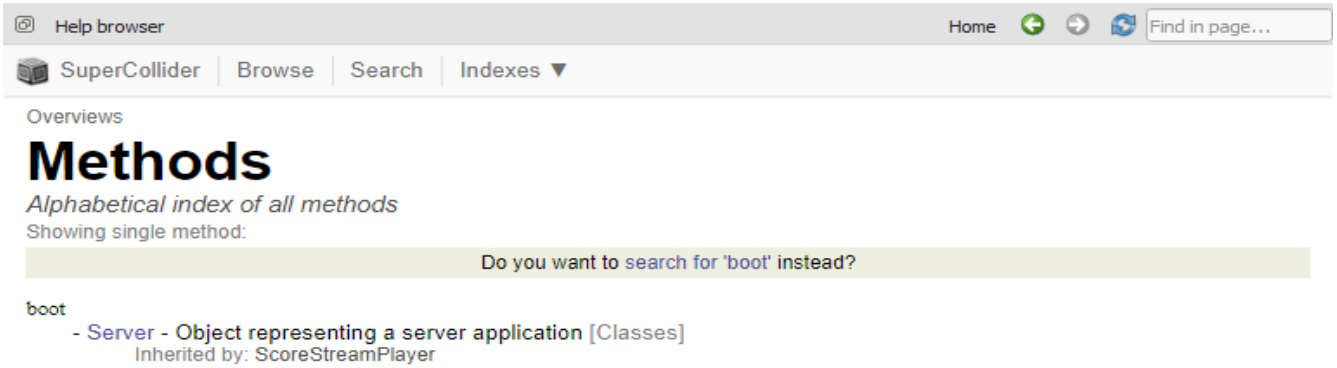
1) From the documentation/download page linked to above, select the version appropriate for your computer, and download the .zip file.
2) Double click the downloaded zip file to unzip the file.
3) Open SuperCollider from the folder that was unzipped.



In the screenshot above you can see that the SuperCollider IDE has three sections. On the left, is the proper IDE. This is where you write your code. You can see that I have one line of code written:

```
s.boot;
```

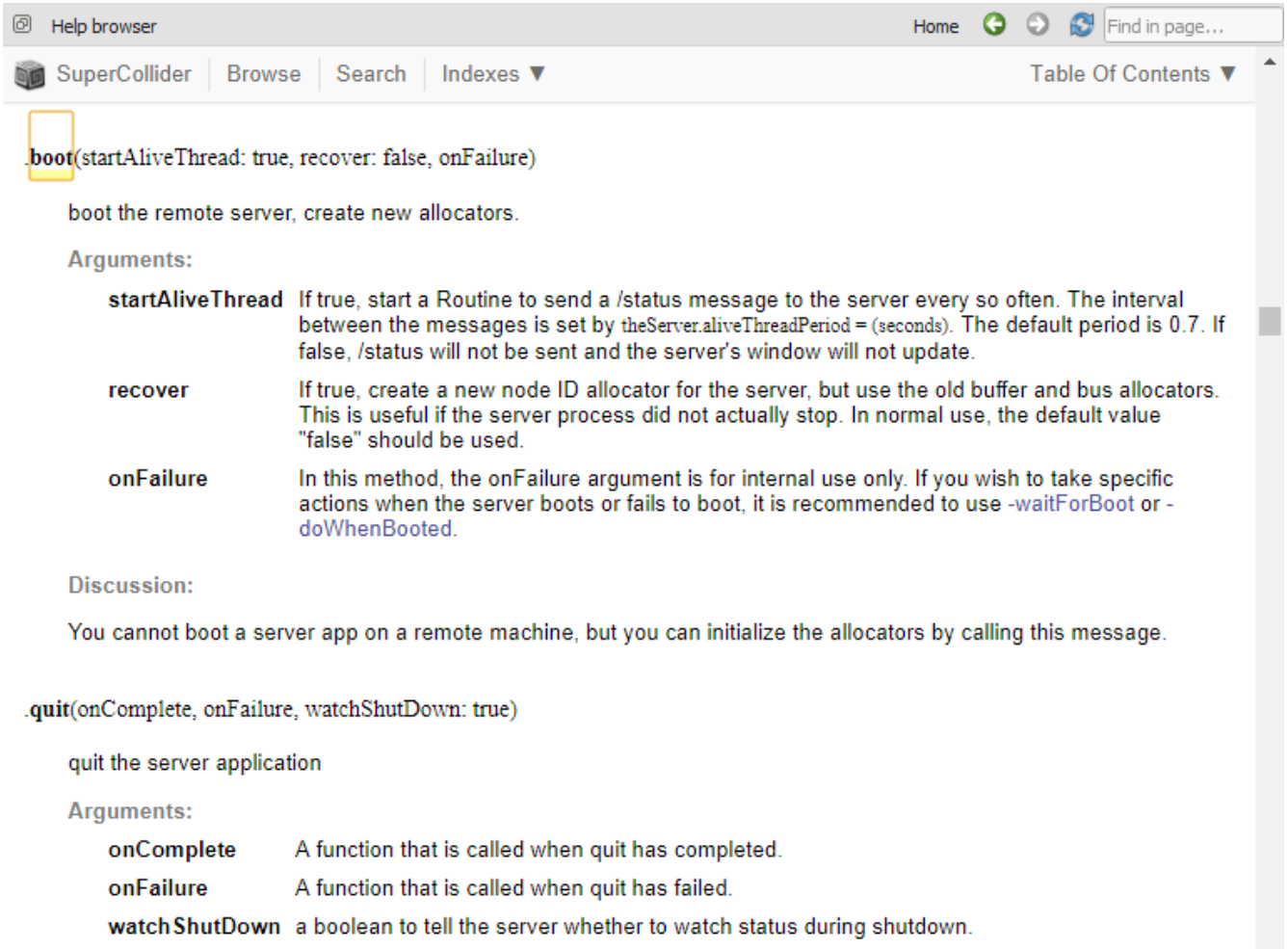
If you type this line of code, select anywhere on the line with your mouse cursor, and press Shift+Enter. When you do, you will see text pop up in the bottom half of the right side of the screen. This section, called the post window, displays all the non-audio output from the SuperCollider server. Here you will find error messages and any other information the server outputs. Lastly, in the top half of the right side of the screen you will see the help browser. Highlight the word 'boot' and press Ctrl+D. When you do this, you will see the following result in the help browser:



