SuperCollider Tutorial

Chapter 3

By Celeste Hutchins

2005

www.celesteh.com

Arrays

You may recall that an **Array** is an indexed list. We can create an array by putting zero or more comma delineated expressions between square brackets. For example, these are some arrays:

```
[1]
[x, y, z]
["in", "the", "house"]
[]
```

There is often more than one way to do something in SuperCollider, as in life. We can also create an Array by using a constructor. The constructor takes one argument, the size of the array.

```
a = Array.new(size);
```

Arrays can hold any kind of object, including numbers, variables, strings or nothing at all. They can even mix types:

```
[3, "French hens", 2 "turtledoves", 1 "partridge"]
```

Arrays can even hold other arrays: `[[1, 2, 3], [4, 5, 6]]`

You can also put an **expression** in an array. The interpreter evaluates it and stores the results. So `[ 3 - 1, 4 + 3, 2 * 6]` is also a valid array, stored as `[2, 7, 12]`. `[x.foo(2), x.bar(3)]` is also an array. It passes those messages to the objects and puts the result into the array. Because commas have extremely low **precedence,** they get evaluated last.

A variable can be part of an expression that is put into an array:

```
(
  var foo, bar;
  ...
  [foo + 1 , bar];
)
```

In that last case, it holds the value of the expressions (including the variables) at the time they became part of the array. For example:

```
(
  var foo, arr;
  foo = 2;
  arr = [foo];
  arr.postln;
  foo. postln;

  " ".postln;

  foo = foo + 1;
  arr.postln;
  foo. postln;
 )
```

Outputs:

```
[ 2 ]
2

[ 2 ]
3
3
```

This is *almost* just what we expected. From whence did the second 3 at the bottom come? SuperCollider's interpreter prints out a return value for every code block that it runs. `bar` was the last object in the code block, so `bar` gets returned. `bar`'s value is 3.

The variables can go on to change, but the array holds the value that was put in, like a snapshot of when the expression was evaluated.

What if we declare a five element array and want to add something to the array? We use the Array.add message.

```
(
	var arr, new_arr;
	arr = ["Mary", "had", "a", "little"];
	new_arr = arr.add("lamb");
	arr.postln;
	new_arr.postln;
)
```

Outputs:

```
[ Mary, had, a, little ]
[ Mary, had, a, little, lamb ]
```

Arrays cannot grow in size once they've been created. So, as stated in the helpfile, “the 'add' method may or may not return the same Array object. It will add the argument to the receiver if there is space, otherwise it returns a new Array object with the argument added.” Therefore, when you add something to an array, you need to assign the result to a variable. `arr` doesn’t change in the example because it is already full.

There are other messages you can send to Arrays, which are detailed in the Array helpfile and the helpfile for its superclass ArrayedCollection. Two of my favorite are **scramble** and **choose.**

The helpfile says that scramble "returns a new Array whose elements have been scrambled. The receiver is unchanged."

The results of scramble are different each time you run it, because it scrambles in random order. When it says, "the receiver is unchanged", it means that if we want to save the scrambled Array, we have to assign that output to a new variable. The **receiver** is the object that **receives** the message "scramble." In this example, the receiver is `arr`, which contains `[1, 2, 3, 4, 5, 6]`.

```
(
  var arr, scrambled;
  arr = [1, 2, 3, 4, 5, 6];
  scrambled = arr.scramble;
  arr.postln;
  scrambled.postln;
)
```

Outputs:

```
[ 1, 2, 3, 4, 5, 6 ]
[ 4, 1, 2, 3, 6, 5 ]
```

Of course, the second array is different every time. And `arr` is unchanged.

Choose is similar. It picks a random element of the Array and outputs it. The receiver is unchanged.

```
[1, 2, 3].choose.postln;
```

Outputs:

```
2
```

Or 1 or 3, potentially changing every time you run it.

Arrays are lists, but they are not merely lists. They are **index**ed lists. You can ask what the value of an item in an array is **at** a particular position.

```
(
	var arr;
	arr = ["Mary", "had", "a", "little"];

	arr.at(1).postln;
	arr.at(3).postln;
)
```

Outputs:

```
had
little
```

Array indexes in SuperCollider start with 0. In the above example, `arr.at(0)` is `"Mary"`.

