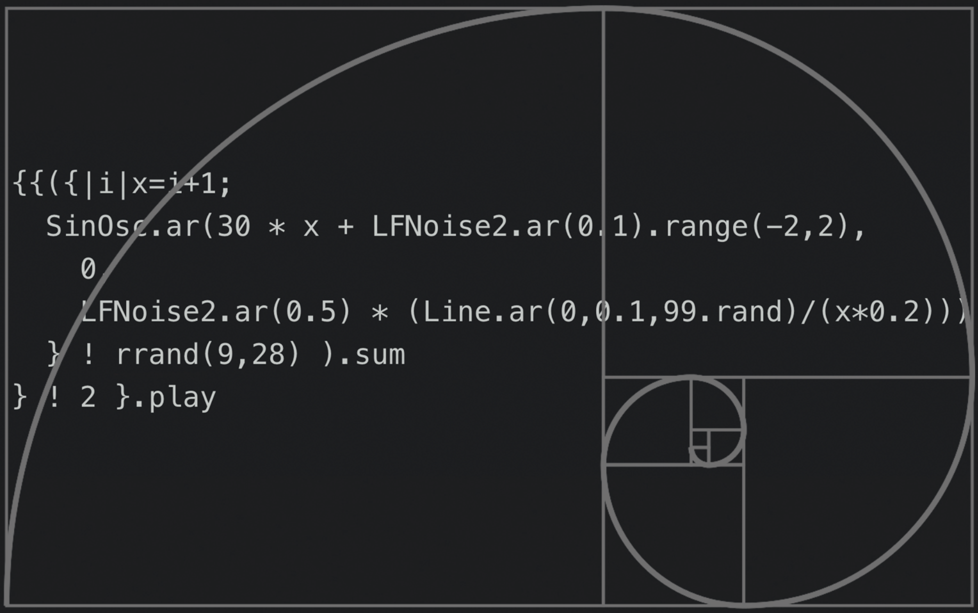
SCORING SOUND: CREATIVE MUSIC CODING WITH SUPERCOLLIDER



```
{{({|i|x=i+1;
  SinOsc.ar(30 * x + LFNoise2.ar(0.1).range(-2,2),
    0,
    LFNoise2.ar(0.5) * (Line.ar(0,0.1,99.rand)/(x*0.2)))
  } ! rrand(9,28) ).sum
} ! 2 }.play
```

THOR MAGNUSSON

Scoring Sound

Creative Music Coding with SuperCollider

Thor Magnusson

This book is for sale at http://leanpub.com/ScoringSound

This version was published on 2021-04-19

Tweet This Book!

Please help Thor Magnusson by spreading the word about this book on Twitter!

The suggested tweet for this book is:

If you are into writing music with code, check out this SuperCollider tutorial by @thormagnusson #scoringsound https://leanpub.com/ScoringSound

The suggested hashtag for this book is #scoringsound.

Find out what other people are saying about the book by clicking on this link to search for this hashtag on Twitter:

#scoringsound

This book is equally dedicated to the author and developers of the SuperCollider programming language as well as its amazingly creative users.

Contents

Part I

Chapter 1 - The SuperCollider language

This chapter will introduce the fundamentals for creating and running a simple SuperCollider program. It will introduce the basic concepts needed for further exploration. We will learn the basic key orientation practices of SuperCollider, that is how to run code, post into the post window and use the documentation system. We will also discuss the key fundamental things needed to understand and write SuperCollider code, namely: variables, arrays, functions and basic data flow syntax. Having grasped the topics introduced in this chapter, you should be able to write practically anything that you want, although later we will go into Object Orientated Programming, which will make things considerably more effective and perhaps easy.

The semicolon, brackets and running a program

The semicolon ";" is what divides one instruction from the next. It defines a *line* of code. After the semicolon, the interpreter looks at next line. There has to be semicolon after each line of code. Forgetting it will give you errors printed in the post console.

This code will work fine if you evaluate only this line:

```
"Hello World".postln
```

But not this, if you evaluate both lines (by highlighting both and evaluating them with Shift+Return):

```
"Hello World".postln
"Goodbye World".postln;
```

Why not? Because the interpreter (the SC language) will not understand

```
"Hello World".postln "Goodbye World".postln;
```

However, this will work:

```
"Hello World".postln; "Goodbye World".postln;
```

It is up to you how you format your code, but you'd typically want to keep it readable for yourself in the future and other readers too. There is however a style of SC coding used for Tweeting, where the 140 character limit introduces interesting constraints for composers. Below is a Twitter composition by Tim Walters, but as you can see, it is not good for human readability although it sounds good (The language doesn't care about human readability, but we do):

```
play{HPF.ar(({|k|({|i|SinOsc.ar(i/96,Saw.ar(2**(i+k))/Decay.ar(Impulse.ar(0.5**i/k),[k*i+1,\
k*i+1*2],3**k))}!6).product}!32).sum/2,40)}
```

It can get tiring to having to select many lines of code and here is where brackets come in handy as they can create a scope for the interpreter. So this following code:

```
var freq = 440;
var amp = 0.5;
{SinOsc.ar(freq, 0, amp}.play;
```

will not work unless you highlight all three lines. Imagine if these were 100 lines: you would have to do some tedious scrolling up and down the document. So using brackets, you can simply double click after or before a bracket, and it will highlight all the text between the matching brackets.

```
(
var freq = 440;
var amp = 0.5;
{SinOsc.ar(freq, 0, amp}.play;
)
```

Matching brackets

Often when writing SuperCollider code, you will experience errors whose origin you can't figure out. Double clicking between brackets and observe whether they are matching properly is one of the key methods of debugging SuperCollider code.

```
(
"you ran the program and ".post;
(44+77).post; " is the sum of 44 and 77".postln;
"the next line - the interpreter posts it twice as it's the last line".postln;
)
```

The following will not work. Why not? Look at the post window.

```
(
(44+77).postln
55.postln;
)
```

Note that the • sign is where the interpreter finds the error.

The post window

You have already posted into the post window (many other languages use a “print” and “println” for this purpose). But let's explore the post window a little further.

```
(
"hello".post; // post something
"one, two, three".post;
)


(
"hello there".postln; // post something and make a line break
"one, two, three".postln;
)

1+4; // returns 5

Scale.minor.degrees // returns an array with the degrees in the minor scale
```

You can also use postf:

```
"the first value is %, and the second one is % \n".postf(1111, 9999);
```

If you are posting a long list you might not get the whole content using .postln, as SC is lazy and doesn't like printing too long data structures, like lists.

For this purpose use the following:

```
Post << "hey"
```

Example

```
Array.fill(1000, {100.rand}).postln; // you see you get ...etc...
```

Whereas,

```
Post << Array.fill(1000, {100.rand}) // you get the whole list
```

The Documentation system (The help system)

The documentation system in SuperCollider is a good source for information and learning. It includes introduction tutorials, overviews and documentation for almost every class in SuperCollider. The documentation files typically contain examples of how to use the specific class/UGen, and thus serves as a great source for learning and understanding. Many SC users go straight into the documentation when they start writing code, using it as a template and copy-paste the examples into their projects.

So if you highlight the word Array in an SC document and hit Cmd+d or Ctrl+d (d for documentation), you will get the documentation for that class. You will see the superclasses/subclasses and learn about all the methods that the Array class has. With no text highlighted, you can search the documentation by hitting Cmd+D (capital d) and you will get a menu asking “Search documentation for” where you can type in your item, such as “LFNoise0”.

Also, if you want to read and browse all the documentation, you can open a help browser: **Help.gui**.

Comments

Comments are information written for humans, but ignored by the language interpreter. It is a good practice to write comments where you think you might forget what a certain block of code will do. It is also a communication to another programmer who might read your code. Feel free to write as many comments as you want, but often it might be a better practice to name your variables and function names (we'll learn later in this section what these words mean) such that you don't need to add a comment.

```
// This is a comment

/*
And this is
also a comment
*/
```

Comments are red by default, but can be any colour (in the Format menu choose 'syntax colorize')

Variables

Here is a mantra to memorise: Variables are containers of some value. They are names or references to values that could change (their value can vary). So we could create a variable that is a property of yourself called age. Every year this variable will increase by one integer (a whole number). So let us try this now:

```
var age = 33;
age = age + 1; // here the variable 'age' gets a new value, or 33 + 1
age.postln; // and it posts 34
```

SuperCollider is not strongly typed so there is no need to declare the data type of variables. Data types (in other languages) include : integer, float, double, string, custom objects, etc... But in SuperCollider you can create a variable that contains an integer at one stage, but later contains reference to a string or a float. This can be handy, but one has to be careful as this can introduce bugs in your code.

