A Gentle Introduction to SuperCollider

by Bruno Ruviaro

Contents

A Gentle Introduction to SuperCollider

Bruno Ruviaro

August 23, 2014

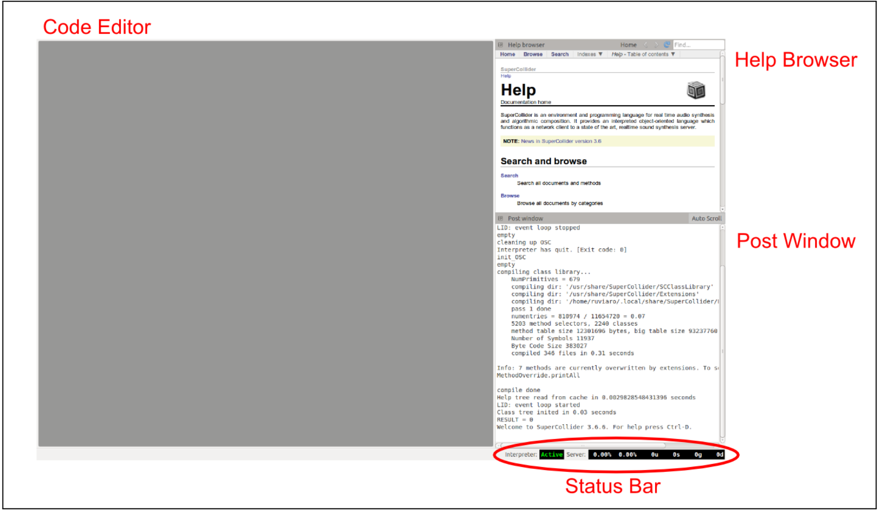
Part I
BASICS

1 Hello World

Ready for creating your first SuperCollider program? Assuming you have SC up and running in front of you, open a new document (menu File→New, or shortcut [ctrl+N]) and type the following line:

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

Leave your cursor anywhere in that line (in doesn't matter if beginning, middle, or end). Press [ctrl+Enter] to evaluate the code. "Hello world" appears in the Post window. Congratulations! That was your first SuperCollider program.



SuperCollider
Help

Help

Documentation home

SuperCollider is an environment and programming language for real time audio synthesis and algorithmic composition. It provides an interpreted object-oriented language which functions as a network client to a state of the art, realtime sound synthesis server.

NOTE: News in SuperCollider version 3.6

Search and browse

Search
Search all documents and methods

Browse
Browse all documents by categories

```
LID: event loop stopped
empty
cleaning up OSC
Interpreter has quit. [Exit code: 0]
init_OSC
empty
compiling class library...
    NumPrimitives = 679
    compiling dir: '/usr/share/SuperCollider/SCClassLibrary'
    compiling dir: '/usr/share/SuperCollider/Extensions'
    compiling dir: '/home/ruviaro/.local/share/SuperCollider/
    pass 1 done
    numentries = 810974 / 11654720 = 0.07
    5203 method selectors, 2240 classes
    method table size 12301696 bytes, big table size 93237760
    Number of Symbols 11937
    Byte Code Size 383027
    compiled 346 files in 0.31 seconds

Info: 7 methods are currently overwritten by extensions. To s
MethodOverride.printAll

compile done
Help tree read from cache in 0.0029828548431396 seconds
LID: event loop started
Class tree inited in 0.03 seconds
RESULT = 0
Welcome to SuperCollider 3.6.6. For help press Ctrl-D.
```

Figure 1: SuperCollider IDE interface.

TIP: Throughout this document, ctrl (control) indicates the modifier key for keyboard shortcuts that is used on Linux and Windows platforms. On Mac OSX, use cmd (command) instead.

Figure 1 shows a screenshot of the SuperCollider IDE (Integrated Development Environment) when you first open it. Let's take a moment to get to know it a bit.

What is the SuperCollider IDE? It is "a cross-platform coding environment developed specifically for SuperCollider (. . .), easy to start using, handy to work with, and sprinkled with powerful features for experienced coders. It is also very customizable. It runs equally well and looks almost the same on Mac OSX, Linux and Windows."[*]

The main parts you see on the SC window are the Code Editor, the Help Browser, and Post Window. If you don't see any of these when you open SuperCollider, simply go to the menu View→Docklets (that's where you can show or hide each of them). There is also the Status Bar, always located on the bottom right corner of the window.

Always keep the Post window visible, even if you don't understand yet all the stuff being printed there. The Post window displays the responses of the program to our commands: results of code evaluation, various notifications, warnings, errors, etc.

2 Server and Language

On the Status Bar you can see the words "Interpreter" and "Server." The Interpreter starts up turned on by default ("Active"), while "Server" is turned off (that's what all the zeros mean). What is the Interpreter, and what is the Server?

SuperCollider is actually made of two distinct applications: the server and the language. The server is responsible for making sounds. The language (also referred to as *client* or *interpreter*) is used to control the server. The first is called scsynth (SC-synthesizer), the second sclang (SC-language). The Status Bar tell us the status (on/off) of each one of these two components.

Don't worry if this distinction does not make much sense for you just now. The two main things you need to know at this point are:

1. Everything that you type in SuperCollider is in the SuperCollider language (the client): that's where you write and execute commands, and see results in the Post window.

[*]Quoted from the SuperCollider Documentation: `http://doc.sccode.org/Guides/SCIde.html`. Visit that page to learn more about the IDE interface.

2. Everything that makes sound in SuperCollider is coming from the server—the "sound engine", so to speak—, controlled by you through the SuperCollider language.

2.1 Booting the Server

Your "Hello World" program produced no sound: everything happened in the language, and the server was not used at all. The next example will make sound, so we need to make sure the Server is up and running.

The easiest way to boot the server is with the shortcut [ctrl+B]. Alternatively, you can also click on the zeros in the Status Bar: a menu pops up, and one of the options is "Boot Server." You will see some activity in the Post window as the server boots up. After you have successfully started the server, the numbers on the Status Bar will turn green. You will have to do this each time you launch SC, but only once per session.

3 Your first sine wave

"Hello World" is traditionally the first program that people create when learning a new programming language. You've already done that in SuperCollider.

Creating a simple sine wave might be the "Hello World" of computer music languages. Let's jump right to it. Type and evaluate the following line of code. Careful—this can be loud. Bring your volume all the way down, evaluate the line, then increase the volume slowly.

```
{SinOsc.ar}.play;
```

That's a beautiful, smooth, continuous, and perhaps slightly boring, sine wave. You can stop the sound with [ctrl+.] (That's the *control* key plus the *period* key.) Memorize this key combination, because you will be using it a lot to stop any and all sounds in SC. Now let's make this sine wave a bit more interesting. Type this:

```
{SinOsc.ar(LFNoise0.kr(10).range(500, 1500), mul: 0.1)}.play;
```

Remember, you just need to leave your cursor anywhere within the line and hit [ctrl+Enter] to evaluate. Alternatively, you could also select the entire line before evaluating it.

TIP: Typing the code examples by yourself is a great learning tool. It will help you to build confidence and familiarize yourself with the language. When reading tutorials in digital format, you may be tempted to simply copy and paste short snippets of code from the examples. That's fine, but you will learn more if you type it up yourself—try that at least in the first stages of your SC learning.

4 Error messages

No sound when you evaluated the last example? If so, your code probably had a typo: a wrong character, a missing comma or parenthesis, etc. When something goes wrong in your code, the Post window gives you an error message. Error messages can be long and cryptic, but don't panic: with time you will learn how to read them. A short error message could look like this:

```
ERROR: Class not defined.
  in file 'selected text'
  line 1 char 19:

  {SinOsc.ar(LFNoiseO.kr(12).range(400, 1600), mul: 0.01)}.play;
```

```
-----------------------------------
nil
```

This error message says, "Class not defined," and points to the approximate location of the error ("line 1 char 19"). Classes in SC are those blue words that start with a capital letter (like `SinOsc` and `LFNoise0`). It turns out this error was due to the user typing LFNoiseO with a capital letter "O" at the end. The correct class is LFNoise0, with the number zero at the end. As you can see, attention to detail is crucial.

If you have an error in your code, proofread it, change as needed, and try again until it's fixed. If you had no error at first, try introducing one now so you can see how the error message would look like (for example, remove a comma).

TIP: Learning SuperCollider is like learning another language like German, Portuguese, Japanese... just keep trying to speak it, work on expanding your vocabulary, pay attention to grammar and syntax, and learn from your mistakes. The worst it can happen here is to crash SuperCollider. Not nearly as bad as taking the wrong bus in São Paulo because of a mispronounced request for directions.

5 Changing parameters

Here's a nice example adapted from the first chapter of the SuperCollider book.[*] As with the previous examples, don't worry trying to understand everything. Just enjoy the sound result

[*]Wilson, S. and Cottle, D. and Collins, N. (Editors). The SuperCollider Book, MIT Press, 2011, p. 5. Several things in the present tutorial were borrowed, adapted from, or inspired by David Cottle's excellent "Beginner's Tutorial," which is the first chapter of the book. This tutorial borrows some examples and explanations from Cottle's chapter, but—differently from it—assumes less exposure to computer music, and introduces the Pattern family as the backbone of the pedagogical approach.

and play with the numbers.

```
{RLPF.ar(Dust.ar([12, 15]), LFNoise1.ar([0.3, 0.2]).range(100, 3000), 0.02)}.play;
```

Stop the sound, change some of the numbers, and evaluate again. For example, what happens when you replace the numbers 12 and 15 with lower numbers between 1 and 5? After `LFNoise1`, what if instead of 0.3 and 0.2 you tried something like 1 and 2? Change them one at a time. Compare the new sound with the previous sound, listen to the differences. See if you can understand what number is controlling what. This is a fun way of exploring SuperCollider: grab a snippet of code that makes something interesting, and mess around with the parameters to create variations of it. Even if you don't fully understand the role of every single number, you can still find interesting sonic results.

TIP: Like with any software, remember to frequently save your work with [ctrl+S]! When working on tutorials like this one, you will often come up with interesting sounds by experimenting with the examples provided. When you want to keep something you like, copy the code onto a new document and save it. Notice that every SuperCollider file has the extension .scd, which stands for "SuperCollider Document."

6 Comments

All text in your code that shows in red color is a *comment*. If you are new to programming languages, comments are a very useful way to document your code, both for yourself and for others who may have to read it later. Any line that starts with a double slash is a comment. You can write comments right after a valid line of code; the comment part will be ignored when you evaluate. In SC we use a semicolon to indicate the end of a valid statement.

```
2 + 5 + 10 — 5; // just doing some math

rrand(10, 20); // generate a random number between 10 and 20
```

You can evaluate a line even if your cursor is in the middle of the comment after that line. The comment part is ignored. The next two paragraphs will be written as "comments" just for the sake of the example.

```
// You can quickly comment out one line of code using the shortcut [ctrl+/].
"Some SC code here...".postln;
2 + 2;

// If you write a really long comment, your text may break into what looks like a
    new line that does *not* start with a double slash. That still counts as a
    single line of comment.

/* Use "slash + asterisk" to start a longer comment with several lines. Close the
    big comment chunk with "asterisk + slash." The shortcut mentioned above also
    works for big chunks: simply select the lines of code you want to comment out,
    and hit [ctrl+/]. Same to un—comment. */
```

7 Precedence

SuperCollider follows a left to right order of precedence, regardless of operation. This means, for example, that multiplication does *not* happen first:

```
// In high school, the result was 9; in SC, it is 14:
5 + 2 * 2;
// Use parentheses to force a specific order of operations:
5 + (2 * 2); // equals 9.
```

When combining messages and binary operations, messages take precedence. For example, in `5 + 2.squared`, the squaring happens first.

8 The last thing always gets posted

A small but useful detail to understand: SuperCollider, by default, always posts to the Post window the result of whatever was the *last thing to be evaluated*. This explains why your Hello World code prints twice when you evaluate. Type the following lines onto a new document, then select all with [ctrl+A] and evaluate all lines at once:

```
"First Line".postln;
"Second Line".postln;
(2 + 2).postln;
3 + 3;
"Finished".postln;
```

All five lines are executed by SuperCollider. You see the result of `2 + 2` in the Post window because there was an explicit `postln` request. The result of `3 + 3` was calculated, but there was no request to post it, so you don't see it. Then the command of the last line is executed (the word "Finished" gets posted due to the `postln` request). Finally, the result of the very last thing to be evaluated is posted by default: in this case, it happened to be the word "Finished."

9 Code blocks

Selecting multiple lines of code before evaluating can be tedious. A much easier way of running a chunk of code all at once is by creating a *code block*: simply enclose in parentheses all lines of code that you want to run together. Here's an example:

```
(
// A little poem
"Today is Sunday".postln;
"Foot of pipe".postln;
"The pipe is made of gold".postln;
"It can beat the bull".postln;
)
```

The outer parentheses are delimiting the code block. As long as your cursor is anywhere within the parentheses, a single [ctrl+Enter] will evaluate all lines for you (they are executed in order from top to bottom, but it's so fast that it seems simultaneous).

Using code blocks saves you the trouble of having to select all the lines again every time you change something and want to re-evaluate. For example, change some of the words between double quotes, and hit [ctrl+Enter] right after making the change. The entire block of code is evaluated without you having to manually select all lines. SuperCollider highlights the block for a second to give you a visual hint of what's being executed.

10 How to clean up the Post window

This is such a useful command for cleaning freaks that it deserves a section of its own: [ctrl+shift+P]. Evaluate this line and enjoy cleaning the Post window afterwards:

```
100.do({"Print this line over and over...".scramble.postln});
```

You're welcome.

11 Recording the output of SuperCollider

Soon you will want to start recording the sound output of your SuperCollider patches. Here's a quick way:

```
// QUICK RECORD
// Start recording:
s.record;
// Make some cool sound
{Saw.ar(LFNoise0.kr([2, 3]).range(100, 2000), LFPulse.kr([4, 5]) * 0.1)}.play;
// Stop recording:
s.stopRecording;
// Optional: GUI with record button, volume control, mute button:
s.makeWindow;
```

The Post Window shows the path of the folder where the file was saved. Find the file, open it with Audacity or similar program, and verify that the sound was indeed recorded. For more info, look at the "Server" Help file (scroll down to "Recording Support"). Also online at `http://doc.sccode.org/Classes/Server.html`.

12 Variables

You can store numbers, words, unit generators, functions, or entire blocks of code in variables. Variables can be single letters or whole words chosen by you. We use the equal sign (=) to "assign" variables. Run these lines one at a time and watch the Post window:

```
x = 10;
y = 660;
y; // check what's in there
x;
```

```
x + y;
y - x;
```

The first line assigns the number 10 to the variable `x`. The second line puts 660 into the variable `y`. The next two lines prove that those letters now "contain" those numbers (the data). Finally, the last two lines show that we can use the variables to do any operations with the data.

Lowercase letters `a` through `z` can be used anytime as variables in SuperCollider. The only single letter that by convention we don't use is `s`, which by default represents the Server. Anything can go into a variable:

```
a = "Hello, World"; // a string of characters
b = [0, 1, 2, 3, 5]; // a list
c = Pbind(\note, Pwhite(0, 10), \dur, 0.1); // you'll learn all about Pbind later,
    don't worry

// ...and now you can use them just like you would use the original data:
a.postln; // post it
b + 100; // do some math
c.play; // play that Pbind
d = b * 5; // take b, multiply by 5, and assign that to a new variable
```

Often it will make more sense to give better names to your variables, to help you remember what they stand for in your code. You need to use a ~ (tilde) to declare a variable with a longer name. Note that there is no space between the tilde and the name of the variable.

```
~myFreqs = [415, 220, 440, 880, 220, 990];
~myDurs = [0.1, 0.2, 0.2, 0.5, 0.2, 0.1];

Pbind(\freq, Pseq(~myFreqs), \dur, Pseq(~myDurs)).play;
```

Variable names must begin with lowercase letters. You can use numbers, underscores, and uppercase letters within the name, just not as the first character. All characters must be contiguous (no spaces or punctuation). In short, stick to letters and numbers and the occasional underscore, and avoid all other characters when naming your variables. `~myFreqs`, `~theBestSineWave`, and `~banana_3` are valid names. `~MyFreqs`, `~theBest&*#SineWave`, and `~banana!!!` are bad names.

There are two types of variables that you can create: *environment* variables and *local* variables.

12.1 Environment Variables vs. Local Variables

The ones you have seen up to now are environment variables (some people loosely call them "global variables"): they are the single lowercase letters `a` through `z`, or those starting with the tilde (~) character. Once declared, they will work anywhere in the patch, in other patches, even in other SC documents, until you quit SuperCollider.

Local variables, on the other hand, are declared with the reserved keyword `var` at the beginning of the line. You can assign an initial value to a variable at declaration time (`var apples = 4`). Local variables only exist within the scope of that code block.