Arrays also understand the message **do**, but treat it a bit differently than an Integer does.

```
(
	[3, 4, 5].do ({ arg item, index;
```

```
            ("index: " ++ index ++ " item: " ++ item).postln;
        });
)
```

Outputs:

```
index: 0 item: 3
index: 1 item: 4
index: 2 item: 5
```

We call our arguments in this example `item` and `index`, but they can have any name we want. It doesn't matter what we call them. The first one always gets the value of the array item that the loop is on and the second one always gets the index.

The ++ means **concatenate**, by the way. You use it to add something to the end of a string. For example:

`"foo " ++ 3` returns the string `"foo 3"`

In the last chapter, I mentioned Nicole, the ex-grad student working at a SuperCollider startup. Since then, she's gotten another assignment from her boss. She has to write a function that takes as arguments, an array of tuning ratios, a base frequency and a detuning amount. It has to figure out the final pitches by first multiplying the base frequency by the ratio and then adding the detuning amount to the result. Then it has to print them all out formatted like:

tuning ratio: <ratio>, pitch: <pitch>

After learning about Arrays, our hero does a bit of research on tuning ratios and comes up with an array of ratios that she will use to test her function. It looks

like: `[ 1/1, 3/2, 4/3, 9/8, 16/9, 5/4, 8/5 ]`

That is 1 divided by 1, 3 divided by 2, four divided by 3, etc. Now remember that precedence means that the interpreter evaluates things in a particular order. It looks at / before it looks at commas. So it seems a bunch of /'s and starts dividing. Then it looks at the commas and treats it as an array. The interpreter stores that array as:

```
[ 1, 1.5, 1.3333333333333, 1.125, 1.7777777777778, 1.25, 1.6 ]
```

Nicole's function cycle through the ratios, taking each one and multiplying it by the base frequency and adding the detuning amount. Remember that SuperCollider is like a cheap calculator and +, -, *, and / all have the same precedence. Math is evaluated from left to right. So `ratio * baseFreq + detune` is not equivalent to `detune + ratio * baseFreq`, like it would be in algebra. However, fortunately, she can use parenthesis like we would in algebra.

She could write out her **expression** as `(ratio * baseFreq) + detune` or `(detune + (ratio * baseFreq))` or in many other ways. Even though she could get the right answer without using parenthesis at all, it's good programming practice to use them.

Nicole has a formula and she has an Array. She just needs a way to step through it. Fortunately, she knows that the 'do' method also exists for Arrays.

She writes her code as follows:

```
(

    var func, arr;

    func = { arg ratio_arr, baseFreq = 440, detune = 10;
```

```
        var pitch;

        ratio_arr.do({ arg ratio, index;
            pitch =  (ratio * baseFreq) + detune;
            ("Ratio: " ++ ratio ++ " Pitch: " ++ pitch).postln;
        });
    };

    arr = [ 1/1, 3/2, 4/3, 9/8, 16/9, 5/4, 8/5];

    func.value(arr, 440, 10);
)
```

Which outputs:

```
Ratio: 1 Pitch: 450
Ratio: 1.5 Pitch: 670
Ratio: 1.3333333333333 Pitch: 596.66666666667
Ratio: 1.125 Pitch: 505
Ratio: 1.7777777777778 Pitch: 792.22222222222
Ratio: 1.25 Pitch: 560
Ratio: 1.6 Pitch: 714
```

Envelopes

The next logical assignment for Nicole is to play those pitches rather than print them out. In the last chapter, we wrote a function to play overtones, but it played them all at the same time. We need a way to tell notes to end and to go from note to note.

"Just like human beings, sounds are born, reach their prime and die; but

the life of a sound, from creation to evanescence last only a few seconds"
Yamaha GX-1 Guide

We control this process with a UGen called an **envelope**.

```
(

    var syn, sound;

    syn = SynthDef.new("example3", {arg out = 0, freq = 440, amp = 0.2,
                                dur = 1;
        var sin, env_gen, env;

        env = Env.triangle(dur, amp);
        env_gen = EnvGen.kr(env);
        sin = SinOsc.ar(freq, mul: env_gen);
        Out.ar(out, sin);
    });

    syn.load(s);
)

Synth.new("example3");
```

There are a few things that make that SynthDef different. One is that it uses variables. Variables are a good way to keep SynthDefs more readable. The easier it is to read, the easier it is to understand, to fix and to change.

The other obvious change is the addition of an envelope. Envelopes have two parts. One is **Env**. The Env class lets you describe what shape the envelope should have. In this case, we're using a **fixed duration** envelope shaped like a

triangle. When we use a fixed duration envelope, we know the length of the envelope when we create an instance of the Synth. There are other envelopes that we will not know the envelope of until we decide to end the note, for instance, in response to a key up on a keyboard.

