A Very Step-By-Step Guide to

SuperCollider

(c) Iannis Zannos
iani@otenet.gr

Draft! April 2005

Preface: Uses of SuperCollider

SuperCollider is an open, extensible, programmable, flexible, all-round tool for creating applications and works in the domain of Sound Arts. It can be applied to almost all related types of data, techniques, and goals in this domain, as shown in the following list:

Data Processed ("Input", "What")
- Sound (Audio Samples)
- MIDI
- Sound Synthesis & Processing Algorithms
- Sound Events (parameter tuples)
- Interactive I/O Data (other than MIDI, such as OSC, HID, Mouse or Keyboard input)

Techniques ("Processing", "How")
-~~Audio/Event "Time-Line" Editing~~
-Sound Processing
-Sound Synthesis Algorithm Design and Application
-Sound-Event Generation Algorithm Design and Application
-Real-Time Interactive Tools Design and Application

End Products ("Applications", "Output", "What For")
- Fixed audio and / or MIDI tracks
- Interactive electroacoustic music performance
- Interactive sound installation

The only technique in the above list that SuperCollider is not suited for, is *Audio "Time-Line" Editing* (shown in the table crossed out). This is the editing by "cut-and-paste" of sound samples of different lengths into specific time points in a number of parallel sound tracks, which result in a sound "piece" of fixed duration and number of sound channels. This important technique is covered by a large number of commercial or free tools such as ProTools, Cubase, Logic Audio, Peak, Sound Edit, Sound Maker etc.

The flexibility and power offered by the open programming environment of SuperCollider comes at a cost, namely the effort and time that it takes to program the algorithms and tools that correspond to the specific artistic goal at hand. In contrast to other programs, SuperCollider does not come with a ready-to-use palette of tools and a predefined way for using them. Instead, it comes with components for developing tools and examples as well as documentation to help one develop ones own. SuperCollider takes more time to master than a specialized application, but gives more freedom in the long term. This is an important asset for artists whose approach to mastering their craft involves not just using already known techniques, but developing new techniques or extending old ones, and who regard expression and invention as inherently linked to each other.

SuperCollider is being used as tool in a significant and growing number of ambitious artistic projects. Still, the non-programmer may wonder whether the effort needed to learn and develop at

the same time justifies the end results. There is however one more factor that speaks in favor of trying it out. That is the "Open Source factor": The SuperCollider programmer can participate creatively in a growing "culture" that improves and extends the tradition of SuperCollider continuously with new tools. These tools are gradually heightening the potential of SuperCollider. This culture is gaining momentum steadily. Thus, powerful and high-level tools will continue to be added at a fairly fast pace to SuperCollider, and these will gradually make it an easier and more rewarding environment for end users. This tutorial is intended as a contribution to this culture. Its aim is to support learning and developing in SuperCollider through documentation and examples for non-programmers as well as for versed users.

Getting started

Installation

Compatibility, Requirements

SuperCollider runs best on an Apple Macintosh with MacOs X. It also runs on Linux, but this tutorial will not deal with the Linux port. At the time of the writing of the present tutorial, a preliminary Windows port has been released.

Any Apple Macintosh running MacOS X is sufficient for working with SuperCollider. Of course, processor and bus speed are essential for real time performance, the faster these are, the more computationally demanding examples the computer will be able to run.

SuperCollider uses Core Audio and will therefore run with any audio cards supporting Core Audio such as MOTU, M-Audio, Edirol etc. Use of ProTools hardware is also possible with drivers that can be downloaded from http:www.digidesign.com.

Downloading and Installation

The latest version of SuperCollider can be downloaded from the SourceForge project website at: sourceforge.net—supercollider (SourceForge.net: Project Info - SuperCollider). Scroll to "Latest File Releases" at the lower part of the page. Click on the item in the left column "Package" which you wish to download (SuperCollider3 MacOSX or SuperCollider3 Win32). On the next page, select the link "Download SuperCollider3_bin" followed by the date of the release. On the next page click on the Download icon from your preferred server location: 

This will start the download of a compressed archive such as: SuperCollider3_bin_20050304.tar.gz. This file is a self-expanding archive and will usually expand automatically to create a folder named *SuperCollider_f*. If it does not, try double-clicking on it or some other method to open it.

Place the *SuperCollider_f* folder in the Applications folder under the standard main directory of the Macintosh - or anywhere else where you may wish. However, if your computer has more than one user accounts be aware of restrictions applying to file access in multi-user envronments, as explained in the next section.

 Using SuperCollider on systems with more than one user account

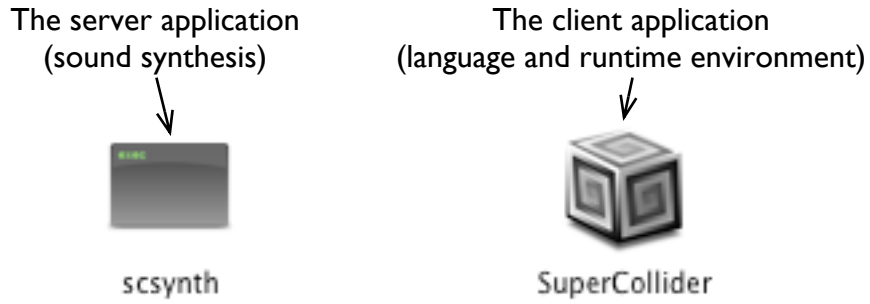
To work with SuperCollider, it is necessary to have write-access to files located inside the *SuperCollider_f* folder. This will not be the case if SuperCollider is used on an account that is

different than the account that SuperCollider was originally installed with. This is because most multi-user systems will by default not permit to write in a folder that was created by another user. Therefore, it is recommended that each user with a separate account should also download and install a separate, private copy of SuperCollider. Otherwise, it will be necessary to set the read-write permissions of the *SuperCollider_f* folder so that it can be written to by all related user accounts. This can be done through the information panel of the folder: Select the folder and type "Command-I". Then set the Ownership & Permissions of the folder under the Details tab to *Others: Read & Write* and press the *Apply to enclosed items* ... button. It is recommended to avoid using the same copy of SuperCollider from many accounts, because this will incur administrative overhead.

☞ **Always use the SuperCollider language application while logged on the user account from which that copy of SuperCollider was originally installed**

The client-server structure of SuperCollider

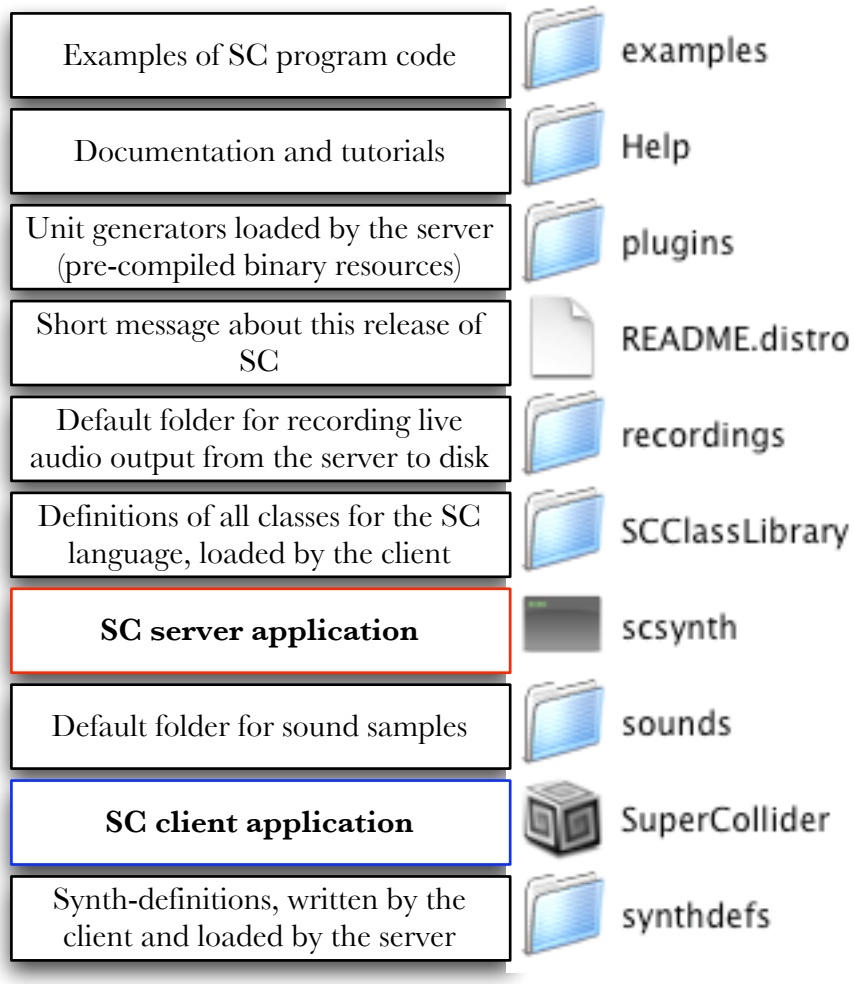
SuperCollider consists of 2 applications, whose files are named *SuperCollider* and *scsynth*.



These two applications perform complementary tasks: The server (*scsynth*) is used to perform sound synthesis, while the client (*SuperCollider)* is used to write, compile and execute programs that define and control sound synthesis, and other types of time-based interaction. The client implements a unique programming language that is called the SuperCollider language. The *SuperCollider* application cannot perform sound synthesis by itself, but it sends commands to the *scsynth* application, requesting the performance of sounds synthesis tasks as a "service" (hence the term "server"). The two applications together form a "client-server" system: The language application is a client that requests sounds to be synthesized and the scsynth application is a server that responds to requests from clients by producing sounds. In this tutorial, *client* refers to the *SuperCollider* application that implements the language and its run-time environment, and *server* refers to the *scsynth* application that implements continuous, real-time sound synthesis.

Contents of the SuperCollider_f Folder

Following figure lists the function of each folder and file at the top-level of the SuperCollider_f folder hierarchy (MacOs X version):



Files and folders at the top level of the SuperCollider_f folder

The scsynth and SuperCollider applications are introduced in the next section, "The Client-Server Structure of SuperCollider". Synth-definitions are introduced in section "Making Sounds" and explained in depth in section ... Unit Generators are discussed in section ...

SuperCollider (client): Development and Runtime Environment

The application named *SuperCollider*, implements the SuperCollider language. This is what the programmer uses to write and execute SuperCollider programs, in other words it is the development as well as the run-time environment. There are no stand-alone SuperCollider programs. Instead, one uses the SuperCollider application both to develop and to run programs.

☛ **SuperCollider client is an environment both for developing and for running programs**

☛ **User interaction and program execution with the SuperCollider client is only possible while the client is running.**

☛ **All SuperCollider programs run "within" the SuperCollider client: SuperCollider client does not create independent, stand-alone SuperCollider client programs that run without the client.**

scsynth (server): Sound Synthesis Engine

The server application (file named *scsynth*) is the sound synthesis engine that loads, inputs, processes, synthesizes and outputs sounds. The programmer does not need to interact directly with the server application, but only via commands written and executed in the Language application. One can start and stop either the client or the server independently of each other. Usually one starts and stops a Server by commands evaluated in the client, as demonstrated in the next section (*Working with SuperCollider*).

☛ **The scsynth application (server) is the sound synthesis engine.**

☛ **Sound synthesis and processing is only possible while a server is running.**

Networking Capabilities of SuperCollider

The language and server communicate with each other entirely via the Open Sound Control protocol (OSC). This an extensible protocol for sending commands and data from one application to another and runs over UDP (a kind of IP protocol) or TCP-IP. Thus, Language and Server can communicate with each other as well as with other applications over any medium that supports IP, such as ethernet or wireless internet.

☛ **The server can be on a different machine than the client application, as long as both of them can communicate via OSC.**

☛ **It is possible to connect several servers and client applications in one sound-generating network, all running on different machines.**

☛ **Both the client and the server can communicate with applications other than SuperCollider over OSC.**

Working with SuperCollider

One one can develop programs in SuperCollider at three different levels. These are:

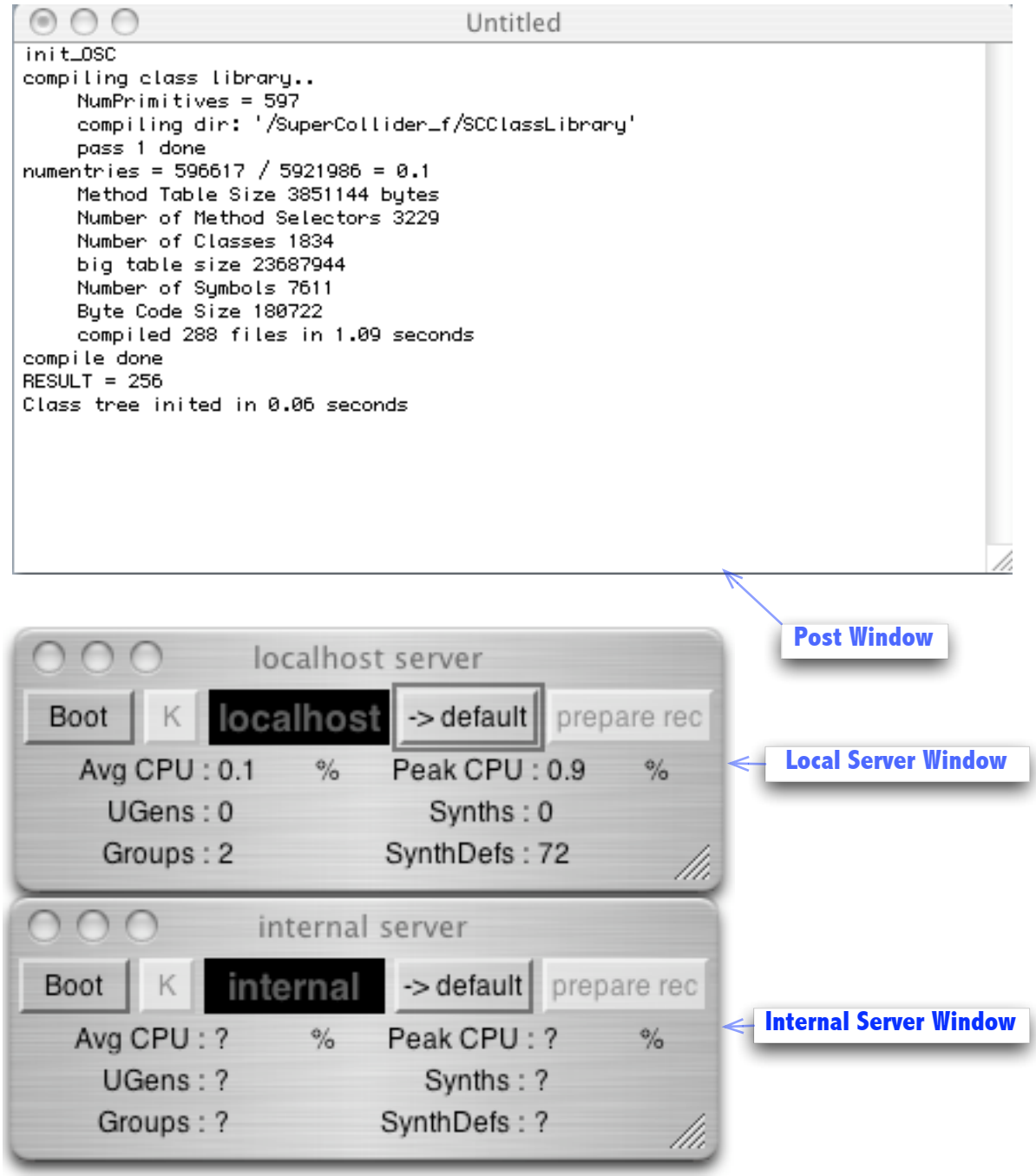
1. Evaluating (executing, running) code that is written in workspace windows of the SuperCollider client. This code is written in SuperCollider's own language.
2. Writing Class code for the SuperCollider Library that is compiled by the SuperCollider client.
3. Modifying and extending the C++ source code of the SuperCollider client, the scsynth server, or the Plug-Ins that are loaded by the server. This requires recompiling the entire source code to produce a new fixed stand-alone version of these applications.

Levels 1 and 2 are sufficient for programming most applications and pieces while technique 3 is for specific advanced needs. This section introduces the basics of level 1, evaluating code in workspace windows. Level 2, defining ones own classes, is described in chapters ... and Level 3, the C++ source code level, is introduced in chapters ...

Startup of the client

Double-click on the SuperCollider Icon to start the SuperCollider client application.

As soon as the SuperCollider client has started up, it will open the windows that are specified in the default setup. The screen should look like this:



If SuperCollider fails to start in the manner described above, consult the section on Troubleshooting **at the end of the present chapter.**

Customizing startup

To customize the actions that are performed at startup, edit the `Main-startup` method. More details are provided in the section *Customizing the Main Class*.

The SuperCollider client application menus

The "Post Window"

The window named "Untitled" is called the *Post Window*, because it is where the SuperCollider application posts its feedback to the user in the form of text. The post window cannot be closed by the user, because feedback is essential for users to interact with the language. The Post Window gives the following kinds of feedback:

1 Information about the system at startup

This concerns mainly checks of the systems basic functionality such as OSC communication. See Reference ...

2 Information about the language during and after compilation

This concerns the number of classes compiled, warnings about modifications of existing classes through code in other other files, and the duration of the compilation process. See section on compiling the class library.

3 Printout of the object returned from code evaluated by the user

Every time some code is evaluated by the user, it returns some object as result of the evaluation to the system. This object is "posted" in the Post Window as feedback for the user. See more under section *Evaluating Code*.

4 Printout of objects requested by the user through the "post" and "postln" messages

One can print any object to the Post Window by sending it the message `post` or `postln`, or by sending it as argument to a stream operator to the class `Post`. [*See more under...*]

5 Error messages

Error messages are posted by the system when it encounters any of several kinds errors during the compilation or the execution of a program. The information they provide is invaluable to the user in identifying and correcting errors.

6 Feedback from the server

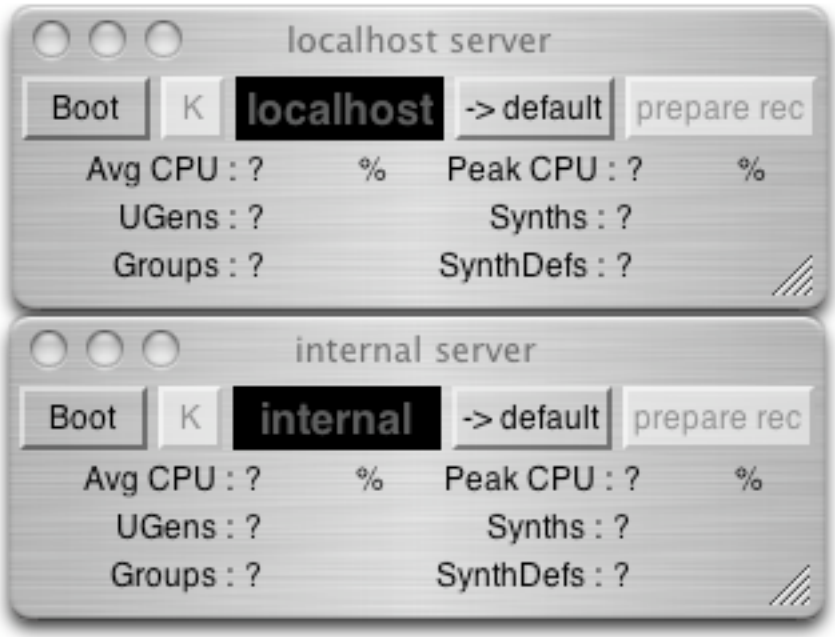
The server can send feedback to the client about changes in its status such as booting and quitting, as well as about possible server errors. These are all posted in the Post Window.