Here's a simple example comparing the two types of variables. Evaluate line by line and watch the Post window.

```
// Environment variables
~galaApples = 4;
~bloodOranges = 5;
~limes = 2;
~plantains = 1;

["Citrus", ~bloodOranges + ~limes];
```

```
["Non-citrus", ~plantains + ~galaApples];

// Local variables: valid only within the code block.
// Evaluate the block once and watch the Post window:
(
var apples = 4, oranges = 3, lemons = 8, bananas = 10;
["Citrus fruits", oranges + lemons].postln;
["Non-citrus fruits", bananas + apples].postln;
"End".postln;
)

~galaApples; // still exists
apples; // gone
```

12.2 Reassignment

One last useful thing to understand about variables is that they can be *reassigned*: you can give them a new value at anytime.

```
// Assign a variable
a = 10 + 3;
a.postln; // check it
a = 999; // reassign the variable (give it a new value)
a.postln; // check it: the old value is gone.
```

A very common practice that is sometimes confusing for beginners is when *the variable itself is used in its own reassignment.* Take a look at this example:

```
x = 10; // assign 10 to the variable x
x = x + 1; // assign x + 1 to the variable x
x.postln; // check it
```

The easiest way to understand that last line is to read it like this: “take the current value of variable x, add 1 to it, and assign this new result to the variable x.” It’s really not complicated, and you will see later on how this can be useful.[*]

[*]This example clearly demonstrates that the equal sign, in programming, is not the same equal sign that you learned in mathematics. In math, $x = x + 1$ is impossible (a number cannot be equal to itself plus one). In a programming language like SuperCollider, the equal sign can be seen as a kind of action: *take the result of the expression on the right side of the sign, and “assign it” to the variable on the left side.*

Part II
PATTERNS

13 The Pattern family

Let's try something different now. Type and run this line of code:

```
Pbind(\degree, Pseries(0, 1, 30), \dur, 0.05).play;
```

13.1 Meet Pbind

`Pbind` is a member of the Pattern family in SuperCollider. The capital P in `Pbind` and `Pseries` stands for *Pattern*; we'll meet other members of the family soon. For now, let's take a closer look at `Pbind` only. Try this stripped down example:

```
Pbind(\degree, 0).play;
```

The only thing this line of code does in life is to play the note *middle C*, one time per second. The keyword `\degree` refers to scale degrees, and the number 0 means the first scale degree (a C major scale is assumed, so that's the note C itself). Note that SuperCollider starts counting things from 0, not from 1. In a simple line like the above, the notes C, D, E, F, G... would be represented by the numbers 0, 1, 2, 3, 4... Try changing the number and notice how the note changes when you re-evaluate. You can also choose notes below middle C by using negative numbers (for example, -2 will give you the note A below middle C). In short, just imagine that the middle C note of the piano is 0, and then count white keys up or down (positive or negative numbers) to get any other note.

Now play around a bit with the duration of the notes. Pbind uses the keyword \dur to specify durations in seconds:

```
Pbind(\degree, 0, \dur, 0.5).play;
```

Of course this is still very rigid and inflexible—always the same note, always the same duration. Don't worry: things will get better very soon. But first let's take a look at the other ways you can specify pitch inside a Pbind.

13.2 Pseq

Let's go ahead and play several notes in sequence, like a scale. Let's also make our notes shorter, say, 0.2 second long.

```
Pbind(\degree, Pseq([0, 1, 2, 3, 4, 5, 6, 7], 1), \dur, 0.2).play;
```

This line introduces a new member of the Pattern family: Pseq. As the name might suggest, this pattern deals with sequences. All that Pseq needs in order to play a sequence is:

- a list of items between square brackets
- a number of repetitions.

In the example, the list is [0, 1, 2, 3, 4, 5, 6, 7], and the number of repeats is 1. This Pseq simply means: "play once all the items of the list, in sequence." Notice that these two elements, list and number of repeats, are inside Pseq's own parentheses, and they are separated by a comma.

Also notice where Pseq appears within the Pbind: it is the input value of \degree. This is important: instead of providing a single, fixed number for scale degree (as in our first simple Pbind), *we are providing a whole Pseq: a recipe for a sequence of numbers.* With this in mind, we can easily expand upon this idea and use another Pseq to control durations as well:

```
Pbind(\degree, Pseq([0, 1, 2, 3, 4, 5, 6, 7], 5), \dur, Pseq([0.2, 0.1, 0.1, 0.2,
    0.2, 0.35], inf)).play;
```

What is happening in this example? First, we have changed the number of repeats of the first `Pseq` to 5, so the entire scale will play five times. Second, we have replaced the previously fixed `\dur` value of 0.2 with another `Pseq`. This new `Pseq` has a list of six items: `[0.2, 0.1, 0.1, 0.2, 0.2, 0.35]`. These numbers become duration values for the resulting notes. The `repeats` value of this second `Pseq` is set to `inf`, which stands for "infinite." This means that the `Pseq` has no limit on the number of times it can repeat its sequence. Does the `Pbind` play forever, then? No: it stops after the *other* `Pseq` has finished its job, that is, after the sequence of scale degrees has been played 5 times.

Finally, the example has a total of eight different notes (the list in the first `Pseq`), while there are only six values for duration (second `Pseq`). When you provide sequences of different sizes like this, `Pbind` simply cycles through them as needed.

Answer these questions to practice what you have learned:

- Try the number 1 instead of inf as the `repeats` argument of the second `Pseq`. What happens?
- How can you make this Pbind play forever?

Solutions are at the end of the book.[1]

13.3 Make your code more readable

You may have noticed that the line of code above is quite long. In fact, it is so long that it wraps to a new line, even though it is technically a single statement. Long lines of code can be confusing to read. To avoid this, it is common practice to break the code into several indented lines; the goal is to make it as clear and intelligible as possible. The same `Pbind` above can be written like this:

```
(
Pbind(
        \degree, Pseq([0, 1, 2, 3, 4, 5, 6, 7], 5),
        \dur, Pseq([0.2, 0.1, 0.1, 0.2, 0.2, 0.35], inf)
).play;
)
```

From now on, get into the habit of writing your `Pbind`s like this. Writing code that looks neatly arranged and well organized will help you a lot in learning SuperCollider.

Also, notice that we enclosed this `Pbind` within parentheses to create a code block (remember section 9?): because it is no longer in a single line, we need to do this to be able to run it all together. Just make sure the cursor is anywhere within the block before evaluating.

13.4 Four ways of specifying pitch

`Pbind` accepts other ways of specifying pitch, not just scale degrees.

- If you want to use all twelve chromatic notes (black and white keys of the piano), you can use `\note` instead of `\degree`. 0 will still mean middle C, but now the steps include black keys of the piano (0=middle C, 1=C♯, 2=D, etc).
- If you prefer to use MIDI note numbering, use `\midinote` (60=middle C, 61=C♯, 62=D, etc).
- Finally, if you'd rather specify frequencies directly in Herz, use `\freq`.

See Figure 2 for a comparison of all four methods.
In the next example, the four Pbinds all play the same note: the A above middle C (A4).

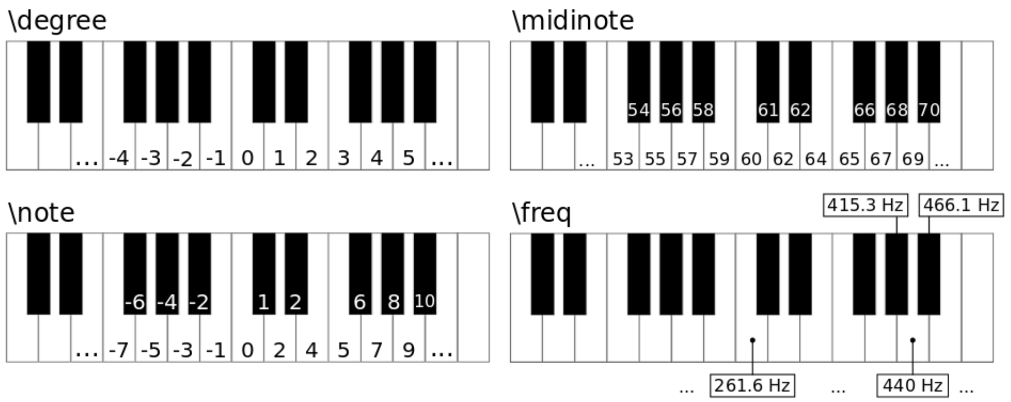


Figure 2: Comparing scale degrees, note numbers, midinotes, and frequencies

```
Pbind(\degree, 5).play;
Pbind(\note, 9).play;
Pbind(\midinote, 69).play;
Pbind(\freq, 440).play;
```

TIP: Remember that each type of pitch specification expects numbers in a different sensible range. A list of numbers like `[-1, 0, 1, 3]` makes sense for `\degree` and `\note`, but doesn't make sense for `\midinote` nor `\freq`. The table below compares some values using the piano keyboard as a reference.

	A0 (lowest piano note)	C4	A4	C5	C8 (highest piano note)
`\degree`	-23	0	5	7	21
`\note`	-39	0	9	12	48
`\midinote`	21	60	69	72	108
`\freq`	27.5	261.6	440	523.2	4186

13.5 More keywords: amplitude and legato

The next example introduces two new keywords: `\amp` and `\legato`, which define the amplitude of events, and the amount of legato between notes. Notice how the code is fairly easy to read thanks to nice indentation and spread over multiple lines. Enclosing parentheses (top and bottom) are used to delimit a code block for quick execution.

```
(
Pbind(
        \degree, Pseq([0, −1, 2, −3, 4, −3, 7, 11, 4, 2, 0, −3], 5),
        \dur, Pseq([0.2, 0.1, 0.1], inf),
        \amp, Pseq([0.7, 0.5, 0.3, 0.2], inf),
        \legato, 0.4
).play;
)
```

`Pbind` has many of these pre-defined keywords, and with time you will learn more of them. For now, let's stick to just a few—one for pitch (choose from `\degree`, `\note`, `\midinote`, or `\freq`), one for durations (`\dur`), one for amplitude (`\amp`), and one for legato (`\legato`). Durations are in beats (in this case, 1 beat per second, which is the default); amplitude should be between 0 and 1 (0 = silence, 1 = very loud); and legato works best with values between 0.1 and 1 (if you are not sure about what legato does, simply try the example above with 0.1, then 0.2, then 0.3, all the way up to 1, and hear the results).

Take the last example as point of departure and create new `Pbind`s. Change the melody. Come up with new lists of durations and amplitudes. Experiment using `\freq` for pitch. Remember, you can always choose to use a fixed number for any given parameter, if that's what you need. For example, if you want all notes in your melody to be 0.2 seconds long, there is no need to write `Pseq[0.2, 0.2, 0.2, 0.2...`, not even `Pseq([0.2], inf)`—simply remove the whole `Pseq` structure and write 0.2 in there.

13.6 Prand

`Prand` is a close cousin of `Pseq`. It also takes in a list and a number of repeats. But instead of playing through the list in sequence, `Prand` *picks a random item from the list every time*. Try it:

```
(
Pbind(
        \degree, Prand([2, 3, 4, 5, 6], inf),
        \dur, 0.15,
        \amp, 0.2,
        \legato, 0.1
).play;
)
```

Replace `Prand` with `Pseq` and compare the results. Now try using `Prand` for durations, amplitudes, and legato.

13.7 Pwhite

Another popular member of the Pattern family is `Pwhite`. It is an equal distribution random number generator (the name comes from "white noise"). For example, `Pwhite(100, 500)` will get you random numbers between 100 and 500.

```
(
Pbind(
	\freq, Pwhite(100, 500),
	\dur, Prand([0.15, 0.25, 0.3], inf),
	\amp, 0.2,
	\legato, 0.3
).trace.play;
)
```

The example above also shows another helpful trick: the message `trace` just before `play`. It prints out in the Post window the values chosen for every event. Very useful for debugging or simply to understand what is going on!

Pay attention to the differences between `Pwhite` and `Prand`: even though both have to do with randomness, they take in different arguments, and they do different things. Inside `Pwhite`'s parentheses you only need to provide a low and a high boundary: `Pwhite(low, high)`. Random numbers will be chosen from within that range. `Prand`, on the other hand, takes in a list of items (necessarily between square brackets), and a number of repeats: `Prand([list, of, items], repeats)`. Random items will be chosen *from the list*.

Play around with both and make sure you fully understand the difference.

TIP: A `Pwhite` with two integer numbers will generate only integers. For example, `Pwhite(100, 500)` will output numbers like 145, 568, 700, but not 145.6, 450.32, etc. If you want floating point numbers in your output, write `Pwhite(100, 500.0)`. This is very useful for, say, amplitudes: if you write `Pwhite(0, 1)` you are only getting 0 or 1, but write `Pwhite(0, 1.0)` and you will get everything in between.

Try the following questions to test your new knowledge:

a) What is the difference in output between `Pwhite(0, 10)` and `Prand([0, 4, 1, 5, 9, 10, 2, 3], inf)`?

b) If you need a stream of integer numbers chosen randomly between 0 and 100, could you use a `Prand`?

c) What is the difference in output between `Pwhite(0, 3)` and `Prand([0, 1, 2, 3], inf)`? What if you write `Pwhite(0, 3.0)`?

d) Run the examples below. We use `\note` instead of `\degree` in order to play a C minor scale (which includes black keys). The list `[0, 2, 3, 5, 7, 8, 11, 12]` has eight numbers in it, corresponding to pitches C, D, E♭, F, G, A♭, B, C, but how many events does each example actually play? Why?

```
// Pseq
(
Pbind(
        \note, Pseq([0, 2, 3, 5, 7, 8, 11, 12], 4),
        \dur, 0.15;
).play;
)
```

```

// Pseq
(
Pbind(
        \note, Prand([0, 2, 3, 5, 7, 8, 11, 12], 4),
        \dur, 0.15;
).play;
)

// Pwhite
(
Pbind(
        \note, Pseq([0, 2, 3, 5, 7, 8, 11, 12], 4),
        \dur, Pwhite(0.15, 0.5);
).play;
)
```

Answers are at the end of this tutorial.[2]

TIP: A Pbind stops playing when the shortest internal pattern has finished playing (as determined by the `repeats` argument of each internal pattern).

13.8 Expanding your Pattern vocabulary

By now you should be able to write simple `Pbind`s on your own. You know how to specify pitches, durations, amplitudes, and legato values, and you know how to embed other patterns (`Pseq`, `Prand`, `Pwhite`) to generate interesting parameter changes.

This section will expand your Pattern vocabulary a bit. The examples below introduce six more members of the Pattern family. Try to figure out by yourself what they do. Use the following strategies:

- Listen the resulting melody; describe and analyze what you hear;
- Look at the Pattern name: does it suggest something? (for example, `Pshuf` may remind you of the word "shuffle");
- Look at the arguments (numbers) inside the new Pattern;
- Use `.trace.play` as seen earlier to watch the values being printed in the Post window;
- Finally, confirm your guesses by consulting the Help files (select the name of the pattern and hit [ctrl+D] to open the corresponding Help file).

```
// Expanding your Pattern vocabulary

// Pser
(
Pbind(
        \note, Pser([0, 2, 3, 5, 7, 8, 11, 12], 11),
        \dur, 0.15;
).play;
)

// Pxrand
// Compare with Prand to hear the difference
(
p = Pbind(
    \note, Pxrand([0, 2, 3, 5, 7, 8, 11, 12], inf),
        \dur, 0.15;
).play;
)

// Pshuf
(
p = Pbind(
    \note, Pshuf([0, 2, 3, 5, 7, 8, 11, 12], 6),
        \dur, 0.15;
).play;
)

// Pslide
// Takes 4 arguments: list, repeats, length, step
(
Pbind(
        \note, Pslide([0, 2, 3, 5, 7, 8, 11, 12], 7, 3, 1),
        \dur, 0.15;
).play;
)

// Pseries
// Takes three arguments: start, step, length
(
Pbind(
    \note, Pseries(0, 2, 15),
        \dur, 0.15;
).play;
)

// Pgeom
// Takes three arguments: start, grow, length
```

```
(
Pbind(
        \note, Pseq([0, 2, 3, 5, 7, 8, 11, 12], inf),
        \dur, Pgeom(0.1, 1.1, 25);
).play;
)

// Pn
(
Pbind(
        \note, Pseq([0, Pn(2, 3), 3, Pn(5, 3), 7, Pn(8, 3), 11, 12], 1),
        \dur, 0.15;
).play;
)
```

Practice using these Patterns—you can do a lot with them. `Pbinds` are like a recipe for a musical score, with the advantage that you are not limited to writing fixed sequences of notes and rhythms: you can describe processes of ever changing musical parameters (this is sometimes called "algorithmic composition"). And this is just one aspect of the powerful capabilities of the Pattern family.

In the future, when you feel the need for more pattern objects, the best place to go is James Harkins' "A Practical Guide to Patterns," available in the built-in Help files.*

*Also online at `http://doc.sccode.org/Tutorials/A-Practical-Guide/PG_01_Introduction.html`

14 More Pattern tricks

14.1 Chords

Want to write chords inside `Pbind`s? Write them as lists (comma-separated values enclosed in square brackets):

```
(
Pbind(
	\note, Pseq([[0, 3, 7], [2, 5, 8], [3, 7, 10], [5, 8, 12]], 3),
	\dur, 0.15
).play;
)
// Fun with strum
(
Pbind(
	\note, Pseq([[-7, 3, 7, 10], [0, 3, 5, 8]], 2),
	\dur, 1,
	\legato, 0.4,
	\strum, 0.1 // try 0, 0.1, 0.2, etc
).play;
)
```

14.2 Scales

When using `\degree` for your pitch specification, you can add another line with the keyword `\scale` to change scales (note: this only works in conjunction with `\degree`, not with `\note`, `\midinote`, or `\freq`):

```
(
Pbind(
        \scale, Scale.harmonicMinor,
        \degree, Pseq([0, 1, 2, 3, 4, 5, 6, 7], 1),
        \dur, 0.15;
).play;
)

// Evaluate this line to see a list of all available scales:
Scale.directory;

// If you need a chromatic note in between scale degrees, do this:
(
Pbind(
        \degree, Pseq([0, 1, 2, 3, 3.1, 4], 1),
).play;
)

// The 3.1 above means one chromatic step above scale degree 3 (in this case, F#
    above F). Note that when you don't explicitly specify a \scale, Scale.major is
    assumed.
```

14.3 Transposition

Use the `\ctranspose` keyword to achieve chromatic transposition. This will work in conjunction with `\degree`, `\note`, and `\midinote`, but not with `\freq`.

```
(
Pbind(
        \note, Pser([0, 2, 3, 5, 7, 8, 11, 12], 11),
        \ctranspose, 12, // transpose an octave above (= 12 semitones)
        \dur, 0.15;
).play;
)
```

14.4 Microtones

How to write microtones:

```
// Microtones with \note and \midinote:
Pbind(\note, Pseq([0, 0.5, 1, 1.5, 1.75, 2], 1)).play;
Pbind(\midinote, Pseq([60, 69, 68.5, 60.25, 70], 1)).play;
```

14.5 Tempo

The values you provide to the `\dur` key of a Pbind are in *number of beats*, that is, 1 means one beat, 0.5 means half a beat, and so on. Unless you specify otherwise, the default tempo is 60 BPM (beats per minute). To play at a different tempo, you simply create a new TempoClock. Here's a `Pbind` playing at 120 beats per minute (BPM):

```
(
Pbind(\degree, Pseq([0, 0.1, 1, 2, 3, 4, 5, 6, 7]),
        \dur, 1;
).play(TempoClock(120/60)); // 120 beats over 60 seconds: 120 BPM
)
```

By the way, did you see that the `Pseq` above is taking only one argument (the list)? Where is the `repeats` value that always came after the list? You can hear that the example plays through the sequence only once, but why? This is a common property of all Patterns (and in fact, of many other objects in SuperCollider): if you omit an argument, it will use a built-in default value. In this case, the default `repeats` for `Pseq` is 1. Remember your first ridiculously simple

Pbind? It was a mere `Pbind(\degree, 0).play` and it only knew how to play one note. You didn't provide any info for duration, amplitude, legato, etc. In theses cases `Pbind` simply goes ahead and uses its default values for those.