As you can see in the screen above, 'boot' is a method of the Server object, which SuperCollider automatically stores in the variable 's'. When we pressed Shift+Enter on the command, we were calling the 'boot' method on the Server object, which is why the post window returns this message (in addition to other information):

```
-> localhost
Booting server 'localhost' on address 127.0.0.1:57110.
```

Go back to the help window on the top half of the right side of your screen, and click on the word 'Server'. The page below will load:



Below the method 'boot' we see another method called 'quit'. Try typing

```
s.quit;
```

Since the 's' variable is the Server object, this code will call 'quit' on the Server which shuts it down. If you press Shift+Enter with your mouse cursor anywhere on that line, the post window will output the following:

```
server 'localhost' disconnected shared memory interface
'/quit' message sent to server 'localhost'.
-> localhost
Server 'localhost' exited with exit code 0.
```

Great! Now we know what SuperCollider is, how to install it, how to execute commands (Shift+Enter while the mouse cursor is on the line) as well as how to boot and quit the server, which needs to be running if we want to produce and hear sound.

And, perhaps most importantly, we now know how to look up things in the documentation (Ctr+D while highlighting a word in the IDE). At any point in this tutorial, you can highlight a word from the code I've given you and look it up, or use the search and browse areas of the help browser. And never forget, the documentation is your friend!

Some Basic Concepts, Classes, and Code Snippets

In this tutorial we are going to create an instrument I call BendyBells, because they sound like bells or chimes, and the pitch bends up and down with each stroke. Before we start constructing the instrument, we need to talk about some SuperCollider fundamentals, using some code snippets and examples along the way. And remember, at any point feel free to explore the docs and start experimenting.

SuperCollider comes with a whole bunch of pre-defined classes and methods that make creating noise very easy. Before I show you how to create your very own Bendy-Bells instrument, we're going to talk about a few of the classes that we will be using.

One class we will work with a lot is called a UGen, or unit generator. (Hint! Hint! Use the docs to search UGen!) As the documentation tells us, a UGen is an abstract class used to generate sound. (An abstract class, in case you're unfamiliar with this language, is a class that we don't instantiate, but which provides the blueprint for sub-classes, which we *do* instantiate.) Two UGens that we will be working with are called SinOsc and Env, short for Sine Wave Oscillator and Envelope, respectively.

If you have the server running, type the following code and execute it (Shift+Enter while the cursor is on the line), but before you do, make sure your volume isn't up to high, and remember that you can stop sound **anytime** by pressing Ctrl+. (That's Control + Period, just to be safe).

```
(
{SinOsc.ar(freq: 440)}.play;
)
```

When executed, you should hear a sine wave at a 440hz frequency. Let's analyze this expression to get a feel for some basic syntax.