`triangle` is a constructor. It creates a new instance of an Env. The arguments it takes are the duration and the maximum amplitude.

The other part of the envelope is the **EnvGen**. Since EnvGen is a class, `kr` must be a constructor. The two most common UGen constructors are `ar` and `kr`. You may recall that ar stands for “audio rate.” kr stands for “control rate.” Because the EnvGen is not actually creating sound, but is controlling the amplitude of a sound, we want the control rate. Control rate signals change less often than audio rate signals, and so using kr reduces the load on our computer and makes our SynthDef more efficient.

EnvGen.kr generates the envelope based on the specifications of its first argument, an instance of Env. So, with Env, we define an Envelope. We pass that definition to an Envelop Generator, which plays back the envelope that we defined. We use that envelope to control the amplitude of a SinOsc. We send the output of the SinOsc to an output bus. When the envelope is done, the amplitude of it has returned to zero, so the SinOsc is no longer audible.

With our previous examples, the Synth never ended. The way to stop sound was to stop execution with apple-. . With our new SynthDef, the sound goes on for exactly the length of dur and then ends with no way to restart it. After we create a Synth and play it, there's nothing that can be done with it.

Try running `Synth.new("example3");` several times. If you look at the localhost server window, you will see the number of UGens and Synths grow every time

you run Synth.new. The Avg and Peak CPU may also increase.

Every time we create a new instance of Synth, we use up more system resources. Eventually we will either run out of memory or the CPU will climb over 100%. We can clear the Synths by hitting apple-., but this still would limit the number of sounds we could play in a row without stopping execution.

We need an automatic way to remove our synths from the server and **deallocate** their resources. That means that some other Synths would be able to use the resources they were taking up.

Fortunately, EnvGens can both silence a Synth forever and lay it to rest so iti s deallocated. EnvGen takes an argument called “doneAction”. Adding a `doneAction` to your EnvGen would look like:

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

The EnvGen helpfile describes doneActions. You should look at the helpfile for EnvGen to look at what all the different values mean. 2 means "remove the synth and deallocate it."

Try changing adding a doneAction to the SynthDef and running `Synth.new("example3");` a few times. Notice that now the Synths don’t accumulate. Now, after a Synth stops making sound, it has ended its useful life and so it gets removed from the server because of the doneAction: 2. However,

note that the doneAction removes only instances of the Synth. The SynthDef persists and can be used as many times as we'd like.

Envelopes don't have to only control amplitude, but can control (almost) any aspect of any UGen. You can't set the bus on Out.ar with an Envelope, but you can control almost anything else.

Routines

Now that we know how to create synths that can stop playing, we just need a way to time their creation. A Routine is a highly versatile object that can be used in many ways. One way it can be used is to add timing to your programs.

```
(

     r  = Routine.new ({

          5.do({ arg index;
               index.postln;
               1.wait;
          });
     });

     r.play;
)
```

That example prints out:

```
0
1
2
```

```
3
4
```

With a once second pause between each number. Try running it.

The class SimpleNumber, which we remember is the superclass of Float and Integer, takes a message called `wait`. That message only works in the context of a routine. If we try `wait` outside of a Routine:

```
(

	f  = {

		5.do({ arg index;
			index.postln;
			1.wait;
		});
	};

	f.play;
)
```

We get an error:

```
• ERROR: yield was called outside of a Routine.
ERROR: Primitive '_RoutineYield' failed.
```

Routines are one powerful way to add timing to your program. Routine.new takes a function as an argument. It runs that function when we send it the **play** message. Hence, `r.play;` in the Routine example. When we play a Routine, it will pause for a duration of SimpleNumber.wait. If we code `5.wait`, it will pause

for five seconds. If we have `0.5.wait`, it will pause for half a second.

Let's try to take Nicole's tuning assignment and have it play sounds instead of just print data. We can't pass arguments to a Routine, but fortunately, because of **scope** and functions **returning** things, there is a clever work around:

```
(

  var func, rout;

  func = { arg ratio_arr, baseFreq = 440, detune = 10;

     Routine.new({

          ratio_arr.postln;
          baseFreq.postln;
          detune.postln;
     });

  };

  rout = func.value([1/1, 3/2, 2/1]);
  rout.play;
)
```

First, we declare a function that takes three arguments. Then, inside the function, we declare a Routine. Because the Routine is inside the function's code block, the Routine is within the scope of the arguments. And because the Routine is the last (and only, aside from the args) thing inside the function, it gets returned when the function gets evaluated. So `rout` gets the Routine that is returned by the function. Then we call `rout.play;` to run the Routine.