14.6 Rests

Here is how to write rests. The number inside `Rest` is the duration of the rest in beats. Rests can go anywhere in the Pbind, not just in the `\dur` line.

```
(
Pbind(
	\degree, Pwhite(0, 10),
	\dur, Pseq([0.1, 0.1, 0.3, 0.6, Rest(0.3), 0.25], inf);
).play;
)
```

14.7 Playing two or more Pbinds together

To start a few Pbinds simultaneously, simply enclose all of them within a single code block:

```
( // open big block
Pbind(
	\freq, Pn(Pseries(110, 111, 10)),
	\dur, 1/2,
	\legato, Pwhite(0.1, 1)
).play;

Pbind(
	\freq, Pn(Pseries(220, 222, 10)),
	\dur, 1/4,
        \legato, Pwhite(0.1, 1)
).play;

Pbind(
        \freq, Pn(Pseries(330, 333, 10)),
        \dur, 1/6,
        \legato, 0.1
).play;
) // close big block
```

In order to play Pbinds in a time-ordered fashion (other than simply evaluating them manually one after the other), you can use `{ }.fork`:

```
// Basic fork example. Watch Post window:
(
{
        "one thing".postln;
        2.wait;
        "another thing".postln;
        1.5.wait;
        "one last thing".postln;
}.fork;
)
// A more interesting example:
(
t = TempoClock(76/60);
{
        Pbind(
                \note, Pseq([[4, 11], [6, 9]], 32),
                \dur, 1/6,
                \amp, Pseq([0.05, 0.03], inf)
        ).play(t);

        2.wait;

        Pbind(
                \note, Pseq([[-25, -13, -1], [-20, -8, 4], \rest], 3),
                \dur, Pseq([1, 1, Rest(1)], inf),
                \amp, 0.1,
                \legato, Pseq([0.4, 0.7, \rest], inf)
        ).play(t);

        2.75.wait;

        Pbind(
                \note, Pseq([23, 21, 25, 23, 21, 20, 18, 16, 20, 21, 23, 21], inf),
                \dur, Pseq([0.25, 0.75, 0.25, 1.75, 0.125, 0.125, 0.80, 0.20, 0.125,
                    0.125, 1], 1),
                \amp, 0.1,
                \legato, 0.5
        ).play(t);
}.fork(t);
)
```

For advanced ways of playing `Pbind`s simultaneously and in sequence, check out `Ppar` and `Pspawner`. For more about `fork`, check out the `Routine` Help file.

14.8 Using variables

In the earlier section "Expanding your Pattern vocabulary," did you notice how you had to type the same note list [0, 2, 3, 5, 7, 8, 11, 12] several times for multiple `Pbind`s? Not very efficient to copy the same thing by hand over and over, right? In programming, whenever you find yourself doing the same task repeatedly, it's probably time to adopt a smarter strategy to accomplish the same goal. In this case, we can use variables. As you may remember, variables allow you to refer to any chunk of data in a flexible and concise way (review section 12 if needed). Here's an example:

```
// Using the same sequence of numbers a lot? Save it in a variable:
c = [0, 2, 3, 5, 7, 8, 11, 12];

// Now you can just refer to it:
Pbind(\note, Pseq(c, 1), \dur, 0.15).play;
Pbind(\note, Prand(c, 6), \dur, 0.15).play;
Pbind(\note, Pslide(c, 5, 3, 1), \dur, 0.15).play;
```

Another example to practice using variables: let's say we want to play two `Pbind`s simultaneously. One of them does an ascending major scale, the other does a descending major scale one octave above. Both use the same list of durations. Here is one way of writing this:

```
~scale = [0, 1, 2, 3, 4, 5, 6, 7];
~durs = [0.4, 0.2, 0.2, 0.4, 0.8, 0.2, 0.2, 0.2];
(
Pbind(
        \degree, Pseq(~scale),
        \dur, Pseq(~durs)
).play;

Pbind(
        \degree, Pseq(~scale.reverse + 7),
        \dur, Pseq(~durs)
).play;
)
```

Interesting tricks here: thanks to variables, we reuse the same list of scale degrees and durations for both `Pbind`s. We wanted the second scale to be descending and one octave above the first. To achieve this, we simply use the message `.reverse` to reverse the order of the list (type ~`scale.reverse` on a new line and evaluate to see exactly what it does). Then we add 7 to transpose it one octave above (test it as well to see the result).[*] We played two `Pbind`s at the same time by enclosing them within a single code block.

Exercise: create one additional `Pbind` inside the code block above, so that you hear three simultaneous voices. Use both variables (~`scale` and ~`durs`) in some different way—for example, use them inside a pattern other than Pseq; change transposition amount; reverse and/or multiply durations; etc.

15 Starting and stopping Pbinds independently

This is a very common question that comes up about `Pbind`s, especially the ones that run forever with `inf`: how can I stop and start individual `Pbind`s at will? The answer will involve using variables, and we'll see a complete example soon; but before going there, we need to understand a little more of what happens when you play a `Pbind`.

15.1 Pbind as a musical score

You can think of `Pbind` as a kind of musical score: it is a recipe for making sounds, a set of instructions to realize a musical passage. In order for the score to become music, you need to give it to a player: someone who will read the score and make sounds based on those instructions. Let's conceptually separate these two moments: the definition of the score, and the performance of it.

```
// Define the score
```

[*]We could also have used `\ctranspose, 12` to get the same transposition.

```
(
p = Pbind(
        \midinote, Pseq([57, 62, 64, 65, 67, 69], inf),
        \dur, 1/7
); // no .play here!
)

// Ask for the score to be played
p.play;
```

The variable `p` in the example above simply holds the score—notice that the `Pbind` does not have a `.play` message right after its closing parenthesis. No sound is made at that point. The second moment is when you ask SuperCollider to play from that score: `p.play`.

A common mistake at this point is to try `p.stop`, hoping that it will stop the player. Try it and verify for yourself that it doesn't work this way. You will understand why in the next couple of paragraphs.

15.2 EventStreamPlayer

Clean the Post window with [ctrl+shift+P] (not really needed, but why not?) and evaluate `p.play` again. Look at the Post window and you will see that the result is something called an `EventStreamPlayer`. Every time you call `.play` on a `Pbind`, SuperCollider creates a player to realize that action: that's what `EventStreamPlayer` is. It's like having a pianist materialize in front of you every time you say "I want this score to be played right now." Nice, huh?

Well, yes, except that after this anonymous virtual player shows up and starts the job, you have no way to talk to it—it has no name. In slightly more technical terms, you have created an object, but you have no way to refer to that object later. Maybe at this point you can see why doing `p.stop` won't work: it's like you are trying to talk to the score instead of talking to the player. The score (the Pbind stored in the variable p) knows nothing about starting or stopping: it is just a recipe. The *player* is the one who knows about starting, stopping, "would you please take from the beginning", etc. In other words, you have to talk to the EventStreamPlayer. All you need to do is to give it a name, in other words, store it into a variable:

```
// Try these lines one by one:
~myPlayer = p.play;
~myPlayer.stop;
~myPlayer.resume;
~myPlayer.stop.reset;
~myPlayer.start;
~myPlayer.stop;
```

In summary: calling .play on a Pbind generates an EventStreamPlayer; and storing your EventStreamPlayers into variables allows you to access them later to start and stop patterns individually (no need to use [ctrl+.], which kills everything at once).

15.3 Example

Here's a more complex example to wrap up this section. The top melody is borrowed from Tchaikovsky's Album for the Youth, and a lower melody is added in counterpoint. Figure 3 shows the passage in musical notation.

```
// Define the score
(
var myDurs = Pseq([Pn(1, 5), 3, Pn(1, 5), 3, Pn(1, 6), 1/2, 1/2, 1, 1, 3, 1, 3], inf
    ) * 0.4;
~upperMelody = Pbind(
        \midinote, Pseq([69, 74, 76, 77, 79, 81, Pseq([81, 79, 81, 82, 79, 81], 2),
            82, 81, 79, 77, 76, 74, 74], inf),
        \dur, myDurs
);
~lowerMelody = Pbind(
        \midinote, Pseq([57, 62, 61, 60, 59, 58, 57, 55, 53, 52, 50, 49, 50, 52, 50,
             55, 53, 52, 53, 55, 57, 58, 61, 62, 62], inf),
        \dur, myDurs
);
)
// Play the two together:
(
~player1 = ~upperMelody.play;
~player2 = ~lowerMelody.play;
)
// Stop them separately:
~player1.stop;
~player2.stop;
// Other available messages
~player1.resume;
~player1.reset;
~player1.play;
~player1.start; // same as .play
```

First, notice the use of variables. One of them, `myDurs`, is a local variable. You can tell it's a local variable because it doesn't start with a tilde (~) and it's declared at the top with the reserved keyword `var`. This variable holds an entire `Pseq` that will be used as `\dur` in both of the `Pbind`s. `myDurs` is really only needed at the moment of defining the score, so it makes sense to use a local variable for that (though an environment variable would work just fine too). The other variables you see in the example are environment variables—once declared, they are valid anywhere in your SuperCollider patches.

Second, notice the separation between score and players, as discussed earlier. When the `Pbind`s are defined, they are not played right away—there is no `.play` immediately after their

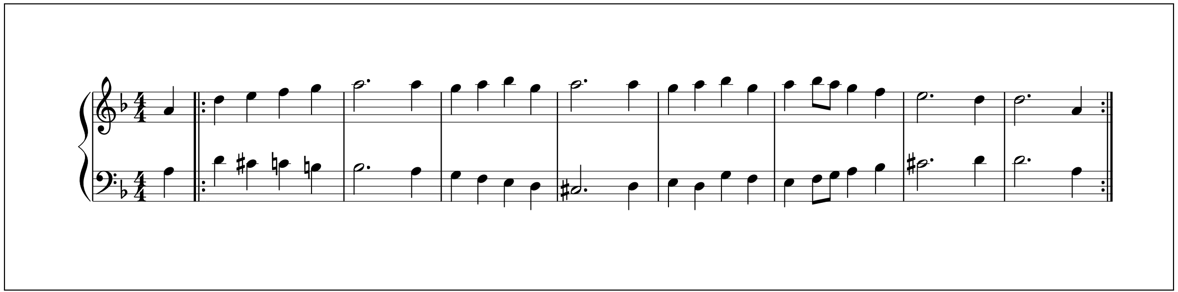


Figure 3: Pbind counterpoint with a Tchaikovsky melody

closing parenthesis. After you evaluate the first code block, all you have is two Pbind definitions stored into the variables ~upperMelody and ~lowerMelody. They are not making sound yet—they are just the scores. The line ~player1 = ~upperMelody.play creates an EventStreamPlayer to do the job of playing the upper melody, and that player is given the name ~player1. Same idea for ~player2. Thanks to this, we can talk to each player and request it to stop, start, resume, etc.

At the risk of being tedious, let's reiterate this one last time:

- A Pbind is just a recipe for making sounds, like a musical score;
- When you call the message play on a Pbind, an EventStreamPlayer object is created;
- If you store this EventStreamPlayer into a variable, you can access it later to use commands like stop and resume.

Part III
MORE ABOUT THE LANGUAGE

16 Objects, classes, messages, arguments

SuperCollider is an Object-Oriented programming language, like Java or C++. It is beyond the scope of this tutorial to explain what this means, so we'll let you search that on the web if you are curious. Here we'll just explain a few basic concepts you need to know to better understand this new language you are learning.

Everything in SuperCollider is an *object*. Even simple numbers are objects in SC. Different objects behave in different ways and hold different kinds of information. You can request some info or action from an object by sending it a *message*. When you write something like `2.squared`, the message `squared` is being sent to the receiver object `2`. The dot between them makes the connection. Messages are also called *methods*, by the way.

Objects are specified hierarchically in *classes*. SuperCollider comes with a huge collection of pre-defined classes, each with their own set of methods.

Here's a good way to understand this. Let's imagine there is an abstract class of objects called `Animal`. The Animal class defines a few general methods (messages) common to all animals. Methods like `age`, `weight`, `picture` could be used to get information about the animal. Methods like `move`, `eat`, `sleep` would make the animal perform a specific action. Then we could have two subclasses of Animal: one called `Pet`, another called `Wild`. Each one of these subclasses could have even more subclasses derived from them (like `Dog` and `Cat` derived from `Pet`). Subclasses inherit all methods from their parent classes, and implement new methods of their own to add specialized features. For example, both Dog and Cat objects would happily respond to the `.eat` message, inherited from the Animal class. `Dog.name` and `Cat.name` would return the name of the pet: this method is common to all objects derived from Pet. `Dog` has a `bark` method, so you can call `Dog.bark` and it will know what to do. `Cat.bark` would throw you an error message: `ERROR: Message 'bark' not understood.`

In all these hypothetical examples, the words beginning with a capital letter are *classes* which represent *objects*. The lowercase words after the dot are *messages* (or *methods*) being sent to those objects. Sending a message to an object always returns some kind of information. Finally, messages sometimes accept (or even require) *arguments*. Arguments are the things that come inside parentheses right after a message. In `Cat.eat("sardines", 2)`, the message `eat` is being sent to `Cat` with some very specific information: what to eat, and quantity. Sometimes you will see arguments declared explicitly inside the parentheses (keywords ending with a colon). This is often handy to remind the reader what the argument refers to. `Dog.bark(volume: 10)` is more self-explanatory than just `Dog.bark(10)`.

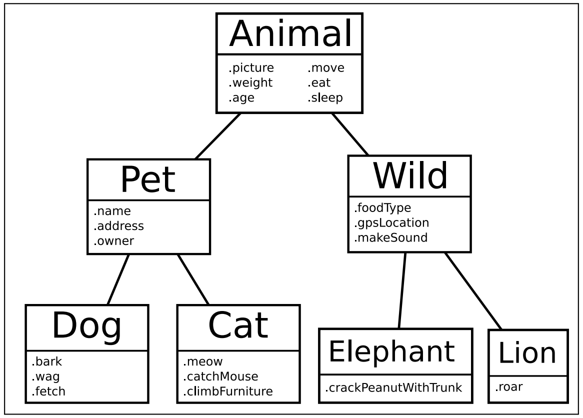


Figure 4: Hypothetical class hierarchy.

OK—enough of this quick and dirty explanation of object-oriented programming. Let's try some examples that you can actually run in SuperCollider. Run one line after the other and see if you can identify the message, the receiver object, and the arguments (if any). The basic structure is `Receiver.message(arguments)` Answers at the end of this document.[3]

```
[1, 2, 3, "wow"].reverse;
"hello".dup(4);
3.1415.round(0.1); // note that the first dot is the decimal case of 3.1415
100.rand; // evaluate this line several times
// Chaining messages is fun:
100.0.rand.round(0.01).dup(4);
```

17 Receiver notation, functional notation

There is more than one way of writing your expressions in SuperCollider. The one we just saw above is called *receiver notation*: `100.rand`, where the dot connects the Object (`100`) to the message (`rand`). Alternatively, the exact same thing can also be written like this: `rand(100)`. This one is called *functional notation*.

You can use either way of writing. Here's how this works when a message takes two or more arguments.

```
5.dup(20);  // receiver notation
dup(5, 20); // same thing in functional notation

3.1415.round(0.1); // receiver notation
round(3.1415, 0.1); // functional notation
```

In the examples above, you might read `dup(5, 20)` as "duplicate the number 5 twenty times," and `round(3.1415, 0.1)` as "round the number 3.1415 to one decimal case." Conversely, the receiver notation versions could be read as “Number 5, duplicate yourself twenty times!” (for `5.dup(20)`) and “Number 3.1415, round yourself to one decimal case!” (for `3.1415.round(0.1)`).

In short: `Receiver.message(argument)` is equivalent to `message(Receiver, argument)`.

Choosing one writing style over another is a matter of personal preference and convention. Sometimes one method can be clearer than the other. Whatever style you end up preferring (and it's fine to mix them), the important thing is to be consistent. One convention that is widespread among SuperCollider users is that classes (words that begin with uppercase letters) are almost always written as `Receiver.message(argument)`. For example, you will always see `SinOsc.ar(440)`, but you will almost never see `ar(SinOsc, 440)`, even though both are correct.

Exercise: rewrite the following statement using functional notation only:

```
100.0.rand.round(0.01).dup(4);
```

Solution at the end.[4]

18 Nesting

The solution to the last exercise has led you to nest things one inside the other. David Cottle has an excellent explanation of nesting in the SuperCollider book, so we will just quote it here.[*]

> *To further clarify the idea of nesting, consider a hypothetical example in which SC will make you lunch. To do so, you might use a* `serve` message. *The arguments might be salad, main course, and dessert. But just saying* `serve(lettuce, fish, banana)` *may not give you the results you want. So to be safe you could clarify those arguments, replacing each with a nested message and argument.*

```
serve(toss(lettuce, tomato, cheese), bake(fish, 400, 20), mix(banana, icecream))
```

[*]Cottle, D. “Beginner's Tutorial.” The SuperCollider Book, MIT Press, 2011, pp. 8-9.