Above we created a variable 'age', and we *declared* that variable by writing 'var' in front of it. All variables have to be declared before you can use them. There are two exceptions, all lowercase letters from a to z (note that 's' is a special variable that is by default used as a reference to the SC Server) can be used without declaration. There are also so called *environmental* variables (which can be considered global variables within a certain context) and they start with the '~' symbol. More on that later.

```
a = 3; // we assign the number 3 to the variable "a"
a = "hello"; // we can also assign a string to it.
a = 0.333312; // or a floating point number;
a = [1, 34, 55, 0.1, "string in a list", \symbol, pi]; // or an array with mixed types

a // hit this line and we see in the post window what "a" contains
```

SuperCollider has scope, so if you declare a variable within a certain scope, such as a function, they can have a local value within that scope. So try to run this code (by double clicking behind the first bracket).

```
(
var v, a;
v = 22;
a = 33;
"The value of a is : ".post; a.postln;
)
"The value of a is now : ".post; a.postln; // then run this line
```

The value of 'a' will be from the code block above this one. So 'a' is a global variable, but because you declared it with a *var* in a scope (the brackets) it did not override the global variable. This is good for prototyping and testing, but not recommended as a good software design. A variable with the name 'myvar' could not be global – only single lowercase characters.

If we want longer variable names, we can use environmental variables (using the ∼ symbol): they can be seen as global variables, accessible from anywhere in your code

```
~myvar = 333;

~myvar // post it;
```

But typically we just declare the variable (var) in the beginning of the program and assign its value where needed. Environmental variables are not necessary, although they can be useful, and this book will not use them extensively.

But why use variables at all? Why not simply write the numbers or the value wherever we need it? Let's take one example that should demonstrate clearly why they are useful:

```
{
 // declare the variables
var freq, oscillator, filter, signal;
freq = 333; // set the frequency variable
 // create a Saw wave oscillator with two channels
oscillator = Saw.ar([freq, freq+2]);
// use a resonant low pass filter on the oscillator
filter = RLPF.ar(oscillator, freq*4, 0.25);
// multiply the signal by 0.5 to lower the amplitude
signal = filter * 0.5;
}.play;
```

As you can see, the 'freq' variable is used in various places in the above synthesizer. You can now change the value of the variable to something like 500, and it the frequency will 'automatically' be turned into 500 Hz in the left channel, 502 Hz in the right, and the cutoff frequency will be 2000 Hz. So instead of changing these variables throughout the code, you change it in one place and its value magically plugged into every location where that variable is used.

Functions

Functions are an important feature of SuperCollider and most other programming languages. They are used to encapsulate algorithms or functionality that we only want to write once, but use in different places at various times. They can be seen as a black box or a factory that takes some input, parses it, and returns some output. Just as a sophisticated coffee machine might take coffee beans and water as input, it then grounds the beans, boils the water, brews the coffee, and finally outputs a lovely drink. The key point is that you don't need (or want) to know precisely how all this happens. It is enough to know where to fill up the beans and water, and then how to operate the buttons of the machine (strength, number of cups, etc.). The coffee machine is a [black box] (http://en.wikipedia.org/wiki/Black_box).

Functions in SuperCollider are notated with curly brackets '{}'

Let's create a function that posts the value of 44. We store it in a variable 'f', so we can *call* it later.

```
f = { 44.postln };
```

When you run this line of code, you see that the SuperCollider post window notifies you that it has been given a function. It does *not* post 44 into the post window. For that we have to call the function, i.e., to ask it to perform its calculation and return some value to us.

```
f.value // to call the function we need to get its value
```

Let us write a more complex function:

```
f = {
    69 + ( 12 * log( 220/440 ) / log(2) )
};
f.value // returns the MIDI note 57 (the MIDI note for 220 Hz)
```

This is a typical function that calculates the MIDI note of a given frequency in Hz (or cycles per second). Most electronic musicians know that the MIDI note 60 is C, and that 69 is A, and that A is 440 Hz. But how is this calculated? Well the function above does return the MIDI note of 220 Hz. But this is a function without any input (or *argument* as it is called in the lingo). Let's open up this input channel, by drilling a hole into the black box, and let's name this argument 'freq' as that's what we want to put in.

```
f = { arg freq;
    69 + ( 12 * log( freq/440 ) / log(2) )
}
```

We have now an *input* into our function, an argument named 'freq'. Note that this argument has been put into the right position inside the calculation. We can now put in any frequency and get the relevant MIDI note.

```
f.value(440) // returns 69
f.value(880) // returns 81
f.value(261) // returns 59.958555396543 (a fractional MIDI note, close to C (or 60))
```

The above is a good example of why functions are so great. The algorithm of calculating the MIDI note from frequency is somewhat complex (or nasty?), and we don't really want to memorise it or write it more than once. We have simply created a black box that we put in to the 'f' variable and now we can call it whenever we want without knowing what is inside the black box.

We will be using functions all the time in the coming chapters. It is vital to understand how they receive arguments, process the data, and return a value.

The final thing to say about functions at this stage is that they can have default values in their arguments. This means that we don't have to *pass* in all the arguments of the function.

```
f = { arg salary, tax=20;
    var aftertax;
    aftertax = salary - (salary * (tax/100) )
}
```

So here above is a function that calculates the pay after tax, with the default tax rate set at 20%. Of course we can't be sure that this is the tax rate forever, or in different countries, so this needs to be an argument that can be set in the different contexts.

```
f.value(2000) // here we use the default 20% tax rate
f.value(2000, 35) // and here the tax rate has become 35%
```

Tip

SuperCollider contains quite a lot of examples of "syntax sugar", i.e., where you can write things slightly differently for the sake of brevity (or perhaps aesthetics?). We will explore more of this sugar later in the book, but for now it suffices to mention some related to the function.

You will see the following

```
f = { arg string; string.postln; } // we will post the string that comes into the function
f.value("hi there") // and here we call the function passing "hi there" as the argument.
```

Often written in this form:

```
f = {|string| string.postln;} // arguments can be defined within two pipes '|'
f.("hi there") // and you can skip the .value and just write a dot (.)
```

Arrays, Lists and Dictionaries

Arrays are one of the most useful things to understand and use in computer music. This is where we can store bunch of data (whether pitches, scales, synths, or any other information you might want to reference). A common thing a novice programmer typically does is to create lots of variables for data that could be stored in an array, so let's dive straight into learning how to use arrays and lists.

An array can be seen as a storage space for things that you need to use later. Like a bag or a box where you keep your things. We typically keep the reference to the array in a variable so we can access it anywhere in our code:

```
a = [11, 22, 33, 44, 55]; // we create an array with these five numbers
```

You will see that the post window posts the array there when you run this line. Now let us try to play a little with the array:

```
a[0]; // we get at the first item in the array (most programming languages index at zero)
a[4] // returns 55, as index 4 into the array contains the value 55
a[1]+a[4] // returns 77 as 22 plus 55 equals 77
a.reverse // we can reverse the array
a.maxItem // the array can tell us what is the highest value
```

and so on. The array we created above had five defined items in it. But we can create arrays differently, where we fill it algorithmically with any data we'd be interested in:

```
a = Array.fill(5, { 100.rand }); // create an array with five random numbers from 0 to 100
```

What happened here is that we tell the Array class to fill a new array with five items, but then we pass it a function (introduced above) and the function will be evaluated five times. Compare that with:

```
a = Array.fill(5, 100.rand ); // create an array with ONE random number from 0 to 100
```

We can now play a little bit with that function that we pass to the array creation:

```
a = Array.fill(5, { arg i; i }); // create a function with the iterator (‘i’) argument
a = Array.fill(5, { arg i; (i+1)*11 }); // the same as the first array we created
a = Array.fill(5, { arg i; i*i });
a = Array.series(5, 10, 2); // a new method (series).
// Fill the array with 5 items, starting at 10, adding 2 in every step.
```