Remember that functions **return** their last value. And remember that **scope** means that inner code blocks can see variables from outer code blocks, but not vice versa. So the Routine can see the args. And there's nothing after the Routine in the function, so it's the last value, so it gets returned when we send a value message to the function. Then we send a play message to the returned routine. We could abbreviate the last two lines of our program by just having one line:

```
func.value([1/1, 3/2, 2/1]).play;
```

The interpreter, reading from right to left, will pass a value message to the func, with an argument of `[1/1, 3/2, 2/1]`, an array. That returns a Routine, which is then passed a message of play.

We could use a Routine to alter Nicole's program so that instead of just printing ratios, it plays them in sequence using our SynthDef "example2".

```
(

	var func, arr;

	func = { arg ratio_arr, baseFreq = 440, detune = 10;
		var pitch;

		Routine.new({
			ratio_arr.do({ arg ratio, index;
				pitch =  (ratio * baseFreq) + detune;
				//("Ratio: " ++ ratio ++ " Pitch: " ++ pitch).postln;
				Synth.new("example3", [\freq, pitch, \dur, 1]);
				1.wait;
```

```
            });
        });
    };

    arr = [ 1/1, 3/2, 4/3, 9/8, 16/9, 5/4, 8/5];

    func.value(arr, 440, 10).play;
)
```

We can change this so it plays the pitches back in a random order, by sending a scramble message to our ratio array, either by changing our function call to:

```
    func.value(arr.scramble, 440, 10).play;
```

Or by changing our do loop:

```
    ratio_arr.scramble.do({ arg ratio, index;
```

This is sounding more like music! Let's try repeating it a few times, with an argument specifying how many times.

```
(

    var func, arr;

    func = { arg ratio_arr, baseFreq = 440, detune = 10, repeats= 1;
        var pitch;

        Routine.new({
            repeats.do ({
                ratio_arr.scramble.do({ arg ratio, index;
```

```
                        pitch =  (ratio * baseFreq) + detune;
                        //("Ratio:  "  ++  ratio  ++  "  Pitch:  "  ++
pitch).postln;
                        Synth.new("example3", [\freq, pitch, \dur, 1]);
                        1.wait;
                    });
                });
            });
        };

        arr = [ 1/1, 3/2, 4/3, 9/8, 16/9, 5/4, 8/5];

        func.value(arr, 440, 10, 2).play;
)
```

What if we wanted to play out a hundred notes? There is often more than one way to do something.

We could figure out the size of our Array:

```
var arr_size;
arr_size = ratio_arr.size;
```

And then divide that into a hundred to get an number for a SimpleNumber.do loop. However, the array might not divide evenly into a hundred.

Or we could just count to 100.

```
100.do({arg count;

        count = count % arr_size;
```

```
    pitch = ratio_arr.at(count);
    pitch = (pitch * baseFreq) + detune;

    . . .
});
```

Remember that the modulus (%) gives us a remainder. This means that if the count gets to be greater than the number of elements in the array, using a modulus will cause it to wrap around to zero when it exceeds the size of the array. There is also a message you can send to arrays that does this for you:

```
arr.wrapAt(index) == arr.at(index % arr.size)
```

Problems

1. Try out the different fixed duration envelopes and try them out with different Oscillators. Especially try out the Env.perc envelope with different noise generators.

2. Create a program that plays notes from a tuning array. Change the duration of notes, using the \dur value in Synth.new. Try also modifying the length of the waits. Create rhythm by using arrays for durations and waits.

3. You can navigate arrays of different length using variables for indexes.

```
(

  var arr, arr2, arr2_index;

  arr = [1, 2, 3];
```

```
arr2 = [3, 4, 5, 6];
arr2_index = 0;

2.do({
     arr.do({ arg item;
          ("arr " ++ item).postln;
          ("\t arr2 " ++ arr2.wrapAt(arr2_index)).postln;
          arr2_index = arr2_index + 1;
     });
});
)
```

("\t" means tab and makes this print more nicely.)
Use this idea to create duration, pitch and wait loops of different lengths.

4. Instead of calculating detuning in your Routine, do it in the SynthDef using a fixed duration envelope. Create an argument to your SynthDef for the peak detuning amount.

Project

Write a one or two minute long piece using Routines.