SC would then serve not just lettuce, fish, and banana, but a tossed salad with lettuce, tomato, and cheese; a baked fish; and a banana sundae. These inner commands can be further clarified by nesting a message(arg) for each ingredient: lettuce, tomato, cheese, and so on. Each internal message produces a result that is in turn used as an argument by the outer message.

```
// Pseudo—code to make dinner:
serve(
        toss(
                wash(lettuce, water, 10),
                dice(tomato, small),
                sprinkle(choose([blue, feta, gouda]))
        ),
        bake(catch(lagoon, hook, bamboo), 400, 20),
        mix(
                slice(peel(banana), 20),
                cook(mix(milk, sugar, starch), 200, 10)
        )
);
```

When the nesting has several levels, we can use new lines and indents for clarity. Some messages and arguments are left on one line, some are spread out with one argument per line—whichever is clearer. Each indent level should indicate a level of nesting. (Note that you can have any amount of white space—new lines, tabs, or spaces—between bits of code.)

[In the dinner example] *the lunch program is now told to wash the lettuce in water for 10 minutes and to dice the tomato into small pieces before tossing them into the salad bowl and sprinkling them with cheese. You've also specified where to catch the fish and to bake it at 400 degrees for 20 minutes before serving, and so on.*

To "read" this style of code you start from the innermost nested message and move out to each successive layer. Here is an example aligned to show how the innermost message is nested inside the outer messages.

```
                exprand(1.0, 1000.0);
           dup({exprand(1.0, 1000.0)}, 100);
      sort(dup({exprand(1.0, 1000.0)}, 100));
round(sort(dup({exprand(1.0, 1000.0)}, 100)), 0.01);
```

The code below is another example of nesting. Answer the questions that follow. You don't have to explain what the numbers are doing—the task is simply to identify the arguments in each layer of nesting. (Example and exercise questions also borrowed and slightly adapted from Cottle's tutorial.)

```
// Nesting and proper indentation
(
{
        CombN.ar(
                SinOsc.ar(
                        midicps(
                                LFNoise1.ar(3, 24,
                                        LFSaw.ar([5, 5.123], 0, 3, 80)
                                )
                        ),
                        0, 0.4
                ),
                1, 0.3, 2)
}.play;
)
```

a) What number is the second argument for `LFNoise1.ar`?

b) What is the first argument for `LFSaw.ar`?

c) What is the third argument for `LFNoise1.ar`?

d) How many arguments are in `midicps`?

e) What is the third argument for `SinOsc.ar`?

f) What are the second and third arguments for `CombN.ar`?

See the end of this document for the answers.[5]

TIP: If for whatever reason your code has lost proper indentation, simply select all of it and go to menu Edit→Autoindent Line or Region, and it will be fixed.

19 Enclosures

There are four types of enclosures: `(parentheses)`, `[brackets]`, `{braces}`, and `"quotation marks"`.

Each one that you open will need to be closed at a later point. This is called "balancing," that is, keeping properly matched pairs of enclosures throughout your code.

The SuperCollider IDE automatically indicates matching parentheses (also brackets and braces) when you close a pair—they show up in red. If you click on a parenthesis that lacks an opening/closing match, you will see a dark red selection telling you something is missing.

Balancing is a quick way to select large sections of code for evaluation, deletion, or copy/paste operations. You can double click an opening or closing parenthesis (also brackets and braces) to select everything within.

19.1 Quotation marks

Quotation marks are used to enclose a sequence of characters (including spaces) as a single unit. These are called Strings. Single quotes create Symbols, which are slightly different than Strings. Symbols can also be created with a backslash immediately before the text. Thus `'greatSymbol'` and `\greatSymbol` are equivalent.

```
"Here's a nice string";
'greatSymbol';
```

19.2 Parentheses

Parentheses can be used to:

- enclose argument lists: `rrand(0, 10);`
- force precedence: `5 + (10 * 4);`
- create code blocks (multiple lines of code to be evaluated together).

19.3 Brackets

Square brackets define a collection of items, like `[1, 2, 3, 4, "hello"]`. These are normally called Arrays. An array can contain anything: numbers, strings, functions, patterns, etc. Arrays understand messages such as **reverse**, **scramble**, **mirror**, **choose**, to name a few. You can also perform mathematical operations on arrays.

```
[1, 2, 3, 4, "hello"].scramble;
[1, 2, 3, 4, "hello"].mirror;
[1, 2, 3, 4].reverse + 10;
// convert midi to frequency in Hz
[60, 62, 64, 65, 67, 69, 71].midicps.round(0.1);
```

More on Arrays coming soon in section 22.

19.4 Curly Braces

Braces (or "curly braces") define functions. Functions encapsulate some kind of operation or task that will probably be used and reused multiple times, possibly returning different results each time. The example below is from the SuperCollider book.

```
exprand(1, 1000.0);
{exprand(1, 1000.0)};
```

David Cottle walks us through his example: *"the first line picks a random number, which is displayed in the post window. The second prints a very different result: a function. What does the function do? It picks a random number. How can that difference affect code? Consider the lines below. The first chooses a random number and duplicates it. The second executes the random-number-picking function 5 times and collects the results in an array."*[*]

```
rand(1000.0).dup(5);  // picks a number, duplicates it
{rand(1000.0)}.dup(5);  // duplicates the function of picking a number
{rand(1000.0)}.dup(5).round(0.1); // all of the above, then round
```

[*]Cottle, D. "Beginner's Tutorial." The SuperCollider Book, MIT Press, 2011, p. 13.

```
// essentially, this (which has a similar result)
[rand(1000.0), rand(1000.0), rand(1000.0), rand(1000.0), rand(1000.0)]
```

More about functions soon. For now, here's a summary of all possible enclosures:

Collections `[list, of, items]`

Functions `{ often multiple lines of code }`

Strings `"words inside quotes"`

Symbols `'singlequotes'` or preceded by a `\backslash`

20 Conditionals: if/else and case

If it's raining, I'll take an umbrella when I go out. If it's sunny, I'll take my sunglasses. Our days are filled with this kind of decision making. In programming, these are the moments when your code has to test some condition, and take different courses of action depending on the result of the test (true or false). There are many types of conditional structures. Let's take a look at two simple ones: *if/else* and *case*.

The syntax for an if/else in SC is: `if(condition, {true action}, {false action})`. The condition is a Boolean test (it must return `true` or `false`). If the test returns true, the first function is evaluated; otherwise, the second function is. Try it:

```
// if / else
if(100 > 50, { "very true".postln }, { "very false".postln });
```

The table below, borrowed from the SuperCollider book*, presents some common Boolean operators that you can use. Note the distinction between a single equal sign (`x = 10`) and two equal signs (`x == 10`). The single sign means "*assign 10 to the variable x*," while the double sign means "*is x equal to 10?*" Type and run some of the examples from the true or false columns, and you will actually see `true` or `false` results in the Post window.

Symbol	**Meaning**	**True Example**	**False Example**
`==`	equal to?	`10 == 10`	`10 == 99`
`!=`	not equal to?	`10 != 99`	`10 != 10`
`>`	greater than?	`10 > 5`	`10 > 99`
`<`	less than?	`10 < 99`	`10 < 5`
`>=`	greater than or equal to?	`10 >= 10`, `10 >= 3`	`10 >= 99`
`<=`	less than or equal to?	`10 <= 99`, `10 <= 10`	`10 <= 9`
`odd`	is it odd?	`15.odd`	`16.odd`
`even`	is it even?	`22.even`	`21.even`
`isInteger`	is it an integer?	`3.isInteger`	`3.1415.isInteger`
`isFloat`	is it a float?	`3.1415.isFloat`	`3.isFloat`
`and`	both conditions	`11.odd.and(12.even)`	`11.odd.and(13.even)`
`or`	either condition	`or(1.odd, 1.even)`	`or(2.odd, 1.even)`

The last two lines (`and`, `or`) show how to write the longer expressions in either receiver notation or functional notation.

Another useful structure is `case`. It works by defining pairs of functions to be evaluated in order until one of the tests returns true:

```
case
```

*Cottle, D. "Beginner's Tutorial." The SuperCollider Book, MIT Press, 2011, p. 33

```
{test1} {action1}
{test2} {action2}
{test3} {action3}
...
{testN} {actionN};
```

The expression inside each test has to return either `true` or `false`. If test1 returns false, the program ignores action1 and moves on to test2. If it's false, action2 is also ignored and we move on to test3. If that turns out to be true, then action3 is executed, and `case` stops (no further tests or actions are executed). Note that there are no commas between functions. Simply use a semicolon at the very end to mark the end of the `case` statement.

```
// case
(
~num = -2;

case
{~num == 0} {"WOW".postln}
{~num == 1} {"ONE!".postln}
{~num < 0} {"negative number!".postln}
{true} {"last case scenario".postln};
)
```

Try changing the value of ~`num` to get all possible results. Notice the useful (and optional) trick in the last line of `case` in the example above: since `true` always evaluate to true, you can define a "last case scenario" action that will always happen if all previous conditions happen to be false.

For more, check out the Control Structures Help file.

21 Functions

When you find yourself doing the same task several times, it may be a good time to create a reusable function. A function, as you learned in the Enclosures section, is something written within curly braces. David Touretzky introduces the idea of function in this way: "think of a function as a box through which data flows. The function operates on the data in some way, and the result is what flows out."*

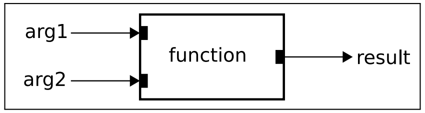


Figure 5: General idea of a function.

The first line in the example below defines a function and assigns it to the variable `f`. The second line puts the function to work.

```
f = { 2 + 2 }; // define the function
f.value; // put the function to work
```

The function above is not terribly useful, as it only knows how to do one thing (add 2 and 2). Normally you will want to define functions that can give you different results depending on the input arguments you give to it. We use the keyword `arg` to specify the inputs that a function can accept. The example below looks more like the drawing from figure 5.

```
f = {arg a, b; ["a plus b", a+b, "a times b", a*b].postln}; // define function
f.value(3, 7); // now you can give any two numbers as arguments to the function
```

*Touretzky, David. COMMON LISP: A Gentle Introduction to Symbolic Computation. The Benjamin/Cummings Publishing Company, Inc, 1990, p. 1. That's the book that inspired the title of this tutorial.

```
f.value(10, 14);

// Compare:
~sillyRand = rrand(0, 10); // not a function
~sillyRand.value; // evaluate several times
~sillyRand2 = {rrand(0, 10)}; // a function
~sillyRand2.value; // evaluate several times
```

As a last example, here's one very useful function.

```
// Use this function to decide how to spend your Summer days
(
~whatToDo = {
var today, dayName, actions;
        today = Date.getDate.dayOfWeek;
        dayName =
        case
        {today==0} {"Sunday"}
        {today==1} {"Monday"}
        {today==2} {"Tuesday"}
        {today==3} {"Wednesday"}
        {today==4} {"Thursday"}
        {today==5} {"Friday"}
        {today==6} {"Saturday"};
        actions = ["boomerang throwing", "arm wrestling", "stair climbing", "playing
             chess", "underwater hockey", "pea shooting", "a nap marathon"];
        "Ah, " ++ dayName ++ "...! " ++ "What a good day for " ++ actions.choose;
};
)

// Run it in the morning
~whatToDo.value;
```

TIP: Another common notation to declare arguments at the beginning of a Function is: `f = {|a, b| a + b}`. This is equivalent to `f = {arg a, b; a + b}`

22 Fun with Arrays

Arrays are the most common type of collections in SuperCollider. Every time you write a collection of items between square brackets, like `[0, 1, 2]`, it is an instance of the class `Array`. You will often find yourself manipulating arrays in various ways. Here is a small selection of some interesting methods that arrays understand:

```
// Create some array
a = [10, 11, 12, 13, 14, 15, 16, 17];

a.reverse;  // reverse
a.scramble; // scramble
a.choose;  // picks one element at random
a.size;   // returns size of array
a.at(0);   // retrieves item at specified position
a[0]    ;  // same as above
a.wrapAt(9); // retrieves item at specified position, wrapping around if > a.size
["wow", 99] ++ a; // concatenates the two arrays into a new one
a ++ \hi;  // a Symbol is a single character
a ++ 'hi'; // same as above
a ++ "hi"; // a String is a collection of characters
a.add(44);    // creates new array with new element at the end
a.insert(5, "wow"); // inserts "wow" at position 5, pushes other items forward (
    returns new array)
a; // evaluate this and see that none of the above operations actually changed the
    original array
```

```
a.put(2, "oops"); // put "oops" at index 2 (destructive; evaluate line above again
    to check)
a.permute(3); // permute: item in position 3 goes to position 0, and vice-versa
a.mirror;  // makes it a palindrome
a.powerset; // returns all possible combinations of the array's elements
```

You can do math with arrays:

```
[1, 2, 3, 4, 5] + 10;
[1, 2, 3, 4, 5] * 10;
([1, 2, 3, 4, 5] / 7).round(0.01); // notice the parentheses for precedence
x = 11; y = 12; // try some variables
[x, y, 9] * 100;
// but make sure you only do math with proper numbers
[1, 2, 3, 4, "oops", 11] + 10; // strange result
```

22.1 Creating new Arrays

Here are a few ways of using the class `Array` to create new collections:

```
// Arithmetic series
Array.series(size: 6, start: 10, step: 3);
// Geometric series
Array.geom(size: 10, start: 1, grow: 2);
// Compare the two:
Array.series(7, 100, -10); // 7 items; start at 100, step of -10
Array.geom(7, 100, 0.9); // 7 items; start at 100; multiply by 0.9 each time
// Meet the .fill method
Array.fill(10, "same");
// Compare:
Array.fill(10, rrand(1, 10));
Array.fill(10, {rrand(1, 10)}); // function is re-evaluated 10 times
```

```
// The function for the .fill method can take a default argument that is a counter.
// The argument name can be whatever you want.
Array.fill(10, {arg counter; counter * 10});
// For example, generating a list of harmonic frequencies:
Array.fill(10, {arg wow; wow+1 * 440});
// The .newClear method
a = Array.newClear(7); // creates an empty array of given size
a[3] = "wow"; // same as a.put(3, "wow")
```

22.2 That funny exclamation mark

It is just a matter of time until you see something like `30!4` in someone else's code. This shortcut notation simply creates an array containing the same item a number of times:

```
// Shortcut notation:
30!4;
"hello" ! 10;
// It gives the same results as the following:
30.dup(4);
"hello".dup(10);
// or
Array.fill(4, 30);
Array.fill(10, "hello");
```

22.3 The two dots between parentheses

Here is another common syntax shortcut used to create arrays.

```
// What is this?
(50..79);
```

```
// It's a shortcut to generate an array with an arithmetic series of numbers.
// The above has the same result as:
series(50, 51, 79);
// or
Array.series(30, 50, 1);
// For a step different than 1, you can do this:
(50, 53 .. 79); // step of 3
// Same result as:
series(50, 53, 79);
Array.series(10, 50, 3);
```

Note that each command implies a slightly different way of thinking. The `(50..79)` allows you to think this way: "*just give me an array from 50 to 79.*" You don't necessarily think about how many items the array will end up having. On the other hand, `Array.series` allows you to think: "*just give me an array with 30 items total, counting up from 50.*" You don't necessarily think about who is going to be the last number in the series.

Also note that the shortcut uses parentheses, not square brackets. The resulting array, of course, will be in square brackets.

22.4 How to "do" an Array

Often you will need to do some action over all items of a collection. We can use the method `do` for this:

```
~myFreqs = Array.fill(10, {rrand(440, 880)});

// Now let's do some simple action on every item of the list:
~myFreqs.do({arg item, count; ("Item " ++ count ++ " is " ++ item ++ " Hz. Closest
    midinote is " ++ item.cpsmidi.round).postln});

```

```
// If you don't need the counter, just use one argument:
~myFreqs.do({arg item; {SinOsc.ar(item, 0, 0.1)}.play});
~myFreqs.do({arg item; item.squared.postln});

// Of course something as simple as the last one could be done like this:
~myFreqs.squared;
```

In summary: when you "do" an array, you provide a function. The message `do` will iterate through the items of the array and evaluate that function each time. The function can take two arguments by default: the array item at current iteration, and a counter that keeps track of number of iterations. The names of these arguments can be whatever you want, but they are always in this order: item, count.

See also the method `collect`, which is very similar to `do`, but returns a new collection with all intermediate results.

23 Getting Help

Learn how to make good use of the Help files. Often you will find useful examples at the bottom of each Help page. Be sure to scroll down to check them out, even (or specially) if you don't fully understand the text explanations at first. You can run the examples directly from the Help browser, or you can copy and paste the code onto a new window to play around with it.

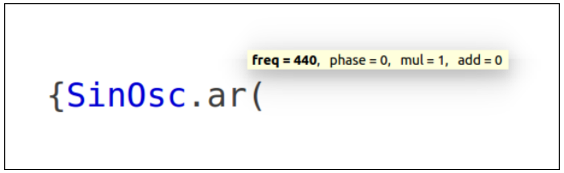
Select any valid class or method in your SuperCollider code (double-clicking the word will select it), and hit [ctrl+D] to open the corresponding Help file. If you select a class name (for example, `MouseX`), you will be directed to the class Help file. If you select a method, you will be directed to a list of classes that understand that method (for example, ask for help on the method `scramble`).*

*Attention: SuperCollider will display in blue any word that starts with a Capital letter. This means that the

Other ways to explore the Help files in the SuperCollider IDE are the "Browse" and "Search" links. Use Browse to navigate the files by categories, and Search to look for words in all Help files. Important note about the Help Browser in the SuperCollider IDE:

- Use the top-right field (where it says "Find...") to look for specific words *within the currently open Help file* (like you would do a "find" on a website);
- Use the "Search" link (to the right of "Browse") to search text *across all Help files*.