Post Window Commands

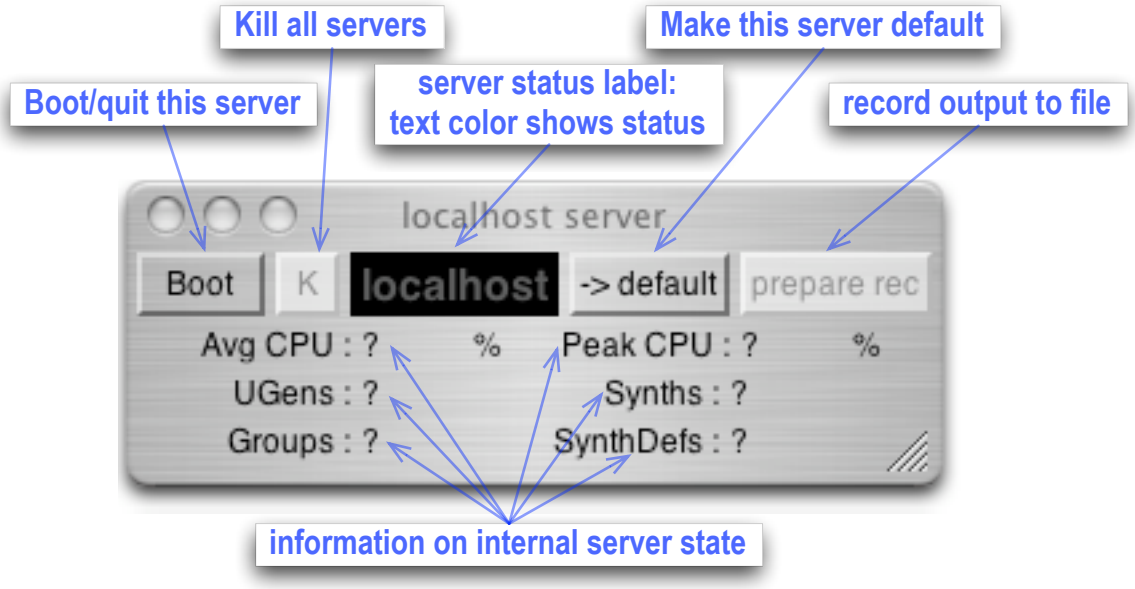
- To clear the contents of the Post Window, type Command-Shift-K.
- To bring the post window to the front, type Command-\
- To make any other text window become the Post Window, select Menu item "Become Post Window" from the Window menu.

The server windows

The two windows at the lower left corner of the screen show information and control items relating to two commonly used SuperCollider servers: The localhost server and the internal server.



The server GUI window shows basic information about the runtime status of the corresponding server and provides buttons for booting (starting), quitting (stopping) and for recording the servers output on a sound file. The label widget with black background shows the servers boot status. When the text of the label has gray color, it means that the server application represented by the GUI is not running and can be started (booted).



Booting and quitting a server through the server GUI

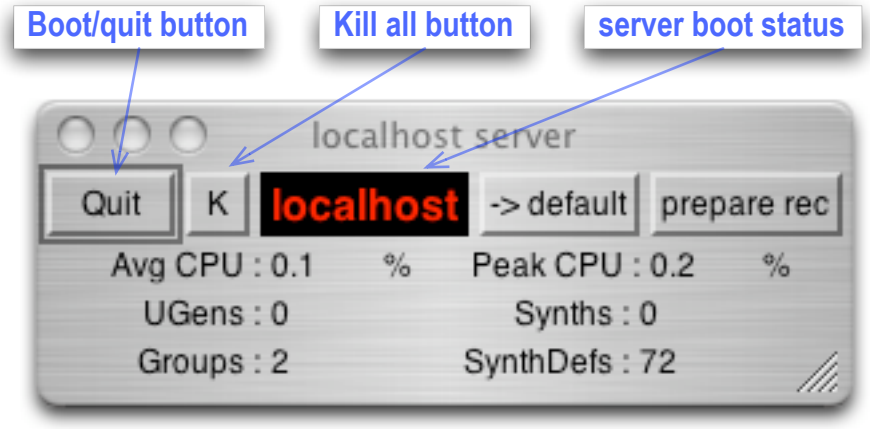
To boot a server through the server GUI window, press on the boot/quit button. The text color of the label widget displaying the servers name will change from gray to yellow to show that the server is in the process of booting, and then after a small time (usually less than one second) it will change to red, to show that the server is running. The label of the boot/quit button will change from "Boot" to "Quit". The Post Window should display the messages:

```
booting 57110
SC_AudioDriver: numSamples=512, sampleRate=44100.000000
start   UseSeparateIO?: 0
PublishPortToRendezvous 0 57110
SuperCollider 3 server ready..
notification is on
```

These show that the server booted successfully (see details in the Reference section).

If the server fails to start, consult the section on Troubleshooting **at the end of the present chapter.**

Caution: Do not click on the *Boot* button more than once in rapid succession. Like any other application, the server needs some time to boot, If you try to boot it twice while already waiting for the results of the first boot command, the state of the GUI representing the server will become out of sync with the actual state of the server application that is booting. Then you may have to use the "K" button explained here below to stop the server and start over again.



In some cases the server application may become "uncoupled" from the language, in which case one cannot stop it through the boot/quit button. For these emergency cases, one can use the "kill all button" (K), which kills all scsynth server processes found by the system by using a UNIX command. This is a foolproof version for stopping runaway server processes.

When a server quits, the system posts the following messages:

```
/quit sent

SC_AudioDriver::Stop
RESULT = 0
```

If the RESULT is not 0, it means that the server may have encountered problems in quitting. This can usually be ignored.

- The reference section *Booting (starting) and quitting (stopping) the server* lists a number of techniques for starting and stopping a server.

Local server and internal server

While there is only one synthesis application for SuperCollider, which is the *scsynth* server, this application can be run in two different ways on the same machine: as an entirely separate application (known as the *local server* or the *localhost server*), or as an application embedded inside the memory space of the SuperCollider client (*internal server*). In other words, the localhost server and the internal server are both scsynth processes that can be started on the same computer as the SuperCollider client is running. They are booted from the same application file, *scsynth*. In terms of their synthesis capabilities, the localhost server and the internal server are identical. They are however different as far as their communication with other programs is concerned. The following sections explain the practical impact of this difference in the way the two types of servers function.

The local server

The local server is the server that runs on the same computer as the client but as an independent application. This has two implications that can be useful under many circumstances: First, the local server will continue running even if the client application crashes, and vice versa. This can be vital in a situation such as a performance or an installation, because in the case of a crash of the client, sound synthesis will continue, thus making the possibility of recovery easier. Second, the server can be controlled by OSC commands coming from any application on any computer that is accessible in the network.

Use the localhost server:

- **if you do not need to have a signal scope**
- **if it is important that sound synthesis continues even when the SuperCollider client program crashes**
- **if there is need to control the server from several different applications or computers**

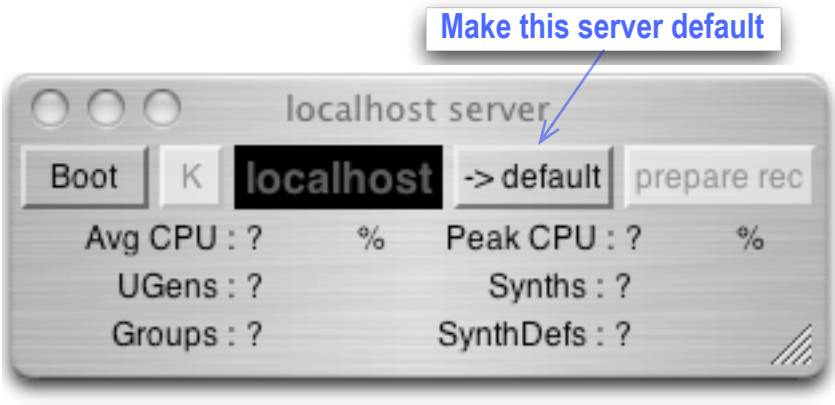
The internal server

Because the SuperCollider client shares its memory space with the internal server, the client has immediate, fast access to the audio and control signals synthesized by the server, and can thus show these in a real-time scope. Therefore, one should use the internal server if one needs to open scope windows that display audio or control signals as they are being processed by the server in real-time.

Use the internal server if you need to have a signal scope window

Setting the default server

Many SuperCollider programs, such as some of those included in the examples and help folders, do not specify which server will be used for synthesis. This has the advantage that the same program can be made to run on a different server without changing the code of the program. For example, one may want to run a program on the internal server instead of the local server, because of the need of visual feedback through use of a scope window. This feature is made possible by the definition of a default server. The default server is the server which is used by client programs when the code of the program does not explicitly specify a server. One can select which server to use as default by clicking on the button "-> default" on a server GUI window.



One can also set the default server by evaluating a line of code, for example:

```
Server.default = Server.local;
```

Reopening a server window

If the GUI window of a server is closed by mistake, one can make another one by sending it the message `makeWindow`. For example, to make a new window for the local server, evaluate the code:

```
Server.local.makeWindow;
```

The details of evaluating code are explained in section *Evaluating code*.

"Workspace" windows and files

The SuperCollider Language application contains a text editor for editing code in any number of windows. This is used to edit and execute SuperCollider programs interactively.

 To open a new window press Command-N. If you have just started SuperCollider and not created any other windows, then the new window will be called "Untitled 2".

You can type any text in the window, format the text with the commands found in the Format menu, paste graphics, and save the contents of the window onto a file.

 Note that writing into a workspace window or opening an existing workspace-text file does ***not*** automatically "load" any code to the system. To make the language application translate the text you have written into a program and to execute that program, you have to explicitly evaluate a text passage, as is explained below in section *Evaluating Code*.

Evaluating code, making sounds

Evaluating code in a workspace window is the most common way to start developing some new idea in SuperCollider: One writes and executes code interactively in the SuperCollider Language Application by editing the code in a workspace window, selecting it with the mouse or cursor keys, and pressing the *enter* key. The code can be as short as a single line or it can fill several pages. One can evaluate various code passages successively or repeatedly at any point, even while the application is running processes started by previous code. The Language Application Library contains a multitude of tools for modeling synthesis algorithms, generating compositional structures, creating Graphic User Interfaces (GUIs), programming interaction, controlling synthesis, and communicating with external devices (MIDI, OSC, HID). By using these tools, it is possible to realize applications for installations or performances without having to write ones own class definitions.

Each time that one evaluates some code, the system of SuperCollider that exists in the memory

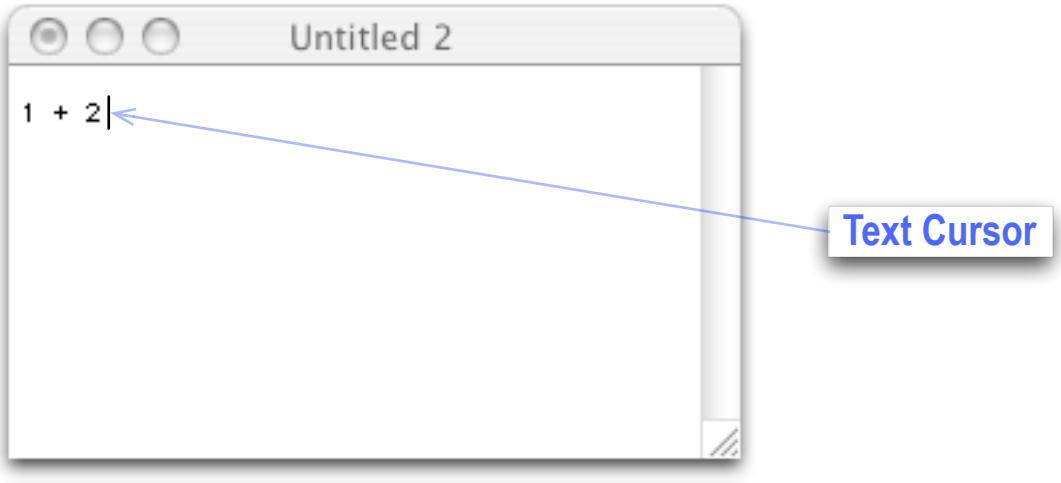
(RAM) of the client program changes: New objects are created, other objects may be marked as free for disposal (destruction) by the Garbage Collector. These changes are not permanent. Compiling the SuperCollider library (Command-K, see Section ...) will clear the memory of the application from all objects and rebuild the system from scratch.

Evaluating one line of code

Type the following text in a workspace window:

```
1 + 2
```

Make sure the text cursor is still on the line you have just typed, like this:



Press the "enter" key (which is different from the "return" key). Watch the output that is printed on the Post Window (the window named *Untitled*). This will be the number 3, which results from calculating the expression `1 + 2`. The last thing that is always output after evaluating some code, is the *return value*, that is, the object that is returned to the system after the end of the execution of the program.

The enter key

The *enter* key is not the same as the *return* key. The enter key is marked on most Apple keyboards with the word "enter". This is different from the return key which is marked with the word "return" and may additionally have above it with smaller font letters the word "enter". On some keyboards the enter key is marked as .

The keyboard-shortcut Control-C also functions as an *enter* command.

Testing some sounds

To play a sound on a server, the server must be booted. Boot the local server by evaluating this line:

```
Server.local.boot;
```

After the server has booted, evaluate this function to start a sine tone:

```
{ SinOsc.ar(440, 0, 0.1) }.play;
```

Stop the sound by typing Command-. This command stops all currently playing sounds. Next play two sounds at the same time, starting and stopping them independently:

```
// Start sound 1 and store its synth in variable a
a = { Ringz.ar(WhiteNoise.ar(Decay.kr(Impulse.kr(2.3,0,0.01))),1000,0.1)
}.play;

// After that, start sound 2 and store its synth in variable b
b ={RLPF.ar(LFSaw.ar([200, 300],0, 0.1), LFNoise0.kr([6.9,7], 600,
800),0.1)}.play;

// to stop sound 1, free the synth stored in variable a
a.free;

// to stop sound 2, free the synth stored in variable b
b.free;
```

To restart either of these sounds, evaluate the functions above again. If you start and store a synth in `a` or `b` twice without stopping it first, you will get stuck with one "runaway synth" because storing a synth in a variable *x* will make you "lose" the copy that was previously stored in that variable. That is, the new synth ("synth 2") replaces the one that was previously stored in *x* so this variable will no longer contain the synth that was first stored in it ("synth 1"). Note that each time that you send the a function the message `play`, a *new* synth is created, even if it is the same function that you are sending the message to. Then you need to type Command-. or to quit the server to stop the sounds.

Return value vs. effects of evaluating code

In many cases, the point of a program is not the result value that is returned by the program, but some intermediate actions that the program executes on the way. It is therefore important to distinguish between the intermediate actions or "*side effects*" of a program and the final return value. A simple example is the following line of code:

```
5.do { ["hello", "there"].choose.postln }
```

The effect of this code is to post five random choices between the strings "hello" and "there", on five separate lines. The value returned by the code is the number 5, which is posted after the code has been evaluated.

Documenting your code: Comments

Comments are useful for explaining to humans what the code does. The compiler (the program that translates any evaluated code to a program) ignores them. There are two ways to write comments in SuperCollider;

1. Everything to the right of "//" on one line counts as a comment:

```
var rpt; // rpt holds the number of repetitions of the pattern
```

2. Anything enclosed between /* and */ is a comment.

```
/*   Everything between /* and */
     will be ignored by the compiler!
     Nesting of /* is allowed as long as it is matched by
     the closing */ Following line of code is still commented out:
     v = v ** x;
     ALL THIS IS JUST A COMMENT!
     USE COMMENTS TO MAKE YOUR CODE READABLE!!!
*/
```

Comments can also be used to remove some code from the program while leaving it in the text.

Include comments in your programs as much as possible.

Comments are always invaluable when reading complicated code written by someone else, or by oneself some while ago.

Program statements. Evaluating blocks of code

Most programs consist of more than just one line of code. Like in many other languages, programs in SuperCollider are organized in statements. A program is thus written as a sequence of syntactically self-contained statements. Each statement is separated from the next one by a semicolon (;). To evaluate a program that extends over more than one line of text, select the entire code of the program with the mouse and hit the *enter* key. Try for example the following program:

```
(
Routine({          // make a routine with the following code:
     w = SCWindow.new(" ", Rect(400, 400, 200, 200)); // make a window
     w.front;      // make the window visible
     30 do: {      // repeat 30 times
                   // set the background of the window to a random color
          w.view.background = Color.rand;
          0.1.wait;// wait 0.1 seconds between repetitions
     };            // end of the repeated part
     1.wait;       // after the repetitions, wait one more second
     w.close;      // then close the window
}).play(AppClock)  // play the routine with the Application Clock
                   // AppClock is required if GUI elements are involved
)
```

The outer parentheses () that enclose this piece of code are there for convenience: Instead of having to select the entire code with the mouse, one may place the text cursor anywhere inside the code and then press Command-shift-B one or more times until the whole code enclosed within the outer parentheses is selected.

If you could not select the entire code with Command-shift-B or got an error message in trying to evaluate the code, see the following section.

Dealing with error messages caused by syntax mistakes

Typing mistakes are very common in writing code. There are chances that you already encountered an error when trying to evaluate a longer example that you typed in, such as the one of the previous section. When the system cannot understand the code that it is called to evaluate, it issues an error message. Learning how to read error messages is essential to correcting your code. For example, consider what happens if one forgets to put a semicolong after closing the function argument to `do`: in line 9 of the previous example:

```
(
Routine({
     w = SCWindow.new(" ", Rect(400, 400, 200, 200));
     w.front;
     30 do: {
          w.view.background = Color.rand;
          0.1.wait;
     } // error here! Missing semicolon
     1.wait;
     w.close;
}).play(AppClock)
)
```

Then the system will respond with this error:

```
· ERROR: Parse error
   in file 'selected text'
   line 9 char 2 :
     1·.wait;
     w.close;
----------------------------------
· ERROR: Command line parse failed
nil
```

It is easy to find out the mistake immediately in this case: The error message says that it encountered a parse error on the second character of the 9th line in the code. Furthermore, it posts the line where the error occured:

```
     1·.wait;
```

The sign · is placed exactly where the system stopped because of the error, so the mistake should be sought immediately before that. So the mistake is not between `1` and the following dot `.`but between 1 and what precedes it. Knowing that, as explained, each statement must be separated from the next one by a semicolon, one recognizes that there should be a semicolon between `}` and the following statement starting with `1`.