You might wonder why this is so fantastic or important. The fact is that arrays are used everywhere in computer music. The sound file you will load in later in this book will be stored in an array, with each sample in its own slot in an array. Then you can jump back and forth in the array, scratching, cutting, break beating or whatever you would like to do, but the fact is that this is all done with data (the samples of your soundfile) stored in an array. Or perhaps you want to play a certain scale.

```
m = Scale.minor.degrees; // the Scale class will return the degrees of the minor scale
```

m is here an array with the following values: [0, 2, 3, 5, 7, 8, 10]. So in a C scale, 0 would be C, 2 would be D (two half notes above C), 3 would be E flat, and so on. We could represent those values as MIDI notes, where 60 is the C note (∼ 261Hz). And we could even look at the actual frequencies in Hertz of those MIDI notes. (Those frequencies would be passed to the oscillators as they are expecting frequencies and not MIDI notes as arguments).

```
m = Scale.minor.degrees; // Scale class returns the degrees of the minor scale
m = m.add(12); // you might want to add the octave (12) into your array
m = m+60 // here we simply add 60 to all the values in the array
m = m.midicps // and here we turn the MIDI notes into their frequency values
m = m.cpsmidi // but let’s turn them back to MIDI values for now
```

We could now play with the ‘m’ array a little. In an algorithmic composition, for example, you might want to pick a random note from the minor scale

```
n = m.choose; // choose a random MIDI note and store it in the variable ’n’
x = m.scramble; // we could create a melody by scrambling the array
x = m.scramble[0..3] // scramble the list and select the first 4 notes
p = m.mirror // mirror the array (like an ascending and descending scale)
```

You will note that in ‘x = m.scramble’ above, the ‘x’ variable contains an array with a scrambled version of the ‘m’ array. The ‘m’ array is still intact: you haven’t scrambled that one, you’ve simply said “put a scrambled version of ‘m’ into variable ‘x’.” So the original ‘m’ is still there. If you really wanted to scramble ‘m’ you would have to do:

```
m = m.scramble; // a scrambled version of the ‘m’ array is put back into the ‘m’ variable
// But now it’s all scrambled up. Let’s sort it into ascending numbers again:
m = m.sort
```

Arrays can contain anything, and in SuperCollider, they can contain values of mixed types, such as integers, strings, floats, and so on.

```
a = [1, "two”, 3.33, Scale.minor] // we mix types into the array.
// This can be dangerous as the following
a[0]*10 // will work
a[1]*10 // but this won’t, as you cant multiply the word "two” with 10
```

Arrays can contain other arrays, containing other arrays of any dimensions.

```
// a function that will create a 5 item array with random numbers from 0 to 10
f = { Array.fill(5, { 10.rand }) }; // array generating function
a = Array.fill(10, f.value);  // create another array with 10 items of the above array
// But the above was evaluated only once. Why?
// Because, you need to pass it a function to get a different array every time. Like this:
a = Array.fill(10, { f.value } );  // create another array with 10 items of the above array
// We can get at the first array and see it’s different from the second array
a[0]
a[1]
// We could put a new array into a[0] (that slot contains an array)
a[0] = f.value
// We could put a new array into a[0][0] (an integer)
a[0][0] = f.value
```

Above we added 12 to the minor scale.

```
m = Scale.minor.degrees;
m.add(12) // but try to run this line many times, the array won’t grow forever
```

Lists

It is here that the List class becomes useful.

```
l = List.new;
l.add(100.rand) // try to run this a few times and watch the list grow
```

Lists are like arrays - and implement many of the same methods - but the are slightly more expensive than arrays. In the example above you could simply do 'a = a.add(100.rand)' if 'a' was an array, but many people like lists for reasons we will discuss later.

Dictionaries

A dictionary is a collection of items where *keys* are mapped to *values*. Here, keys are keywords that are identifiers for slots in the collection. You can think of this like names for values. This can be quite useful. Let's explore two examples:

```
a = Dictionary.new
a.put(\C, 60)
a.put(\Cs, 61)
a.put(\D, 62)
a[\Ds] = 63 // same as .put
// and now, let's get the values
a.at(\D)
a[\D#] // same as .at

a.keys
a.values
a.getPairs
a.findKeyForValue(60)
```

Imagine how you would do this with an Array. One way would be

```
a = [\C, 60, \Cs, 61, \D, 62, \Ds, 63]
// we find the slot of a key:
x = a.indexOf(\D) // 4
a[x+1]
// or simply
a[a.indexOf(\D)+1]
```

but using an array you need to keep track of the how things are organised and indexed.

Another Dictionary example:

```
b = Dictionary.new
b.put(\major, [ 0, 2, 4, 5, 7, 9, 11 ])
b.put(\minor, [ 0, 2, 3, 5, 7, 8, 10 ])
b[\minor]
```

Methods?

We have now seen things as 100.rand and a.reverse. How does .rand and .reverse work? Well, SuperCollider is an Object Orientated language[1] and these are *methods* of the respective classes. So an integer (like 100), has methods like .rand, .midicps, or .neg. It does not have a .reverse method. Why not? Because you can't reverse a number. However, an array (like [11,22,33,44,55]) can be reversed or added to. We will explore this later in the chapter about Object Orientated programming in SC, but for now it is enough to think that the object (an instantiation of the class) has relevant methods. Or to use an analogy: let's say we have a class called Car. This class is the information needed to build the car. When we build a Car, we instantiate the class and we have an actual Car. This car can then have some methods, for instance: start, drive, turn, putWipersOn. And these methods could have arguments, like speed(60), or turn(-60). You could think about the object as the noun, the method as the verb, and the argument as the adjective. (As in: John (object) walks (method) fast (adjective)).

```
// we create a new car. 4 indicating for example number of seats
c = Car.new(4);
c.start;
c.drive(40); // the car drives 40 miles per hour
c.turn(-60); // the car turns 60 degrees to the left
```

So to really understand a class like Array or List you need to read the documentation and explore the methods available. Note also that the Array is subclassing (or getting methods from its superclass) the ArrayedColldection class. This means that it has all the methods of its superclass. Like a class "Car" might have a superclass called "Vehicle" of which a "Motorbike" would also be a subclass (a sibling to "Car"). You can explore this by peeking under the hood of SC a little:

```
Array.openHelpFile // get the documentation of the Array class
Array.dumpInterface // get the interface or the methods of the Array class
Array.dumpFullInterface // get the methods of Array's superclasses as well.
```

You can see that in the .dumpFullInterface method will tell you all the methods Array *inherits* from its superclasses.

Now, this might give you a bit of a brainache, but don't worry, you will gradually learn this terminology and what it means for you in your musical or sound practice with SuperCollider. Wikipedia is good place to start reading about [Object Oriented Programming] (https://en.wikipedia.org/wiki/Object-oriented_-programming).

Conditionals, data flow and control

The final thing we should discuss before we start to make sounds with SuperCollider is how we control data and take decisions. This is about logic, about human thinking, and how to encode such decisions in the form of code. Such logic the basic form of all clever systems, for example in artificial intelligence. In short it is about establishing conditions and then decide what to do with them. For example: if it is raining and I'm going out, I take my umbrella with me, else I leave it at home. It's about basic logic that humans do all the time throughout the day. And programming languages have ways formalise such conditions, most typically with an if-else statement.

[1]https://en.wikipedia.org/wiki/Object-oriented_programming

In pseudocode it looks like this: if(condition, { then do this }, { else do this }); as in: if(rain, { umbrella }, { no umbrella });

So the condition represents a state that is either true or false. If it is true (there is rain), then it evaluates the first function, if false (no rain) it evaluates the second condition.

Another form is a simple if statement where you don't need to specify what to do if it's false: if(hungry, { eat });

So let's play with this:

```
if( true, { "condition is TRUE".postln;}, {"condition is FALSE".postln;});
if( false, { "condition is TRUE".postln;}, {"condition is FALSE".postln;});
```

You can see that true and false are keywords in SuperCollider. They are so called Boolean values. You should not use those as variables (well, you can't). In digital systems, we operate in binary code, in 1s and 0s. True is associated with 1 and false with 0.

```
true.binaryValue;
false.binaryValue;
```

Boolean logic is named after George Boole who wrote an important paper in 1848 ("The Calculus of Logic") on expressions and reasoning. In short it involves the statements AND, OR, and not.