When you first open parentheses to add arguments to a given method, SC displays a little "tooltip help" to show you what the expected arguments are. For example, type the beginning of line that you see in figure 6. Right after opening the first parenthesis, you see the tooltip showing that the arguments to a `SinOsc.ar` are `freq`, `phase`, `mul`, and `add`. It also shows us what the default values are. This is exactly the same information you would get from the `SinOsc` Help file. If the tooltip has disappeared, you can bring it back with [ctrl+Shift+Space].



```
{SinOsc.ar(
```

Figure 6: Helpful info is displayed as you type.

Another shortcut: if you would like to explicitly name your arguments (like `SinOsc.ar(freq: 890)`), try hitting the tab key right after opening the parentheses. SC will autocomplete the

color blue *does not guarantee* that the word is typo-free: for example, if you type Sinosc (with wrong lowercase "o"), it will still show up in blue.

correct argument name for you, in order, as you type (hit tab after the comma for subsequent argument names).

TIP: Create a folder with your own "personalized help files." Whenever you figure out some new trick or learn a new object, write a simple example with explanations in your own words, and save it for the future. It may come in handy a month or a year from now.

The exact same Help files can also be found online at `http://doc.sccode.org/`.

Part IV
SOUND SYNTHESIS AND PROCESSING

At this point you already know quite a lot about the SuperCollider. The last part of this tutorial introduced you to nitty-gritty details about the language itself, from variables to enclosures and more. You also learned how to create interesting `Pbind`s using several members of the Pattern family.

This part of the tutorial will (finally!) introduce you to sound synthesis and processing with SuperCollider. We will start with the topic of Unit Generators (UGens).[*]

24 UGens

You have already seen a few Unit Generators (UGens) in action in sections 3 and 18. What is a UGen? A unit generator is an object that generates sound signals or control signals. These signals are always calculated in the server. There are many classes of unit generators, all of which derive from the class UGen. `SinOsc` and `LFNoise0` are examples of UGens. For more details, look at the Help files called "Unit Generators and Synths," and "Tour of UGens."

When you played your Pbinds earlier in this tutorial, the default sound was always the same: a simple piano-like synth. That synth is made of a combination of unit generators.[†] You will learn

[*]Most tutorials start with Unit Generators right away; in this intro to SC, however, we chose to emphasize the Pattern family first (`Pbind` and friends) for a different pedagogical approach.

[†]Since you used `Pbind`s to make sound in SuperCollider so far, you may be tempted to think: *"I see, so the `Pbind` is a Unit Generator!"* That's not the case. `Pbind` is not a Unit Generator—it is just a recipe for making musical events (score). *"So the `EventStreamPlayer`, the thing that results when I call `play` on a `Pbind`, THAT must be a UGen!"* The answer is still no. The `EventStreamPlayer` is just the player, like a pianist, and

how to combine unit generators to create all sorts of electronic instruments with synthetic and processed sounds. The next example builds up from your first sine wave to create an electronic instrument that you can perform live with the mouse.

24.1 Mouse control: instant Theremin

Here's a simple synth that you can perform live. It is a simulation of the Theremin, one of the oldest electronic music instruments:

```
{SinOsc.ar(freq: MouseX.kr(300, 2500), mul: MouseY.kr(0, 1))}.play;
```

If you don't know what a Theremin is, please stop everything right now and search for "Clara Rockmore Theremin" ou YouTube. Then come back here and try to play the Swan song with your SC Theremin.

`SinOsc`, `MouseX`, and `MouseY` are UGens. `SinOsc` is generating the sine wave tone. The other two are capturing the motion of your cursor on the screen (X for horizontal motion, Y for vertical motion), and using the numbers to feed frequency and amplitude values to the sine wave. Very simple, and a lot of fun.

24.2 Saw and Pulse; plot and scope

The theremin above used a sine oscillator. There are other waveforms you could use to make the sound. Run the lines the below—they use the convenient `plot` method—to look at the shape of

the pianist does not generate sound. In keeping with this limited metaphor, the *instrument piano* is the thing that actually vibrates and generates sound. That is a more apt analogy for a UGen: it's not the score, nor the player: it's the instrument. When you made music with `Pbinds` earlier, SC would create an `EventStreamPlayer` to play your score with the built-in piano synth. You didn't have to worry about creating the piano or any of that—SuperCollider did all the work under the hood for you. That hidden piano synth is made of a combination of a few Unit Generators.

`SinOsc`, and compare it to `Saw` and `Pulse`. The lines below won't make sound—they just let you visualize a snapshot of the waveform.

```
{ SinOsc.ar }.plot; // sine wave
{ Saw.ar }.plot; // sawtooth wave
{ Pulse.ar }.plot; // square wave
```

Now rewrite your theremin line replacing `SinOsc` with `Saw`, then `Pulse`. Listen how different they sound. Finally, try `.scope` instead of `.play` in your theremin code, and you will be able to watch a representation of the waveform in real time (a "Stethoscope" window will pop up).

25 Audio rate, control rate

It is very easy to spot a UGen in SuperCollider code: they are nearly always followed by the messages `.ar` or `.kr`. These letters stand for Audio Rate and Control Rate. Let's see what this means.

From the "Unit Generators and Synths" Help file:

> A unit generator is created by sending the `ar` or `kr` message to the unit generator's class object. The `ar` message creates a unit generator that runs at audio rate. The `kr` message creates a unit generator that runs at control rate. Control rate unit generators are used for low frequency or slowly changing control signals. Control rate unit generators produce only a single sample per control cycle and therefore use less processing power than audio rate unit generators.[*]

In other words: when you write `SinOsc.ar`, you are sending the message "audio rate" to the `SinOsc` UGen. Assuming your computer is running at the common sampling rate of 44100 Hz,

[*] `http://doc.sccode.org/Guides/UGens-and-Synths.html`

this sine oscillator will generate 44100 samples per second to send out to the loudspeaker. Then we hear the sine wave.

Take a moment to think again about what you just read: when you send the `ar` message to a UGen, you are telling it to generate *forty-four thousand and one hundred* numbers per second. That's a lot of numbers. You write `{SinOsc.ar}.play` in the language, and the language communicates your request to the server. The actual job of generating all those samples is done by the server, the "sound engine" of SuperCollider.

Now, when you use `kr` instead of `ar`, the job is also done by the server, but there are a couple of differences:

1. The amount of numbers generated per second with `.kr` is much smaller.`{SinOsc.ar}.play` generates 44100 numbers per second, while `{SinOsc.kr}.play` outputs a little under 700 numbers per second (if you are curious, the exact amount is 44100 / 64, where 64 is the so-called "control period.")

2. The signal generated with `kr` does not go to your loudspeakers. Instead, it is normally used to control parameteres of other signals—for example, the `MouseX.kr` in your theremin was controlling the frequency of a `SinOsc`.

OK, so UGens are these incredibly fast generators of numbers. Some of these numbers become sound signals; others become control signals. So far so good. But what numbers are these, after all? Big? Small? Positive? Negative? It turns out they are very small numbers, often between -1 and +1, sometimes just between 0 and 1. All UGens can be divided in two categories according to the range of numbers they generate: unipolar UGens, and bipolar UGens.

Unipolar UGens generate numbers between 0 and 1.

Bipolar UGens generate numbers between -1 and +1.

25.1 The `poll` method

Snooping into the output of some UGens should make this clearer. We can't possibly expect SuperCollider to print thousands of numbers per second in the Post window, but we can ask it to print a few of them every second, just for a taste. Type and run the following lines one at a time (make sure your server is running), and watch the Post window:

```
// just watch the Post window (no sound)
{SinOsc.kr(1).poll}.play;
// hit ctrl+period, then evaluate the next line:
{LFPulse.kr(1).poll}.play;
```

The examples make no sound because we are using `kr`—the result is a control signal, so nothing is sent to the loudspeakers. The point here is just to watch the typical output of a `SinOsc`. The message `poll` grabs 10 numbers per second from the `SinOsc` output and prints them out in the Post window. The argument 1 is the frequency, which just means the the sine wave will take one second to complete a whole cycle. Based on what you observed, is `SinOsc` unipolar or bipolar? What about `LFPulse`?[6]

Bring down the volume before evaluating the next line, then bring it back up slowly. You should hear soft clicks.

```
{LFNoise0.ar(1).poll}.play;
```

Because we sent it the message `ar`, this Low Frequency Noise generator is churning out 44100 samples per second to your sound card—it's an audio signal. Each sample is a number between -1 and +1 (so it's a bipolar UGen). With `poll` you are only seeing ten of those per second. `LFNoise0.ar(1)` picks a new random number every second. All of this is done by the server.

Stop the clicks with [ctrl+.] and try changing the frequency of `LFNoise0`. Try numbers like 3, 5, 10, and then higher. Watch the output numbers and hear the results.

26 UGen arguments

Most of the time you will want to specify arguments to the UGens you are using. You have already seen that: when you write `{SinOsc.ar(440)}.play`, the number `440` is an argument to the `SinOsc.ar`; it specifies the frequency that you will hear. You can be explicit about naming the arguments, like this: `{SinOsc.ar(freq: 440, mul: 0.5)}.play`. The argument names are `freq` and `mul` (note the colon immediately after the words in the code). The `mul` stands for "multiplier", and is essentially the amplitude of the waveform. If you don't specify `mul`, SuperCollider uses the default value of 1 (maximum amplitude). Using a `mul: 0.5` means multiplying the waveform by half, in other words, it will play at half of the maximum amplitude.

In your theremin code, the `SinOsc` arguments `freq` and `mul` were explicitly named. You may recall that `MouseX.kr(300, 2500)` was used to control the frequency of the theremin. `MouseX.kr` takes two arguments: a low and a high boundary for its output range. That's what the numbers 300 and 2500 were doing there. Same thing for the `MouseY.kr(0, 1)` controlling amplitude. Those arguments inside the mouse UGens were not explicitly named, but they could be.

How do you find out what arguments a UGen will accept? Simply go to the corresponding Help file: double click the UGen name to select it, and hit [ctrl+D] to open the Documentation page. Do that now for, say, MouseX. After the Description section you see the Class Methods section. Right there, it says that the arguments of the `kr` method are minval, maxval, warp, and lag. From the same page you can learn what each of them does.

Whenever you don't provide an argument, SC will use the default values that you see in the Help file. If you don't name the arguments explicitly, you have to provide them in the exact order shown in the Help file. If you do name them explicitly, you can put them in any order, and even skip some in the middle. Naming arguments explicitly is also a good learning tool, as it helps you to better understand your code. An example is given below.

```
// minval and maxval provided in order, no keywords
```

```
{MouseX.kr(300, 2500).poll}.play;
// minval, maxval and lag provided, skipped warp
{MouseX.kr(minval: 300, maxval: 2500, lag: 10).poll}.play;
```

27 Scaling ranges

The real fun starts when you use some UGens to control the parameters of other UGens. The theremin example did just that. Now you have all the tools to understand exactly what was going on in one of the examples of section 3. The three last lines of the example demonstrate step-by-step how the `LFNoise0` is used to control frequency:

```
{SinOsc.ar(freq: LFNoise0.kr(10).range(500, 1500), mul: 0.1)}.play;

// Breaking it down:
{LFNoise0.kr(1).poll}.play; // watch a simple LFNoise0 in action
{LFNoise0.kr(1).range(500, 1500).poll}.play; // now with .range
{LFNoise0.kr(10).range(500, 1500).poll}.play; // now faster
```

27.1 Scale with the method `range`

The method `range` simply rescales the output of a UGen. Remember, `LFNoise0` produces numbers between -1 and +1 (it is a bipolar UGen). Those raw numbers would not be very useful to control frequency (we need sensible numbers in the human hearing range). The `.range` takes the output betwen -1 and +1 and scales it to whatever low and high values you provide as arguments (in this case, 500 and 1500). The number 10, which is the argument to `LFNoise0.kr`, specifies the frequency of the UGen: how many times per second it will pick a new random number.

In short: in order to use a UGen to control some parameter of another UGen, first you need to know what range of numbers you want. Are the numbers going to be frequencies? Do you want them between, say, 100 and 1000? Or are they amplitudes? Perhaps you want amplitudes to be between 0.1 (soft) and 0.5 (half the maximum)? Or are you trying to control number of harmonics? Do you want it to be between 5 and 19?

Once you know the range you need, use the method `.range` to make the controlling UGen do the right thing.

Exercise: write a simple line code that plays a sine wave, the frequency of which is controlled by a `LFPulse.kr` (provide appropriate arguments to it). Then, use the `.range` method to scale the output of `LFPulse` into something that you want to hear.

27.2 Scale with `mul` and `add`

Now you know how to scale the output of UGens in the server using the method `.range`. The same thing can be accomplished on a more fundamental level by using the arguments `mul` and `add`, which pretty much all UGens have. The code below shows the equivalence between `range` and `mul/add` approaches, both with a bipolar UGen and a unipolar UGen.

```
// This:
{SinOsc.kr(1).range(100, 200).poll}.play;
// ...is the same as this:
{SinOsc.kr(1, mul: 50, add: 150).poll}.play;

// This:
{LFPulse.kr(1).range(100, 200).poll}.play;
// ...is the same as this:
{LFPulse.kr(1, mul: 50, add: 100).poll}.play;
```

Figure 7 helps visualize how `mul` and `add` work in rescaling UGen outputs (a `SinOsc` is used as demonstration).

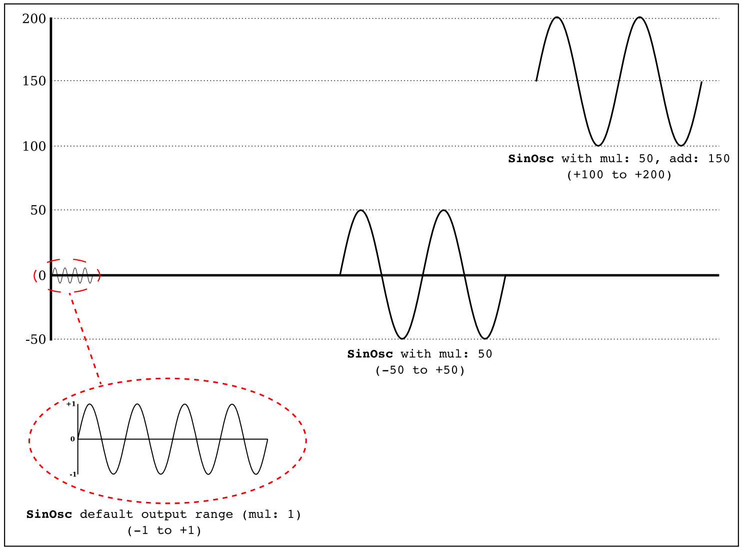


Figure 7: Scaling UGen ranges with mul and add

27.3 `linlin` and friends

For any other arbitrary scaling of ranges, you can use the handy methods `linlin`, `linexp`, `explin`, `expexp`. The method names hint at what they do: convert a linear range to another linear range (`linlin`), linear to exponential (`linexp`), etc.

```
// A bunch of numbers
a = [1, 2, 3, 4, 5, 6, 7];
// Rescale to 0—127, linear to linear
a.linlin(1, 7, 0, 127).round(1);
// Rescale to 0—127, linear to exponential
a.linexp(1, 7, 0.01, 127).round(1); // don't use zero for an exponential range
```

28 Stopping individual synths

Here's a very common way of starting several synths and being able to stop them separately. The example is self-explanatory:

```
// Run one line at a time (don't stop the sound in between):
a = { Saw.ar(LFNoise2.kr(8).range(1000, 2000), mul: 0.2) }.play;
b = { Saw.ar(LFNoise2.kr(7).range(100, 1000), mul: 0.2) }.play;
c = { Saw.ar(LFNoise0.kr(15).range(2000, 3000), mul: 0.1) }.play;
// Stop synths individually:
a.free;
b.free;
c.free;
```

29 The `set` message

Just like with any function (review section 21), arguments specified at the beginning of your synth function are accessible by the user. This allows you to change synth parameters on the fly (while the synth is running). The message `set` is used for that purpose. Simple example:

```
x = {arg freq = 440, amp = 0.1; SinOsc.ar(freq, 0, amp)}.play;
x.set(\freq, 778);
x.set(\amp, 0.5);
x.set(\freq, 920, \amp, 0.2);
x.free;
```

It's good practice to provide default values (like the 440 and 0.1 above), otherwise the synth won't play until you set a proper value to the 'empty' parameters.

30 Audio Buses

Audio buses are used for routing audio signals. They are like the channels of a mixing board. SuperCollider has 128 audio buses by default. There are also control buses (for control signals), but for now let's focus on Audio Buses only.*

Hit [ctrl+M] to open up the Meter window. It shows the levels of all inputs and outputs. Figure 8 shows a screenshot of this window and its correspondence to SuperCollider's default buses. In SuperCollider, audio buses are numbered from 0 to 127. The first eight (0-7) are by default reserved to be the output channels of your sound card. The next eight (8-15) are reserved for the inputs of your sound card. All the others (16 to 127) are free to be used in any way you want, for example, when you need to route audio signals from one UGen to another.

30.1 `Out` and `In` UGens

Now try the following line of code:

```
{Out.ar(1, SinOsc.ar(440, 0, 0.1))}.play; // right channel
```

*We will take a quick look at control buses in section 41.