A similar error will occur if you omit a comma between the elements of a collection, as is the case between `3` and `4` in the following example:

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

Balancing parentheses

Another common source of error is the wrong balancing of parentheses in a nested statement. Parentheses are considered balanced when for every opening parenthesis "(" or square bracket "[" or brace "{" there is exactly one closing element of the same kind matching it, and the closing elements follow in the reverse order as the opening, i.e. the element opened last must close first:

```
{ }       // balanced: matching { }
{ ( ) }   // balanced: matching () enclosed in matching {}
{ (   } ) // not balanced: opening ( does not match closing }
[[ ]      // not balanced: one closing ] missing
[]]       // not balanced: one closing ] too much
```

Command-shift-B selects successive balanced portions of code. For example, place the cursor on the number 500 in the example below and press Command-shift-B several times. The portions of code selected successively by the command are:

```
{ Pan2.ar(SinOsc.ar(LFNoise0.kr(10, 500, 600), 0, 0.1), LFNoise0.kr(5)) }
{ Pan2.ar(SinOsc.ar(LFNoise0.kr(10, 500, 600), 0, 0.1), LFNoise0.kr(5)) }
{ Pan2.ar(SinOsc.ar(LFNoise0.kr(10, 500, 600), 0, 0.1), LFNoise0.kr(5)) }
{ Pan2.ar(SinOsc.ar(LFNoise0.kr(10, 500, 600), 0, 0.1), LFNoise0.kr(5)) }
{ Pan2.ar(SinOsc.ar(LFNoise0.kr(10, 500, 600), 0, 0.1), LFNoise0.kr(5)) }
```

Now test what happens when a parenthesis is missing or in the wrong order. Try to evaluate the following line:

```
{ Ringz.ar(WhiteNoise.ar(Decay.kr(Impulse.kr(2,0,0.01)),440,0.1) }.play
```

The resulting error message includes a second section that specifies the cause of the parse error:

```
opening bracket was a '(', but found a '}'
  in file 'selected text' line 1 char 66
• ERROR: Parse error
   in file 'selected text'
   line 1 char 66 :
  { Ringz.ar(WhiteNoise.ar(Decay.kr(Impulse.kr(2,0,0.01)),440,0.1) }•.play
```

Some care is required to pinpoint the source of the mistake. Using the Command-shift-B technique above can save time. Here are the steps:

Step 1. Finding where the balanced part stops

As a general rule, move the text cursor inside the ***second*** closing bracket immediately preceding the error location indicated by •, then type Command-shift-B several times. Each time, the editor

will try to match a larger stretch of text that is enclosed in balanced brackets, until it reaches the largest stretch that is balanced. In this case this would be:

```
{ Ringz.ar(WhiteNoise.ar(Decay.kr(Impulse.kr(2,0,0.01)),440,0.1) }.play
```

Step 2: Identifying the unmatched element:

The `(` left of `Decay` and the `)` right of `0.1` are the last balanced brackets, so look immediately beyond them for the mistake: To the left is the `(` left of `WhiteNoise.ar`. This does not match the `}` that immediately preceds `.play`, to the right of the selected stretch of text. Therefore, the closing bracket to `(WhiteNoise.ar` is missing.

Step 3: Providing the matching element in the right place:

This is the tricky part. Since the balanced portion ends after 0.1, one might try adding a closing parenthesis after 0.1.

```
{ Ringz.ar(WhiteNoise.ar(Decay.kr(Impulse.kr(2,0,0.01)),440,0.1)) }.play
```

However this is wrong. The closing element needs to be provided exactly in the position required by the meaning of the code. In this case, this depends on the arguments that are required by the Unit Generators (UGens) that make up this synthesis function. Fortunately, there exists a help file that documents what arguments are required by the UGen `Ringz.ar`. To open that file, select the name of the UGen thus: `Ringz` (a double-click on the name will do it), and then type Command-?. The help file starts by giving a summary of the usage of this UGen on one line, and then proceed to explain the meaning of each argument.

Ringz.ar(in, freq, decaytime, mul, add)

Note that the first argument (`in`) is an input signal, and the second argument the resonant frequency. Looking at our code, it becomes clear that here the argument `in` is the WhiteNoise component, while the number 440 must be the resonant frequency of `Ringz.ar`. By contrast `WhiteNoise.ar` component will have only one argument in this case (see also the Help file to WhiteNoise). Therefore the parenthesis opening at `WhiteNoise.ar(` must close immediately after the first argument. The correct code is:

```
{ Ringz.ar(WhiteNoise.ar(Decay.kr(Impulse.kr(2,0,0.01))),440,0.1) }.play
```

Syntax-coloring

SuperCollider can color the code in a workspace window to distinguish comments (colored red) and certain other parts of code such as class names and reserved keywords (blue), strings (gray) and symbols (green). To apply coloring, type Command-'.

Getting help from the system

Several keyboard shortcuts provide information to help you write and debug your code. The most basic of these is Command-?. It tries to open the help file whose name corresponds to the selected text. If no such file is found, it opens the default help file, which is: Help.help.rtf. This lists many further help files, grouped into topics. Clicking on the name of a file on the Help.help.rtf window opens that file.

To get help about the Server, select the text Server thus:

```
Server.local
```

and type the keyboard combination Command-?.

Here is a list of keyboard commands that offer information about the system:
Command-?: Opens the help file corresponding to the selected text, or the default help file.
Command-J: Opens the class definition of the selected text. If the selected text includes a message as in: `Object-dump`, then the cursor jumps to that method.
Command-Y: Lists all classes that define methods for the selected message.
Command-shift-Y: Lists all methods whose contains the selected message or class name.

In addition to the above commands, one can obtain information about a class through a series of messages such as: `dumpInterface`, `dumpClassSubtree`, `dumpFullInterface` and others. More information on this as well as on other techniques for looking into certain kinds of objects can be found in the help file *InternalSnooping* and the help file for *Class*.

Finally, one can look at the contents of objects by sending them the message inspect:

```
Server.local.inspect;
```

This opens an inspector window. Clicking on the small buttons labeled "|" next to each component (variable) on the window opens a new inspector at the bottom of the parent inspector, showing the contents of the variable.

Saving your work

You can save the contents of a window in a file on disk under any name. Type Command-S to save the code, or Command-shift-S to save it under a different file name.

At the bottom of save file dialog is a menu for selecting among the following file types:

1. **NeXT Rich Text Format v1.0 pasteboard type**: The ending .rtf will be appended to the file name. This format preserves the formating of the text (font, size, color, alignment) but cannot save graphics
2. **NeXT RTFD pasteboard type**: The ending .rtfd will be appended to the file name. This is the same as type 1 above, but has the additional capacity to save pasted graphics (pictures). This is actually a directory that contains the text and the pictures separately.
3. **NsStringPboardType**: Saves the text only as plain text.
4. **Apple Simple Text Document**: Saves the text with formating as document that can be opened also by TextEdit application. Cannot save graphics.

5 **SuperCollider document**: Saves the text only without formatting. When the document opens, SuperCollider will use the default Monaco font and will syntax-color the text.
6 **SC help (HTML):** Saves the text in HTML format. Does not save graphics.

The preferred formats are 1 and 2. Use 2 only if your window contains graphics. All of the above formats are compatible with Apple's TextEdit application.

Do not append the ending .sc to your file name if it contains code to be evaluated interactively. This is reserved for class definition code. Files that are in the SCClassLibrary folder and end in .sc or sc.rtf are compiled by the client at startup or with Command-K.

Troubleshooting

First thing to check: Is this the right copy of SuperCollider?

A very common cause of problems is starting SuperCollider from another copy of the applicatoin than that intended. If you or another user has downloaded many copies of SuperCollider on the same computer, then

Perhaps the most failsafe method to make sure that you are running a

SuperCollider client cannot start / crashes upon start

It is possible for different users on the same computer to start concurrently several copies of the SuperCollider client application from the same SuperCollider application file residing in the SuperCollider_f folder under /Applications, and to control independently one single local server application (scsynth) running on this computer.
Check these possible error sources:
- Is another copy of SuperCollider running in parallel on the same computer, possibly on another user account?
- Has this copy of SuperCollider been installed from another user account, and especially, is that user currently logged in?
- If none of the above is the case, then

SuperCollider server cannot start / crashes upon start

- Is another server running on the same user account in the background? (show how to find this out with terminal and `ps -ax | grep scsynth`, refer to Meta_Server-killAll ...
- Incompatibility with other applications
- Incompatibility with audio hardware and drivers: It is possible that
- Does the startup code of SuperCollider client or the server contain custom changes that involve loading large buffers or complicated synthdefs? In that case either
-

Cannot get sound output or input from the server

Refer to section above on SuperCollider server cannot start - about incompatibility with other applications and drivers
Also include notes on:
- testing your sound hardware and connections (basics ;-)
- checking that your computer is not on mute ("silenced")
- checking that the audio I/O settings (preferences) of your computer are set to the hardware that is actually connected or that your are expecting the input or output to take place at.

SuperCollider library cannot compile

First of all consult the error messages ...

- Has this copy of SuperCollider been installed from another user account (no write access to needed folders inside *SuperCollider_f*).
- Are you sure you are running the right copy of SuperCollider? If there are multiple copies of SuperCollider ...

No GUI windows appear when SuperCollider client starts up

This is most probably an effect of SuperCollider library not being able to compile. Look at the post window (Untitled) and consult the previous section: *SuperCollider library cannot compile*.

My own library does not work - cannot be found

(See case of which copy of SuperCollider is running in section SuperCollider library cannot compile above.)

(Note: include extra comments here ...: Maybe the supercollider library compiles but is not the library which includes the code you want to use!)

Making Sounds

Playing a function

The shortest way to make a sound in SuperCollider is to "play" a function. Sending the message `play` to a function will cause the client to convert this function to a *synthesis definition* (SynthDef), that is, the definition of a synthesis algorithm that the server uses to produce signals. The client then sends the resulting SynthDef to the server. After this has been loaded on the server, the client requests from the server to make a sound from the SynthDef by creating a new synth. Here is a very short session that makes a sound:

Before starting, boot the server:

```
Server.default.boot;    // Boot the default server
```

After the server has booted, play this function to produce a sine tone:

```
// Play a sine tone of frequency 440 Hz, phase 0, amplitude 0.1
{  SinOsc.ar(440, 0, 0.1) }.play;
```

Since no server was specified with the play message, the default server will be used.

The keyboard shortcut Command-. stops all synths on the server and all scheduled processes (routines, tasks etc) on the client.

Note that this method of playing sounds is a shortcut for trying things out or for sounds that are played once in a piece. To create sound algorithms that can be re-used, one stores these in SynthDefs, as will be explained in section *Storing synth algorithms in synth definitions (synthdefs)*.

Stopping all sounds

To stop all sounds, press Command-. (Command-dot). This also stops all routines.
Other commands that one can enter as code to stop servers are:

```
Server.quitAll;         // quit all servers (will require rebooting them!)

Server.default.freeAll; // stop all sounds on the default server
```

Viewing sound in a scope

The internal server can display a scope of both audio and control signals. To prepare for using it, boot the internal server and preferably also set it as default and store it in s, so that all synths will by default get started in it:

```
Server.default = s = Server.internal; // set defaults for this session
s.boot;                               // boot the internal server
```

To open a scope on the internal server, send it the scope message:

```
s.scope; // open default scope on internal server
```

The server will only open one scope. Resending it the message scope has no effect, unless you close the existing scope first. You may want to customize the buffer size, that is the size of the signal that is displayed by the scope and the zoom factor, that is how much of the signal is displayed on the same stretch (width area) of the screen. These two parameters are set with the `bufsize` and `zoom` arguments to scope. Both of these should be given in powers of 2. One can calculate what amount of time of the audio signal a certain buffer size will display, by dividing the number of samples of the buffer by the sampling rate. Here are the durations of some useable buffer sizes:

```
// calculate and collect buffer sizes and their corresponding signal
durations
// (using a List Comprehension. See help file ListComprehensions)
all {: [x, (x/44100)round:0.01], x<-2**(10..16) }

// Result - durations rounded to 0.01 for brevity:
[1024,0.02],[2048,0.05],[4096,0.09],[8192,0.19],[16384,0.37],
[32768,0.74],[65536,1.49]
```

For larger buffer sizes you should set also a larger zoom factor, otherwise the signal will stretch outside the bounds of the screen. You can resize the window manually do obtain a better vertical resolution or to view the entire portion of the signal displayed by the buffer. To make these effects visible, start a sound first:

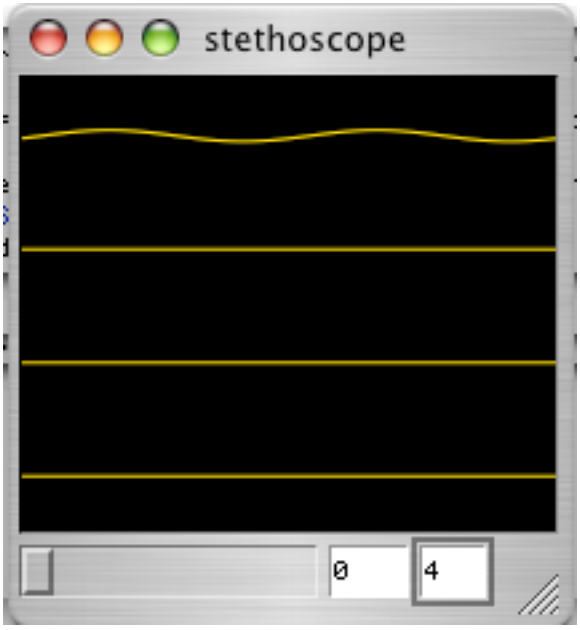
```
{ SinOsc.ar(440, 0, 0.1) }.play

s.scope; // default zoom is 1. Cannot stretch window to see all the signal
```

Close the window each time to make a new scope with different settings:

```
// Default buffer size: 4096 ~ 0.09 sec & zoom to see all the buffer.
// stretch the width of the window by about 2 cm to see all the signal:
s.scope(zoom: 16);

// zoom to see about 1.49 seconds of audio at once:
s.scope(2, bufsize: 2 ** 16, zoom: 64);
```

The slider at the bottom of the window allows you to scroll between all audio busses of the server, meaning that you can see the signals not just of the busses that are being output, but also of all

internal busses. The left number box displays the number of the uppermost bus that is displayed. The right number box displays and sets the number of busses that are displayed at once. The values of both number boxes can be set by selecting them, typing in a number, and entering it with the enter key.



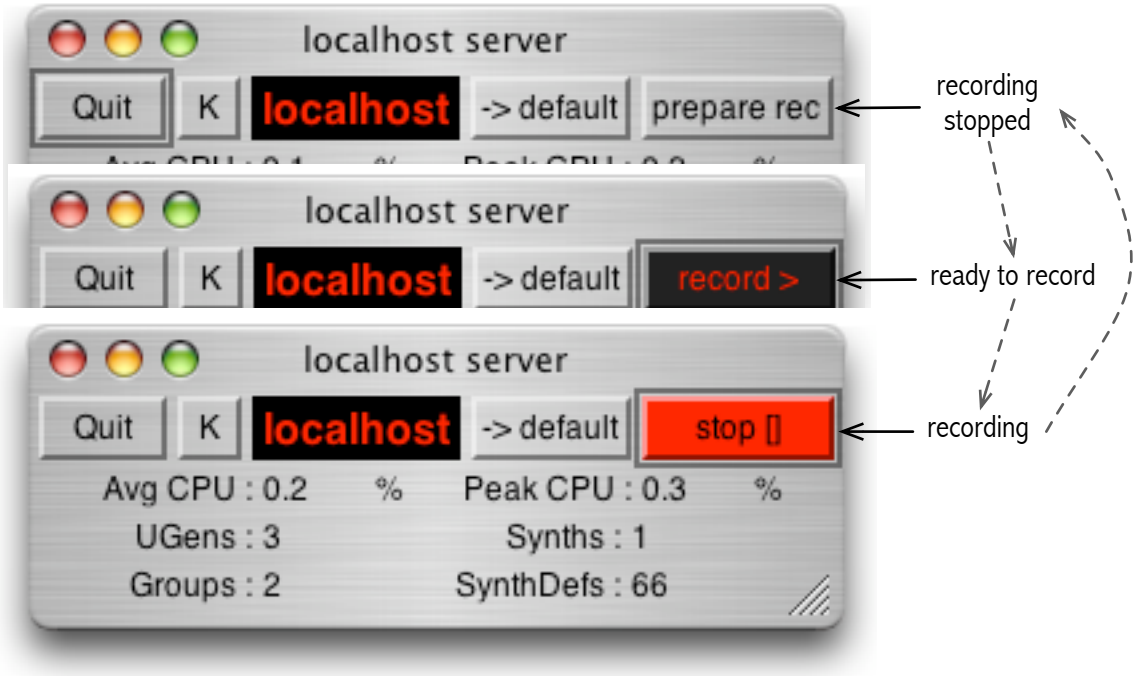
Viewing 4 audio busses, starting with channel 0 on top

One can toggle the view between audio busses and control busses by typing the key K when the scope window is in the front. A scope can also be opened by sending the message `scope` to a function:

```
{Ringz.ar(WhiteNoise.ar(Decay.kr(Impulse.kr(2,0,0.01))),440,0.1)}.scope(zoom:
16)
```

Recording audio output to disk

The upper right button of a server window allows one to record the output of the server to disk. Each time one starts a recording with this button, the output is recorded in a new sound file which is placed inside the "recordings" folder and is named with a date and time stamp. The "prepare record" button is enabled only if the server is running. Click on the button once to get ready to record. The button text will change to "record>". This means that the server has created a file for the recording but has not started writing on it yet. Click on the record button once more to start the recording. The button will turn red and the text will now show "stop []". To stop the recording, click on the same button yet again. The button will return to its initial state "prepare rec", meaning that the system is ready to prepare another recording. The upper part of the server window will look as follows during the three phases of the recording process:



This method is adequate for most applications that record the full output of the server exactly as it is sent to the output hardware. To record only specific busses or from a higher stage of the synthesis process before the output, one can build custom recording techniques making use of the classes `Buffer` and `DiskOut`. Another class, `RecordBuf`, allows you to record the output in RAM so that you perform real-time sampling of the input or output for further use during a performance. For more details, consult the help files of these classes.

Synths and UGens

When "playing a function" as in the previous section, the client creates a *synth*. A synth is an object that represents a synthesis process that runs on the server (the synth *process* runs on the server, the synth *object* that represents it exists on the client). One can control a synth process by sending messages to the synth object that represents it. To start a synthesis process on the server, one creates a new synth object. Playing a function as shown above creates a synth implicitly, later sections will show how to create a synth explicitly. One can follow the changes that happen in the system when one creates or stops ("frees") a synth through information on the Post Window and on the GUI window of the corresponding server. Before starting the next example, make sure the default server is stopped, so that no synths are running on it:

```
// Boot the default server anew to start with a clean slate:
Server.default.boot;
```

Now look at the default server window (normally the local host server). If no synths are playing, then the entry "Synths : " will show a count of 0 and so will the entry "UGens : ". Next create a synth by playing a function:

```
// Create a synth from a function
{ PinkNoise.ar(LFPulse.kr(LFNoise1.kr(0.5, 15, 15.05), 0, 0.4, 0.1)) }.play
```

Watch the changes on the server window:

- The *Synths* entry now shows a count of 1. This signifies that a new synth has been created and is running on the server.
- The *UGens* entry shows a count of 11. This means that the new synth contains 11 UGens. All Synths are made up of UGens, that is of unit generators which process and synthesize signals.

