SuperCollider Tutorial

Chapter 4

By Celeste Hutchins

2005

www.celesteh.com

Controls

In our last chapter, we began to create something more musical by playing every note in a scale, although out of order. Sometimes, we might just want to skip a note. We need a way to make decisions. One such way is with **if**. `if` has it's own helpfile. Highlight lowercase if and press apple-shift-?. If is also explained briefly in the helpfile for **Boolean**.

A **Boolean** is a value that is either true or false. `true` and `false` are reserved words in SuperCollider. We can send an `if` message to Booleans.

```
(

	([true, false].choose).if(
		{
			"true".postln;

		}, {
			"false".postln;
		}
	);
)
```

If you run the above code several times, “true” and “false” shout print out about the same number of times in a random order, because `[true, false].choose` ought to be true half the time and false the other half. The result of that expression is a Boolean. We send an if message to the Boolean, which has two arguments, both functions. The first function is evaluated if the Boolean is true. The second function is evaluated if the Boolean is false.

```
boolean.if(trueFunction, falseFunction);
```

You can omit the false function if you want.

This syntax that we've been using, `object.message(argument1, argument2, . . . argumentN);`, is the most commonly used syntax in SuperCollider programs. It's called **receiver notation**. However, there is more than one correct syntax in SuperCollider. There also exists a syntax called **functional notation**. It is more commonly used with if messages than receiver notation. When you see if in the helpfiles, the examples almost always use functional notation. Functional notation is:

```
message(object, argument1, argument2,  . . . argumentN);
```

The two notations are equivalent. You can replace one with the other at any place in any program and it will not change the program. What this means for if, is that you very commonly see:

```
if(boolean, trueFunc, falseFunc);
```

So our example would change to:

```
(
     if([true, false].choose, {
          "true".postln;

     }, {
          "false".postln;
     });
)
```

It works in exactly the same way.

Why are there multiple correct notations? It's confusing!

SuperCollider is based on many other programming languages, but the language that it borrows most heavily on is one called Smalltalk. Smalltalk, like SuperCollider, is an object-oriented language. When I took Programming Languages, my teacher said that Smalltalk was the best object oriented language and the only reason it wasn't the most popular was that the syntax was insane.

James McCartney, the author of SuperCollider perhaps was trying to spare us from the horrors of Smalltalk syntax and let us use receiver notation, which is common across many object-oriented languages. Functional notation, however, persists in if, probably because other languages have different ways of thinking about if.

Let's go back to our musical program and give it a 50% chance of playing a note and a 50% chance of resting:

```
(

	var func, arr;

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

		Routine.new({
			ratio_arr.scramble.do({ arg ratio, index;
				if( [true, false].choose, {
					pitch =  (ratio * baseFreq) + detune;
					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;
)
```

If we want to give it a 33.3% chance of resting (a 66% chance of playing), we could change out if to look like `if ([true, true, false].choose, {` and expand our array every time we want to change the probability. But what if we want something to play 99% of the time? We would have to have 99 trues and one false. Fortunately, there is a message you can use that returns a Boolean based on percentage. To play 66% of the time, we would change our if to `if (0.66.coin, {`

```
(

     var func, arr;

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

          Routine.new({
               ratio_arr.scramble.do({ arg ratio, index;
                    if( ((2/3).coin), {
                         pitch =  (ratio * baseFreq) + detune;
                         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;
)
```

`coin` is a message you can pass to SimpleNumber. It returns a true or false value. The number that receives the message is the percent likelihood that it will return true.

Boolean Expressions

Many arithmetic operations return Booleans. For example, we can test for **equivalency** with **==**

```
(
    a = 3;
    if (a == 3, {
        "true".postln;
    }, {
        "false".postln;
    });
)
```

Note that is two equal signs next to each other when we're testing for equivalency. If you just use one equal sign, it means assignment. I often accidentally type one equals sign when I mean to type two.

We can test for **greater than** or **less than**:

```
(
	a = 3;
	if (a > 4, {
		"true".postln;
	}, {
		"false".postln;
	});
)

(
	a = 3;
	if (a < 4, {
		"true".postln;
	}, {
		"false".postln;
	});
)
```

We can do Boolean operations. Some of the most important ones are **not**, **and**, and **or**.