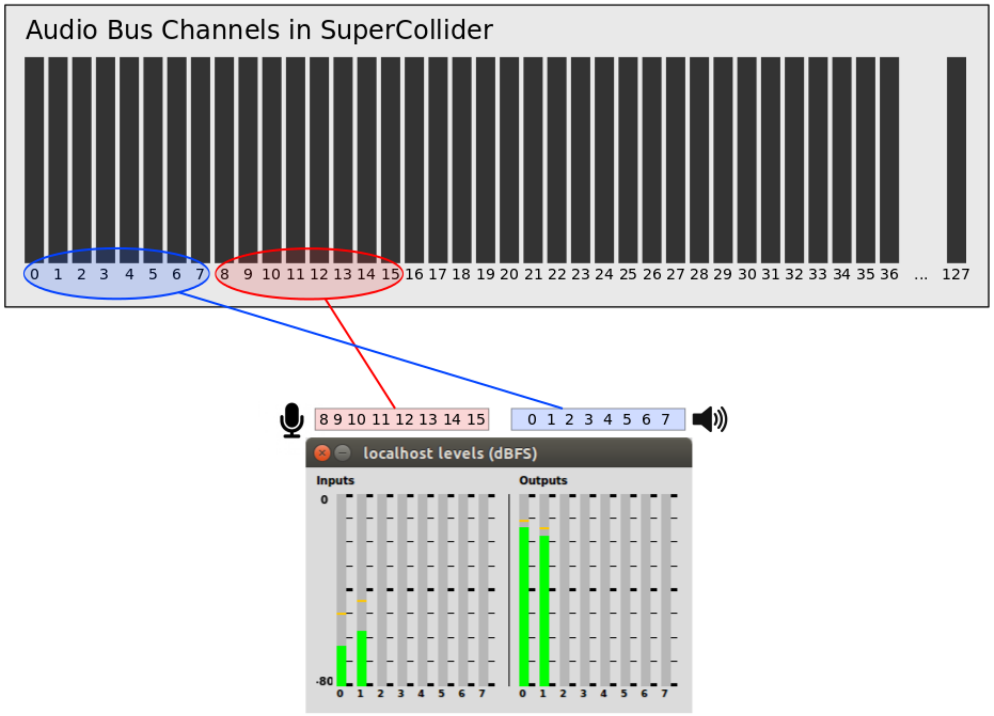


Figure 8: Audio buses and Meter window in SC.

The `Out` UGen takes care of routing signals to specific buses.

The first argument to `Out` is the target bus, that is, where you want this signal to go. In the example above, the number `1` means that we want to send the signal to bus 1, which is the right channel of your sound card.

The second argument of `Out.ar` is the actual signal that you want to "write" into that bus. It can be a single UGen, or a combination of UGens. In the example, it is just a sine wave. You should hear it only on your right speaker (or on your right ear if using headphones).

With the Meter window open and visible, go ahead and change the first argument of `Out.ar`: try any number between 0 and 7, and watch the meters. You will see that the signal goes wherever you tell it to.

TIP: Most likely you have a sound card that can only play two channels (left and right), so you will only hear the sine tone when you send it to bus 0 or bus 1. When you send it to other buses (3 to 7), you will still see the corresponding meter showing the signal: SC is in fact sending the sound to that bus, but unless you have an 8-channel sound card you will not be able to hear the output of buses 3-7.

One simple example of an audio bus being used for an effect is shown below.

```
// start the effect
f = {Out.ar(0, BPF.ar(in: In.ar(55), freq: MouseY.kr(1000, 5000), rq: 0.1))}.play;
// start the source
n = {Out.ar(55, WhiteNoise.ar(0.5))}.play;
```

The first line declares a synth (stored into the variable `f`) consisting of a filter UGen (a Band Pass Filter). A band pass filter takes any sound as input, and *filters out all frequencies except the one frequency region that you want to let through.* `In.ar` is the UGen we use to read from an audio bus; so with `In.ar(55)` being used as input of the `BPF`, any sound that we send to bus

55 will be passed to the band pass filter. Notice that this first synth does not make any sound at first: when you evaluate the first line, bus 55 is still empty. It will only make sound when we send some audio into bus 55, which happens on the second line.

The second line creates a synth and stores it into the variable `n`. This synth simply generates white noise, and outputs it *not to the loudspeakers directly, but to audio bus 55 instead.* That is precisely the bus that our filter synth is listening to, so as soon as you evaluate the second line, you should start hearing the white noise being filtered by synth `f`. In short, the routing looks like this:

$$\textit{noise synth} \rightarrow \textit{bus 55} \rightarrow \textit{filter synth}$$

The order of execution is important. The previous example won't work if you evaluate the source *before* the effect. This will be discussed in more detail in section 42, "Order of Execution."

One last thing: when you wrote in earlier examples synths like `{SinOsc.ar(440)}.play`, SC was actually doing `{Out.ar(0, SinOsc.ar(440))}.play` under the hood: it assumed you wanted to send the sound out to bus 0, so it automatically wrapped the first UGen with an `Out.ar(0, ...)` UGen. In fact a few more things are happening there behind the scenes, but we'll come back to this later (section 39).

31 Microphone Input

The example below shows how you can easily access sound input from your sound card with the `SoundIn` UGen.[*]

[*]Since `In.ar` will read from any bus, and you know that your soundcard inputs are by default assigned to buses 8-15, you could write `In.ar(8)` to get sound from your microphone. That works just fine, but `SoundIn.ar` is a more convenient option.

```
// Warning: use headphones to avoid feedback
{SoundIn.ar(0)}.play; // same as In.ar(8): takes sound from the first input bus

// Stereo version
{SoundIn.ar([0, 1])}.play; // first and second inputs

// Some reverb just for fun?
{FreeVerb.ar(SoundIn.ar([0, 1]), mix: 0.5, room: 0.9)}.play;
```

32 Multichannel Expansion

With your Meter window open—[ctrl+M]—, watch this.

```
{Out.ar(0, Saw.ar(freq: [440, 570], mul: Line.kr(0, 1, 10)))}.play;
```

We are using a nice `Line.kr` UGen to ramp up the amplitude from 0 to 1 in 10 seconds. That's neat. But there is a more interesting magic going on here. Did you notice that there are 2 channels of output (left and right)? Did you hear that there is a different note on each channel? And that those two notes come from a *list*—[440, 570]—that is passed to `Saw.ar` as its `freq` argument?

This is called Multichannel Expansion.

David Cottle jokes that "multichannel expansion is one [application of arrays] that borders on voodoo."[*] It is one of the most powerful and unique features of SuperCollider, and one that may puzzle people at first.

In a nutshell: if you use an array anywhere as one of the arguments of a UGen, *the entire patch is duplicated*. The number of copies created is *the number of items in the array*. These duplicated

[*]Cottle, D. "Beginner's Tutorial." The SuperCollider Book, MIT Press, 2011, p. 14

UGens are sent out to as many *adjacent buses* as needed, starting from the bus specified as the first argument of `Out.ar`.

In the example above, we have `Out.ar(0, ... )`. The `freq` of the Saw wave is an array of two items: `[440, 570]`. What does SC do? It "multichannel expands," creating two copies of the entire patch. The first copy is a sawtooth wave with frequency 440 Hz, sent out to bus 0 (your left channel); the second copy is a sawtooth wave with frequency 570 Hz, sent out to bus 1 (your right channel)!

Go ahead and check that for yourself. Change those two frequencies to any other values you like. Listen to the results. One goes to the left channel, the other goes to the right channel. Go even further, and add a third frequency to the list (say, `[440, 570, 980]`). Watch the Meter window. You will see that the first three outputs are lighting up (but you will only be able to hear the third one if you have a multichannel sound card).

What's more: you can use additional arrays in other arguments of the same UGen, or in arguments of other UGens in the same synth. SuperCollider will do the housekeeping and generate synths that follow those values accordingly. For example: right now both frequencies [440, 570] are fading in from 0 to 1 in 10 seconds. But change the code to `Line.kr(0, 1, [1, 15])` and you'll have the 440 Hz tone take 1 second to fade in, and the 570 Hz tone take 15 seconds to fade in. Try it.

Exercise: listen to this simulation of a "busy tone" of an old telephone. It uses multichannel expansion to create two sine oscillators, each playing a different frequency on a different channel. Make the left channel pulse 2 times per second, and the right channel pulse 3 times per second.[7]

```
a = {Out.ar(0, SinOsc.ar(freq: [800, 880], mul: LFPulse.ar(2)))}.play;
a.free;
```

33 The Bus object

Here's an example that uses everything you just learned in the previous two sections: audio buses, and multichannel expansion.

```
// Run this first ('turn reverb on' -- you won't hear anything at first)
r = {Out.ar(0, FreeVerb.ar(In.ar(55, 2), mix: 0.5, room: 0.9, mul: 0.4))}.play;

// Now run this second ('feed the busy tone into the reverb bus')
a = {Out.ar(55, SinOsc.ar([800, 880], mul: LFPulse.ar(2)))}.play;
a.free;
```

Thanks to multichannel expansion, the busy tone uses two channels. When (in synth `a`) we route the busy tone to bus 55, two buses are actually being used up—number 55, and the immediately adjacent bus 56. In the reverb (synth `r`), we indicate with `In.ar(55, 2)` that we want to read 2 channels starting from bus 55: so both 55 and 56 get into the reverb. The output of the reverb is in turn also expanded to two channels, so synth `r` sends sound out to buses 0 and 1 (left and right channels of our sound card).

Now, this choice of bus number (55) to connect a source synth to an effect synth was arbitrary: it could have been any other number between 16 and 127 (remember, buses 0-15 are reserved for sound card outputs and inputs). How inconvenient it would be if we had to keep track of bus numbers ourselves. As soon as our patches grew in complexity, imagine the nightmare: "What bus number did I choose again for reverb? Was it 59 or 95? What about the bus number for my delay? I guess it was 27? Can't recall..." and so on and so forth.

SuperCollider takes care of this for you with Bus objects. We only hand-assigned the infamous bus 55 in the examples above for the sake of demonstration. In your daily SuperCollider life, you should simply use the Bus object. The Bus object does the job of choosing an available bus for you and keeping track of it. This is how you use it:

```
// Create the bus
~myBus = Bus.audio(s, 2);
// Turn on the reverb: read from myBus (source sound)
r = {Out.ar(0, FreeVerb.ar(In.ar(~myBus, 2), mix: 0.5, room: 0.9, mul: 0.4))}.play;
// Feed the busy tone into ~myBus
b = {Out.ar(~myBus, SinOsc.ar([800, 880], mul: LFPulse.ar(2)))}.play;
// Free both synths
r.free; b.free;
```

The first argument of `Bus.audio` is the variable `s`, which stands for the server. The second argument is how many channels you need (2 in the example). Then you store that into a variable with a meaningful name (`~myBus` in the example, but it could be `~reverbBus`, `~source`, `~tangerine`, or whatever makes sense to you in your patch). After that, whenever you need to refer to that bus, just use the variable you created.

34 Panning

Panning is the spreading of an audio signal into a stereo or multichannel sound field. Here's a mono signal bouncing between left and right channel thanks to `Pan2`:*

```
p = {Pan2.ar(in: PinkNoise.ar, pos: SinOsc.kr(2), level: 0.1)}.play;
p.free;
```

From the `Pan2` Help file, you can see that the argument `pos` (position) expects a number between -1 (left) and +1 (right), 0 being center. That's why you can use a `SinOsc` directly into that

*For multichannel panning, take a look at `Pan4` and `PanAz`. Advanced users may want to take a look at SuperCollider plug-ins for Ambisonics.

argument: the sine oscillator is a bipolar UGen, so it outputs numbers between -1 and +1 by default.

Here's a more elaborated example. A sawtooth wave goes through a very sharp band pass filter (`rq: 0.01`). Notice the use of local variables to modularize different parts of the code. Analyze and try to understand as much as you can in the example above. Then answer the questions below.

```
(
x = {
        var lfn = LFNoise2.kr(1);
        var saw = Saw.ar(
                freq: 30,
                mul: LFPulse.kr(
                        freq: LFNoise1.kr(1).range(1, 10),
                        width: 0.1));
        var bpf = BPF.ar(in: saw, freq: lfn.range(500, 2500), rq: 0.01, mul: 20);
        Pan2.ar(in: bpf, pos: lfn);
}.play;
)
x.free;
```

Questions:

(a) The variable `lfn` is used in two different places. Why? (What is the result?)

(b) What happens if you change the `mul:` argument of the `BPF` from 20 to 10, 5, or 1? Why such a high number as 20 was used?

(c) What part of the code is controlling the rhythm?

Answers are at the end of this document.[8]

35 Mix and Splay

Here's a cool trick. You can use multichannel expansion to generate complex sounds, and then mix it all down to mono or stereo with `Mix` or `Splay`:

```
// 5 channels output (watch Meter window)
a = { SinOsc.ar([100, 300, 500, 700, 900], mul: 0.1) }.play;
a.free;
// Mix it down to mono:
b = { Mix(SinOsc.ar([100, 300, 500, 700, 900], mul: 0.1)) }.play;
b.free;
// Mix it down to stereo (spread evenly from left to right)
c = { Splay.ar(SinOsc.ar([100, 300, 500, 700, 900], mul: 0.1)) }.play;
c.free
// Fun with Splay:
(
d = {arg fundamental = 110;
        var harmonics = [1, 2, 3, 4, 5, 6, 7, 8, 9];
        var snd = BPF.ar(
                in: Saw.ar(32, LFPulse.ar(harmonics, width: 0.1)),
                freq: harmonics * fundamental,
                rq: 0.01,
                mul: 20);
        Splay.ar(snd);
}.play;
)
d.set(\fundamental, 100); // change fundamental just for fun
d.free;
```

Can you see the multichannel expansion at work in that last `Splay` example? The only difference is that the array is first stored into a variable (`harmonics`) before being used in the

UGens. The array `harmonics` has 9 items, so the synth will expand to 9 channels. Then, just before the `.play`, `Splay` takes in the array of nine channels and mix it down to stereo, spreading the channels evenly from left to right.*

`Mix` has another nice trick: the method `fill`. It creates an array of synths and mixes it down to mono all at once.

```
// Instant cluster generator
c = { Mix.fill(16, {SinOsc.ar(rrand(100, 3000), mul: 0.01)}) }.play;
c.free;
// A note with 12 partials of decreasing amplitudes
(
n = { Mix.fill(12, {arg counter;
        var partial = counter + 1; // we want it to start from 1, not 0
        SinOsc.ar(partial * 440, mul: 1/partial.squared) * 0.1
        })
}.play;
FreqScope.new;
)
n.free;
```

You give two things to `Mix.fill`: the size of the array you want to create, and a function (in curly braces) that will be used to fill up the array. In the first example above, `Mix.fill` evaluates the function 16 times. Note that the function includes a variable component: the frequency of the sine oscillator can be any random number between 100 and 3000. Sixteen sine waves will be created, each with a different random frequency. They will all be mixed down to mono, and you'll hear the result on your left channel. The second example shows that the function can

*The last line before the `.play` could be explicitly written as `Out.ar(0, Splay.ar(snd))`. Remember that SuperCollider is graciously filling in the gaps and throwing in a `Out.ar(0...)` there—that's how the synth knows it should play into your channels left (bus 0) and right (bus 1).

take a “counter” argument that keeps track of the number of iterations (just like `Array.fill`). Twelve sine oscillators are generated following the harmonic series, and mixed down to a single note in mono.

36 Playing an audio file

First, you have to load the sound file into a buffer. The second argument to `Buffer.read` is the path of your sound file between double quotes. You will need to change that accordingly so that it points to a WAV or AIFF file on your computer. After buffers are loaded, simply use the `PlayBuf` UGen to play them back in various ways.

TIP: A quick way to get the correct path of a sound file saved on your computer is drag the file onto a blank SuperCollider document. SC will give you the full path automatically, already in double quotes!

```
// Load files into buffers:
~buf1 = Buffer.read(s, "/home/Music/wheels-mono.wav"); // one sound file
~buf2 = Buffer.read(s, "/home/Music/mussorgsky.wav"); // another sound file

// Playback:
{PlayBuf.ar(1, ~buf1)}.play; // number of channels and buffer
{PlayBuf.ar(1, ~buf2)}.play;

// Get some info about the files:
[~buf1.bufnum, ~buf1.numChannels, ~buf1.path, ~buf1.numFrames];
[~buf2.bufnum, ~buf2.numChannels, ~buf2.path, ~buf2.numFrames];

// Changing playback speed with 'rate'
```

```
{PlayBuf.ar(numChannels: 1, bufnum: ~buf1, rate: 2, loop: 1)}.play;
{PlayBuf.ar(1, ~buf1, 0.5, loop: 1)}.play; // play at half the speed
{PlayBuf.ar(1, ~buf1, Line.kr(0.5, 2, 10), loop: 1)}.play; // speeding up
{PlayBuf.ar(1, ~buf1, MouseY.kr(0.5, 3), loop: 1)}.play; // mouse control

// Changing direction (reverse)
{PlayBuf.ar(1, ~buf2, -1, loop: 1)}.play; // reverse sound
{PlayBuf.ar(1, ~buf2, -0.5, loop: 1)}.play; // play at half the speed AND reversed
```

37 Synth Nodes

In the previous `PlayBuf` examples, you had to hit [ctrl+.] after each line to stop the sound. In other examples, you assigned the synth to a variable (like `x = {WhiteNoise.ar}.play`) so that you could stop it directly with `x.free`.

Every time you create a synth in SuperCollider, you know that it runs in the server, our "sound engine." Each running synth in the server is represented by a *node*. We can take a peek at this tree of nodes with the command `s.plotTree`. Try it. A window named `NodeTree` will open.

```
// open the GUI
s.plotTree;
// run these one by one (don't stop the sound) and watch the Node Tree:
w = { SinOsc.ar(60.midicps, 0, 0.1) }.play;
x = { SinOsc.ar(64.midicps, 0, 0.1) }.play;
y = { SinOsc.ar(67.midicps, 0, 0.1) }.play;
z = { SinOsc.ar(71.midicps, 0, 0.1) }.play;
w.free;
x.free;
y.free;
```

```
z.free;
```

Every rectangle that you see in the Node Tree is a synth node. Each synth gets a temporary name (something like temp_101, temp_102, etc) and stays there for as long as it is running. Try now playing the four sines again, and hit [ctrl+.] (watch the Node Tree window). The shortcut [ctrl+.] ruthlessly stops at once all nodes that are running in the Server. On the other hand, with the `.free` method, you can be more subtle and free up specific nodes one at a time.