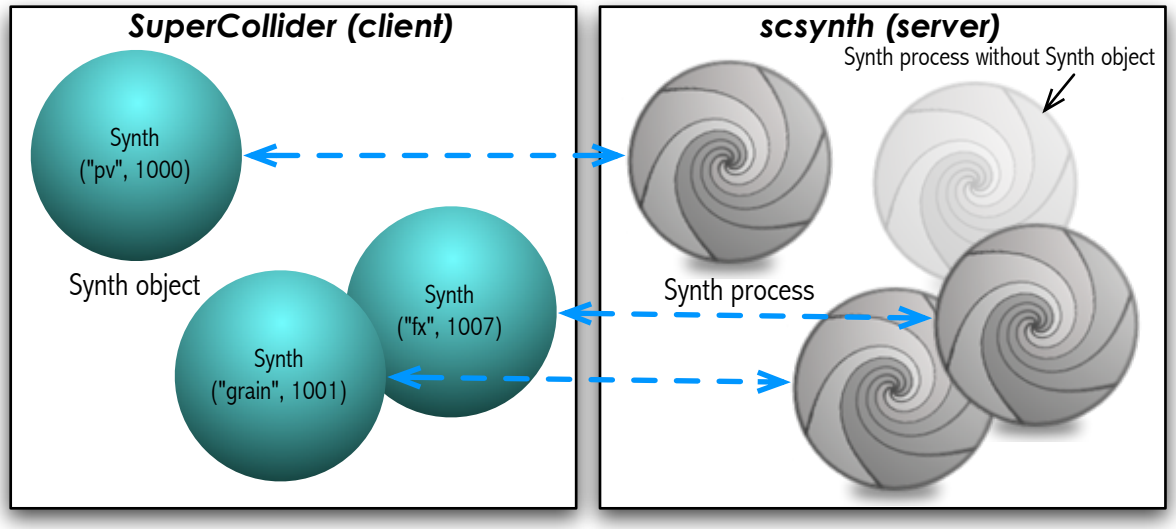
Watch the output on the post window. It should show something like:

```
Synth("1438563257" : 1000)
```

This means that playing the above function has created a new synth object. The first element inside the parentheses is some number enclosed in double quotes: `"1438563257"`. This is the name of the synth. Since the synth was created by the system from the function, the system provided as name the internal number that represents the function. The second number (`1000`) is the ID number of the synth on the server. This number can be used to address the synth while it is running in order to change some of its characteristics (parameters) or to stop it.

Stop the synth by typing Command-. before proceeding to the next section. Note that the *UGen* and *Synth* counts go back to 0 after the synth stops.

A synth object created in the client (language) is actually different entity from the synth process created in the server. The synth in the language is merely an "avatar" of the real synthesis process that runs on the server, and is used to control that process. The figure below shows how the synth instance Synth("sine", 1000) represents one of the many synth processes that may run on the scsynth server at any moment.



There may be synth processes running on the server that have no synth instance on the client. Conversely, when a synth process stops, any synth object that represents it no longer has a counterpart on the server.

Creating (starting) and freeing (stopping) individual synths

The previous section demonstrated how to start a sound in SuperCollider, by creating a new synth. To stop a synth, one sends it the message `free`. In order to be able to do this, one must first store the synth somewhere. One way to do this, is to store a synth in an *interpreter variable* (For more information on variables, see chapter *The Language*). One can start or stop any number of synths individually at any moment. In the following example, we start and stop two different synths, stored in variables *a* and *b* respectively.

```
// Store the synth created from the function in variable "a"
( // ("Alien subtropic drummers")
// Select all 3 following lines and evaluate:
a = { RLPF.ar(LFTri.ar(Demand.kr(Impulse.kr([10, 5.1]), 0,
     Dser((200,300..1100).scramble, inf)), 0, 0.05),
          LFNoise0.kr([0.2,0.21], 300, 400), 0.01, 0.1) }.play;
)

// ------------------
// ("Data-space in a slightly lyric mood")
// Store another synth in variable "b"
b={SinOsc.ar(LFNoise0.kr([0.3, 0.31], 700, 900), 0,
Trig.kr(Dust.kr([10,10],0.1)))}.play;

// ------------------

a.free // Stop the synth stored in a

// ------------------

b.free // Stop the synth stored in b
```

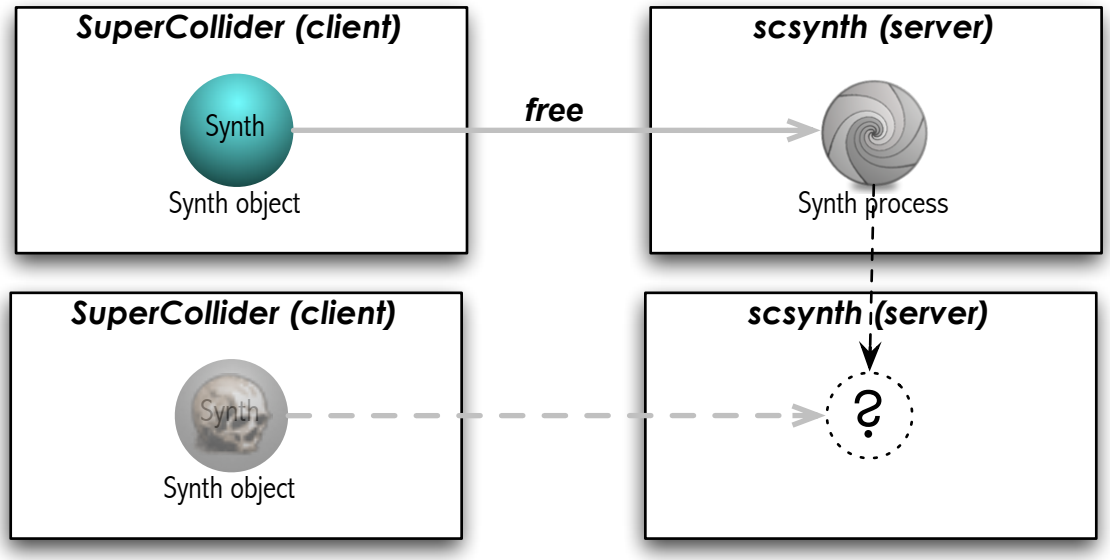

If you try sending the message `free` to a synth that has already stopped, you will get a warning message from the server. For example, try stopping the synth in *a* again:

```
a.free // Warning: Trying to stop a synth that has already stopped.
```

The error issued is:

```
Synth("1189365759" : 1000)
FAILURE /n_free Node not found
```

This means that the node which was holding the synth number 1000 cannot be found in the server, because it was removed from the server when the synth was stopped, as shown in the figure below. (Node is a generic concept for synths or groups of synths that can be controlled together.)



Summary:

- To start a new sound, create a synth
- Playing a function with the message "play" creates and returns a synth
- Store synths in variables to be able to access them later
- To stop a synth, send it the message "free"
- Once a synth has been stopped, it can neither be stopped nor started again

Warning: To play sound, the server must be booted

If you try to play a function on a server that is not booted you will get a warning:

```
{ SinOsc.ar(440, 0, 0.1) }.play
WARNING:
server 'localhost' not running.
nil
```

The operation returned `nil` because no synth was created, because no server was running to create it in.

From now on, make sure that the server is booted before trying any examples that make sound.

Anatomy of a synth

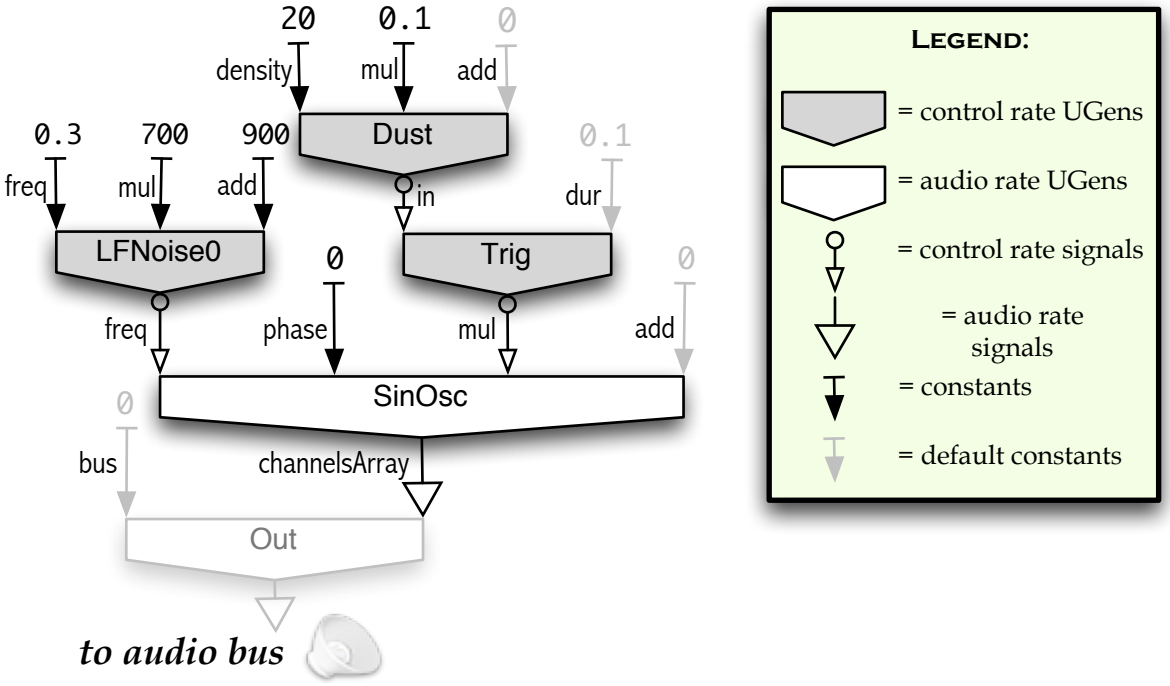
This section shows how the synth-definition code that is contained in a function is translated by the language into a synthesis algorithm that runs inside the synth on the server. All synths are composed of unit generators (*UGens*), which are connected to each other via signals. A *UGen* is a fundamental synthesis processing unit. It has inputs in the form of constant numbers or signals, and it outputs signals (in most cases just one signal). Signals are continuous streams of numbers that are produced at a steady rate. SuperCollider defines three rates of signals that run between the UGens inside of a synth:

Audio rate signals (`ar`): These run at a rate that can be used to input or output audio on the computer (current default is 44100 Hz).
Control rate signals (`kr`): These run at a fraction of the audio rate depending on the internal buffer size of the synthesis engine. They are used as inputs to UGens and provide continuous control to their parameters.
Initial rate signals (`ir`): An `ir` signal has an unchanging (constant) value that is initialized when the synth is created.

A UGen is audio rate, control rate or intial rate, depending on the kind of signal that it outputs. UGens are created in synthesis-definition code by sending to a UGen class one of the messages `ar` (for audio rate UGens), `kr` (for control rate UGens) or `ir` (for initial rate UGens). As an example, let us look one of the synthesis functions of the previous section:

```
b = { SinOsc.ar(LFNoise0.kr(0.3, 700, 900), 0, Trig.kr(Dust.kr(10, 0.1)))
}.play;
```

The code contains three control-rate UGens (`LFNoise.kr`, `Dust.kr`, `Trig.kr`) and one audio rate UGen (`SinOsc.ar`). In addition, there is one output UGen which is provided by the system through the method of `Function-play`, as a convenience to the user. This is the UGen `Out.ar` which directs the output from the last statement of the function to an audio bus that leads to the computers audio output. The UGens are interconnected to each other as shown in the following figure:



The basic principle for translating the code into a configuration such as above is simple: The arguments following each UGen represent inputs of the UGen. These inputs can be either a constant number or the signal output from another UGen. The output of the last UGen is sent to `Out.ar` which sends it to the system for audio output. The last UGen is the one which is not contained in another UGen and is not followed by any other statement in the function. In this case this is the leftmost UGen: `SinOsc.ar` (see note on computation order below). Its first input is the UGen `LFNoise0.kr`. The inputs of `LFNoise0.kr` are the constants 0.3, 700, and 900.

A NOTE ON COMPUTATION ORDER:

The arguments of a message are computed first, *then* they are sent to the receiver with the message. Thus, the last object computed in a nested message statement is the *leftmost* receiver.

The last two UGens show how defaults are provided by the system: The statement `Dust.kr(10, 0.1)` declares a kr UGen with two constant inputs: 10, 0.1. But the Dust UGen requires three inputs: `density`, `mul` and `add`. Most inputs, and especially the inputs named mul and add, have defaults, so their values do not need to be specified by the user unless they deviate from those defaults. The most reliable way of finding out the default values of arguments is to look at the code of the method concerned. To find the code of the method corresponding to `kr` as message sent to the class `Dust` we need to find the ***class*** method `kr` of `Dust`. For this we need to address the ***class of the class*** Dust. Write and select: `Meta_Dust-kr` (the dash `"-"` is correct), then type Command-J. The system opens the code at this location:

```
*kr { arg density = 0.0, mul = 1.0, add = 0.0;
```

The * before `kr` indicates that this is a class method. The input names and their defaults are given here as arguments, listed after the argument declaration keyword `arg`, and each argument followed by = and the corresponding default value. Thus, the inputs of `Dust.kr` are `density` (default: 0.0), `mul` (default: 1.0) and `add` (default: 0.0).

To look for the code of a method for a message sent to a class, add `Meta_` before the name of the class. To look up `Dust.ar`, write and select `Meta_Dust-ar`, then type Command-J.

UGens vs Synths

UGens are the units that do the actual signal processing. The relation of UGens to Synths is similar to that of sub-atomic particles inside of atoms in physics. Atoms consist of sub-atomic particles as Synths consist of UGens. But UGens cannot exist outside of Synths. That is, you cannot perform in the interpreter:

m = MouseX.kr(-1.0, 1.0);

Anything that takes the message .ar, .kr or .ir must be included inside a SynthDef. It will then be run when a Synth is created from that SynthDef. SynthDefs are explained in the following chapters.

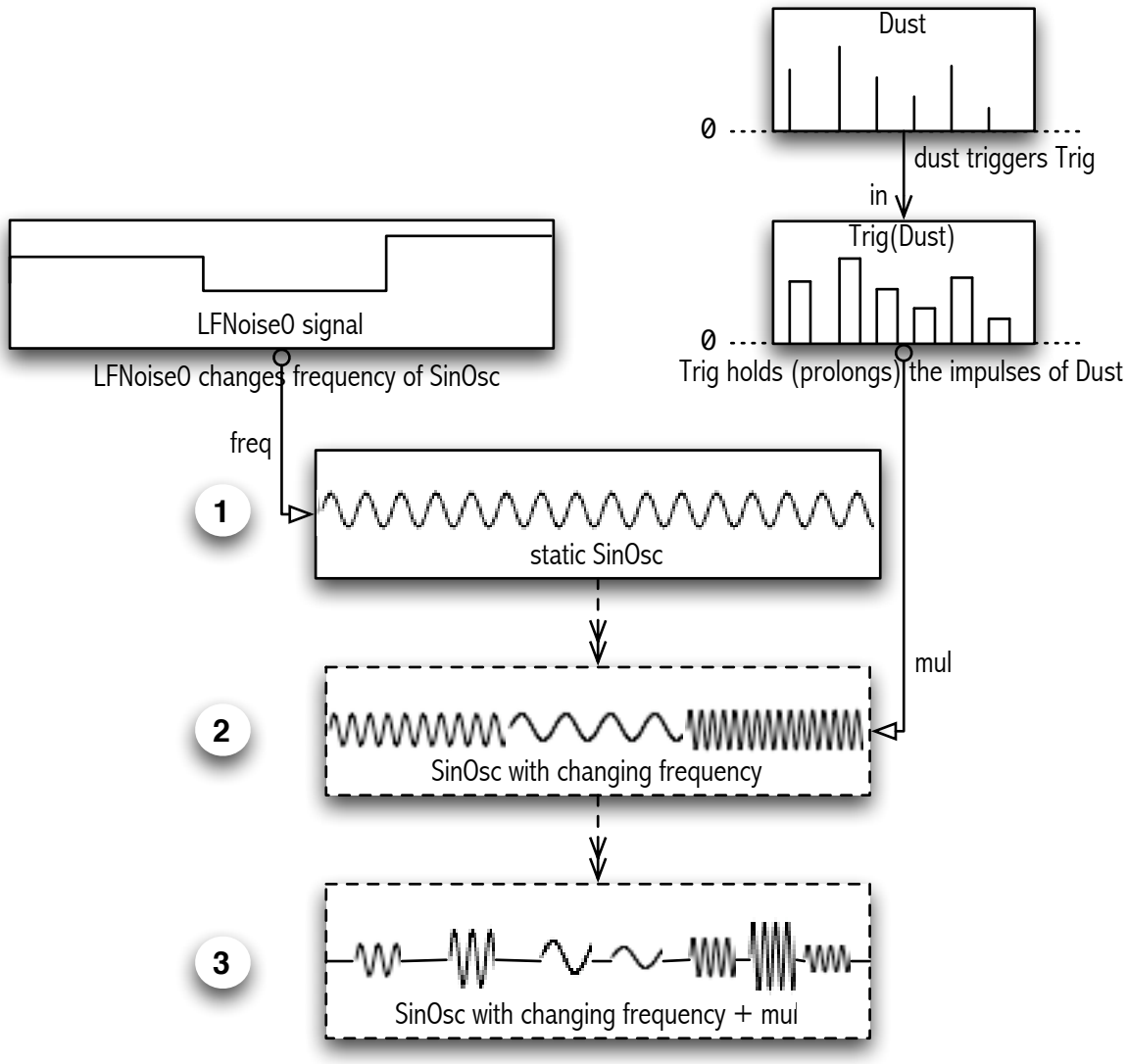
Parameter change and signal transformation

A signal that is connected to an input of a UGen will in most cases affect it in one of two ways:

1. It can change the parameters of the algorithm that produce the output signal, thereby changing the way that the UGen synthesizes its output.
2. It can function as an input which is transformed by the UGen that operates on it.

The change of a parameter is illustrated in the example of the previous section by both the input of `LFNoise0` as freq parameter input and the input of `Trig` as mul parameter input to `SinOsc`. The effect of the change of the freq paremeter on `SinOsc` is shown below by the change in the signal between (1) and (2), while the effect of the change in the `mul` signal is shown between (2) and (3). The effect of `Trig` on the signal output by `Dust` is shown in the box *Trig(Dust)*. The "transformation" of the signal from `Dust` consists in prolonging single impulses to make them last for 0.1 seconds each (the default value of the parameter `dur` in `Trig`).

The parameters `mul` and `add` exist in the majority of UGens. `mul` scales the signal, that is, it changes its amplitude. `add` transposes the values signal by its value. The default values of `mul` (1) and `add` (0) leave the signal unchanged. UGens that transform the signal of one of their inputs usually name that input "`in`".



Chapter *Synthesis Techniques: Generation, Control and Transformation*, sections *Control* and *Transformation* explains the different techniques related to these topics in detail. The present chapter introduces the basic elements of such techniques such as envelopes, control with the mouse, sending set and map messages, filters and other transforms applied to signals.

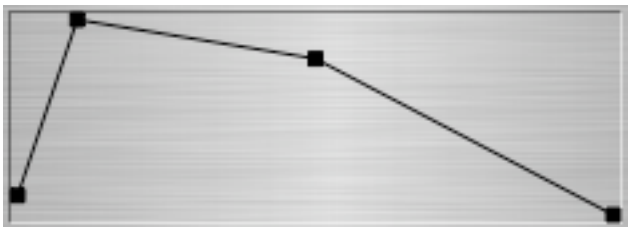
Defining synths of limited duration: Envelopes

Envelopes are objects that are used to create shapes of control signals of a specific duration. The most common application of an envelope is to control the amplitude of the overall output signal of a synth and to stop that synth after a certain duration. For example:

```
{ SinOsc.ar(400, 0, 0.1 * EnvGen.kr(Env.perc, doneAction: 2)) }.play
```

An envelope describes the fixed shape of a signal through a limited number of points in time,

called *break-points* or *nodes*.



An envelope with 4 break-points (nodes)

The shape above corresponds to an envelope created by the specification:

```
Env([0.1,1.0,0.8,0.0],[0.1, 0.4, 0.5])
```

The first array of numbers specifies the value of each break-point on the y-axis and the second array of numbers specifies the distance (a time interval in seconds) between each successive pair of points on the x-axis. Playing an envelope creates a signal that takes successive values between the points in the y axis, travelling between each break-point and the next one within the time-interval specified on the x-axis. When the envelope has reached the last point, the signal will stay constant at the y-value of the last node. Additionally, one can ask the envelope to stop the sound that it controls when it reaches the last node.