A simple Boolean truth table might look like this

```
true AND true = true
true AND false = false
false AND false = false
true OR true = true
true OR false = true
false OR false = false
```

And also

```
true AND not false = true
```

etc. Let's try this in SuperCollder code and observe the post window. But first we need to learn the basic syntax for the Boolean operators:

== stands for *equal* != stands for *not equal* && stands for *and* || stands for *or*

And we also use comparison operators

">" stands for *more than*
"<" stands for *less than*
">=" stands for *more than or equal to*
"<=" stands for *less than or equal to*

```
true == true // returns true
true != true // returns false (as true does indeed equal true)
true == false // returns false
true != false // returns true (as true does not equal false)
3 == 3 // yes, 3 equals 3
3 != 4 // true, 3 does not equal 4
true || false // returns true, as one of the elements are true
false || false // returns false, as both of the elements are false
3 > 4 // false, as 3 is less than 4
3 < 4 // true
3 < 3 // false
3 <= 3 // true, as 3 is indeed less than or equal to 3
```

You might not realise it yet, but knowing what you now know is very powerful and it is something you will use all the time for synthesis, algorithmic composition, instrument building, sound installations, and so on. So make sure that you understand this properly. Let's play with this a bit more in if-statements:

```
if( 3==3, { "condition is TRUE".postln;}, {"condition is FALSE".postln;});
if( 3==4, { "condition is TRUE".postln;}, {"condition is FALSE".postln;});
// and things can be a bit more complex:
if( (3 < 4) && (true != false), {"TRUE".postln;}, {"FALSE".postln;});
```

What happened in that last statement? It asks: is 3 less than 4? Yes. AND is true not equal to false? Yes. Then both conditions are true, and that's what it posts. Note that of course the values in the string (inside the quotation marks) could be anything, we're just posting now. So you could write:

```
if( (3 < 4) && (true != false), {"VERDAD".postln;}, {"FALSO".postln;});
```

in Spanish if you'd like, but you could not write this:

verdad == verdad

as the SuperCollider language is in English.

But what if you have lots of conditions to compare? Here you could use a *switch* statement:

```
(
a = 4.rand; // a will be a number from 0 to 4;
switch(a)
        {0} {"a is zero".postln;} // runs this if a is zero
        {1} {"a is one".postln;} // runs this if a is one
        {2} {"a is two".postln;} // etc.
        {3} {"a is three".postln;};
)
```

Another way is to use the case statement, and it might be faster than the switch.

```
(
a = 4.rand; // a will be a number from 0 to 4;
case
	{a == 0} {"a is zero".postln;} // runs this if a is zero
	{a == 1} {"a is one".postln;} // runs this if a is one
	{a == 2} {"a is two".postln;} // etc.
	{a == 3} {"a is three".postln;};
)
```

Note that both in switch and case, the semicolon is only after the last testing condition. (so the line evaluation goes from “case...... to that semicolon”)

Looping and iterating

The final thing we need to learn in this chapter is looping. Looping is one of the key tricks used in programming. Say we want to generate 1000 synths at once. It would be tedious to write and evaluate 1000 lines of code one after another, but it’s easy to loop one line of code 1000 times!

In many programming languages this is done with a [for-loop] (http://en.wikipedia.org/wiki/For_loop):

```
for(int i = 0; i > 10, i++) {
	println("i is now" + i);
}
```

The above code will work in Java, C, JavaScript and many other languages. But since SuperCollider is a fully object orientated language, where everything is an object - which can have methods - so an integer can have a method like .neg, or .midicps, but also .do (which is our loop).

So in SuperCollider we can simply do:

```
10.do({ "SCRAMBLE THIS 10 TIMES".scramble.postln; })
```

What happened is that it loops through the command 10 times and evaluates the function (which scrambles and posts the string we wrote) every time. We could then make a counter:

```
(
var counter = 0;
10.do({
	counter = counter + 1;
	"counter is now: ".post;
	counter.postln;
})
)
```

But instead of such counter we can use the argument passed into the function in a loop:

```
10.do({arg counter; counter.postln;});
// you can call this argument whatever you want:
10.do({arg num; num.postln;});
// and the typical convention is to use the character "i" (for iteration):
10.do({arg i; i.postln;});
```

Let’s now try to make a small program that gives us all the prime numbers from 0 to 10000. There is a method of the Integer class that is called isPrime which comes in handy here. We will use many of the things learned in this chapter, i.e., creating a List, making a do loop with a function that has a iterator argument, and then we’ll ask if the iterator is a prime number, using an if-statement. If it is (i.e. true), we add it to the list. Finally we post the result to the post window. But note that we’re only posting after we’ve done the 10000 iterations.

```
(
p = List.new;
10000.do({ arg i; // i is the iteration from 0 to 10000
        if( i.isPrime, { p.add(i) }); // no else condition - we don't need it
});
Post << p;
)
```

We can also loop through an Array or a List. Then the do-loop will pick up up all the items of the array and pass it into the function that you write inside the do loop. Additionally, it will add an iterator. So we have two arguments to the function:

```
(
[ 11, 22, 33, 44, 55, 66, 77, 88, 99 ].do({arg item, counter;
        item.post; " is in the array at slot: ".post; counter.postln;
});
)
```

So it posts the slot (the counter/iterator always starts at zero), and the item in the list. You can call the arguments whatever you want of course. Example:

```
[ 11, 22, 33, 44, 55, 66, 77, 88, 99 ].do({arg aa, bb; aa.post; " is in the array at slot: \
".post; bb.postln });
```

Another looping technique is to use the for-loop:

```
for(startValue, endValue, function); // this is the syntax
for(100, 130, { arg i; i = i+10; i.postln; }) // example
```

We might also want to use the forBy-loop:

```
forBy(startValue, endValue, stepValue, function); // the syntax
forBy(100, 130, 4, { arg i; i = i+10; i.postln; }) // example
```

While is another type of loop:

```
while (testFunc, bodyFunc); // syntax
(
i = 0;
while ({ i < 30 }, {  i = i + 1; i.postln; });
)
```

This is enough about the language. Now is the time to dive into making sounds and explore the synthesis capabilities of SuperCollider. But first let us learn some tricks of peeking under the hood of the SuperCollider language:

Peaking under the hood

Each UGen or Class in SuperCollider has a class definition in a class file. These files are compiled every time SuperCollider is started and become the application environment we are using. SC is an "interpreted" language. (As opposed to a "compiled" language like C or JavaScript). If you add a new class to SuperCollider, you need to *recompile* the language (there is a menu item for that), or simply restart SuperCollider.

Tip

For checking the sourcefile, type Apple + i (or cmd + i) when a class is highlighted (say SinOsc).

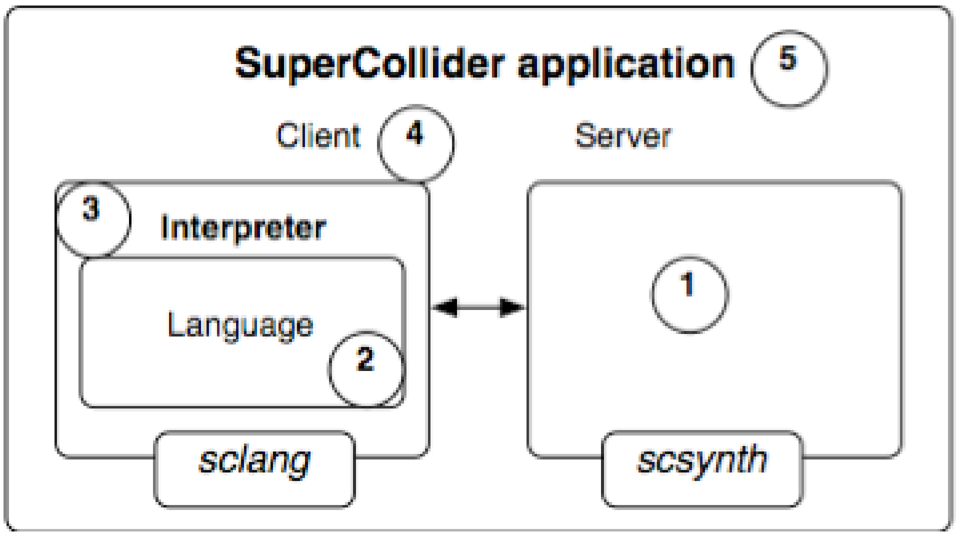
```
UGen.dumpSubclassList // UGen is a class. Try dumping LFSaw for example

UGen.browse  // examine methods interactively in a GUI (OSX)

SinOsc.dumpFullInterface  // list all methods for the classhierarchically
SinOsc.dumpMethodList  // list instance methods alphabetically
SinOsc.openHelpFile
```

