SuperCollider Tutorial

Chapter 1

By Celeste Hutchins

2005

www.celesteh.com

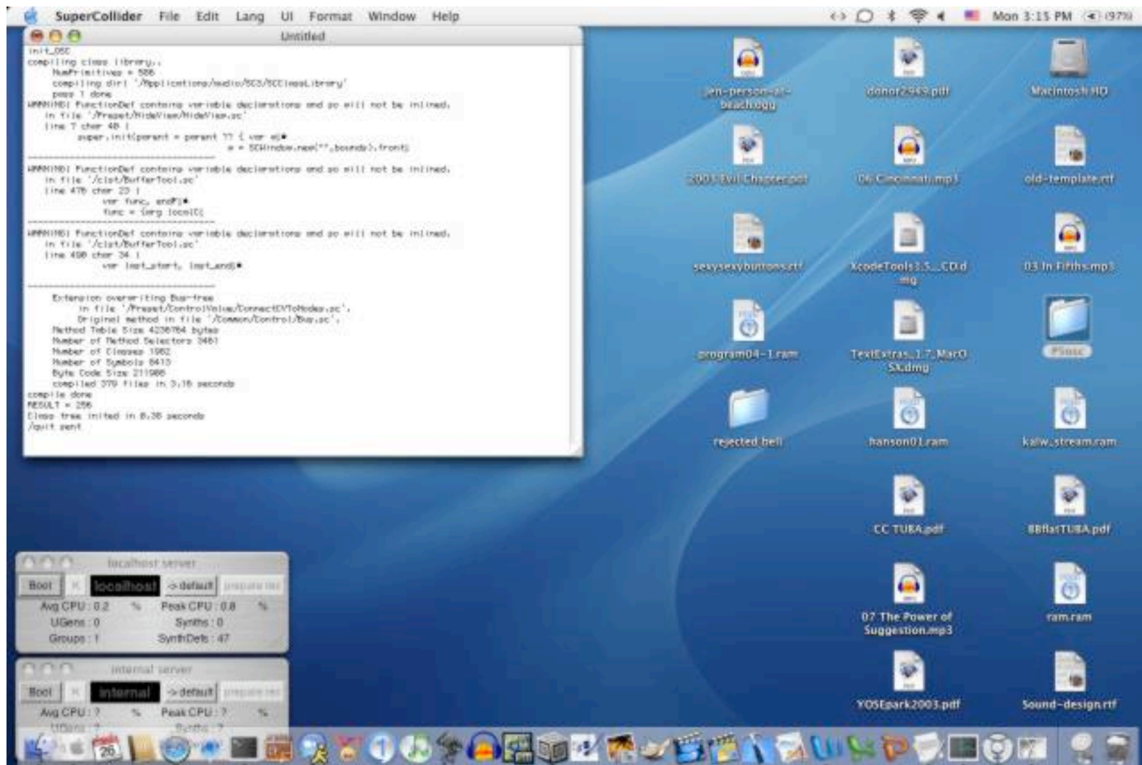
Creative Commons License: Attribution Only

Chapter 1: An Introduction

The goal of this document is to teach you SuperCollider, and some basic programming concepts and terminology. These concepts are the same across many programming languages

What is a **program**? It's a series of instructions that a computer follows to complete a task. A program is a lot like a score. You play a score in order from left to right, playing each note or rest one after each other, jumping backwards in case of repeats and forwards in cases of things like second endings or codas. In the same way, you can tell your computer to play a B for two seconds and then an A and tell it to repeat back and so on. You're able to tell your computer to do more complicated things to, like play a C if you move your mouse to the upper right hand corner and a D in the left hand, or make more complicated sounds. In order to tell all these things to your computer, you need to be able to speak the same language as it does. SuperCollider is a programming language designed for writing musical programs.

When you double click on the SuperCollider icon, three windows should open on your screen. A big text window called "Untitled" should print out some information and there should be two smaller windows below it called "localhost server" and "internal server." If there is an error in the Untitled window, the two server windows will not open. Try downloading a different build of SuperCollider or running it on a different machine.