The manner in which the signal changes ("interpolates") between one break-point and the next one is called in SuperCollider the type of "curve". The actual shape of an envelope is not limited to straight line segments between the points. SuperCollider provides the following different ways for interpolating between these: *step*, *linear*, *exponential*, *sine*, *welch*, *float* (individual curvature for each segment). One can also construct an envelope by blending between other envelopes (see help file *Env*).

Envelopes are produced by instances of the `Env` class, and then played by the UGen `EnvGen`. Use the messages `plot` to display envelopes on a GUI window and `test` to play them with a test-tone:

```
// Evaluate each line after one another - not all lines at the same time:
Env.perc.test.plot; // (close each plot window before making the next one)
Env.sine.test.plot;
Env.new([0,1, 0.3, 0.8, 0], [2, 3, 1, 4],'linear').test.plot;
Env.new([0.001, 1, 0.3, 0.8, 0.001], [2, 3, 1, 4],'exponential').test.plot;
Env.new([0, 1, 0.3, 0.8, 0], [2, 3, 1, 4],'sine').test.plot;
Env.new([0.001, 1, 0.3, 0.8, 0.001],[2,3,1,4],'welch').test.plot;
Env.new([0, 1, 0.3, 0.8, 0], [2, 3, 1, 4],'step').test.plot;
```

In a synth, envelopes are played by envelope generators. An envelope generator (`EnvGen`) is a UGen whose output is a signal of fixed shape and duration. The most common use of an envelope is to provide control input for the `mul` (amplitude) parameter of a UGen and at the same time to limit the duration of the synth that contains that UGen. To stop (remove) the synth after the envelope has ended, one has to specify a value of 2 for the argument `doneAction` of `EnvGen`:

```
{ WhiteNoise.ar(EnvGen.kr(Env.perc(level: 0.1), doneAction: 2)) }.play;
```

```
{ GrayNoise.ar(EnvGen.kr(Env.perc(1.5, 0.01, 0.1, 5), doneAction: 2)) }.play;
```

While above the envelope was provided as a multiplier argument to a UGen, it can also figure by itself as a UGen that is mutliplied with the signal some output. Notice that multiplying an audio rate UGen with a control rate UGen is allowed:

```
(
{ BrownNoise.ar(Decay.kr(Impulse.kr(3, 0, 0.1)))
     * EnvGen.kr(Env.perc(releaseTime: 10), doneAction: 2) }.play
)
```

One can use envelopes to control also other parameters. The following example uses one envelope to control the overall amplitude and another one to control the changing frequency of an impulse that drives the repetition rate of a percussive tone:

```
(
{ GrayNoise.ar(Decay2.kr(Impulse.kr(
     EnvGen.kr(Env([3, 10, 2, 25, 1], {1.5}!4, \exponential)))))
     * EnvGen.kr(Env.perc(1.5, 15, 0.05, -1), doneAction: 2) }.play
)
```

Storing synth algorithms in synth definitions (synthdefs)

A synth definition is a way to store a synthesis algorithm in the server, so that many synths can be created from it at any time. Consider an algorithm that creates a short sound such as this:

```
{ WhiteNoise.ar(EnvGen.kr(Env.perc(0.01, 0.25, 0.1), doneAction: 2)) }.play;
```

To repeat this kind of sound many times one might write a routine that plays the same function:

```
(
Routine({                          // create a routine
     30.do {                       // repeat 30 times
          {                        // play a synth function
               WhiteNoise.ar(
                    EnvGen.kr(Env.perc(0.01, 0.25, 0.1), doneAction: 2))
          }.play;
          // wait for a random interval between 0.01 and 0.15 sec:
          0.01.rrand(0.15).wait;
     }                             // end of repeated function
}).play                            // play the routine
)
```

However, this is not economical, because it means that the computer must translate the code of the function into a synthesis algorithm and send this algorithm to the server for each single synth that is created. This can be an enormous waste if a piece creates many synths, and can easily result in unwanted delays during performance. So instead, the algorithm can be loaded on the server once under a name given by the user, and then one can create any number of synths from it. The synth

engine is designed to store each synthesis algorithm under a different name, so that it can be recalled to create a synth. A synthesis algorithm that can be loaded to a synth under a name is called a *synthdef*. The number of synthdefs that can be stored is limited (the default is 1024). This means that normally, one can only play up to 1024 different sound events by playing functions directly. The number of events (synths) that can be created from any single synthdef stored in the server is unlimited.

```
// Create a new SynthDef
SynthDef("white_burst",// name of the synthdef is "white_burst"
                       // Start of synthesis function:
     { Out.ar(0,       // specification of output bus!
          WhiteNoise.ar(
               EnvGen.kr(Env.perc(0.01, 0.25, 0.1), doneAction: 2)))
     }                 // end of synthesis funtion
).load(Server.local);  // Load the synthdef to the local server
```

After the SynthDef has been loaded, synths can be created from it by specifying its name:

```
Synth("white_burst"); // Create a synth from synthdef "white_burst"
```

The above routine will run much more efficiently if one creates synths directly using the preloaded synthdef:

```
(
Routine({                          // create a routine
     30.do {                       // repeat 30 times
          Synth("white_burst");    // make a synth
          // wait for a random interval betweeb 0.01 and 0.15 sec:
          0.01.rrand(0.15).wait;
     };                            // end of repeated function
}).play                            // play the routine
)
```

The function used in the SynthDef was not exactly the same as the function that was used for playing directly. This is because the play message to functions provides an additional shortcut for the user: It sends the output of the function by default to the left channel - and to the channels following it if the output contains more channels than 1. But synthdefs require that the output channel be specified in the code of the function. This is necessary in order to give the user the possibility of sending the output to any channel.

For a full discussion of synth definitions see section *Anatomy of a synthdef* in the *Reference* chapter.

How synthdefs work

Sending vs. loading synthdefs

(include diagram of the sending and loading procedures, the client and server applications, and the synthdef folder)

Explain that synthdefs sent are not permanent while those loaded will be there the next time that the local or internal servers are booted.

Control techniques

Creating interfaces for external control of a synth

The synth examples shown up to here could not be "tuned" or "customized" in any way. We could start and stop a synth but we could not affect its behavior while it is running, or even at its start. This section shows how to control the parameters of a synth while it is running, by defining controls for that synth. A synth control is a UGen object whose signal output value can be changed either by sending it an OSC message, or by binding it to a bus. In this way, the parameter of a synth that is bound to that control can be changed at any moment during the function of the synth. A convenient way for creating controls for a synth is to declare them as arguments to the synth function:

```
// Create a synth with controls for the parameters freq and amp:
a = { arg freq = 400, amp = 0.1; SinOsc.ar(freq, 0, amp) }.play
```

The values 400 and 0.1 are default values for freq and amp respectively. These must be provided because they are needed to initialize the value of these parameters when the synth is created.

To change the value of a parameter, one sends the synth the message `set`. The syntax is:

```
a.set(\freq, 500); // set the value of freq to 500
```

One can change many parameter-value pairs with one message:

```
a.set(\freq, 1000, \amp, 0.05);
```

Following routine changes the freq and amp parameters 50 times:

```
(
Routine({
     50.do { |i|
          a.set(\freq, ((i * 50) + 200), \amp, 1 / (1 + i));
          0.2.wait;
     };
}).play;
)

// stop the synth in a to end the example:
a.free;
```

Controls have many other features:

- The value of a control can be set at the time that the synth is created (see Setting parameters of

a synth at its creation)
- A control can be made to change between values "smoothly" by setting a lag factor
- A control name can be bound to an array of control values so that one can set several values with one control name, for example to specify the frequencies of a filter bank.
- A control can be made to read its values continuously from a bus by "mapping" that control to the bus.

Setting parameters of a synth at its creation

To "customize" the sound of a new synth immediately, one can override the default initial values of controls by passing them as extra arguments to the synth creation message.

```
// Create and load a synthdef with controls:
SynthDef("ping", { | freq = 400, amp = 0.1, dur = 1.0 |
     Out.ar(0, SinOsc.ar(freq, 0,
          EnvGen.kr(Env.perc, 1.0, amp, 0, dur, 2)))
}).load(Server.local);

// Create a "ping" synth with default parameters:
Synth("ping");

// Create a "ping" synth with customized parameters:
Synth("ping", [\freq, 350, \amp, 0.4, \dur, 5]);
```

It is slightly faster to address the controls by their number , in the order they appear as arguments in the synth definition function - instead of by their name. In cases where performance optimization is needed, instead of:

```
     Synth("ping", [\freq, 350, \amp, 0.4, \dur, 5]);
```

write:

```
     Synth("ping", [0, 350, 1, 0.4, 2, 5]);
```

A little rhythmic improvisation with a routine:

```
(
Routine({
     5.do {
     ([0, -4, -3, 3, 11].choose +
     [60, 64, 65, 67, 69, 71]).midicps.scramble.pyramid(5).curdle(0.8) do:
          { |c| c do: { |f| Synth("ping",
          // control 0 is freq, 1 is amp, 2 is dur:
               [0, f, 1, 0.01.rrand(0.2), 2, 0.1 exprand: 1.5])
          };
          [0.1, 0.1, 0.1, 0.2].choose.wait;
     };
     }
}).play;
)
```

Synth parameter control from a GUI

Section "Providing external controls to a Synth"

Setting the lag parameter of a control

Mapping a control to a bus

(note for writing: point out in intro that this is useful for creating various shapes?)

Note: Distinguish between mapping to a bus and In.kr / In.ar ?

Synth Order

Effects of various control "shapes"

Control arrays

Interconnecting synths: Busses

Audio and Control Busses

Working with live sound input

Sound input from the computer hardware is accessible on the input busses of the server. These are the busses

Playing sound files

There are two ways for playing sound files:

1. Play while reading continuously from disk. This has the advantage that audio files of any length can be played, but playback is only possible at the original rate that the file was recorded and no jumps are allowed to different locations in the sound file after the start of playback. This method uses the UGen `DiskIn`. It also requires preloading an initial amount of the file into a buffer.

2. Load the entire file into a buffer on RAM. This has the advantage that you can move to any point in the file at any time and can change the speed and direction of playback, but the

disadvantage that the total duration of sounds that can be loaded is limited by the amount of RAM available to SuperCollider server. This method uses the UGen `PlayBuf`.

Playing samples from disk with DiskIn

```
b = Buffer.cueSoundFile(s, p, 0, 1);
```

Playing samples from RAM with Playbuf

Multichannel expansion

Examples to illustrate with diagrams:

1. Compare the two variants below to show the effect of .dup on Dust versus duplicating RLPF only with the freq parameter.

```
(
a = { arg freq = 440,  density = 1, rq = 0.05;
     var trigger;
     freq = [freq, freq * 1.25];
     trigger = Dust.kr(density, 0.01);
     RLPF.ar(GrayNoise.ar(Decay2.kr(trigger, 0.1)), freq, rq)
}.play;
)

(
a = { arg freq = 440,  density = 1, rq = 0.05;
     var trigger;
     freq = [freq, freq * 1.25];
     trigger = { Dust.kr(density, 0.01) }.dup;
     RLPF.ar(GrayNoise.ar(Decay2.kr(trigger, 0.1)), freq, rq)
}.play;
)
```

Explain dup, i.e [] i.e. { } ! 2 notation.
Show that `{ Dust.kr(density, 0.01) }.dup;`
is not the same as `Dust.kr(density, 0.01).dup;`

2. Another example, with variants. Comment!

```
b = {RLPF.ar(LFSaw.ar(300,0, 0.01), LFNoise0.kr([9,3], 600, 800),0.1)}.play;

c = {RLPF.ar(LFSaw.ar([250, 200],0, 0.01), LFNoise0.kr([9,3], 600,
800),0.1)}.play;

(
d = {
     RLPF.ar(LFSaw.ar(300,0, 0.01), LFNoise0.kr([9,3], 600, 800),0.1) +
```

```
    RLPF.ar(LFSaw.ar([250, 200],0, 0.01), LFNoise0.kr([9,3], 600, 800),0.1)
}.play
)

(
e = {
    RLPF.ar(LFSaw.ar(300,0, 0.01), LFNoise0.kr([9.01,3.01], 800, 840),0.1) +
    RLPF.ar(LFSaw.ar([250, 200],0, 0.01), LFNoise0.kr([9,3], 800, 840),0.1)
}.play
)

f = {RLPF.ar(LFSaw.ar([250, 200],0, 0.01), LFNoise0.kr([9,3], 600,
800).lag,0.1)}.play;
f = {RLPF.ar(LFSaw.ar([250, 200],0, 0.01), LFNoise0.kr([9,3], 800,
850).lag,0.1)}.play;
f = {RLPF.ar(LFSaw.ar([250, 200],0, 0.01), LFNoise0.kr([9,3], 800,
850),0.1)}.play;


f = {RLPF.ar(LFSaw.ar([250, 200],0, 0.01), LFNoise0.kr([9,3], 3000,
3000).lag,0.1)}.play;
f = {RLPF.ar(LFSaw.ar([250, 200],0, 0.01), LFNoise0.kr([9,3], 3000,
3000),0.1)}.play;
```

Timed event structures: Routines, Streams, Patterns

Scheduling events in time

Clocks, the system process and the application process. {}.defer

Routines

OK ...

Demand-UGen Streams

First explain Demand-UGens as a simple way to introduce the role of streams as generators for musical structures.
Show how Demand-UGens can be concatenated in series and nested. Give diagrams to show how streams work (how they produce their elements, how nesting works ?) The idea that a stream keeps producing the next element until nil. Why this is powerful? Because its simplicitly endows it with the capability to support nesting of different kinds of streams.

Demand-UGens as discrete value generators

Demand UGens are generators that run at control rate but generate a new value only when triggered ("at demand"). A value generating Demand-UGen is used as argument to the UGen

Demand and then triggering that UGen with a trigger that is control rate UGen that produces a signal with transitions from 0 to 1. Whenever there is a transition in the trigger signal from 0 to 1 the Demand UGen will produce a new value which it obtains from the value generator.

Patterns

Show that Patterns in the language have even more power than Demand-UGen streams because of the binding of multiple parameters in events. Explain environments - probably separately in the next section:

Events

(Must be introduced after PBind. PBind probably introduced in previous section). The idea of event stems from the realization that a musical event is a tuple of parameters. There are always many parameters involved in defining one sound event, some of them change, others stay constant, which change and which stay constant may differ from portion to portion - from phrase to phrase - of a piece.

Non-real time sound synthesis, event scores

Summary

- Before starting any sounds, make sure that the server is booted
- Sending the message 'play' to a function will attempt to make and play a synth from it
- Sending the message 'scope' to a function will attempt to play it and open a scope
- Synths are the basic synthesis processes in the server and are created from synthdefs
- Synthdefs are named definitions of synthesis algorithms that are stored in the server for immediate reuse
- Arguments in a synthdef function create inputs for controling its synths parameters
- Buffers can be used to load and play sound samples from files
- Envelopes can be used in synthdefs for synths that stop after a specific time interval
- The inputs and outputs of different synths can be connected via busses
- Routines or patterns can be used to define timed sound structures composed of many synths
- To stop all sounds and routines, type Command-.

Synthesis Techniques: Generation, Control and Transformation

Most synthesis and processing algorithms are made of combinations of components, each of which performs one of three basic functions: (1) *generation* of a signal, (2) *control* of a synthesis parameter or (3) *transformation* of a signal. This section introduces each of these types of components in turn and shows how they can be combined into increasingly complex algorithms. This can serve as a basis for learning how to program sound synthesis not just in SuperCollider but also in most other commercial as well as experimental software.

_____ *Following brief part of this section is still under development /under discussion / may or may not be included:*

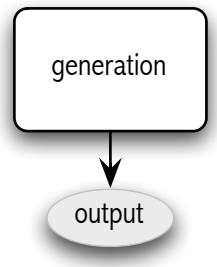
Probably here include brief reference to basic diagrams exemplifying the principles: Very brief - just max 3 sentences for each case: AND GIVE AT LEAST ONE SC EXAMPLE FOR EACH.

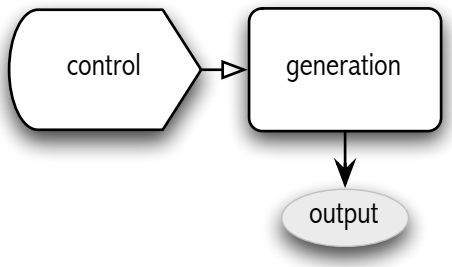
Maybe this will become one table / figure: "Configuration types of synthesis algorithms". Give a ***briefest possible code example*** for each of the 5 following types:

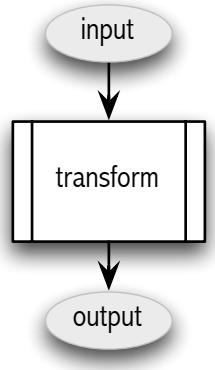
Possible begginning of the text introducing the table and its discussion:

To give a more concrete idea of how the 3 fundamental kinds of components work together, consider the following 5 prototypes or basic kinds of configurations:

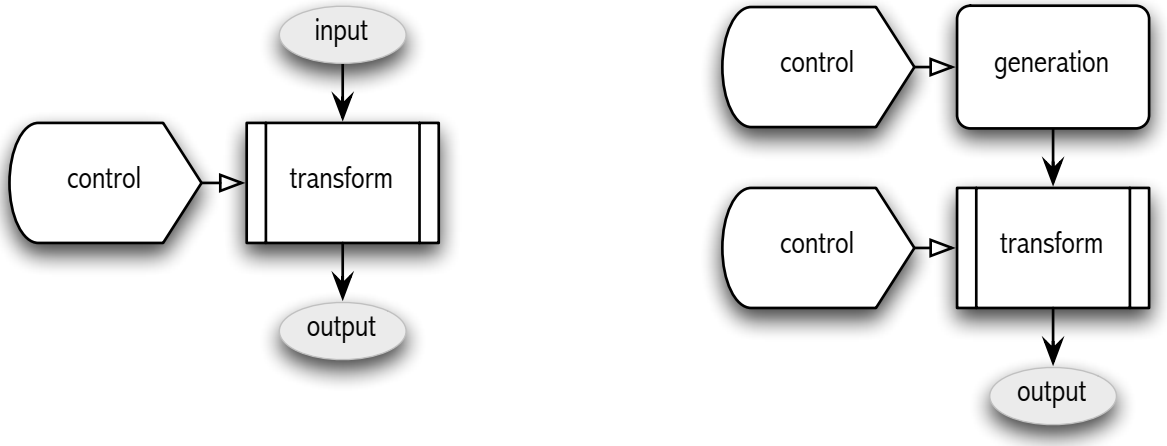
1. Generation alone (hypothetical): / 2. Control of Generation / 3. Transform alone (rare)