Chapter 2 - The SuperCollider Server

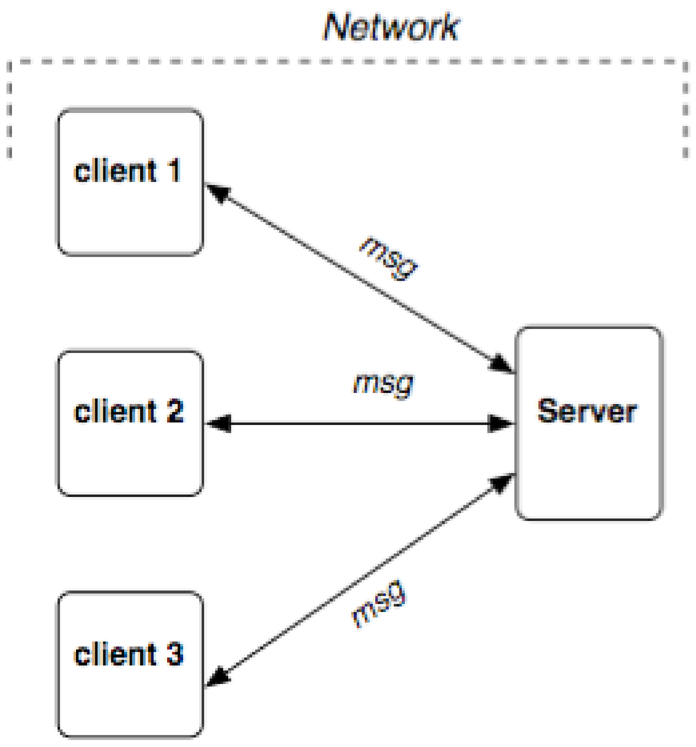
The SuperCollider Server, or SC Synth as it's also known as, is an elegant and great sounding audio engine. As mentioned earlier, SuperCollider is traditionally separated between a server and a client, that is, an audio server (the SC Synth) and the SuperCollider language client (sc-lang). When the server is booted, it connects to the default audio device (such as internal or external audio cards), but you can set it to any audio device available to your computer (for example using virtual audio routing software like Jack).



A figure illustrating the structure of SuperCollider: 1) the server (scsynth); 2) the language; 3) an interpreter for the language; 4) the client for the server; and 5) the SuperCollider IDE - from http://doc.sccode.org/Guides/ClientVsServer.html

The SC Synth renders audio and has an elegant structure of Busses, Groups, Synths and UGens, and it works a bit like a modular synth, where the output of certain chain of oscillators and filters can be routed into another module. The audio is created through creating graphs called *synth definitions* or SynthDefs. These are definitions of synths, but in a wide sense as they can do practically anything audio related (for example performing audio analysis rather than synthesis).

The SC Synth is a program that runs independently from the SuperCollider IDE or language. You can use any software to control it, like C/C++, Java, Python, Lua, Pure Data, Max/MSP or any other.



A figure illustrating how different clients can communicate with the sc server - from http://doc.sccode.org/Guides/ClientVsServer.html

This chapter will introduce the SuperCollider server for the most basic purposes of getting started with this amazing engine for audio work. This section will be fundamental for the succeeding chapters.

Booting the Server

When you "boot the server", you are basically starting a new process on your computer that does not have a GUI (Graphical User Interface). If you observe the list of running processes of your computer, you will see that when you boot the server, a new process will appear (try typing "top" into a Unix Terminal). The server can be booted through a menu command (Menu-> Server -> Boot Server), or through code (s.boot). You can also boot it from the command line if you know where the server is on your system, as it is independent of the SuperCollider application.

The 's' Variable

The 's' variable is a unique variable in SuperCollider, as there is a convention that the SC Server has been assigned to this variable. So never assign anything else to the 's' variable.

```
// let us explore the 's' variable, that stands for the synth:
s.postln; // we see that it contains a localhost synth
s.addr // the address of the synth (IP address and Port)
s.name // the localhost server is the default server (see Main.sc file)
s.serverRunning // is it running?
s.avgCPU // how much CPU is it using right now?

// Let's boot the server. Look at the post window
s.boot
```

We can explore creating our own servers with specific ports and IP addresses:

```
n = NetAddr("127.0.0.1", 57200); // IP (get it from whatsmyip.org) and port
p = Server.new("hoho", n); // create a server with the specific net address
p.makeWindow; // make a GUI window
p.boot; // boot it

// try the server:
{SinOsc.ar(444)}.play(p);
// stop it
p.quit;
```

From the above you might start to think about possibilities of having the server running on a remote computer with various clients communicating to it over network, and yes, that is precisely one of the innovative ideas of SuperCollider 3. You could put any server (with a remote IP address and port) into your server variable and communicate to it over a network. Or have many servers on diverse computers, instructing each of them to render audio. All this is common in SuperCollider practice, but the most common setup is using the SuperCollider IDE to write SC Lang code to control a localhost audio server (localhost meaning "on the same computer"). And that is what we will focus on for a while.

The Unit Generators

Unit Generators have been the key building blocks of digital synthesis systems, since Max Matthews' Music N systems in the 1960s. Written in C++ and compiled as plugins for the SC Server, they encapsulate complex calculations into a simple black box that returns to us - the synth builders or musicians - what we are after, namely an output that could be in the form of a wave or a filter. The Unit Generators, or UGens as they are commonly called, are modular and the output of one can be the input of another. You can think of them like units in a modular synthesizer, for example the Moog:

A Moog Modular Synth

UGens typically have audio rate (.ar) and control rate (.kr) methods. Some have initialization rate as well. The difference here is that an **audio rate** UGen will output as many samples as the sample rate per second. A computer with 44.1kHz sample rate will require each UGen to calculate 44100 samples per second. **Control rate** is of much lower rate than the sample rate and gives the synth designer the possibility of saving computational power (or CPU cycles) if used wisely.

Control Rate

The control rate frequency can be found out by dividing the sample rate with the server's block size. The block size is the number of samples that the server calculates in one go, typically 64 samples, but this can be increased or decreased. So, for example, if your sample rate is 44.1 kHz, and the block size is 64, your control rate will be ~689 Hz (i.e., 44100/64 = 689.0625). This also means that the latency of audio processing using this block size is ~1.5 ms (i.e. 64/44100).

Control rate is used where we are controlling parameters that do not need to be audio rate. For example the frequency of an oscillator. It is perfectly adequate to control frequency changes with control rate, since our ear would not detect any difference whether the resolution was audio rate or not.

```
// Here is a sine wave unit generator
// it has an audio rate method (the .ar)
// and its argument order is frequency, phase and multiplication
{SinOsc.ar(440, 0, 1)}.play
// now try to run a SinOsc with control rate:
{SinOsc.kr(440, 0, 1)}.play // and it is inaudible
```

The control rate SinOsc is inaudible, but it is running fine on the server. We use control rate UGens to **control** other UGens, for example frequency, amplitude, or filter frequency. Let's explore that a little:

```
// A sine wave of 1 Hz modulates the 440 Hz frequency
{SinOsc.ar(440*SinOsc.kr(1), 0, 1)}.play
// A control rate sine wave of 3 Hz modulates the amplitude
{SinOsc.ar(440, 0, SinOsc.kr(3))}.play
// An audio rate sine wave of 3 Hz modulates the amplitude
{SinOsc.ar(440, 0, SinOsc.ar(3))}.play
// and as you can hear, there is no difference

// 2 Hz modulation of the cutoff frequency of a Low Pass Filter (LPF)
// we add 1002, so the filter does not go into negative range
// which might blow up the filter
{LPF.ar(Saw.ar(440), SinOsc.kr(2, 0, 1000)+1002)}.play
```

The beauty of UGens is how one can connect the output of one to the input of another. Oscillator UGens typically output values between -1 and 1, in a certain pattern (e.g., sine wave, saw wave, or square wave) and in a certain frequency. Other UGens such as filters or FFT processing do calculations on an incoming signal and output a new signal. Let's explore one more example of connected UGens that demonstrates their modular power:

```
{
        // we create a slow oscillator in control rate
        a = SinOsc.kr(1);
        // the output of 'a' is used to multiply the frequency of a saw wave
        // resulting in a frequency from 440 to 660. Why?
        b = Saw.ar(220*(a+2), 0.5);
        // and here we use 'a' to control amplitude (from -0.5 to 0.5)
        c = Saw.ar(110, a*0.5);
        // we add b and c, and use a to control the filter cutoff frequency
        // we simply added a .range method to a so it now outputs
        // values between 100 and 2000 at 1 Hz
        d = LPF.ar(b+c, a.range(100, 2000));
        Out.ar(0, Pan2.ar(d, 0));
}.play
```