The Untitled window is where text output goes. The other two windows control two different version of the audio Server. The examples in this document use the localhost server. If you want to hear audio, you must boot the audio server, which you can do by pressing the "Boot" button. We'll come back to what this means later.

To write your own code, you must open a new window, which you can do under the File menu or by typing apple-n. To execute code, highlight it with the mouse and then press the Enter key, NOT the return key. (The enter key may be located next to your arrows or in your number pad.) To stop code that is running, hit apple-period.

To get help, hit apple-shift-?. To get help on a specific topic, for instance on SynthDef, highlight the word SynthDef and hit apple-shift-?.

In Supercollider, blocks of code are enclosed in parentheses. To run an entire block, double click to the right of the open-paren and hit enter. For example, if you double click to right of the open paren of:

```
(
  "hello world".postln;
)
```

then `hello world` will print out in the Untitled window.

Objects

SuperCollider is an **object-oriented language**. `"hello world"` is a type of **object** called a **String**. A string is any piece of text surrounded by double quotes. There are other objects, for example, integers, or floating point numbers, or Arrays, which we'll talk about later. We can communicate with objects by using **methods** defined by the author of the object. In the above example, we're sending a **message** of `postln` to the string. That message tells the string to print itself.

We can think of the example of a light switch. If you want to turn on a light, you flip on the switch. This is like sending a message to the light bulb saying, "Hey, turn on." If we flip the switch down, it's like sending a message of "hey, turn off."

Variables

Let's think of a person, Nicole, as an object. Now, Nicole, herself is an object, but the word "Nicole" is a name that refers to Nicole the person. Similarly, we can give names to our objects. These names are called **variables**.

```
(
  var greeting;

  greeting = "hello world";
  greeting.postln;
)
```

What's going on there? The first word, "var," is short for **variable**. A variable is a storage location. It's a place to store a piece of data. The contents of data stored may vary, which is why it's called a variable. The second word "greeting" is a name. It is the name of the variables. Here we are **declaring** to the **interpreter** that we will have a variable named "greeting." The interpreter is the part of SuperCollider that reads and runs our programs. So, when the interpreter is reading our program and sees the word "greeting" it will know that greeting is a variable that we've created. Otherwise, it wouldn't know what we were talking about.
All variable names in SuperCollider must start with a lowercase letter and cannot be a **reserved word**. You cannot have a variable called "var" because var already has a special meaning to the interpreter.

Next we are **assigning** a value to the variable. `greeting` gets `"hello world".` The variable is on the left of the equal sign. It must always be on the left. There can only ever be one thing on the left of an equals sign. That sign is saying, hey, take whatever is on the right of this equals sign and store it under the variable name on the left.

In the last line, we're sending a `postln` message to the variable. The SuperCollider interpreter sends that message to `greeting`. `greeting` is a String. Strings print themselves with the postln message. So because `greeting` is a

String, the contents of `greeting`, `"hello world"`, print out.

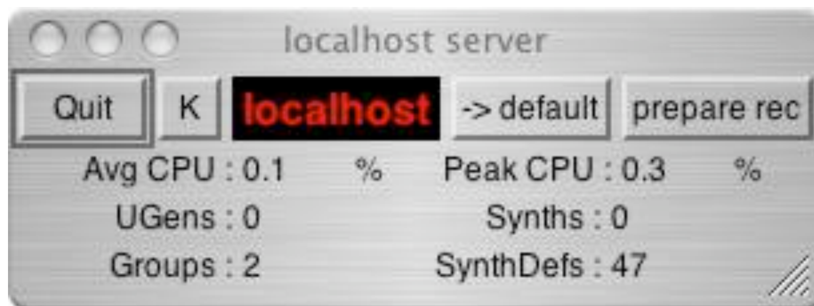
On the left is the variable name, an object. Then is a period. Then is the name of the message. There are a few different coding styles allowable in SuperCollider, but we're going to focus on **receiver notation** because it is common across many programming languages. That is

```
object.message;
```

Notice that every line ends with a semi-colon. In Supercollider, all lines must **terminate** with a semi-colon. Later on, when you're trying to figure out why some program isn't working, you're going to discover a missing semicolon. The semicolon tells the interpreter that you're done with that instruction. A single instruction can span as many lines as you need it to, but it must end with a semicolon.