4. Controlled Transformation / 5. Controlled Generation and Transformation



Prototype 1, generation alone, is hardly ever encountered, although theoretically it can exist if the algorithm is complex or "interesting" enough. It is possible to regard the control aspect over time as incorporated in the algorithm itself. Such sounds are mostly useful as textures that can be combined with other sounds during a piece. Examples:

```
(/* 1. From an earlier section. "Alien subtropic drummers" */
{ RLPF.ar(LFTri.ar(Demand.kr(Impulse.kr([10, 5.1]), 0,
     Dser((200,300..1100).scramble, inf)), 0, 0.05),
          LFNoise0.kr([0.2,0.21], 300, 400), 0.01, 0.1) }.play;
)

( // 2. ("the winds whims")
{
     var src, filt, mod, aburbl, fburbl;

     mod = { LFNoise1.kr(    // fluctuate the amplitude
          // fluctuate the speed of the fluctuation of the amplitude
          LFNoise1.kr(0.005, 2, 2.001)
          // Add some intermittent extra fluctuation
               + Trig.kr(Dust.kr(0.1, 10), LFNoise1.kr(0.1, 5, 5.1)),
          // fluctuate the amplitude of the fluctuation of the amplitude
          LFNoise1.kr(0.1, 0.1)
     ); };     // this is a function making 2 independent copies later
     // "fluttery ornament", similar technique as with "mod" above
     aburbl = { LFPulse.kr(LFNoise1.kr(0.05, 10,11) // pulse -> "flutter"
          + Trig.kr(Dust.kr(0.25, 5), 2), 0,
          LFNoise1.kr(0.1 + Trig.kr(Dust.kr(0.25, 3), 5),
               0.8, 0.9),
          LFNoise1.kr(0.5, 0.1, 0.1001)) * Trig.kr(Dust.kr(0.1), 5);
     }; // second function, to be combined with mod
     // source:            double copies for 2 independent voices, L-R
     src = WhiteNoise.ar(({ mod.value + aburbl.value } ! 2) * 0.5);
     fburbl = LFPulse.kr(LFNoise1.kr(0.05, 10,15)
          + Trig.kr(Dust.kr(0.25, 5), 2), 0,
          LFNoise1.kr(0.1 + Trig.kr(Dust.kr(0.25, 3), 5),
               0.5, 0.7),
          LFNoise1.kr(0.5, 500.0)) * Trig.kr(Dust.kr(0.25), 5);
```

```
    // Filter source with fluctuating frequency and amplitude:
    filt = Ringz.ar(
        src,
        // fluctuate frequency with same curve as amplitude,
        // but scaled for frequency:
        (mod * 1200) + 1250 + fburbl,    // ornament
        // fluctuate the ring time
        LFNoise1.kr([0.05, 0.06], 0.1, 0.101),
        0.2
    );
    Out.ar(0, filt);   // final output
}.play;
)

( /* 3.
   From the Gendy1 help file, with "smoothed" LFNoise0 stubstituted
   for the MouseY and MouseX controls.
   ("Another traffic moment")
   This is a rough texture!
  */
{
var n;
n=10;     // try also n=20 and n=30
Resonz.ar(
Mix.fill(n,{
var freq, numcps;
freq= rrand(50,560.3);
numcps= rrand(2,20);
Pan2.ar(Gendy1.ar(6.rand,6.rand,1.0.rand,1.0.rand,freq,freq,1.0.rand,
1.0.rand, numcps, SinOsc.kr(exprand(0.02,0.2), 0, numcps/2, numcps/2), 0.5/
(n.sqrt)), 1.0.rand2)
})
, Lag.kr(LFNoise0.kr(0.1, 2000, 2100), 1.5),
  Lag.kr(LFNoise0.kr(0.21, 1.0, 1.01), 1.5)
);
}.play
)
```

Prototype 2, control of generation parameters, is often encountered in simple synth examples. However, it is seldom used just by itself in the final form of a performance. Most often, some kind of transformation is applied to the output of the synths in the form of one or more "effects" (see 5 below).

....

Prototype 3, transformation of a signal, is frequently encountered as as a type of "effect" ("fx") that is applied to part or all of the sounds produced throughout the duration of a piece. This is seldom applied without any small adjustments during the piece; thus the actual

Prototype 5 ...: (... is he most general and most common of cases ...) ... Most commercial synthesizer modules, whether hardware or software contain some variant of this prototype, extended and elaborated through multiple, configurable chains of control - generation and transformation in parallel and/or in series.

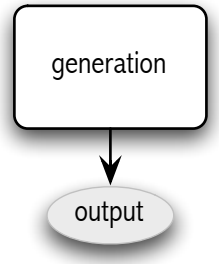
Example ...
... Here the parameter x of generator ... is controlled by ... and the parameter y of transform ... is controlled by ...

NOTE: REFERENCE MUST BE MADE IN THIS CHAPTER AT APPROPRIATE PLACES TO THE BORDERLINE CASES BETWEEN GENERATION AND TRANSFORMATION (WAVETABLES/ GRAINS) AS WELL AS BETWEEN CONTROL AND TRANSFORMATION (SHAPER/WAVESHAPING, FEEDBACK/AM, FM, PM)

Generation

Generation is the process of calculating and outputting, sample by sample, an audio or control signal, on the basis of some algorithm.

The simplest case: A single static generator



Example:
{ SinOsc.ar(400, 0, 0.1) }.play;

Create a synth with a single sine generator and send its output to

Periodic Oscillators

Periodic oscillators are generators that create a waveform which repeats its shape after a regular time interval - the period. The both the shape of the waveform and the period can be varied while the generator is running. The period is controlled by the parameter freq (frequency, that is 1/ period). The variation of the shape is limited by the kind of the generator, and is controlled by additional parameters, depending on the type of the oscillator (mul, add , phase, kwidth i.e. pulse width ratio etc.). Wavetables are also periodic oscillators but they are treated here in a separate category because they have many distinct features and techniques, due to the fact that the shape of their waveform is completely data-driven.

Aperiodic Oscillators

Wavetables

Can be

In this sense (Note: Maybe sampling and granulation can be considered data-driven generation as distinct from algorithm-driven generation - see possibly this distinction in classifications by Serra, or by classifications reviewed in Välimäki et al. book: Evaluation of sound synthesis techniques.)

[Note : This can be considered as a borderline case between generation and transformation. Comment ...]

Experimenting with wavetable shapes

```
Wavetable.sineFill(1024, 1 ! 1, [pi]).asSignal.plot;
Wavetable.sineFill(1024, 1 ! 2, pi ! 2).asSignal.plot;
Wavetable.sineFill(1024, 1 ! 3, pi ! 3).asSignal.plot;
Wavetable.sineFill(1024, 1 ! 4, pi ! 4).asSignal.plot;
Wavetable.sineFill(1024, 1 ! 5, pi ! 5).asSignal.plot;
Wavetable.sineFill(1024, 1 ! 6, pi ! 6).asSignal.plot;
Wavetable.sineFill(1024, 1 ! 7, pi ! 7).asSignal.plot;
Wavetable.sineFill(1024, 1 ! 8, pi ! 8).asSignal.plot;

Wavetable.sineFill(1024, (1..2).reciprocal, pi ! 2).asSignal.plot;
Wavetable.sineFill(1024, (1..3).reciprocal, pi ! 3).asSignal.plot;
Wavetable.sineFill(1024, (1..4).reciprocal, pi ! 4).asSignal.plot;
Wavetable.sineFill(1024, (1..5).reciprocal, pi ! 5).asSignal.plot;
Wavetable.sineFill(1024, (1..6).reciprocal, pi ! 6).asSignal.plot;
Wavetable.sineFill(1024, (1..7).reciprocal, pi ! 7).asSignal.plot;
Wavetable.sineFill(1024, (1..8).reciprocal, pi ! 8).asSignal.plot;

(
g = { | pat, phasepat |
     var table;
     pat = pat.asStream;
     pat = pat.next.value(_) ! 64;
     pat.postln;
     phasepat = (phasepat ? pi).asStream;
     phasepat = phasepat.next.value(_) ! 64;
     phasepat.postln;
     table = Wavetable.sineFill(1024, pat, phasepat); // Array.rand(64, 0,
pi)
     table.asSignal.plot;
     table;
}
)

g.(Pwhite(0.1, 0.9, inf), Pwhite(0.0, 2* pi, inf));
```

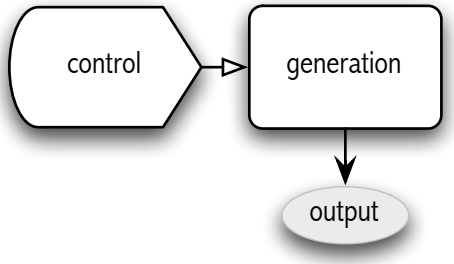
Combining several generators in parallel

+
*
Mix
Klang
Wavetables
Chorusing oscillators (COsc), Multiple oscillators:

Control

Control is the customization or continuous modification of some synthesis process by changing the values of parameters on which the synthesis process depends.

The sources of control signals



Control of unit generators through parameters

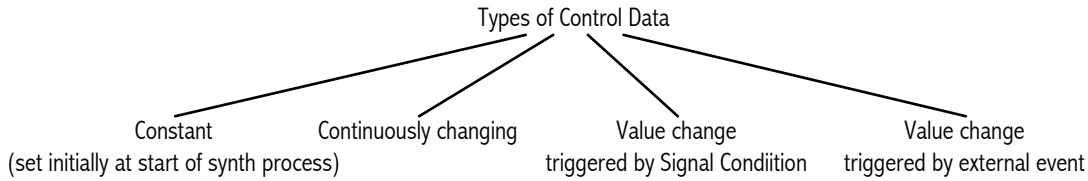
(this is an introductory section to the chapter. Give basic examples)

The mul and add parameters

Scaling, mapping -> Transformation ugens

Types of Control Data

(To be discussed. Give an overview. ***Refer to chapters/sections with more info on specific types.*** *Probably improve following drawing)*:



Identify the basic SC programming techniques for each type of control data. Give (interesting, several!) examples for each.

Control data initialized at the start of a synthesis process

Continuously generated control signals

Control signal changes triggered by signal conditions

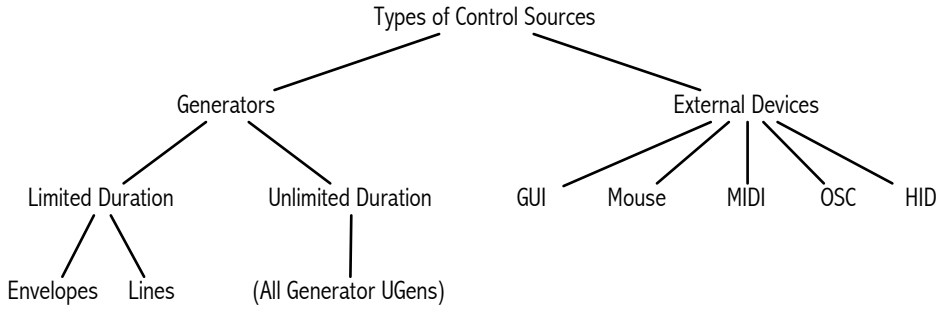
Explain the principles of function and give examples of the basic kinds of techniques.

Explain impulse, dust, trig, gate, ... not clear how much must be here and how much just forward references to the 2 independent sections dealing with this topic later on:
- 4.2.6 "Feature Detection" of the present section *Control*)
- 4.5 "Discrete generation of control data"

Control signal changes triggered by external events

Types of input sources for control signals

(To be discussed. Give an overview. Refer to chapters/sections with more info on specific types. Probably improve following drawing):



Envelope techniques

Line techniques

Oscillators as control sources

GUI widgets as control source

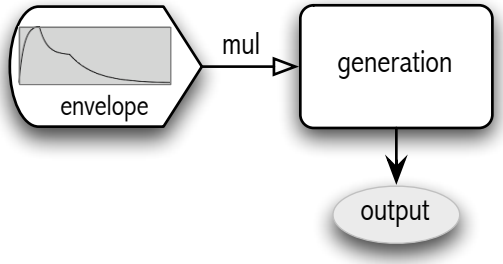
The mouse as control source

MIDI input as control source

OSC commands as control source

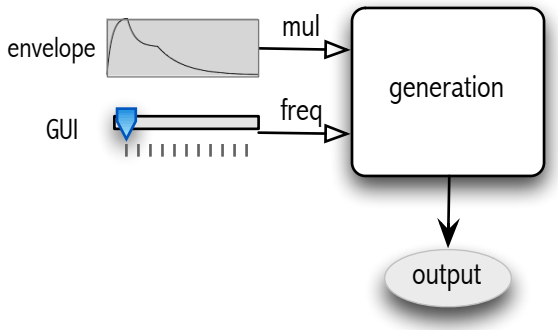
HID input as control source

Basic case: Envelopes as predefined control shapes

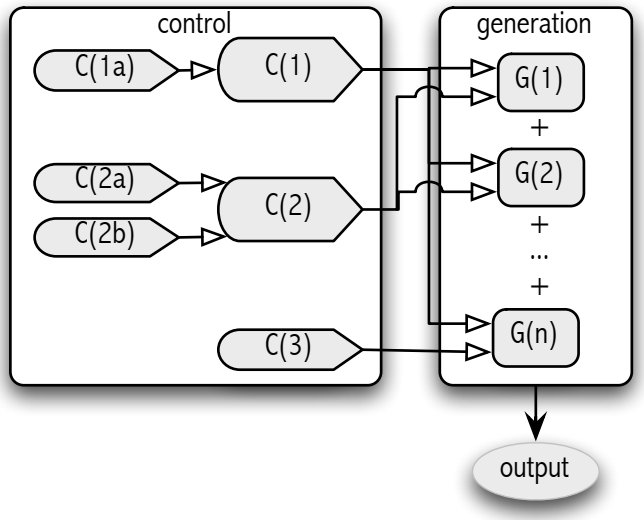


Give basic example. Refer to detailed treatment of envelope techniques in separate section "Envelope techniques" under 4.2.3 Types of input sources for control signals.

Controlling multiple parameters of one generator in parallel



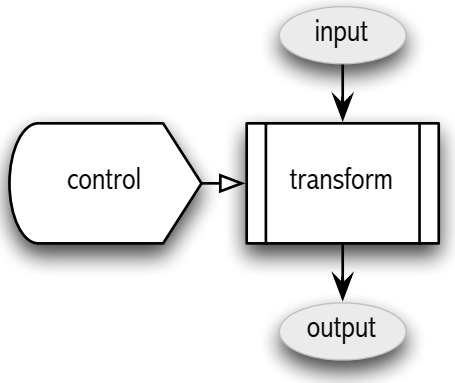
Multiple, complex control and generator configurations



Feature Detection

Transformation

Transformation components are those that accept one kind of signal as input ... In some cases



Filters

Delays [: Echo and reverb]

Waveshaping [: Borderline between control, generation and transformation]

Sampling

(could be considered data-driven generation?)

Granulation

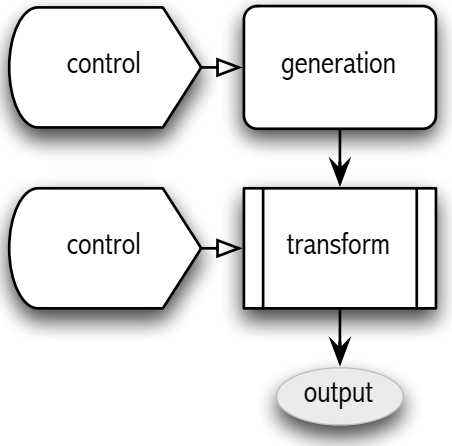
(could be considered also borderline between generation and transformation?)

Spectral transformations

Multichannel, spatialization

Feedback: Control as input to transformation

Combining generation, control and transformation



(Complex configurations ...)

Implementing "traditional" effects

Show the implementation of "traditional" effects such as flanger, wah-wah, echo, reverb, tremolo, phaser, vibrato, chorus, distortion, vocoder, delay (some of these have UGens with analogous names)

Discrete [Event-based?] generation of control data

GUI based

MIDI based

Triggers (should they be here???)

Routines

Streams and demand-rate generators

Patterns (should they be here???)

Here an intro do the principles of patterns and to the most common used kinds of patterns - but maybe put this section as first section of chapter 8: "Sequencing with patterns"

Graphical User Interface

Windows

SCWindow.new and its arguments
- bringing the window to the front
- Note that bounds are calculated from left bottom
- Moving the window around by setting the bounds
- Setting the color (this is a property of ***SCView***!)
- The top view and its children
- The onClose variable

Slider

Label

Number box

Menu

Multislider

Envelope view

Signal view

Drag Sinks and Sources

File Dialog Windows

(Explain about asynchronous return!)

Communication: MIDI, OSC, HID

MIDI

Essentials of MIDI communication in SuperCollider

Programming MIDI communication in SuperCollider involves three stages:

1. **Initialize**: Find out what devices exist on the system. This must be done once only at the beginning of a setup, that is, when the MIDI devices are first connected to the system, or whenever there is some change in the number or connection configuration of devices.

2. **Connect to specific devices**: For MIDI input, activate (or deactivate) a connection to a device for input. For MIDI output, create an object that can sends MIDI to a specific device. This is also done once only to establish or close a connection to a specific device, it is not done when actually sending or receiving MIDI commands.

3. **Handle or send MIDI messages**: For input, specify what to do when receiving a specific kind of MIDI message (once only for each specification of a kind of action); for output, send a message to a device (must be done individually for each MIDI message that is sent).

Once all 3 steps above have been completed, the system will automatically call the functions that were stored in the corresponding variables of MIDIin, whenever a MIDI command is received. The class MIDIResponder provides a convenient way to bind actions to be performed for specific midi-in keys or controller numbers and it is recommended to use this class because of the flexibility that it affords.

Initialization of the MIDIClient

To recognize all devices that are currently set up in SuperCollider, send "init" to MIDIClient:

```
MIDIClient.init
```

Do this once only before starting any further setup of MIDI connections. If the MIDI configuration changes (MIDI hardware or software devices are added to or removed from the system), then it is necessary to perform MIDIClient.init again and to reinitialize the input and output connections to MIDI ports.

 **DO NOT SKIP THE NEXT SECTION**

The structure of MIDI connections: MIDIEndPoints = Ports of Devices

The class variable `sources` of MIDIClient holds all ports of all devices from which SuperCollider can receive MIDI, in the form of instances of `MIDIEndPoint`. In the same way, the class variable `destinations` holds all those ports to which SuperCollider can send MIDI. Each MIDIEndPoint corresponds to one port of one device as it can be seen in the info on MIDI devices given by the Audio MIDI Setup Utility (on MacOS X). The variable `device` of the MIDIEndPoint holds the name of the device, the variable `name` holds the name of the port, and the variable `uid` holds a numerical ID that is used by SuperCollider internally to connect to the port in question. The correspondences are thus:

SuperCollider:	**Audio MIDI Setup Utility (MacOS X):**
`MIDIEndPoint` instance	Port of a Device
variable `device`	Device Name
variable `name`	Port (name)
variable `uid`	(used internally for communication with the O.S.)

If you use more than one device, or if your device has many ports, it is essential that you familiarize yourself with the ports of your device and with the MIDI setup utility panel of your computer, and that you understand the above correspondence chart, in order to be able to address the desired device on the desired port.

All MIDI communication is based on the MIDIEndPoints stored in MIDIClient's sources and destinations. These are accessed by numerical index, in the order in which they are stored by the system inside these variables:

```
MIDIClient.sources[0]; // obtain the first input port
MIDIClient.destinations[1]; // obtain the second output port
```

MIDIIn uses these MIDIEndPoints implicitly on the base of the index provided to argument destination of messages connect and disconnect. MIDIOut needs to be passed the uids of the ports explicitly. The next sections show how to use MIDIn and MIDIOut. For more details about the correspondence of MIDI devices and ports to MIDIEndPoints see section *MIDI connections in SuperCollider*.

Receiving MIDI.

Configuring MIDI input involves two stages: (a) Activating or deactivating a connection, and (b) specifying what to do when a MIDI message is received.