The outermost part of the expression is a set of parentheses '( )' In SuperCollider we can execute large blocks of code by putting all of it in parentheses, double-clicking on the line with either the opening or closing parethesis, and pressing Shift+Enter while the whole thing is highlighted.

Inside the parentheses we find this: { }.play; Like many programming languages, the semicolon at the end marks the end of a statement. The curly braces indicate a function that is being defined, and .play is a method that we can call on various objects in SuperCollider. So, we have a function that is being played, but what is inside that function?

It's a UGen: 'SinOsc.ar(440)' Here we have a Sine wave oscillator unit generator. On this unit generator the '.ar' method is being called, which stands for Audio Rate. What this means is that the Sine Oscillator will be generating sound we want to *hear* and not merely sound that is used for the processing of other signals, which we will get to next. The only argument being passed into the '.ar' method is frequency, which you can change to see that changing the argument changes the pitch being played.

The other UGen I mentioned is an Envelope. If you are familiar with audio science, mixing boards, or recording technology you may already know what an envelope is, but if not, don't worry. Put simply, an envelop is the shape that a sound wave has. Usually, when we talk about envelopes we are talking about the shaping amplitude or volume level of a sound. The example above will never stop playing until you press Ctrl+. or the server shuts down. This is because the sound being produced doesn't have an envelope. Let's create an envelope to learn more about it.

```
env = Env.new(levels: [0, 1, 0], times: [2, 5]);
```

Here we have declared a variable, 'env', which is defined as a new Env, the envelope class. When we create a new envelope the arguments 'levels' and 'times' are passed in. 'Levels' is an array of values that the sound wave being altered will pass through. Since this envelope will be used to control the amplitude/volume of the pitch, thee pitch will start at 0% volume, go to 100% volume and then proceed back to 0% volume. The values in the 'times' array indicate how long it will take (in seconds) to move from each value in the 'levels' array to the next value in the 'levels' array. So this envelope will start at 0% volume, then in 2 seconds it will reach 100% volume, then in another 5 seconds, the volume will drop back down to 0% volume.

We want to apply this envelope to the simple sine wave we created above, but we can't do that yet. The envelope we just created isn't a UGen, which means it can't generate or control sound. What we have created above are instructions that we can give to a class called EnvGen, which will produce an Envelope Generator, which is a type of UGen.

```
env = Env.new(levels: [0, 1, 0], times: [2, 5]);

amplitudeEnvGen = EnvGen.kr(envelope: env, doneAction: 2);
```

We now have a variable called 'amplitudeEnvGen' which holds an envelope unit generator used to control other sound waves--this what the method .kr does (kontrol rate), as opposed to the .ar method we saw earlier (audio rate). To create the envelope unit generator we passed in two arguments. 1) the

Envelope instructions that we defined above, and 2) an argument called doneAction, which tells the server what to do with the unitGenerator after it's work is done. Should it sit on the server waiting for further instructions? No, in this case we want the unit generator to destroy itself, so we passed in the correct code. Go to the documentation and look up EnvGen, where you can learn more about the doneAction parameter and other options.

Now let's see how it sounds when applied to the sine wave from earlier:

```
(
{
var env, amplitudeEnvGen;
env = Env.new(levels: [0, 1, 0], times: [2, 5]);
amplitudeEnvGen = EnvGen.kr(envelope: env, doneAction: 2);
SinOsc.ar(freq: 440, mul: amplitudeEnvGen)
}.play;
)
```

Remember that we put all our code inside of parentheses for easy execution. And more importantly, all of our code must be within the function-defining curly braces. Also important to note is that variables must be defined explicitly, and that this should be done at the very start of our function.

Before we move onto our BendyBells instrument, we need to introduce two more classes: SynthDef and Synth. SythDef contains the blueprints and instructions for our synthesized instrument. We can instantiate this instrument by loading a Synth. Let's see it in practice.

Let's Build an Instrument!