The easiest way to illustrate these is with **truth tables**. A truth table shows you all possible combinations of true and false variables and what the results would be. Any Boolean variable can be either true or false. This is a truth table for **not**:

Not:

true	**false**
false	true

Not is a **unary operator**. That means it only involves one object. The top of the table shows a true input and a false input. The bottom of the table shows the result. `true.not` returns false and `false.not` returns true.

And is a **binary operator**. Like +, -, *, / and %, it operates on two objects. Lets' say we have two variables, a and b, and either of them can be true or false. We can put a along the top of the table and b down the left side. In the middle we put the possible results of a **and** b.

	true	**false**
true	true	false
false	false	false

Or is also binary:

	true	**false**
true	true	true
false	true	false

So how do we code these? Let's look again at not. Not is represented as **!**

```
(
    a = 2;
```

```
    if (!(a == 4) , {
        "true".postln;
    }, {
        "false".postln;
    });
)
```

Not just negates. It turns a false into a true and a true into a false. It can also be combined with equivalency to test for **not equals**.

```
(
    a = 2;

    if (a != 4 , {
        "true".postln;
    }, {
        "false".postln;
    });
)
```

The last two examples are the same.

And is represented by &&:

```
(
    a = 3;
    b = 4;

    if ( (a > 2) && (b < 5), {
        "true".postln;
    }, {
        "false".postln;
```

```
    });
)
```

Both `(a > 2)` **and** `(b < 5)` must be true for this expression to evaluate as true. If one of them is false, the whole thing is false.

Or is represented by `||` (Those are vertical lines. On your Macintosh with an American keyboard, they're over the slash \)**:**

```
(
    a = 3;
    b = 4;

    if ( (a > 2) || (b < 5), {
        "true".postln;
    }, {
        "false".postln;
    });
)

(
    a = 3;
    b = 4;

    if ( (a < 2) || (b < 5), {
        "true".postln;
    }, {
        "false".postln;
    });
)

(
```

```
	a = 3;
	b = 4;

	if ( (a > 2) || (b == 5), {
		"true".postln;
	}, {
		"false".postln;
	});
)
```

For these expressions to evaluate to true, only one part of it needs to be true. If neither part of an or is true, then the whole thing is false.

While

We've used Boolean expressions to control the flow of execution of a program with if. Another **control structure** is **while**. While is a message passed to a function. The function must return either true or false. It is a test function. There is another function passed in as an argument. `test_function.while(loop_function);` If the test function is true, the loop function gets run. Then the test function is run again. If it returns true again, the loop function is run again. This continues until the test function returns false. *While* the condition is true, the loop is executed.

```
(

	var add_amt, max_add, total;

	max_add = 0.5;
	total = 0;
```

```
    {total < 1}.while({

        add_amt = max_add.rand;
        // . . .
        total = total + add_amt;
        "in while loop".postln;
    });
)
```

While is a message sent to a function. Remember that receiver notation and functional notation are equivalent. The following two lines are the same:

```
test_func.while(body_func);
while(test_func, body_func);
```

There are other **Control Structures** detailed in a help file called Control-Structures. Highlight "Control-Structures" and press shift-apple-?

Problems

1. Re-write hello world from the first chapter using functional notation.

2. Write if statements using and or and not using Boolean values of true and false to illustrate the truth tables, using all possible combinations of true and false. For example:

```
(
        if (! true, {

        "true".postln;
        }, {
        "false".postln;
        });
```

)

What do you expect each of them to print? Did the results you got from running them match your expectations?

3. Write a function with returns a Routine. The function should take three arguments: an array of pitches to chose from, and array of durations to chose from and a total duration for the Routine. Play pitches in rhythm for the duration and then stop. You may wish use arrays of arrays to create rhythmic motifs or repeating pitch themes.

Project

Some songs, like *Bingo* or some German drinking songs, leave out particular pitches on repetitions. Write a one or two minute piece that repeats a short phrase but leaves notes out on the repeats.