(a) Activating and deactivating MIDI input connections

The ports of the devices that send MIDI to SuperCollider are stored in an array in `MIDIClient.sources`. To start receiving MIDI from the first port of the first device in that array, activate the first MIDIEndPoint of MIDIClient.sources by sending the message connect to MIDIIn, with device argument value 0:

```
// Activate input from the first port-device in sources:
MIDIIn.connect(device: 0);
```

Once this is done, all MIDI messages sent to the computer on that port will be received by SuperCollider and will trigger the actions stored in MIDIIn, as explained in the following section (b). If you want SuperCollider to stop receiving MIDI from some MIDIEndPoint x, send message `disconnect` to MIDIIn:

```
// Deactivate input from port-device with index x in sources:
MIDIIn.disconnect(device: x);
```

(b) Specifying actions that will be performed when a MIDI message is received

To specify what SuperCollider must do when it receives a MIDI message of a certain kind, store some action in the corresponding variable of MIDIIn:

```
/* Upon receipt of a MIDI note-on message, play a short sound with
    corresponding frequency and amplitude: */
(
MIDIIn.noteOn = { | src, chan, key, veloc |
     var freq, amp;
     freq = key.midicps;     // convert MIDI key to frequency
     amp = veloc / 1270;     // scale: max. veloc 127 -> amp 0.1
     { SinOsc.ar(freq, 0, EnvGen.kr(Env.perc,
          levelScale: amp, doneAction: 2));
     }.play;
};
)

/* Upon receipt of a MIDI control message, post the names
of the device and port from which it was received as well as
the channel number, the controller number, and the control value: */

(
// create a dictionary for accessing the MIDIEndPoints by their uid:
var sourceDict;
sourceDict = IdentityDictionary.new;
MIDIClient.sources.do { |s| sourceDict[s.uid] = s; };

// Store action in MIDIIn.control that find the device and posts details
MIDIIn.control = { | src, chan, num, val |
     var device;
     device = sourceDict[src];   // find the device from its uid
                                 // post all the info
     Post << "Device: " <<< device.device << " port: "
          <<< device.name << " channel: " <<
          chan << " number: " << num << " value: " << val << "\r";
}
)
```

Sending MIDI

Like MIDI input, setting up MIDI output also involves 2 stages: First create a MIDIOut instance that will be used to send MIDI messages to a specific port-device. Then use that instance to send any number of MIDI messages of any kind at any moment.

The device where MIDIOut will send its messages must be specified by its uid number! Therefore, to create a MIDIOut, use

```
MIDIOut(0, MIDIClient.destinations[0].uid);
```

Not:

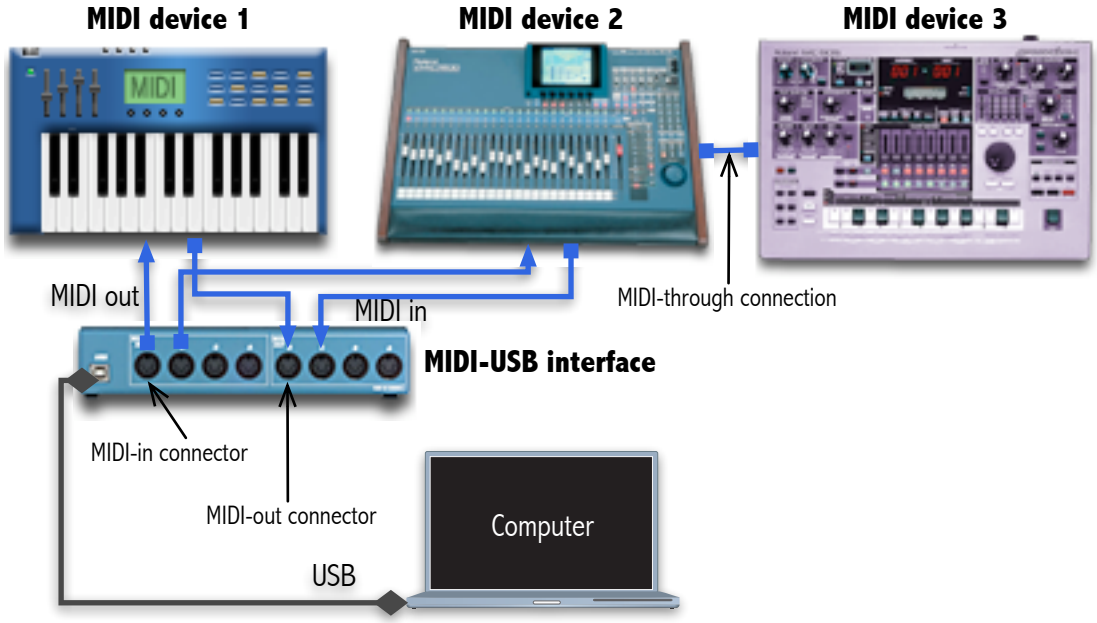
```
MIDIOut(0, MIDIClient.destinations[0]);
```

```
/* 1. Create a MIDIOut instance for sending MIDI
    to the first port-device in MIDIClient.destinations: */

m = MIDIOut(0, MIDIClient.destinations[0].uid); // use uid!!!

/* 2. Use that instance to send some MIDI messages to the specific port */

Routine({
     var note;
     100.do {
          // send a note on message on channel 0 (MIDI channel 1)
          m.noteon(0, note = 50 rrand: 90, 40 rrand: 100);
          0.1.rrand(0.5).wait;    // wait a little
          // send a control 33 message (modulation wheel) on channel 0
          m.control(0, 33, 127.rand);
          0.1.rrand(0.5).wait;    // wait a little
          // send a note off message on channel 0 (MIDI channel 1)
          m.noteon(0, note, 0);
     }

}
```

MIDI devices

Note: This section and the following one explain the concepts hat are involved in connecting MIDI devices together: Device, channel, port and connector. Readers familiar with these concepts may skip to section *Setting up MIDI connections in SuperCollider*.

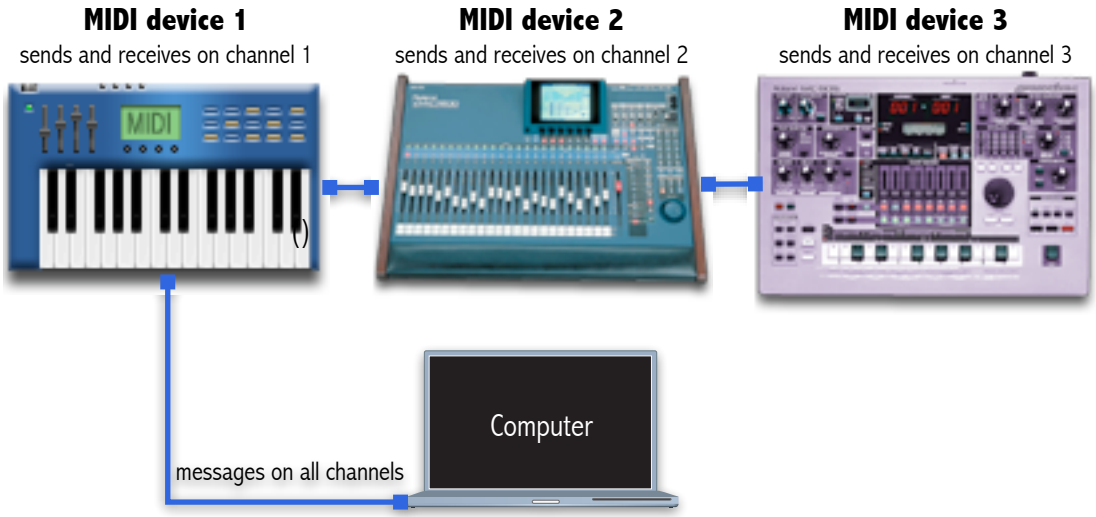
MIDI is a communication protocol between musical devices. MIDI keyboards or other MIDI controller hardware constitute hardware devices whereas software applications are software devices. MIDI devices are connected to a computer either directly via a USB or Firewire output available on the device itself or indirectly via a MIDI-USB interface. Following figure shows 3 devices connected to a computer via a MIDI-USB interface. Device 3 is connected to the interface indirectly, by chaining it via the "MIDI through" connector of device 2:



It would be logical to expect that a computer should recognize each device that is connected to it and be able to distinguish which messages arrive from which device or to send messages selectively to a specific device, addressing it by name or by number. However, as needs of studios grew beyond the capacity originally provided by MIDI for addressing individual devices, new concepts and techniques were added to support more devices. As a result, the current state of the technology involves several different concepts that one must deal with in order to configure MIDI connections in a computer. The next section will explain the differences between these concepts. These are important, because they are reflected in the way in which SuperCollider implements MIDI. They are indispensable if one wants to connect not just one but several different devices with SuperCollider, whether these devices be software applications or actual external hardware components.

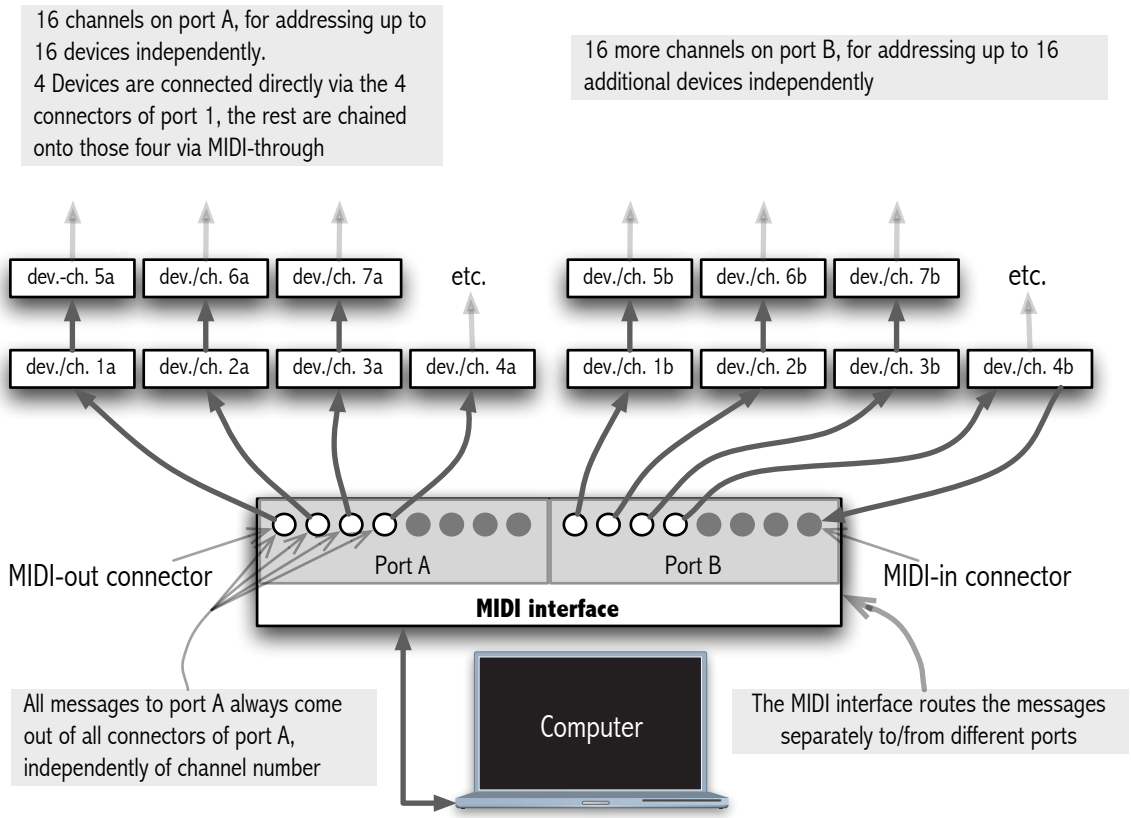
Channels, ports, connectors

The MIDI protocol standard defines 16 channels as a way to distinguish between devices. This standard was originally conceived so that each device would send and receive all its commands on one channel. The number of that channel is contained in every single MIDI command as part of the message, in the format that is prescribed by the MIDI protocol standard. For example, if MIDI device 1 is programmed to receive commands on channel 1, it will only react to those commands that bear the tag identifying them as belonging to channel 1 and it will ignore any commands that belong to channel 2, 3 etc. Thus, it is possible to connect up to 16 devices together in a chain, where each device will pass on any commands received from another device, but will only react to those commands that belong to its own assigned channel:



Addressing devices based on MIDI channels alone

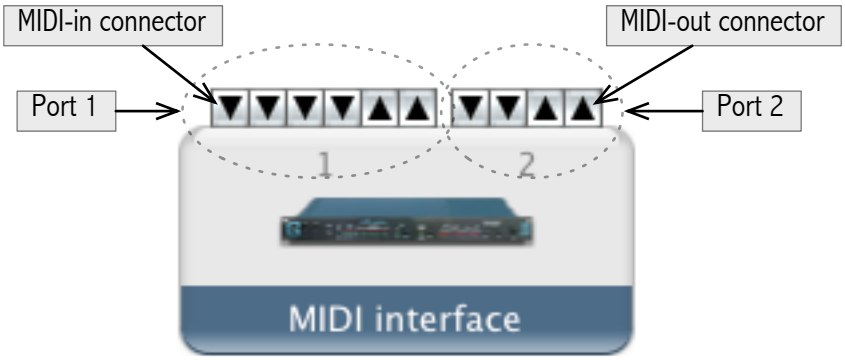
Very soon, MIDI synthesizers became capable of emulating several different synthesizers in one hardware "box", so one device would require several channels, each channel addressing a different type of synthesizer within the same device. Thus, studios with many devices would need more than 16 channels. In order to enable the addressing of more than 16 devices, a new virtual addressing concept was added, called port. Port is an additional "identification tag" for a MIDI message besides the channel number. Depending on its hardware or software properties, a device may be able to distinguish 1, 2 or more ports. If, for example a device has 2 ports, it can address individually 32 devices: 16 devices for 16 channels on port 1 and 16 devices on 16 additional channels on port 2. Following figure shows how up to 32 devices can be connected to a MIDI interface that has two ports. That interface is capable of separating the messages that are sent to it from the computer so that all message tagged as belonging to port A are sent out of the MIDI-out connectors of that port, and those of port B are sent out of the connectors of port B. So now, up to 16 independently controllable devices can be connected to each port, each addressed on a separate MIDI channel. Note that in this example, each port has only 4 MIDI-out connectors, and each connector sends messages for all 16 channels on that port. It does not matter into which connector a device is plugged, it will always receive all messages sent to the port to which the connector belongs, independently of their channel number. So a connector is a physical connection for plugging a cable in, it is independent of channel number, and it is assigned to a specific port.



Expanding the address space of MIDI by introducing ports

The port is not part of the of the original MIDI protocol standard. The information about which port is passed on is tagged "on top" of each MIDI message, in other words, it is contained in some extra bits that are not part of the MIDI message itself but that are attached to the message. Not all devices decode this extra information, but MIDI interfaces that are conceived as hubs for many devices are designed to provide it to computers so that one computer may manage many devices at once.

As was shown in the above example, *ports* are not the same thing as *connectors*. A port is an additional piece of information passed along with a MIDI message. A connector is an actual hardware receptacle for connecting a device to another one by plugging in one end of a MIDI cable. Most modern devices come with drivers that can inform the operating system of a computer about the actual configuration of ports and connectors of the hardware device. For instance, after installing the driver of a MIDI to USB interface device *y* on an Apple Macintosh and scanning the MIDI connections with the *Audio-MIDI Setup* utility of MacOs X, the interface *y* will show up on the configuration window of the utility with its actual ports and connectors. Following example shows a hypothetical interface device *y* with two ports, where port 1 has 4 MIDI-in connectors for receiving messages and 2 MIDI-out connectors for sending messages, while port 2 has 2 MIDI-in and 2 MIDI-out connectors.



Ports and connectors on the panel of a MIDI device in the MIDI Setup utility of Mac OS X

Any MIDI devices connected to the MIDI-in connectors of port 1 will be received by the computer with a tag showing that they come from port 1, while those connected to the in-connectors of port 2 will be tagged with port 2 respectively. Conversely, messages sent from the computer addressed to port 1 will only be sent out via the MIDI-out connectors of port 1, while messages sent to port 2 will come out of the connectors of port 2, on the actual interface hardware.

Note: Unfortunately, the terminology employed by various manufactures is not consistent: The software synthesizer *Reason* names the different ports "busses". On the other hand, *Max/MSP* calls them "Input Devices" and "Output Devices" in the MIDI Setup dialog window, but the documentation on the Max help patch window "inport.help" uses *device* and *port* as interchangeable, equivalent concepts. Therefore, after having explained the concepts above as far as was possible based on the information given by SuperCollider and MacOS X MIDI Setup utility, the remaining details of interconnecting specific devices must be left to the user with the help of documentation from the manufacturer.

MIDI connections in SuperCollider

Connections to MIDI devices and their ports are realized through the class `MIDIClient`. This class reads the current MIDI input and output ports from the operating system when it is sent the message init:

```
// Call this at the beginning of a session or whenever the MIDI setup
MIDIClient.init // of your computer changes, to update the configuration
```

The MIDIClient class variable `sources` stores all ports of all devices that can send MIDI to SuperCollider, while its class variable `destinations` holds all ports of those devices to which SuperCollider can send MIDI. Each port of each device is contained in a separate instance of `MIDIEndPoint` which holds the name of the device, the name of the port, and a unique identity number (`uid`). The names of the device and of the port are the same as those used by the *Audio MIDI Setup* utility (located in folder /Applications/Utilities on MacOS X). One may post all details of all input ports with this code:

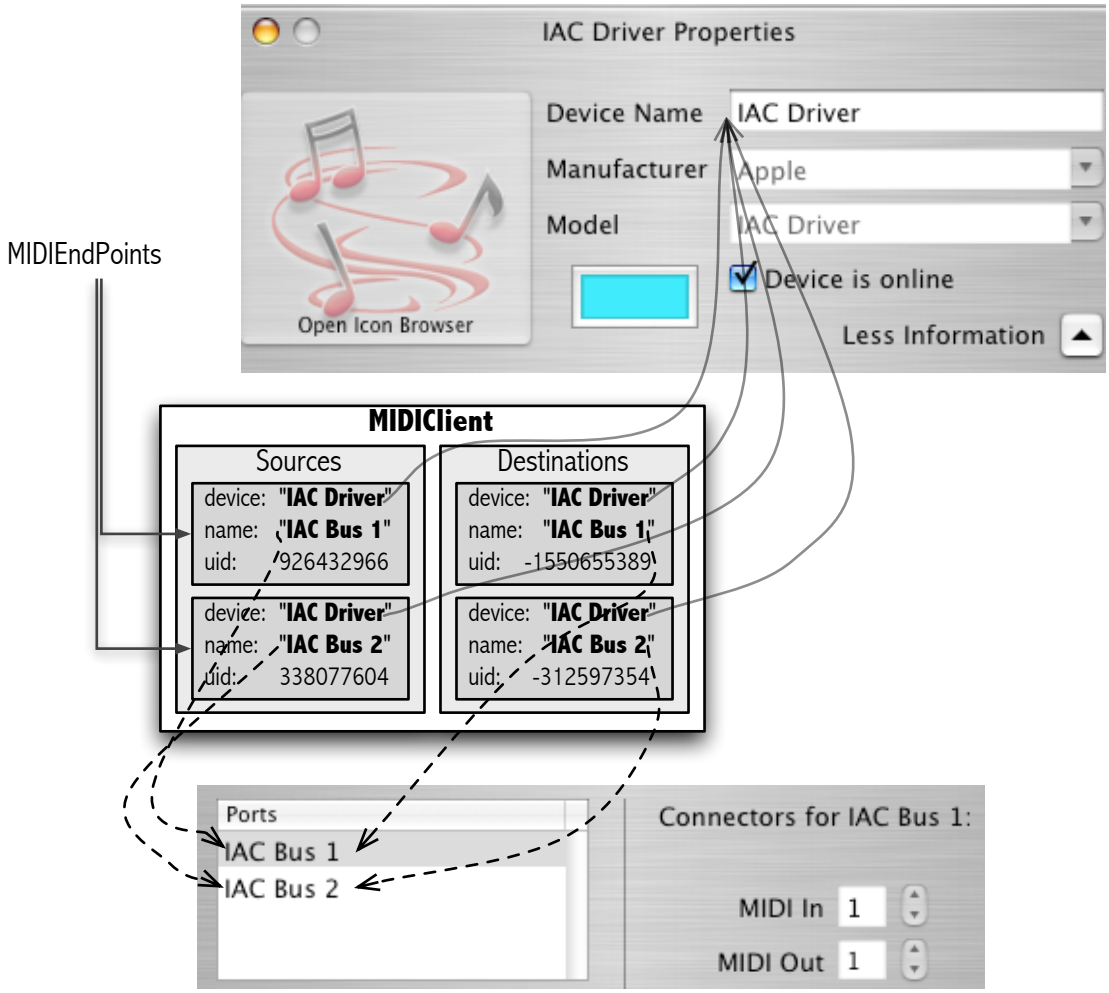
```
(
MIDIClient.sources.do {|s, count|
     Post << "(" << count << ") device: " <<< s.device << " name: "
          <<< s.name << " uid: " << s.uid << "\r";
}
)
```