This is a simple case study of how UGens can be added (b+c), and used in any calculation (such as a*0.5 - which is an amplitude modulation, creating a tremolo effect) of the signal. For a bit of fun, let's try to use a microphone and make a little effect of your voice:

```
{
        // we take sound in from the sound card
        a = SoundIn.ar(0);
        // and we ring modulate using the mouse to control frequency
        b = a * SinOsc.ar(MouseX.kr(100, 3000));
        // we also use the mouse (vertical) to control delay
        c = b + AllpassC.ar(b, 1, MouseY.kr(0.001, 0.2), 2);
        // and here, instead of Pan2, we simply use an array [c, c]
        Out.ar(0, [c, c]);
}.play
```

A good way to explore UGens is to browse them in the documentation.

```
UGen.browse; // XXX check if this works
```

The SynthDef

Above we explored UGens by wrapping them in a function and call .play on that function ({}.play). What this does is to turn the function (indicated by {}, as we learned in the chapter 1) into a synth definition that is sent to the server and then played. The {}.play (or Function:play, if you want to peek into the source code â€“ by highlighting "Function:play" and hit Cmd+I â€“ and explore how SC compiles the function into a SynthDef under the hood) is how many people sketch sound in SuperCollider and it's good for demonstration purposes, but for all real synth building, we need to create a synth definition, or a SynthDef.

A SynthDef is a pre-compiled graph of unit generators. This graph is written to a binary file and sent to the server over OSC (Open Sound Control - See chapter 4). This file is stored in the "synthdefs" folder on your system. In a way you could see it as your own VST plugin for SuperCollider, and you don't need the source code for it to work (although it does not make sense to throw that away).

It is recommended that the SynthDef help file is read carefully and properly understood. The SynthDef is a key class of SuperCollider and very important. It adds synths to the server or writes synth definition files to the disk, amongst many other things. Let's start by exploring how we can turn a unit generator graph function into a synth definition:

```
// this simple synth
{Saw.ar(440)}.play
// becomes equivalent to this synth definition
SynthDef(\mysaw, {
        Out.ar(0, Saw.ar(440));
}).add;
```

You notice that we have done two things: given the function a name (\mysaw), and we've wrapped our saw wave in an 'Out' UGen which defines which 'Bus' the audio is sent to. If you have an 8 channel sound card, you could send audio to any bus from 0 to 7. You could also send it to bus number 20, but we would not be able to hear it then. However, we could put another synth there that routes the audio back onto audio card busses, for example 0-7.

```
// you can use the 'Out' UGen in Function:play
{Out.ar(1, Saw.ar(440))}.play // out on the right speaker
```

NOTE: There is a difference in the Function-play code and the SynthDef, in that we need the Out Ugen in a synth definition to tell the server which audiobus the sound should go out of. (0 is left, 1 is right)

But back to our SynthDef, we can now try to instantiate it, and create a Synth. (A Synth is an instantiation (child) of a SynthDef). This synth can then be controlled if we reference it with a variable.

```
// create a synth and put it into variable 'a'
a = Synth(\mysaw);
// create another synth and put it into variable 'b'
b = Synth(\mysaw);
a.free; // kill a
b.free; // kill b
```

This is obviously not a very interesting synth. It is 'hardcoded', i.e., the parameters in it (such as frequency and amplitude) are static and we can't change them. This is only done in very specific situations, as normally we would like to specify the values of our synth both when initialising the synth and after it has been started.

In order to open the SynthDef up for specified parameters and enabling it to be changed, we need to put arguments into the UGen function graph. Remember in chapter 1 how we created a function with arguments:

```
f = {arg a, b;
        c = a + b;
        postln("c is now: " + c)
};
f.value(2, 3);
```

Note that you cannot write 'f.value', as you will get an error trying to add 'nil' to 'nil' ('a' and 'b' are both nil in the arg slots in the function. To solve that we can give them default values:

```
f = {arg a=2, b=3;
        c = a + b;
        postln("c is now: " + c)
};
f.value(22, 33);
f.value;
```

So we add the arguments for the synthdef, and we add a Pan2 UGen that enables us to pan the sound from the left (-1) to the right (1). The centre is 0:

```
SynthDef(\mysaw, { arg freq=440, amp=0.2, pan=0;
        Out.ar(0, Pan2.ar(Saw.ar(freq, amp), pan));
}).add;
// this now allows us to create a new synth:
a = Synth(\mysaw); // explore the Synth help file
// and control it, using the .set, method of the Synth:
a.set(\freq, 220);
a.set(\amp, 0.8);
a.set(\freq, 555, \amp, 0.4, \pan, -1);
```

This synth definition could be written better and more understandable. Let's say we were to add a filter to the synth, it might look like this:

```
SynthDef(\mysaw, { arg freq=440, amp=0.2, pan=0, cutoff=880, rq=0.3;
        Out.ar(0, Pan2.ar(RLPF.ar(Saw.ar(freq, amp), pan), cutoff, rq));
}).add;
```

But this is starting to be hard to read. Let us make the SynthDef easier to read (although for the computer it is the same, as it only cares about where the semicolons (;) are).

```
// the same as above, but more readable
SynthDef(\mysaw, { arg freq=440, amp=0.2, pan=0, cutoff=880, rq=0.3;
        var signal, filter, panned;
        signal = Saw.ar(freq, amp);
        filter = RLPF.ar(signal, cutoff, rq);
        panned = Pan2.ar(filter, pan);
        Out.ar(0, panned);
}).add;
```

This is roughly how you will write and see other people write synth definitions from now on. The individual parts of a UGen graph are typically put into variables to be more human readable and easier to understand. The exception are SuperCollider tweets (#supercollider) where we have the 280 character limit. We can now explore the synth definition a bit more:

```
a = Synth(\mysaw); // we create a synth with the default arguments
b = Synth(\mysaw, [\freq, 880, \cutoff, 12000]); // we pass arguments
a.set(\cutoff, 500);
b.set(\freq, 444);
a.set(\freq, 1000, \cutoff, 1200);
b.set(\cutoff, 4000);
b.set(\rq, 0.1);
```

Observing server activity (Poll, Scope and FreqScope)

SuperCollider has various ways to explore what is happening on the server, in addition to the most obvious one: sound itself. Due to the separation between the SC server and the sc-lang, this means that data has to be sent from the server and back to the language, since it's the language that prints or displays the data. The server is just a lean mean sound machine and doesn't care about anything else. Firstly we can try to poll (get) the data from a UGen and post it to the post window:

```
// we can explore the output of the SinOsc
{SinOsc.ar(1).poll}.play // you won't be able to hear this
// and compare to white noise:
{WhiteNoise.ar(1).poll}.play // the first arg of noise is amplitude
// we can explore the mouse:
{MouseX.kr(10, 1000).poll}.play // nothing to hear

// we can poll the frequency of a sound:
{SinOsc.ar(LFNoise2.ar(1).range(100, 1000).poll)}.play
// or we poll the amplitude of it
{SinOsc.ar(LFNoise2.ar(1).range(100, 1000)).poll}.play
// and we can add a label (first arg is poll rate, second is label)
{SinOsc.ar(LFNoise2.ar(1).range(100, 1000).poll(10, "freq"))}.play
```

People often use poll to explore what is happening in the synth, to debug, or try to understand why something is not working. But it is typically not used in software that is to be shipped or used in performance as it actually takes some computing power to be sending the messages from the server to the language. Another way to explore the server state is to use scope:

```
// we can explore the output of the SinOsc
{SinOsc.ar(1)}.scope // you won't be able to hear this
// and compare to white noise:
{WhiteNoise.ar(1)}.scope // the first arg of noise is amplitude
// we can scope the mouse state (but note the control rate):
{MouseX.kr(-1, 1)}.scope // nothing to hear
// the range method maps the output from -1 to 1 into 100 to 1000
{SinOsc.ar(LFNoise2.ar(1).range(100, 1000))}.scope;
// same here, we explore the saw wave form at different frequencies
{Saw.ar(220*SinOsc.ar(0.5).range(1, 10))}.scope
```