One thing that is important to realize is that synths may stay running in the server even if they are generating only silence. Here's an example. The amplitude of this `WhiteNoise` UGen will go from 0.2 to 0 in two seconds. After that, we don't hear anything. But you will see that the synth node is still there, and won't go away until you free it.

```
// Evaluate and watch the Node Tree window for a few seconds
x = {WhiteNoise.ar(Line.kr(0.2, 0, 2))}.play;
x.free;
```

37.1 The glorious doneAction: 2

Luckily there is a way to make synths smarter in that regard: for example, wouldn't it be great if we could ask `Line.kr` to notify the synth when it has finished its job (the ramp from 0.2 to 0), upon which the synth would automatically free itself up?

Enter the argument `doneAction: 2` to solve all our problems.

Play the examples below and compare how they behave with and without doneAction: 2. Keep watching the Node Tree as you run the lines.

```
// without doneAction: 2
{WhiteNoise.ar(Line.kr(0.2, 0, 2))}.play;
{PlayBuf.ar(1, ~buf1)}.play; // PS. this assumes you still have your sound file
    loaded into ~buf1 from previous section
```

```

// with doneAction: 2
{WhiteNoise.ar(Line.kr(0.2, 0, 2, doneAction: 2))}.play;
{PlayBuf.ar(1, ~buf1, doneAction: 2)}.play;
```

The synths with doneAction: 2 will free themselves up automatically as soon as their job is done (that is, as soon as the `Line.kr` ramp is over in the first example, and as soon as the `PlayBuf.ar` has finished playing the sound file in the second example). This knowledge will be very useful in the next section: Envelopes.

38 Envelopes

Up to now most of our examples have been continuous sounds. It is about time to learn how to shape the amplitude envelope of a sound. A good example to start with is a percussive envelope. Imagine a cymbal crash. The time it takes for the sound to go from silence to maximum amplitude is very small—a few miliseconds, perhaps. This is called the *attack time*. The time it takes for the sound of the cymbal to decrease from maximum amplitude back to silence (zero) is a little longer, maybe a few seconds. This is called the *release time*.

Think of an amplitude envelope simply as a number that changes over time to be used as the multiplier (`mul`) of any sound-producing UGen. These numbers should be between 0 (silence) and 1 (full amp), because that's how SuperCollider understands amplitude. You may have realized by now that the last example already included an amplitude envelope: in `{WhiteNoise.ar(Line.kr(0.2, 0, 2, doneAction: 2))}.play`, we make the amplitude of the white noise go from 0.2 to 0 in 2 seconds. A `Line.kr`, however, is not a very flexible type of envelope.

`Env` is the object you will be using all the time to define all sorts of envelopes.`Env` has many useful methods; we can only look at a few here. Feel free to explore the `Env` Help file to learn more.

38.1 Env.perc

`Env.perc` is a handy way to get a percussive envelope. It takes in four arguments: attackTime, releaseTime, level, and curve. Let's take a look at some typical shapes, outside of any synth.

```
Env.perc.plot; // using all default args
Env.perc(0.5).plot; // attackTime: 0.5
Env.perc(attackTime: 0.3, releaseTime: 2, level: 0.4).plot;
Env.perc(0.3, 2, 0.4, 0).plot; // same as above, but curve:0 means straight lines
```

Now we can simply hook it up in a synth like this:

```
{PinkNoise.ar(Env.perc.kr(doneAction: 2))}.play; // default Env.perc args
{PinkNoise.ar(Env.perc(0.5).kr(doneAction: 2))}.play;
{PinkNoise.ar(Env.perc(0.3, 2, 0.4).kr(2))}.play;
{PinkNoise.ar(Env.perc(0.3, 2, 0.4, 0).kr(2))}.play;
```

All you have to do is to add the bit `.kr(doneAction: 2)` right after `Env.perc`, and you are good to go. In fact, you can even remove the explicit declaration of doneAction in this case and simply have `.kr(2)`. The `.kr` is telling SC to "perform" this envelope at control rate (like other control rate signals we have seen before).

38.2 Env.triangle

`Env.triangle` takes only two arguments: duration, and level. Examples:

```
// See it:
Env.triangle.plot;
// Hear it:
```

```
{SinOsc.ar([440, 442], mul: Env.triangle.kr(2))}.play;
// By the way, an envelope can be a multiplier anywhere in your code
{SinOsc.ar([440, 442]) * Env.triangle.kr(2)}.play;
```

38.3 Env.linen

Env.linen describes a line envelope with attack, sustain portion, and release. You can also specify level and curve type. Example:

// See it: Env.linen.plot; // Hear it: SinOsc.ar([300, 350], mul: Env.linen(0.01, 2, 1, 0.2).kr(2)).play;

38.4 Env.pairs

Need more flexibility? With **Env.pairs** you can have envelopes of any shape and duration you want. **Env.pairs** takes two arguments: an array of [time, level] pairs, and a type of curve (see the Env Help file for all available curve types).

```
(
{
	var env = Env.pairs([[0, 0], [0.4, 1], [1, 0.2], [1.1, 0.5], [2, 0]], \lin);
	env.plot;
	SinOsc.ar([440, 442], mul: env.kr(2));
}.play;
)
```

Read the array of pairs like this:

At time 0, be at level 0;
At time 0.4, be at level 1;

At time 1, be at level 0.2;
At time 1.1, be at level 0.5;
At time 2, be at level 0.

38.4.1 Envelopes—not just for amplitude

Nothing is stopping you from using these same shapes to control something other than amplitude. You just need to scale them to the desired range of numbers. For example, you can create an envelope to control change of frequencies over time:

```
(
{
	var freqEnv = Env.pairs([[0, 100], [0.4, 1000], [0.9, 400], [1.1, 555], [2,
	    440]], \lin);
	SinOsc.ar(freqEnv.kr, mul: 0.2);
}.play;
)
```

Envelopes are a powerful way to control any synth parameters that need to change over time.

38.5 ADSR Envelope

All envelopes seen up to now have one thing in common: they have a fixed, pre-defined duration. There are situations, however, when this type of envelope is not adequate. For example, imagine you are playing on a MIDI keyboard. The *attack* of the note is triggered when you press a key. The *release* is when you take your finger off the key. But the amount of time that you keep the finger down is not known in advance. What we need in this case is a so-called "sustained envelope." In other words, after the attack portion, the envelope must hold the note for an indefinite amount of time, and only trigger the release portion after some kind of cue, or message—i.e., the moment you 'release the key'.

An ASR (Attack, Sustain, Release) envelope fits the bill. A more popular variation of it is the ADSR envelope (Attack, Decay, Sustain, Release). Let's look at both.

```
// ASR
// Play note ('press key')
// attackTime: 0.5 seconds, sustainLevel: 0.8, releaseTime: 3 seconds
x = {arg gate = 1, freq = 440; SinOsc.ar(freq: freq, mul: Env.asr(0.5, 0.8, 3).kr(
    doneAction: 2, gate: gate))}.play;
// Stop note ('finger off the key' — activate release stage)
x.set(\gate, 0); // alternatively, x.release

// ADSR (attack, decay, sustain, release)
// Play note:
(
d = {arg gate = 1;
        var snd, env;
        env = Env.adsr(0.01, 0.4, 0.7, 2);
        snd = Splay.ar(BPF.ar(Saw.ar((32.1, 32.2..33)), LFNoise2.kr(12).range(100,
            1000), 0.05, 10));
        Out.ar(0, snd * env.kr(doneAction: 2, gate: gate));
}.play;
)
// Stop note:
d.release; // this is equivalent to d.set(\gate, 0);
```

Key concepts:

Attack The time (in seconds) that it takes to go from zero (silence) to peak amplitude

Decay The time (in seconds) that it takes to go down from peak amplitude to sustain amplitude

Sustain The amplitude (between 0 and 1) at which to hold the note (important: this has nothing to do with time)

Release The time (in seconds) that it takes to go from sustain level to zero (silence).

Since sustained envelopes do not have a total duration known in advance, they need a notification as to when to start (trigger the attack) and when to stop (trigger the release). This notification is called a *gate*. The gate is what tells the envelope to 'open' (1) and 'close' (0), thus starting and stopping the note.

For an ASR or ADSR envelope to work in your synth, you must declare a `gate` argument. Normally the default is `gate = 1` because you want the synth to start playing right away. When you want the synth to stop, simply send either the `.release` or `.set(\gate, 0)` message: the release portion of the envelope will be then triggered. For example, if your release time is 3, the note will take three seconds to fade away *from the moment you send the message* `.set(\gate, 0)`.

39 Synth Definitions

So far we have been seamlessly *defining* synths and *playing* them right away. In addition, the `.set` message gave us some flexibility to alter synth controls in real time. However, there are situations when you may want to just define your synths first (without playing them immediately), and only play them later. In essence, this means we have to separate the moment of writing down the recipe (the synth definition) from the moment of baking the cake (creating the sound).

39.1 SynthDef and Synth

SynthDef is what we use to "write the recipe" for a synth. Then you can play it with **Synth**. Here is a simple example.

```
// Synth definition with SynthDef object
SynthDef("mySine1", {Out.ar(0, SinOsc.ar(770, 0, 0.1))}).add;
// Play a note with Synth object
x = Synth("mySine1");
x.free;

// A slightly more flexible example using arguments
// and a self-terminating envelope (doneAction: 2)
SynthDef("mySine2", {arg freq = 440, amp = 0.1;
        var env = Env.perc(level: amp).kr(2);
        var snd = SinOsc.ar(freq, 0, env);
        Out.ar(0, snd);
}).add;

Synth("mySine2"); // using default values
Synth("mySine2", [\freq, 770, \amp, 0.2]);
Synth("mySine2", [\freq, 415, \amp, 0.1]);
Synth("mySine2", [\freq, 346, \amp, 0.3]);
Synth("mySine2", [\freq, rrand(440, 880)]);
```

The first argument to **SynthDef** is a user-defined name for the synth. The second argument is a function where you specify the UGen graph (that's how your combination of UGens is called). Note that you have to explicitly use **Out.ar** to indicate to which bus you want to send the signal to. Finally, **SynthDef** receives the message **.add** at the end, which means you are adding it to the collection of synths that SC knows about. This will be valid until you quit SuperCollider.

After you have created one or more synth definitions with SynthDef, you can play them with Synth: the first argument is the name of the synth definition you want to use, and the second (optional) argument is an array with any parameters you may want to specify (freq, amp, etc.)

39.2 Example

Here's a longer example. After the SynthDef is added, we use an array trick to create a 6-note chord with random pitches and amplitudes. Each synth is stored in one of the slots of the array, so we can release them independently.

```
// Create SynthDef
(
SynthDef("wow", {arg freq = 60, amp = 0.1, gate = 1, wowrelease = 3;
        var chorus, source, filtermod, env, snd;
        chorus = freq.lag(2) * LFNoise2.kr([0.4, 0.5, 0.7, 1, 2, 5, 10]).range(1,
            1.02);
        source = LFSaw.ar(chorus) * 0.5;
        filtermod = SinOsc.kr(1/16).range(1, 10);
        env = Env.asr(1, amp, wowrelease).kr(2, gate);
        snd = LPF.ar(in: source, freq: freq * filtermod, mul: env);
Out.ar(0, Splay.ar(snd))
}).add;
)

// Watch the Node Tree
s.plotTree;

// Create a 6-note chord
a = Array.fill(6, {Synth("wow", [\freq, rrand(40, 70).midicps, \amp, rrand(0.1, 0.5)
    ])}); // all in a single line

```

```
// Release notes one by one
a[0].set(\gate, 0);
a[1].set(\gate, 0);
a[2].set(\gate, 0);
a[3].set(\gate, 0);
a[4].set(\gate, 0);
a[5].set(\gate, 0);

// ADVANCED: run 6-note chord again, then evaluate this line.
// Can you figure out what is happening?
SystemClock.sched(0, {a[5.rand].set(\midinote, rrand(40, 70)); rrand(3, 10)});
```

To help you understand the SynthDef above:

- The resulting sound is the sum of seven closely-tuned sawtooth oscillators going through a low pass filter.
- These seven oscillators are created through multichannel expansion.
- What is the variable `chorus`? It is the frequency multiplied by a `LFNoise2.kr`. The multichannel expansion starts here, because an array with 7 items is given as an argument to LFNoise2. The result is that seven copies of LFNoise2 are created, each one running at a different speeds taken from the list `[0.4, 0.5, 0.7, 1, 2, 5, 10]`. Their outputs are constrained to the range 1.0 to 1.02.
- The source sound `LFSaw.ar` takes the variable `chorus` as its frequency. In a concrete example: for a `freq` value of 60 Hz, the variable `chorus` would result in an expression like

$$60 * [1.01, 1.009, 1.0, 1.02, 1.015, 1.004, 1.019]$$

in which the numbers inside the list would be constantly changing up and down according to the speeds of each LFNoise2. The final result is a list of seven frequencies always sliding between 60 and 61.2 (60 * 1.02). This is called *chorus effect*, thus the variable name.

- When the variable `chorus` is used as freq of `LFSaw.ar`, multichannel expansion happens: we have now seven sawtooth waves with slightly different frequencies.
- The variable `filtermod` is just a sine oscillator moving very slowly (1 cycle over 16 seconds), with its output range scaled to 1-10. This will be used to modulate the cutoff frequency of the low pass filter.
- The variable `snd` holds the low pass filter (LPF), which takes `source` as input, and filters out all frequencies above its cutoff frequency. This cutoff is not a fixed valued: it is the expression `freq * filtermod`. So in the example assuming freq = 60, this becomes a number between 60 and 600. Remember that filtermod is a number oscillating between 1 and 10, so the multiplication would be 60 * (1 to 10).
- `LPF` also multichannel expands to seven copies. The amplitude envelope `env` is also applied right there.
- Finally, `Splay` takes this array of seven channels and mixes it down to stereo.

39.3 Under the hood

This two-step process of first creating the SynthDef (with a unique name) and then calling a Synth is what SC does all the time when you write simple statements like `{SinOsc.ar}.play`. SuperCollider unpacks that into (a) the creation of a temporary SynthDef, and (b) the immediate playback of it (thus the names temp_01, temp_02 that you see in the Post window). All of it behind the scenes, for your convenience.

```
// When you do this:
{SinOsc.ar(440)}.play;
// What SC is doing is this:
{Out.ar(0, SinOsc.ar(440))}.play;
// Which in turn is really this:
SynthDef("tempName", {Out.ar(0, SinOsc.ar(440))}).play;

// And all of them are shortcuts to this two-step operation:
SynthDef("tempName", {Out.ar(0, SinOsc.ar(440))}).add; // create a synth definition
Synth("tempName"); // play it
```

40 Pbind can play your SynthDef

One of the beauties of creating your synths as `SynthDef`s is that you can use `Pbind` to play them.

Assuming the `"wow"` SynthDef is still loaded in memory (it should, unless you quit and reopened SC after the last example), try the `Pbind`s below:

```
(
Pbind(
        \instrument, "wow",
        \degree, Pwhite(-7, 7),
        \dur, Prand([0.125, 0.25], inf),
        \amp, Pwhite(0.5, 1),
        \wowrelease, 1
).play;
)

(
Pbind(
        \instrument, "wow",
        \scale, Pstutter(8, Pseq([
                Scale.lydian,
                Scale.major,
                Scale.mixolydian,
                Scale.minor,
                Scale.phrygian], inf)),
        \degree, Pseq([0, 1, 2, 3, 4, 5, 6, 7], inf),
        \dur, 0.2,
        \amp, Pwhite(0.5, 1),
        \wowrelease, 4,
        \legato, 0.1
).play;
)
```

When using `Pbind` to play one of your custom `SynthDef`s, just keep in mind the following points:

- Use the `Pbind` key `\instrument` to declare the name of your `SynthDef`.
- All the arguments of your SynthDef are accessible and controllable from inside `Pbind`: simply use them as `Pbind` keys. For example, notice the argument called `\wowrelease` used above. It is not one of the default keys understood by `Pbind`—rather, it is unique to the synth definition `wow` (the silly name was chosen on purpose).
- In order to use all of the pitch conversion facilities of `Pbind` (the keys `\degree`, `\note`, and `\midinote`), make sure your `SynthDef` has an argument input for `freq` (it has to be spelled exactly like that). Pbind will do the math for you.
- If using a sustained envelope such as `Env.adsr`, make sure your synth has a default argument `gate = 1` (`gate` has to be spelled exactly like that, because `Pbind` uses it behind the scenes to stop notes at the right times).

- If not using a sustained envelope, make sure your SynthDef includes a doneAction: 2 in an appropriate UGen, in order to automatically free synth nodes in the server.

Exercise: write one or more `Pbinds` to play the `"pluck"` SynthDef provided below. For the `mutedString` argument, try values between 0.1 and 0.9. Have one of your `Pbinds` play a slow sequence of chords. Try arpeggiating the chords with `\strum`.

```
(
SynthDef("pluck", {arg amp = 0.1, freq = 440, decay = 5, mutedString = 0.1;
var env, snd;
env = Env.linen(0, decay, 0).kr(doneAction: 2);
snd = Pluck.ar(
        in: WhiteNoise.ar(amp),
        trig: Impulse.kr(0),
        maxdelaytime: 0.1,
        delaytime: freq.reciprocal,
        decaytime: decay,
        coef: mutedString);
    Out.ar(0, [snd, snd]);
}).add;
)
```

41 Control Buses

Earlier in this tutorial we talked about Audio Buses (section 30) and the Bus Object (section 33). We chose to leave aside the topic of Control Buses at that time in order to focus on the concept of audio routing.

Control Buses in SuperCollider are for routing control signals, not audio. Except for this difference, there is no other practical or conceptual distinction between audio and control buses.

You create and manage a control bus the same way you do with audio buses, simply using `.kr` instead of `.ar`. SuperCollider has 4096 control buses by default.