Classes

Let's make some sound. First, make sure to boot the localhost server. The localhost server is one of the two small gray boxes at the bottom left of your screen. Press the “Boot” button. When it's booted, the word "localhost" will be red and the "Boot" button will change to say "Quit."



Then, select this code and press enter (not return!). You can select it by double-clicking to the right of the open parenthesis.

```
(
```

```
	var syn, sound;

	syn = SynthDef.new("example1", {
		Out.ar(0, SinOsc.ar(440));
	});

	syn.load(s);

	sound = Synth.new("example1");
)
```

When you want to stop the program, press apple-period.

What's going on here? Programs in SuperCollider are **interpreted**. That means that you have to run them from within the SuperCollider **interpreter**. You won't get a little icon that you can double click on like Microsoft Word. When you highlight text in SuperCollider and press enter, you're telling SuperCollider to run it as a program.

Our program first declares a variable, and then it defines a SynthDef and loads it on the server, then creates a Synth object that plays the sound. This is a complicated example, but I wanted to get you making sounds as soon as possible. Let's look at the example line by line.

The first line of our program is an open paren. **Blocks** of **code** are surrounded by parentheses in SuperCollider.

The next line is `var syn, sound;`. We are declaring two variables. One is called syn and the other is called sound.

The next line, translated into English, means "I want to create a new **SynthDef** called 'example1' and store it in the variable syn." Whoa, what's a SynthDef? The SynthDef help file told me that a SynthDef is a "definition of a synth architecture." I got to the help file by highlighting the word SynthDef and pressing apple-shift-? When you see a term that you don't understand, you can get to the help file (if one exists) by selecting the term and pressing apple-shift-?

SuperCollider is not just one program; it is two. One part of the program is the interpreter, which is what we've been talking about so far. The other part of the program is **audio server**, which is what actually makes the sound. The audio server runs separately from the interpreter, but they can communicate with each other using something called **OSC**, which we will talk about later.

The server does not know anything but what it needs to know. This makes it run much faster and more efficiently. You can tell it to make certain kinds of sounds, but in order to run quickly, it wants to know what kinds of sounds you want before you actually make them. You write a description of what you want the server to do. This description, or definition, is called a SynthDef.

`SynthDef` is the name of a **class**. Object definitions are called classes. An object is a particular **instance** of a class. In SuperCollider, all class names start with a capital letter.

Examples: Nicole is an example of a person. Remember that Nicole is an object. "Person" would be his class. Likewise, SynthDef is a class.

There are certain types of messages that you can send to a class. One of those is called a **constructor**. The message ".new" is a constructor. It tells the class to make a new instance of itself. So when we call SynthDef.new, we get back a new SynthDef, which we're storing, in the variable name syn.

Inside the parenthesis are some **arguments**. Some messages, like postln, know what to do when they're called - if we put postln after a String like we did above with "hello world".postln, the message just knows to print whatever the string is. Other messages require more information, which we call **arguments**. Notice that we have an object, a message and arguments. The way we code those is:

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

Or

```
Class.message(argument1, argument2, … argumentN);
```

In the case of our SynthDef, `SynthDef` is the class. `new` is the message. And `"example1"` and the stuff in between the curly brackets are the two arguments.

The stuff in the curly brackets is telling the SynthDef what to play. Things between curly brackets {} are called **functions**. A function is a special type of object that is made up of a code block that you can run. We'll come back to this later.