At the beginning of this tutorial, I said that SuperCollider has a very steep learning curve, and I wasn't kidding. The language is so powerful, and there is so much to explore that I can't teach you very much in a short guide like this. Hopefully, after understanding the basics you can explore the documentation and start creating your own sounds with SuperCollider. Without further ado, here is the code for the Bendy-Bells instrument:

```
s.boot;

s.quit;

(
SynthDef(\BendyBells, {
	var sig, tone1, tone2, ampEnv, ampEnvGen, freqEnv1, freqEnvGen1, freqEnv2, freqEnvGen2,
rand1, rand2;

	rand1 = Rand(50, 200);
	rand2 = Rand(400, 1600);

	ampEnv = Env.new(levels: [0, 1, 0], times: [0.3, 4], curve: [0, -4]);
	ampEnvGen = EnvGen.kr(envelope: ampEnv, doneAction: 2);

	freqEnv1 = Env.new(levels: [rand1, rand1 + 10, rand1 - 25], times: [0.5, 4]);
```

```
        freqEnvGen1 = EnvGen.kr(envelope: freqEnv1, doneAction: 2);

        freqEnv2 = Env.new(levels: [rand2, rand2 + 10, rand2 - 25], times: [0.25, 4]);
        freqEnvGen2 = EnvGen.kr(envelope: freqEnv2, doneAction: 2);

        sig = SinOsc.ar(freq: rand1 + freqEnvGen1, mul: 0.6);
        sig = sig + SinOsc.ar(freq: rand2 + freqEnvGen2, mul: 0.1);
        sig = BPF.ar(in: sig, freq: 1000, rq: 0.1) * 10;
        sig = sig * ampEnvGen;

        Out.ar([0, 1], sig);
}).add;
)

x = Synth(\BendyBells);
```

Let's walk through this code line by line:

```
s.boot;

s.quit;
```

As before these two commands are included so you can start and stop the SuperCollider server.

```
(
SynthDef(\BendyBells, {
```

Here we have our opening paranthesis and the declaration of our SynthDef. The SynthDef here takes two arguments: 1) the name of the SynthDef we are defining. In this case it is BendyBells, and because it is preceded by a back-slash it is a “symbol”, which is a unique identifying name. After the name comes 2) the the function that defines what the instrument will do when played. At the end of the line here we see the opening curly brace that begins the function.

```
        var sig, tone1, tone2, ampEnv, ampEnvGen, freqEnv1, freqEnvGen1, freqEnv2, freqEnvGen2,
rand1, rand2;
```

In SuperCollider it is best practice to declare all of your variables at the top of a function. You do this by starting the line with the keyword 'var' and then listing all your variables, separated by commas, and ending the line with a semi-colon.

```
        rand1 = Rand(50, 200);
        rand2 = Rand(400, 1600);
```

'rand1' and 'rand2' are two variables. Here we define them using the Rand class, which generates a random number with the lower and upper limits being passed in as arguments. Rand generates a new random number every time the instantiated Synth is called. This means that if you “play” the instrument three times in a row, you will get three different pairs of pitches, adding some nice randomness to the instrument. (If you'd like random pitches that are determined once at the time of instantiation, and never change, check out 'rrand' in the docs.)

The way these random number generators are set up, we have one high tone and one low tone. But this can be altered, and it would be easy to add a third or fourth tone to the instrument.

```
        ampEnv = Env.new(levels: [0, 1, 0], times: [0.3, 4], curve: [0, -4]);
        ampEnvGen = EnvGen.kr(envelope: ampEnv, doneAction: 2);
```

Here we have the definition of the envelope to control the amplitude and the Envelope Generator that produces a UGen needed to actually modify the amplitude of the sound. As above, when creating a new envelope we need to pass in 1) the volume levels the tone will pass through, 2) the time (in seconds) to move from each volume level to the next, and 3) we can specify the shape of the curve used to move from one volume level to the next.

I'll let you look at the docs (search “Env” and look at the “curve” section) to see how this works. But as a tip, type the following into the IDE (outside of any function defining curly braces) and execute it:

```
Env.new(levels: [0, 1, 0], times: [0.3, 4], curve: [0, -4]).plot;
```