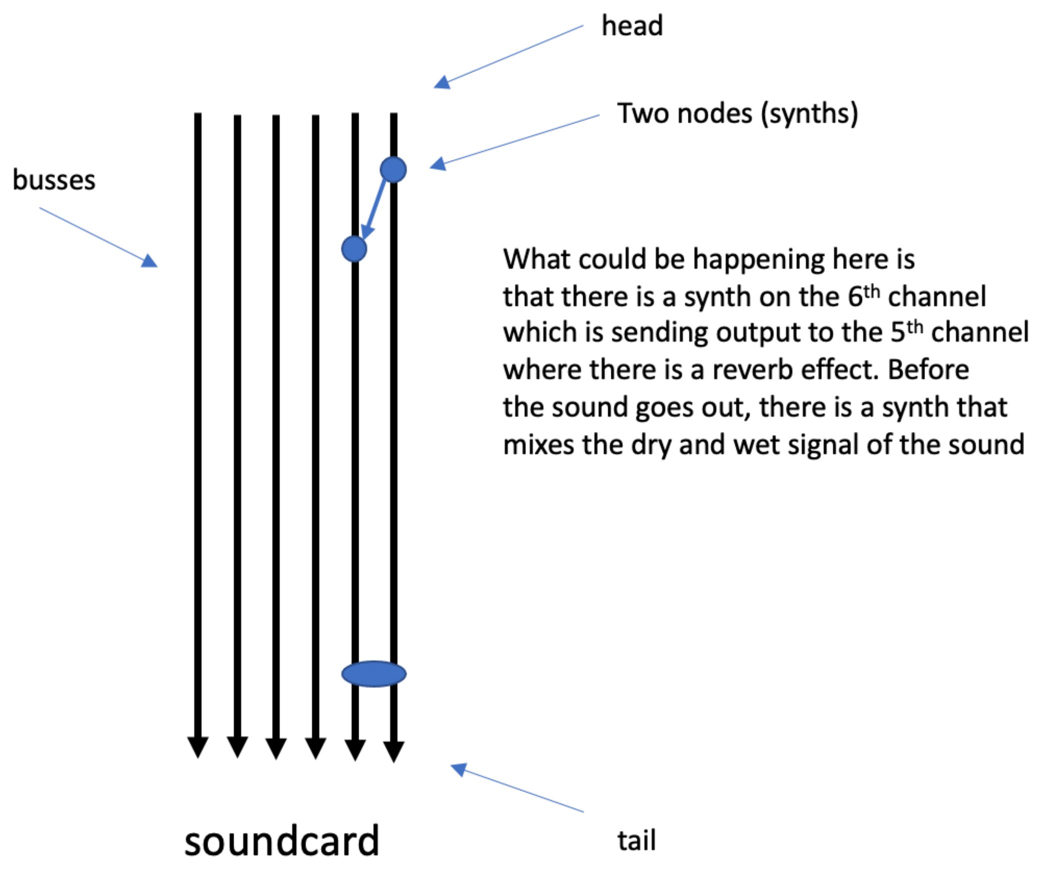
The scope shows amplitude over time, that is: the horizontal axis is **time** and the vertical axis is **amplitude**. This is often called a time-domain view of the signal. But we can also explore the frequency content of the sound, a view we call frequency-domain view. This is achieved by performing an FFT analysis of the signal which is then displayed to the scope (don't worry, this happens 'under the hood' and we'll learn about this in chapter 13). Now let's explore the freqscope:

```
// we see the wave at 1000 Hz, with amplitude modulated
{SinOsc.ar(1000, 0, SinOsc.ar(0.25))}.freqscope
// some white noise again:
{WhiteNoise.ar(1)}.freqscope // random values throughout the spectrum
// and we can now experienc the power of the scope
{RLPF.ar(WhiteNoise.ar(1), MouseX.kr(20, 12000), MouseY.kr(0.01, 0.99))}.freqscope
// we can now explore various wave forms:
{Saw.ar(440*XLine.ar(1, 10, 5))}.freqscope // check the XLine helpfile
// LFTri is a non-bandlimited UGen, so explore the mirroring or 'aliasing'
{LFTri.ar(440*XLine.ar(1, 10, 25))}.freqscope
```

Futhermore, there is a Spectrogram Quark that shows a spectrogram view of the audio signal, but this is not part of the SuperCollider distribution. However, it's easy to install and we will cover this in the chapter on the Quarks.

A quick intro to busses and multichannel expansion

Chapter 14 will go deeper into busses, groups, and how to route the audio signals through the SC Server. However, it is important at this stage to understand how the server works in terms of channels (or busses). Firstly, all oscillators are mono. Many newcomers to SuperCollider find it strange that they only hear a signal in their left ear when using headphones running a SinOsc. Well, it would be strange to have it in stereo, quadrophonic, 5.1 or any other format, unless we specifically ask for that! We therefore need to copy the signal into the next bus if we want stereo. The image below shows a rough sketch of how the sc synth works.


head
Two nodes (synths)
busses
What could be happening here is that there is a synth on the 6th channel which is sending output to the 5th channel where there is a reverb effect. Before the sound goes out, there is a synth that mixes the dry and wet signal of the sound
soundcard
tail

A sketch illustrating busses in the SC Synth

By default SuperCollider has 8 output channels, 8 input channels, and 112 private audio bus channels (where we can run effects and other things). This means that if you have an 8 channel sound card, you can send a signal out on any of the first 8 busses. If you have a 16 channel sound card, you need to enter the ServerOptions class and change the 'numOutputBusChannels' variable to 16. More on that later, but let's now look at some examples:

```
// sound put out on different busses
{ Out.ar(0, LFPulse.ar(220, 0, 0.5, 0.3)) }.play; // left speaker (bus 0)
{ Out.ar(1, LFPulse.ar(220, 0, 0.5, 0.3)) }.play; // right speaker (bus 1)
{ Out.ar(2, LFPulse.ar(220, 0, 0.5, 0.3)) }.play; // third speaker (bus 2)

// Pan2 makes takes the signal and converts it into an array of two signals
{ Out.ar(0, Pan2.ar(PinkNoise.ar(1), 0)) }.scope(8)
// or we can play it out on bus 6 (and you probably won't hear it)
{ Out.ar(0, Pan2.ar(PinkNoise.ar(1), 0)) }.scope(8)
// but the above is the same as:
{ a = PinkNoise.ar(1); Out.ar(0, [a, a]) }.scope(8)
// and (where the first six channels are silent):
{ a = PinkNoise.ar(1); Out.ar(0, [0, 0, 0, 0, 0, 0, a, a]) }.scope(8)
// however, it's not the same as:
{ Out.ar(0, [PinkNoise.ar(1), PinkNoise.ar(1)]) }.scope(8)
```

```
// why not? -> because we now have TWO signals rather than one
```

It is thus clear how the busses of the server are represented by an array containing signals (as in: [signal, signal, signal, signal, etc.]). We can now take a mono signal and ‘expand’ it into other busses. This is called multichannel expansion:

```
{ SinOsc.ar(440) }.scope(8)
{ [SinOsc.ar(440), SinOsc.ar(880)] }.scope(8)
// same as:
{ SinOsc.ar([440, 880]) }.scope(8)
// a trick to 'expand into an array'
{ SinOsc.ar(440) ! 2 }.scope(8)
// if that was strange, check this:
123 ! 30
```

Enough of this. We will explore busses and audio signal routing in chapter 14 later. However, it is important to understand this at the current stage.