Out is a **UGen**. The help file for UGens says, "A UGen or **unit generator** is an object for generating or processing audio or control signals." UGens exist in SynthDefs. They are what make up a SynthDef. Out is a UGen that writes a signal to a **bus**, which, in this case, sends it's output to the left channel. .ar is a type of constructor. So Out.ar creates a new instance of a an Out UGen running at the **audio rate**. ar stands for "audio rate" and is a common constructor name for UGens.

Looking at the arguments to Out.ar, 0 means left channel. If that were a 1, it would mean right channel. And the next thing is what gets sent out, which is a sine tone generator. The argument to SinOsc.ar, 440, is the frequency to play.

So SinOsc.ar(440) creates a sine wave at 440 Hz. Out.ar takes that sine wave and sends it out to the left channel, which is channel 0.

The next line, is a closing curly bracket, a close paren and a semicolon. It's the end of the function, the end of the SynthDef and the end of a statement. We've created a new SynthDef that includes in it a function describing what the SynthDef should do when it gets instantiated.

The next line says, take that SynthDef and load it on the server. We're sending a message to syn, saying load. The argument, s, is the server to send it to. In SuperCollider, a lowercase s by itself refers to the audio server.

The next line asks to create a new Synth. A Synth is an instance of a SynthDef. The interpreter sends a message to the server, saying, "Hey, could you make a synth that plays the SynthDef called 'example1'?" The server looks and says, "oh yeah, I have a SynthDef called that" and makes a new instance of a Synth, which is running the function that we defined.

We get a new Synth object back from this and store it in "sound."

Let's say we don't want to play an A. Let's say we want to play an E. We can change 440 to 660.

```
(
	var sdef;
	sdef = SynthDef.new("example1", {

		Out.ar(0, SinOsc.ar(660));
	});

	sdef.play;
```

```
)
```

But when we're writing a piece, we don't want to have to write a new SynthDef for every note that we're going to play. We can create out own argument, which will tell the SynthDef what frequency to play.

```
(

	var syn, sound;

	syn = SynthDef.new("example1", { arg freq;
		Out.ar(0, SinOsc.ar(freq));
	});

	syn.load(s);

	sound = Synth.new("example1", [\freq, 440]);
)
```

We call that argument "freq." An argument is a special kind of variable. You declare then at the top of a code block by using the reserved word "arg." So the "`arg freq;`" part of "`syn = SynthDef.new("example1", { arg freq;`" tells SuperCollider that our SynthDef function takes a single argument called freq. We can then use freq like we would any other variable. Here, we're passing to SinOsc.ar, to use for the frequency.

Passing variables to Synths is a bit different than normal variable passing. The interpreter has to communicate with the audio server. They use a protocol called OSC. The Synth object will handle the OSC for you, but the server does not know which argument you're trying to pass a value to unless you specifically tell it. So Synth.new takes an optional second argument, which is an **Array**. An

array is a list separated by commas and surrounded by square brackets. When you are passing arguments to the server via a synth, the array must be made up of pairs. The first item in a pair is a **symbol** or a string and the second is a value. The symbol or string must match the name of the argument. So for an argument called freq, we use the symbol \freq or the string “freq”. If we had an argument called foo, we would use the symbol \foo or the string “foo”. A symbol always starts with a forward slash: ‘\’. So to pass a value of 440 to the argument freq, our array contains [\freq, 440]. If we had two arguments, one freq and the other foo, we could pass values to freq and foo with an array that looks like [\freq, 440, \foo 647]. The array is made up of pairs, to the 440 goes to freq and 647 goes to foo. The symbols or strings must match the names of the arguments taken by the SynthDef function.

Problems

When you are writing code, if you want to syntax colorize it, so that reserved words are blue, strings are grey, symbols are green, and so on, you can find that in the Format menu or just type apple-‘ (apple-single quote).

1. Write your own version of “hello world.”

2. Write your own SynthDefs, using some of the oscillator UGens. To find a list of them, highlight the word “UGens” and type apple-shift-?. Some oscillators to try are Saw and Pulse.