The first part of the example below uses an arbitrary bus number just for the sake of demonstration. The second part uses the Bus object, which is the recommended way of creating buses.

```
// Write a control signal into control bus 55
{Out.kr(55, LFNoise0.kr(1))}.play;
// Read a control signal from bus 55
{In.kr(55).poll}.play;

// Using the Bus object
~myControlBus = Bus.control(s, 1);
{Out.kr(~myControlBus, LFNoise0.kr(5).range(440, 880))}.play;
{SinOsc.ar(freq: In.kr(~myControlBus))}.play;
```

The next example shows a single control signal being used to modulate two different synths at the same time. In the `Blip` synth, the control signal is rescaled to control number of harmonics between 1 and 10. In the second synth, the same control signal is rescaled to modulate the frequency of the `Pulse` oscillator.

```
// Create the control bus
~myControl = Bus.control(s, 1);

// Feed the control signal into the bus
c = {Out.kr(~myControl, Pulse.kr(freq: MouseX.kr(1, 10), mul: MouseY.kr(0, 1)))}.
    play;

// Play the sounds being controlled
// (move the mouse to hear changes)
(
{
        Blip.ar(
                        freq: LFNoise0.kr([1/2, 1/3]).range(50, 60),
                        numharm: In.kr(~myControl).range(1, 10),
                        mul: LFTri.kr([1/4, 1/6]).range(0, 0.1))
}.play;

{
            Splay.ar(
                        Pulse.ar(
                                    freq: LFNoise0.kr([1.4, 1, 1/2, 1/3]).range(100, 1000)
                                    * In.kr(~myControl).range(0.9, 1.1),
                                    mul: SinOsc.ar([1/3, 1/2, 1/4, 1/8]).range(0, 0.03))
            )
}.play;
)

// Turn off control signal to compare
c.free;
```

41.1 asMap

In the next example, the method `asMap` is used to directly map a control bus to a running synth node. This way you don't even need `In.kr` in the synth definition.

```
// Create a SynthDef
SynthDef("simple", {arg freq = 440; Out.ar(0, SinOsc.ar(freq, mul: 0.2))}).add;
// Creat control buses
~oneBus = Bus.control(s, 1);
~anotherBus = Bus.control(s, 1);
// Start controls
{Out.kr(~oneBus, LFSaw.kr(1).range(100, 1000))}.play;
{Out.kr(~anotherBus, LFSaw.kr(2, mul: -1).range(500, 2000))}.play;
```

```
// Start a note
x = Synth("simple", [\freq, 800]);
x.set(\freq, ~oneBus.asMap);
x.set(\freq, ~anotherBus.asMap);
x.free;
```

42 Order of Execution

When discussing Audio Buses in section 30 we hinted at the importance of order of execution. The code below is an expanded version of the filtered noise example from that section. The discussion that follows will explain the basic concept of order of execution, demonstrating why it is important.

```
// Create an audio bus
~fxBus = Bus.audio(s, 1);
~masterBus = Bus.audio(s, 1);
// Create SynthDefs
(
SynthDef("noise", {Out.ar(~fxBus, WhiteNoise.ar(0.5))}).add;
SynthDef("filter", {Out.ar(~masterBus, BPF.ar(in: In.ar(~fxBus), freq: MouseY.kr
    (1000, 5000), rq: 0.1))}).add;
SynthDef("masterOut", {arg amp = 1; Out.ar(0, In.ar(~masterBus) * Lag.kr(amp, 1))}).
    add;
)
// Open Node Tree window:
s.plotTree;
// Play synths (watch Node Tree)
m = Synth("masterOut");
f = Synth("filter");
n = Synth("noise");
```

```
// Master volume
m.set(\amp, 0.1);
```

First, two audio buses assigned to the variables ~`fxbus` and ~`masterBus`.
Second, three `SynthDefs` are created:

- `"noise"` is a noise source that sends white noise to an effects bus;
- `"filter"` is a band pass filter which takes its input from the effects bus, and sends the processed sound out to the master bus;
- `"masterOut"` takes in the signal from the master bus and applies a simple volume control to it, sending the final sound with adjusted volume to the loudspeakers.

Watch the Node Tree as you run the synths in order.

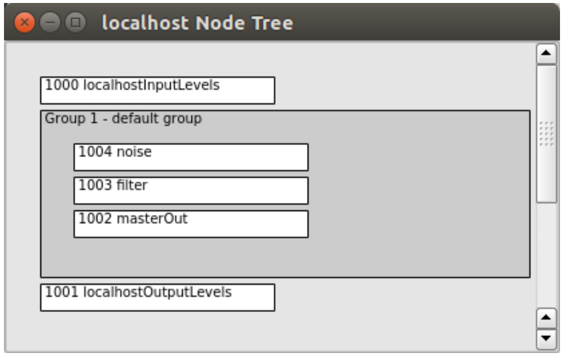


Figure 9: Synth nodes in the Node Tree window

Synth nodes in the Node Tree window run from *top to bottom*. The most recent synths get added to the top by default. In figure 9, you can see that `"noise"` is on top, `"filter"` comes second, and `"masterOut"` comes last. This is the right order we want: reading from top to bottom, the noise source flows into the filter, and result of the filter flows into the master bus. If you now try running the example again, but evaluating the lines `m`, `f`, and `n` in reverse order, you will hear nothing, because the signals are being calculated in the wrong order.

Evaluating the right lines in the right order is fine, but it might get tricky as your code becomes more complex. In order to make this job easier, SuperCollider allows you to explicitly define where to place synths in the Node Tree. For this we use the `target` and `addAction` arguments.

```
n = Synth("noise", addAction: 'addToHead');
m = Synth("masterOut", addAction: 'addToTail');
f = Synth("filter", target: n, addAction: 'addAfter');
```

Now, no matter in what order you execute the lines above, you can be sure that nodes will fall in the right places. The `"noise"` synth is explicitly told to be added to the head of the Node Tree; `"masterOut"` is added to the tail; and `filter` is explicitly added right after target `n` (the noise synth).

42.1 Groups

When you start to have lots of synths—some of them for source sounds, other for effects, or whatever you need—it may be a good idea to organize them into groups. Here's a basic example:

```
// Keep watching everything in the NodeTree
s.plotTree;
// Create some buses
~reverbBus = Bus.audio(s, 2);
```

```
~masterBus = Bus.audio(s, 2);
// Define groups
(
~sources = Group.new;
~effects = Group.new(~sources, \addAfter);
~master = Group.new(~effects, \addAfter);
)
// Run all synths at once
(
// One source sound
{
        Out.ar(~reverbBus, SinOsc.ar([800, 890])*LFPulse.ar(2)*0.1)
}.play(target: ~sources);
// Another source sound
{
        Out.ar(~reverbBus, WhiteNoise.ar(LFPulse.ar(2, 1/2, width: 0.05)*0.1))
}.play(target: ~sources);
// This is some reverb
{
        Out.ar(~masterBus, FreeVerb.ar(In.ar(~reverbBus, 2), mix: 0.5, room: 0.9))
}.play(target: ~effects);
// Some silly master volume control with mouse
{Out.ar(0, In.ar(~masterBus, 2) * MouseY.kr(0, 1))}.play(target: ~master);
)
```

For more information about order of execution, look up the Help files "Synth," "Order of Execution," and "Group."

Part V

FURTHER STUDY TOPICS

43 MIDI

An extended presentation of MIDI concepts and tricks is beyond the scope of this tutorial. The examples below assume some familiarity with MIDI devices, and are provided just to get you started.

```
// Quick way to connect all available devices to SC
MIDIIn.connectAll;

// Quick way to see all incoming MIDI messages
MIDIFunc.trace(true);
MIDIFunc.trace(false); // stop it

// Quick way to inspect all CC inputs
MIDIdef.cc(\someCC, {arg a, b; [a, b].postln});

// Get input only from cc 7, channel 0
MIDIdef.cc(\someSpecificControl, {arg a, b; [a, b].postln}, ccNum: 7, chan: 0);

// A SynthDef for quick tests
SynthDef("quick", {arg freq, amp; Out.ar(0, SinOsc.ar(freq) * Env.perc(level: amp).
    kr(2))}).add;

// Play from a keyboard or drum pad
(
MIDIdef.noteOn(\someKeyboard, { arg vel, note;
        Synth("quick", [\freq, note.midicps, \amp, vel.linlin(0, 127, 0, 1)]);
});
)

// Create a pattern and play that from the keyboard
(
a = Pbind(
        \instrument, "quick",
        \degree, Pwhite(0, 10, 5),
        \amp, Pwhite(0.05, 0.2),
        \dur, 0.1
);
)

// test
a.play;

// Trigger pattern from pad or keyboard
MIDIdef.noteOn(\quneo, {arg vel, note; a.play});
```

A frequently asked question is how to manage note on and note off messages for sustained notes. In other words, when using an ADSR envelope, you want each note to be sustained for as long as a key is pressed. The release stage comes only when the finger comes off the corresponding key (review section on ADSR envelopes if needed).

In order to do that, SuperCollider simply needs to keep track of which synth node corresponds to each key. We can use an Array for that purpose, as shown in the example below.

```
 // A SynthDef with ADSR envelope
SynthDef("quick2", {arg freq = 440, amp = 0.1, gate = 1;
        var snd, env;
        env = Env.adsr(0.01, 0.1, 0.3, 2, amp).kr(2, gate);
        snd = Saw.ar([freq, freq*1.5], env);
        Out.ar(0, snd)
}).add;

// Play it with a MIDI keyboard
(
var noteArray = Array.newClear(128); // array has one slot per possible MIDI note

MIDIdef.noteOn(\myKeyDown, {arg vel, note;
        noteArray[note] = Synth("quick2", [\freq, note.midicps, \amp, vel.linlin(0,
            127, 0, 1)]);
        ["NOTE ON", note].postln;
});

MIDIdef.noteOff(\myKeyUp, {arg vel, note;
        noteArray[note].set(\gate, 0);
        ["NOTE OFF", note].postln;
});
)
// PS. Make sure SC MIDI connections are made (MIDIIn.connectAll)
```

To help you understand the code above:

- The SynthDef `"quick2"` uses an ADSR envelope. The `gate` argument is the one responsible for turning notes on and off.
- An Array called `"noteArray"` is created to keep track of notes being played. The indices of the array are meant to correspond to MIDI note numbers being played.
- Every time a key is pressed on the keyboard, a Synth starts playing (a synth node is created in the server), and *the reference to that synth node is stored in a unique slot in the array*; the array index is simply the MIDI note number itself.

- Whenever a key is released, the message `.set(\gate, 0)` is sent to the appropriate synth node, retrieved from the array by note number.

In this short MIDI demo we only discussed getting MIDI *into* SuperCollider. To get MIDI messages *out* of SuperCollider, take a look at the `MIDIOut` Help file.

44 OSC

OSC (Open Sound Control) is a great way to communicate any kind of message between different applications or different computers over a network. In many cases, it is a much more flexible alternative to MIDI messages. We don't have space to explain it in more detail here, but the example below should serve as a good starting point.

The goal of the demo is to send OSC messages from a smartphone to your computer, or computer to computer.

On the receiver computer, evaluate this simple snippet of code:

```
(
OSCdef(
        key: \whatever,
        func: {arg ...args; args.postln},
        path: '/stuff')
)
```

Note: hitting [ctrl+.] will interrupt the `OSCdef` and you won't receive any more messages.

44.1 Sending OSC from another computer

This assumes that both computers are running SuperCollider and connected to a network. Find the IP address of the receiver computer, and evaluate the following lines in the sender computer:

```
// Use this on the machine sending messages
~destination = NetAddr("127.0.0.1", 57120); // use correct IP address of destination
    computer

~destination.sendMsg("/stuff", "heelloooo");
```

44.2 Sending OSC from a smartphone

- Install any free OSC app on the phone (for example, gyrosc);
- Enter the IP address of the receiver computer into the OSC app (as 'target');
- Enter SuperCollider's receiving port into the OSC app (usually 57120);
- Check the exact message path that the app uses to send OSC to, and change your OSCdef accordingly;
- Make sure your phone is connected to the network

As long as your phone is sending messages to the proper path, you should see them arriving on the computer.

45 Quarks and plug-ins

You can extended the functionality of SuperCollider by adding classes and UGens created by other users. *Quarks* are packages of SuperCollider classes, extending what you can do in the SuperCollider language. *UGen plugins* are extensions for the SuperCollider audio synthesis server.

Please visit `http://supercollider.sourceforge.net/` to get up-to-date information on how to add plug-ins and quarks to your SuperCollider installation. The "Using Quarks" Help file is also a good starting point: `http://doc.sccode.org/Guides/UsingQuarks.html`. From any SuperCollider document, you can evaluate `Quarks.gui` to see a list of all available quarks (it opens in a new window).

46 Extra Resources

This is the end of this introduction to SuperCollider. A few extra learning resources are listed below. Enjoy!

- An excellent series of YouTube tutorials by Eli Fieldsteel: `http://www.youtube.com/playlist?list=PLPYzvS8A_rTaNDweXe6PX4CXSGq4iEWYC`.
- The standard SC get-started tutorial by Scott Wilson and James Harkins, available online and in the built-in Help files: `http://doc.sccode.org/Tutorials/Getting-Started/00-Getting-Started-With-SC.html`
- The official SuperCollider mailing list is the best way to get friendly help from a large pool of users. Beginners are very welcome to ask questions in this list. You can sign

up here: `http://www.birmingham.ac.uk/facilities/BEAST/research/supercollider/mailinglist.aspx`

- Find a SuperCollider meet-up group in your city. The official sc-users mailing list is the best way to find out if there is one where you live. If there is no meet-up group in your area, start one!

- Lots of interesting snippets of code can be found here: `http://sccode.org/`. Sign up for an account and share your code too.

- Have you heard of SuperCollider tweets? `http://supercollider.sourceforge.net/sc140/`

Notes

[1]First question: when you use the number 1 instead of `inf` as the `repeats` argument of the second `Pseq`, the Pbind will stop after 6 notes have been played (that is, after one full sequence of duration values has been performed). Second question: to make a Pbind play forever, simply use `inf` as the `repeats` value of all inner patterns.

[2]

a) `Pwhite(0, 10)` will generate any number between 0 and 10. `Prand([0, 4, 1, 5, 9, 10, 2, 3], inf)` will only pick from the list, which has *some* numbers between 0 and 10, but not all (6, 7, 8 are not there, so they will never occur in this `Prand`).
b) Technically you could use a `Prand` if you provide a list with all numbers between 0 and 100, but it makes more sense to use a `Pwhite` for this task: `Pwhite(0, 100)`.
c) `Prand([0, 1, 2, 3], inf)` picks items from the list at random. `Pwhite(0, 3)` arrives at the same kind of output through different means: it will generate random integer numbers between 0 and 3, which ends up being the same pool of options than the `Prand` above. However, if you write `Pwhite(0, 3.0)`, the output is now different: because one of the input arguments of `Pwhite` is written as a float (3.0), it will now output any floating point number between 1 and 3, like 0.154, 1.0, 1.45, 2.999.
d) The first `Pbind` plays 32 notes (4 times the sequence of 8 notes). The second `Pbind` plays only 4 notes: four random choices picked from the list (remember that `Prand`, unlike `Pseq`, has no obligation to play all the notes from the list: it will simply pick as many random notes as you tell it to). The third and last `Pbind` plays 32 notes, like the first.

[3]First line: the Array `[1, 2, 3, "wow"]` is the receiving object; `reverse` is the message. Second line: the String "hello" is the receiving object; `dup` is the message; `4` is the argument to `dup`. Third line: 3.1415 is the receiving object; `round` is the message; `0.1` is the argument to `round`. Fourth line: `100` is the receiver object, `rand` is the message. Last line: `100.0` is the receiver of the message `rand`, the result of which is a random number between 0 and 100. That number becomes the receiver of the message `round` with argument `0.01`, so that the random number is rounded to two decimal cases. Then this result becomes the receiving object of the message `dup` with argument `4`, which creates a list with four duplicates of that number.

[4]Rewriting using functional notation only: `dup(round(rand(100.0), 0.01), 4);`

[5] Answers:

a) 24

b) $[5, 5.123]$ [both numbers and brackets)

c) Entire `LFSaw` line

d) Only one

e) 0.4

f) 1 and 0.3

[6]`SinOsc` is bipolar because it outputs numbers between -1 and +1. `LFPulse` is unipolar because its output range is 0-1 (in fact, `LFPulse` in particular only outputs zeros or ones, nothing in between)

[7]Solution: `a = {Out.ar(0, SinOsc.ar(freq:  [800, 880], mul:  LFPulse.ar([2, 3])))}.play;`

[8](a) The variable `lfn` simply holds a `LFNoise2`. The role of `LFNoise2` in life is to generate a new random number every second (between -1 and +1), and slide to it from the previous random number (differently from `LFNoise0`, that jumps to the new number immediately). The first use of this variable `lfn` is in the `freq` argument of the BPF: `lfn.range(500, 2500)`. This takes the numbers between -1 and +1 and scales them to the range 500-2500. These numbers are then used as the center frequency of the filter. These frequencies are the pitches that we hear sliding up and down. Finally, `lfn` is used again to control the position of the panner `Pan2`. It is used directly (without a `.range` message) because the numbers are already in the range we want (-1 to +1). The nice result of this is that we couple the change of frequency with the change of position. How? Every second, `LFNoise2` starts to slide toward a new random number, and this becomes a synchronized change in frequency of the filter and panning position. If we had two different `LFNoise2` in each place, the changes would be uncorrelated (which might be fine too, but it's a different aural result).
(b) a `mul:` of 1 would just be too soft. Because the filter is so sharp, it takes so much out of the original signal that the amplitude drops too much. We need to boost the signal back to a reasonably audible range, so that's why we have `mul:  20` at the end of the `BPF` line.
(c) The rhythm is driven by the `LFPulse` that is the `mul:` argument of the `Saw`. `LFPulse` frequency (how many pulses per second) is controlled by an `LFNoise1` that produces numbers between 1 and 10 (interpolating between them). Those numbers are the "how many notes per second" of this patch.