Getting values back to the language

As we have discussed, the SuperCollider language and server are two separate applications. They communicate through the OSC protocol. This means that the communication between the two is **asynchronous**, or in other words, that you can’t know precisely how long it takes for a message to arrive. Also, you would not know in which order things will happen if you were to depend on a value from the server in a code block in the language. However, if we would like to do something with audio data in the language, such as visualising it, posting it, or such, we need to send a message to the server and wait for it to respond back. This can happen in various ways, but a typical way of doing this is to use the SendTrig Ugen:

```
// this is happening in the language
OSCdef(\listener, {arg msg, time, addr, recvPort; msg.postln; }, '/tr', n);
// and this happens in the server
{
	var freq;
	freq = LFSaw.ar(0.75, 0, 100, 900);
	SendTrig.kr(Impulse.kr(10), 0, freq);
	SinOsc.ar(freq, 0, 0.5)
}.play
```

What we see above is the SendTrig, sending 10 messages every second to the language (the Impulse triggers those messages). It sends a ‘/tr’ OSC message to port 57120 locally. (Don’t worry, we’ll explore this later in a chapter on OSC). The OSCdef then has a function that posts the message from the server.

```
// this is happening in the language
OSCdef(\listener, {arg msg, time, addr, recvPort; msg.postln; }, '/tr', n);
// and this happens on the server
{
        var freq;
        freq = LFSaw.ar(0.75, 0, 100, 900);
        SendTrig.kr(Impulse.kr(10), 0, freq);
        SinOsc.ar(freq, 0, 0.5)
}.play
```

A little bit more complex example might involve a GUI (Graphical User Interfaces are part of the language) and synthesis on the server:

```
(
// this is happening in the language
var win, freqslider, mouseslider;
win = Window.new.front;
freqslider = Slider(win, Rect(20, 10, 40, 280));
mouseslider = Slider2D(win, Rect(80, 10, 280, 280));

OSCdef(\sliderdef, {arg msg, time, addr, recvPort;
        {freqslider.value_(msg[3].linlin(600, 1400, 0, 1))}.defer;
}, '/slider', n); // the OSC message we listen to
OSCdef(\sliderdef2D, {arg msg, time, addr, recvPort;
        { mouseslider.x_(msg[3]); mouseslider.y_(msg[4]); }.defer;
}, '/slider2D', n); // the OSC message we listen to

// and this happens on the server
{
        var mx, my, freq;
        freq = LFSaw.ar(0.75, 0, 400, 1000); // outputs 600 to 1400 Hz. Why?
        mx = LFNoise2.kr(2).range(0,1);
        my = LFNoise2.kr(2).range(0, 1);
        SendReply.kr(Impulse.kr(10), '/slider', freq); // sending the OSC message
        SendReply.kr(Impulse.kr(10), '/slider2D', [mx, my]);
        (SinOsc.ar(freq, 0, 0.5)+RLPF.ar(WhiteNoise.ar(0.3), mx.range(100, 3000), my))!2 ;
}.play;
 )
```

We could also write values to a control bus on the server, from which we can read in the language. Here is an example:

```
b = Bus.control(s,1); // we create a control bus
{Out.kr(b, MouseX.kr(20,22000))}.play // and we write the output of some UGen to the bus
b.get({arg val; val.postln;}); // we poll the puss from the language
// or even:
fork{loop{ b.get({arg val; val.postln;});0.1.wait; }}
```

Check the source of Bus (by hitting Cmd+I) and locate the .get method. You will see that the Bus .get method is using an OSCresponder underneath. It is therefore “asynchronous”, meaning that it will not happen in the linear order of your code. (The language is asking server for the value, and the server then sends back to language. This takes time).

Here is a program that demonstrates the asynchronous nature of b.get. The {}.play from above has to be running. Note how the numbered lines of code appear in the post window “in the wrong order”! (Instead of a synchronous posting of 1, 2 and 3, we get the order of 1, 3 and 2). It takes between 0.1 and 10 milliseconds to get the value on a 2.8 GHz Intel computer.

```
(
x = 0; y= 0;
b = Bus.control(s,1); // we create a control bus
{Out.kr(b, MouseX.kr(20,22000))}.play;
t = Task({
        inf.do({
                "1 - before b.get : ".post; x = Main.elapsedTime.postln;
                b.get({|val|
                        "2 - ".post; val.postln;
                        y = Main.elapsedTime.postln;
                        "the asynchronious process took : ".post; (y-x).post; " seconds".postln;
                }); //  this value is returned AFTER the next line
                "3 - after b.get : ".post;  Main.elapsedTime.postln;
                0.5.wait;
        })
}).play;
)
```

This type of communication from the server to the language is not very common. The other way (from language to server) is however. This section is therefore not vital for your work in SuperCollider, but you will at some point stumble into the question of synchronous and asynchronous communication with the server and this section should prepare you for that.

ProxySpace

SuperCollider is an extremely wide and flexible language. It is profoundly deep and you will find new things to explore for years to come. Typically SC users find their own way of working in the language and then explore new areas when they find they need so, or are curious.

ProxySpace is one such area. It makes live coding and various on line coding extremely flexible. Effects can be routed in and out of proxies, and source changed. Below you will find a quick examples that are useful when testing UGens or making prototypes for synths that you will write as synthdefs later. ProxySpace is also often used in live coding. Evaluate the code below line by line:

```
p= ProxySpace.push(s.boot)

~signal.play;
~signal.fadeTime_(2) // fading in and out in 2 secs
~signal= {SinOsc.ar(400, 0, 1)!2}
~signal= {SinOsc.ar([400, 404], 0, LFNoise0.kr(4))}
~signal= {Saw.ar([400, 404],  LFNoise0.kr(4))}
~signal= {Saw.ar([400, 404],  Pulse.ar(2))}
~signal= {Saw.ar([400, 404],  Pulse.ar(Line.kr(1, 30, 20)))}
~signal= {LFSaw.ar([400, 404],  LFNoise0.kr(4))}
~signal= {Pulse.ar([400, 404],  LFNoise0.kr(4))}
~signal= {Blip.ar([400, 404],  12, Pulse.ar(2))}
~signal= {Blip.ar([400, 404],  24, LFNoise0.kr(4))}
~signal= {Blip.ar([400, 404],  4, LFNoise0.kr(4))}
~signal= {Blip.ar([400, 404],  MouseX.kr(4, 40), LFNoise0.kr(4))}
~signal= {Blip.ar([200, 204],  5, Pulse.ar(1))}

// now let's try to add some effects

~signal[1] = \filter -> {arg sig; (sig*0.6)+FreeVerb.ar(sig, 0.85, 0.86, 0.3)}; // reverb
~signal[2] = \filter -> {arg sig; sig + AllpassC.ar(sig, 1, 0.15, 1.3 )}; // delay
~signal[3] = \filter -> {arg sig; (sig * SinOsc.ar(2.1, 0, 5.44, 0))*0.5}; // tremolo
~signal[4] = \filter -> {arg sig; PitchShift.ar(sig, 0.008, SinOsc.ar(2.1, 0, 0.11, 1))}; /\
/ pitchshift
~signal[5] = \filter -> {arg sig; (3111.33*sig.distort/(1+(2231.23*sig.abs))).distort*0.2};\
 // distort
~signal[1] = nil;
~signal[2] = nil;
~signal[3] = nil;
~signal[4] = nil;
~signal[5] = nil;
```

Another ProxySpace example:

```
p = ProxySpace.push(s.boot);
~blipper = { |freq=20, nHarm=30, amp=0.1| Blip.ar(freq, nHarm, amp)!2 };
~blipper.play;
~lfo = { MouseX.kr(10, 100, 1) };
~blipper.map(\freq, ~lfo);
~blipper.set(\nHarm, 50)
~lfn = { LFDNoise3.kr(15, 30, 40) };
~blipper.map(\nHarm, ~lfn);
~lfn = 30;
~blipper.set(\nHarm, 50);
```

Ndef

Ndef is an alternative and more dynamic way of working than using SynthDefs. They can be rewritten on the fly whilst running. They are using the ProxySpace like the code above. Example (from the documentation) here below:

```
Ndef(\sound).play;
Ndef(\sound).fadeTime = 1;
Ndef(\sound, { SinOsc.ar([600, 635], 0, SinOsc.kr(2).max(0) * 0.2) });
Ndef(\sound, { SinOsc.ar([600, 635] * 3, 0, SinOsc.kr(2 * 3).max(0) * 0.2) });
Ndef(\sound, { SinOsc.ar([600, 635] * 2, 0, SinOsc.kr(2 * 3).max(0) * 0.2) });
Ndef(\sound, Pbind(\dur, 0.17, \freq, Pfunc({ rrand(300, 700) })) );

Ndef(\lfo, { LFNoise1.kr(3, 400, 800) });
Ndef(\sound).map(\freq, Ndef(\lfo));
Ndef(\sound, { arg freq; SinOsc.ar([600, 635] + freq, 0, SinOsc.kr(2 * 3).max(0) * 0.2) });
Ndef(\lfo, { LFNoise1.kr(300, 400, 800) });

Ndef.clear; //clear all Ndefs
```