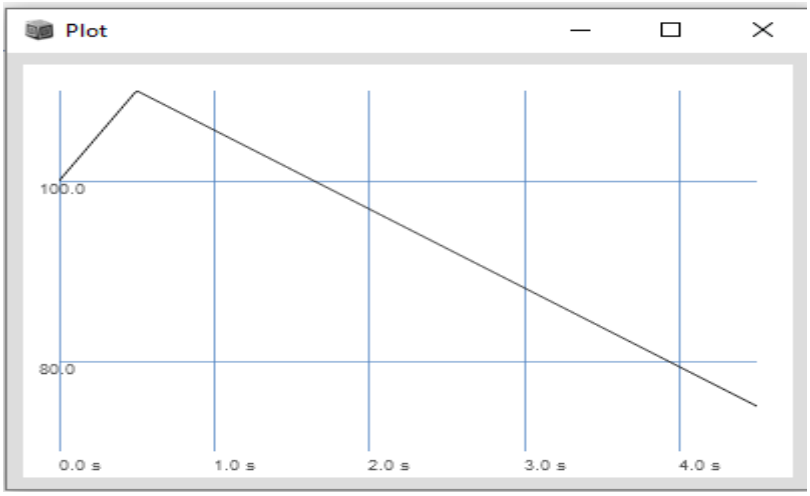
Just remember to remove the variable name and equals sign, and add '.plot' after the closing parenthesis.

```
        freqEnv1 = Env.new(levels: [rand1, rand1 + 10, rand1 - 25], times: [0.5, 4]);
        freqEnvGen1 = EnvGen.kr(envelope: freqEnv1, doneAction: 2);

        freqEnv2 = Env.new(levels: [rand2, rand2 + 10, rand2 - 25], times: [0.25, 4]);
        freqEnvGen2 = EnvGen.kr(envelope: freqEnv2, doneAction: 2);
```

Here we have two more envelopes and envelope generators. These envelope UGens will be used to control the pitch, unlike the previous envelop which modified the amplitude of the tone. Unless otherwise specified, pitch in SuperCollider is measure in herz. Let's imagine that the 'rand1' holds the number 100. Our first envelope would be equivalent to the code below. I've also added the plotted graph so we can see what the wave will look like, and consequently, how it will sound.

```
Env.new(levels: [100, 110, 75], times: [0.5, 4]).plot;
```



Because both tones have the same shape (up, then down using the same 'times' array), both tones move in parallel and sound like they have the same source. Try changing the 'levels' array or 'times' array arguments on one or both of the envelopes and see how it sounds. You can also add a 'curve' array argument like we did on the amplitude envelope above. Try experimenting and see what you can come up with!

```
        sig = SinOsc.ar(freq: rand1 + freqEnvGen1, mul: 0.6);
        sig = sig + SinOsc.ar(freq: rand2 + freqEnvGen2, mul: 0.1);
        sig = BPF.ar(in: sig, freq: 1000, rq: 0.1) * 8;
        sig = sig * ampEnvGen;
```

Here we construct the signal that will be output to the speakers. We start by defining the signal ('sig') as a Sine Wave Oscillator. For it's frequency we use the variable 'rand1', which is the low tone. To apply the filter to the frequency, we add the frequency to the EnvGen we wrote earlier. As always, try experimenting with multiplication or division so see what else happens, just be sure to have to your speakers at a low to medium level. After the frequency argument comes the amplitude, which is set to 0.6 for tone 1 and 0.1 for tone 2. This makes the lower tone louder than than the higher. On the second line, we take the signal we created on the previous line and simply add it to a second Sine Wave Oscillator, constructed just like the first, but with a different random number and it's corresponding EnvGen.

On the third line we have a brand new class, BPF, which stands for “band pass filter”. Check it out in the docs, and be sure to search “filter” to see all the other filter classes built in to SuperCollider. For this particular filter, the important thing to know is that we pass in our signal to the 'in' argument, along with a cut-off frequency ('freq') and the reciprocal of the cut-off frequency ('rq'). Finally, the whole signal is multiplied by 8 to compensate for the volume lost in the filtering. Try experimenting with all of these values.

Last, we take our two-toned, filtered signal and multiply it by the amplitude EnvGen to control the amplitude and volume of the sound.

```
        Out.ar([0, 1], sig);
}).add;
)
```

The last line inside the SynthDef function calls the Out class. This class takes two arguments: 1) it takes an array of busses, which tell your computer where to send the signal. In almost every case, '0' is your left headphone speaker or monitor and '1' is your right headphone speaker or monitor. If you have 7.1 surround sound, you could use an array with values 0-7 to send signals to different speakers. 2) is the signal we want sent out, which is the signal we just constructed.

After that we see the closing function-enclosing curly brace, as well as the closing SynthDef parenthesis, after which is called '.add' on the whole function. And finally, the last closing parenthesis that we use in SuperCollider to make executing large code blocks easy.

What does ".add" do on the SynthDef? It adds the definition to the server so that we can instantiate it. Now SuperCollider knows what we are referring to when we tell it to play BendyBells, which is what we do in the very last line of code by creating a Synth, using the SynthDef from above:

```
x = Synth(\BendyBells);
```

After the SynthDef definition had been added to the server, execute this line of code to play it. Note that you don't have to wait for a previous instance to stop to start a new one.

Thanks for following along with my beginner's introduction to SuperCollider. I hope you've learned something and have developed a taste for computer composition. The possibilities are endless!