If only the IAC Driver device is activated, then these will be:

```
(0) device: "IAC Driver" name: "IAC Bus 1" uid: 926432966
(1) device: "IAC Driver" name: "IAC Bus 2" uid: 338077604
```

- ☛ **All known MIDI ports and devices are stored as MIDIEndPoints in MIDIClient sources and destinations.**
- ☛ **Each MIDIEndpoint represents one port on one device.**
- ☛ **In a MIDIEndPoint, the name of the device is stored in varialbe "device" and the name of the port is stored in the variable "name" (!!!).**
- ☛ **The variable "uid" (numeric ID) of a MIDIEndPoint is used to create a connection to the port of the device.**
- ☛ **If in doubt about the identity of a MIDIEndPoint compare the names of the device and port with those shown for the devices in Audio MIDI Setup (MacOS X) or other operating system setup utility.**

It may be misleading that a `MIDIEndPoint` contains the variables "`device`" and "`name`" but not "*device*" and "*port*". In reality, the string stored in variable `device` corresponds to the name of a MIDI device and the string stored under "`name`" corresponds to the name of the port in the MIDI Setup utility. To clear all doubts, following figure shows the relevant parts of a MIDI Setup panel for the IAC Driver device (MacOS X) and the contents of the MIDIClients sources and destinations variables. One sees that the value of the `device` variable of the MIDIEndPoints corresponds to the field "Device Name" (in this case: IAC Driver) and the `name` variable corresponds to the values of the field "Ports" (in this case: IAC Bus 1 and IAC Bus 2):



Example 1: Communication with software via the IAC Driver

Following test can be performed on MacOS X even without any external MIDI devices connected. It uses the IAC driver, which applications can use to send MIDI messages to each other through the operating system within the same computer.

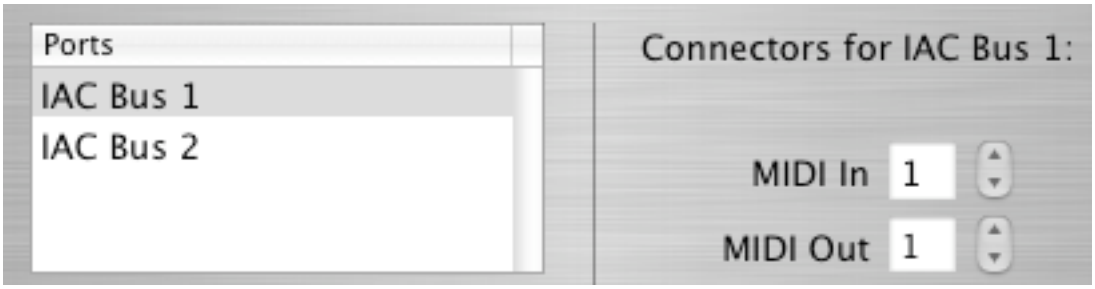
Open the utility Audio MIDI Setup (located in folder /Applications/Utilities), and select the tab *MIDI Devices*. If no MIDI device has ever been installed and the IAC driver has not been activated, then the window will only show the IAC driver in its inactive state thus:

First perform `MIDIClient.init` with the IAC Driver inactive. This should post the following:

```
WARNING:
MIDIClient-init requested 1 inport(s) and 1 outport(s),
but found only 0 inport(s) and 0 outport(s).
Some expected MIDI devices may not be available.
Sources: [  ]
Destinations: [  ]
MIDIClient
```

The message from MIDIClient shows that the arrays that store the sources and destinations are empty (`[ ]`) which means that there are no MIDI devices to receive from or send to at all. Now activate the IAC driver: Double click on its icon in the MIDI Setup utility window. This brings up the set up window "IAC Driver Properties". Click on the checkbox "Device is Online" to activate the driver.

...



☛ For any devices to be known to the system, the drivers of these devices must have been installed and activated (enabled).

☛ Any time that a new device is installed, MIDIClient.init must be evaluated again to update the list of devices known to SuperCollider.

Evaluate `MIDIClient.init` again to read the new connections created by the operating system. This time, MIDIClient prints out the names of these connections:

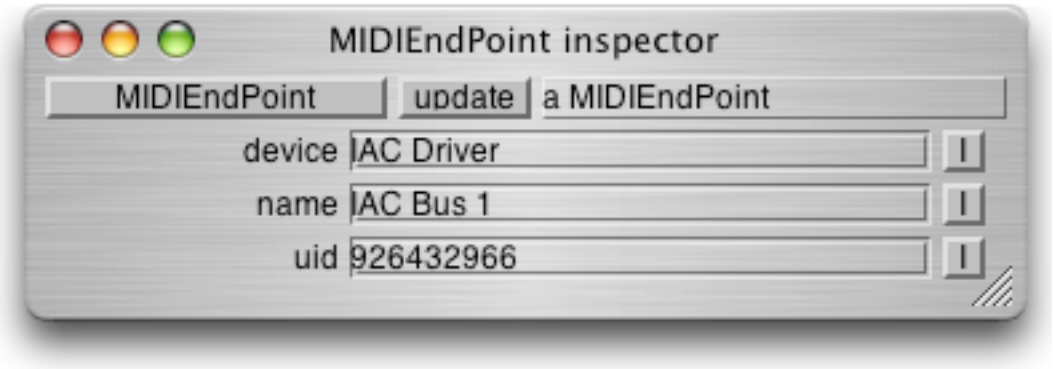
```
Sources: [ IAC Driver : IAC Bus 1, IAC Driver : IAC Bus 2 ]
Destinations: [ IAC Bus 1 : IAC Bus 1, IAC Bus 2 : IAC Bus 2 ]
```

Both the sources and the destinations are instances of `MIDIEndPoint`. Lets look at the sources:

```
MIDIClient.sources;     // post the array of current MIDI sources
[ a MIDIEndPoint, a MIDIEndPoint ]  // array contains 2 end points
```

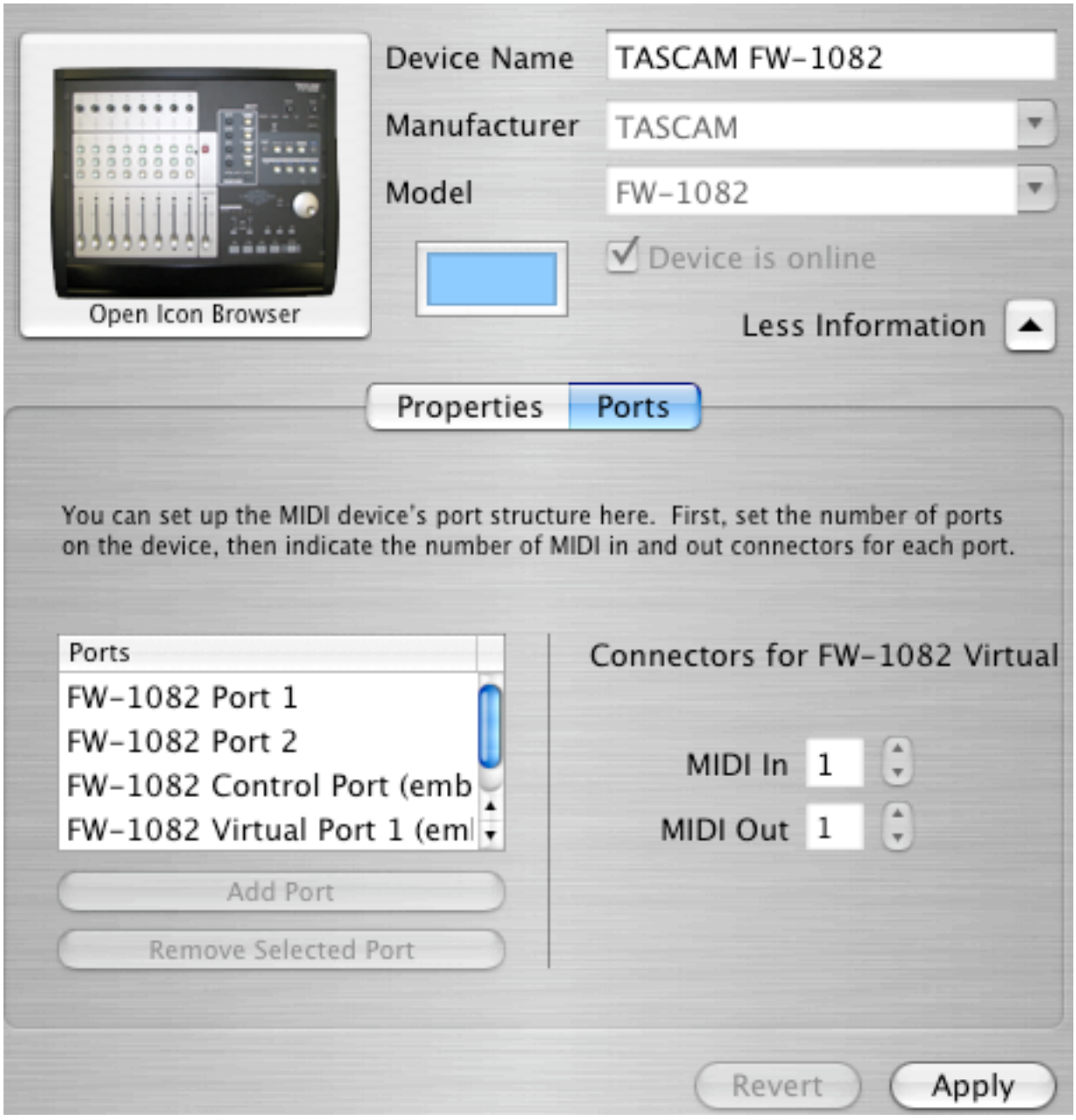
A `MIDIEndPoint` corresponds to one port on one device. It holds the name of the hardware or software device, the name of the port, and a unique identity number (`"uid"`) for addressing that port. As a concrete example, inspect the first instance of MIDIClient's sources:

```
MIDIClient.sources.first.inspect; // show instance on GUI window:
```



Example 2: Communication with a hardware device (TASCAM FW -1082/1884)

The TASCAM FW-1082 and 1884 are two similar multichannel audio interfaces that connect to the computer via Firewire and double as MIDI-Controllers as well as multi-port MIDI-Interfaces.




TASCAM FW-1082
F
F

File I/O

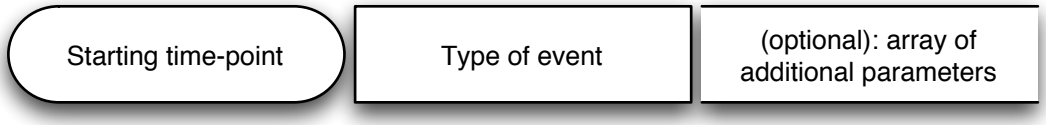
Scores and Patterns

Scores

In computer music, a score consists a list of sound events occurring at specified points in time. A sound event is a process that generates or processes sound, starting at a specific point in time and lasting for a specific time interval. For example a sound event could be a percussive "note" played for 1 second:

```
{ WhiteNoise.ar(EnvGen.kr(Env.perc(0.01, 0.99, 0.1), doneAction: 2))
}.play;
```

Thus, a sound event in a score must specify at least 2 things: the time point at which the event occurs and the type of the event. An event may additionally be contain parameters that further determine its characteristics, such as duration, frequency etc. In most cases an event type contains predefined defaults for every parameter that are used when no values for that parameter are provided. An event definition could therefore be schematically represented as follows:



The starting time point is mostly understood as a time interval in relation to a previous event, which may be either the previous sound event played, or the beginning of the "piece" (the music performance in question). A time interval of 0 indicates

Patterns - Streams - Events

In SuperCollider, the three classes: Pattern, Stream and Event form a system for defining and playing sound-generating algorithms that freely combine sequential techniques with interactive or rule-based techniques. This works as follows:

- A pattern specifies an algothm that generates events. The pattern does not generate the events itself, instead it is used to create any number of copies of the algorithm, called streams.

Patterns as interactive scores

```
Pattern.allSubclasses.size;
93
```

Why learn to use a library of 93 Pattern classes when there are graphic MIDI track editors? There

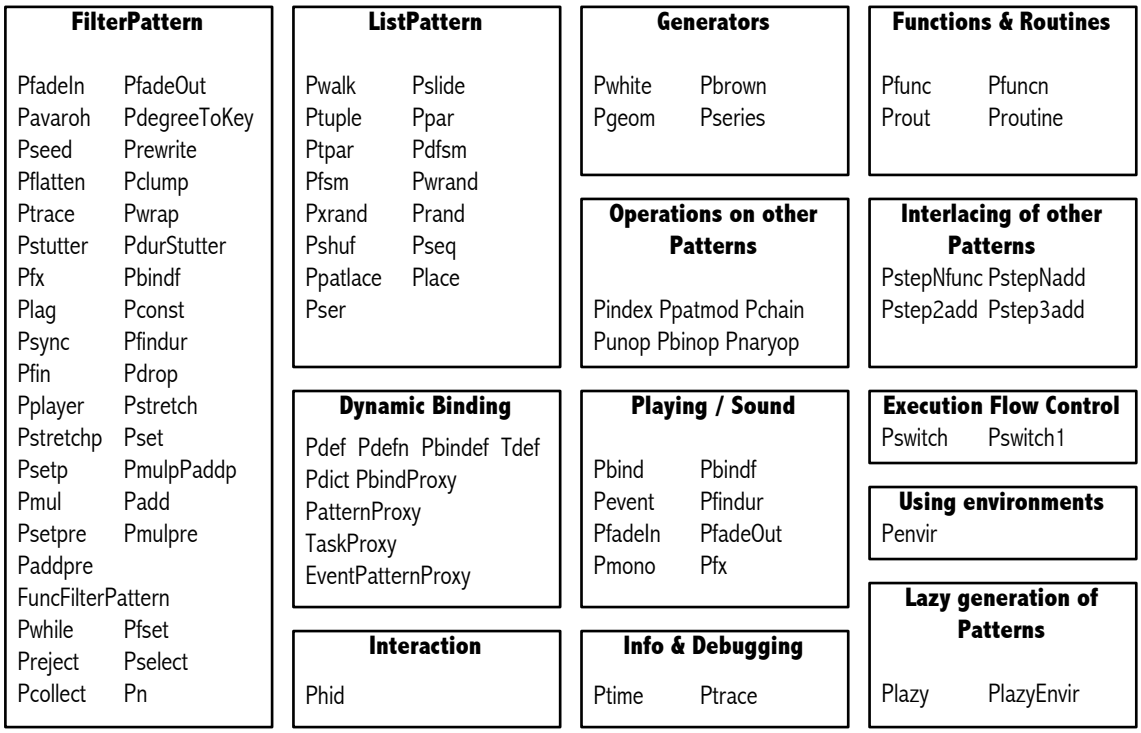
are two reasons:

1. With Patterns you can mix algorithmic and interactively controlled generation of score data of any kind together with fixed sequences of timed events.
2. Patterns can be nested to any depth to form more complex patterns out of groups of simpler patterns.

Examples:

...

Pattern map



Combining patterns

```
a = Pwhite(-1, 1, inf);
b = Pstutter(4, Pseq((0..10), inf));
c = (a * b).asStream;
{ c.next } ! 80;

[ 0, 0, 0, 0, -1, -1, -1, 0, 0, 2, -2, 0, 3, -3, 0, 3, 4, 4, 4, 0, -5, 0, 0,
5, 6, -6, -6, 6, -7, 0, 7, 0, -8, 0, -8, 8, -9, 9, 0, 9, 10, 10, 0, -10, 0,
0, 0, 0, 0, 0, -1, 1, 2, 2, 0, 2, -3, 3, 3, -3, 0, 0, -4, -4, 0, 5, 5, 5, 6,
-6, 6, -6, 0, 0, -7, 0, -8, 8, 8, 0 ]
```

Which parameters can be patterns?

This does not work (Discuss Pwhite source code and explain that hi and lo arguments are not used as streams with "next")

```
(
a = Pwhite(Pseq((0..10),inf), Pseq((0..-10),inf)).asStream;
{ a.next } ! 15;
)
```

For such cases, one must compose the pattern with other means. Instead of Pwhite, compose the rrand message through a Pbinop:

```
(
a = Pbinop('rrand', Pseq((0..10)), Pseq((0..-10))).asStream;
{ a.next } ! 15;
)

( // sounding example:
a = Pbinop('rrand', Pstutter(20, Pseq((0..-10))), Pstutter(10,
Pseq((0..20))));
Pbind(\dur, 0.1, \degree, a).play;
)
```

(Possibly list / classify patterns whose arguments can be patterns and those whose arguments are constants.)

Playing patterns

Patterns and Interaction

How to change the parameters of patterns interactively? One possible basic technique is using Pbindf.

```
(
a = (dur: 0.1);
Pbindf(a).play;
)
// change the octave in a:
a.octave = 6;

// and again:
a.octave = a.octave - 2;
```

Combining with Pfunc to read the values of environment variables from an event whose values one can set interactively. The playing must be evaluated within the event of Pbindf as current environment in order for the other patterns to have access to the environment:

```
(
a = (dtest: 0);
              // wrong: pfunc does not have a as current environment
b = Pbindf(a, \dur, 0.1, \degree, Pfunc { ~dtest + 3.rand } );
b.play;
)

(
a = (dtest: 0);
b = Pbindf(a, \dur, 0.1, \degree, Pfunc { ~dtest + 3.rand } );
              // correct:
a.use { b.play };
)
// now try modifying dtest:
a.dtest = 5;

( // another way: provide a as environment only for the Pfunc:
a = (dtest: 0);
b = Pbindf(a, \dur, 0.1, \degree, Pfunc { a.use {~dtest + 3.rand } });
b.play;
)

a.dtest = 3;
```

Events

When a pattern

Event objects are used in SuperCollider to store all the information that may be required to create a sound event in a flexible way.

Notes - Basics about what an event is: An environment + custom "properties" in the environment that enable the playing of "musical score events".
- How events are used: `Event.use` …

The effect of "know"

Event inherits its environment from its parent

The method Event-play sets the parent event of an event to the defaultParentEvent (!)

Environments

The current environment

Event inheritance

The effect of "know"

Structure of the defaultParentEvent

Modifying Pbinds interactively with Pbindef

Adding interactive controls to patterns

(Use *references* (`Ref`) to write to)

From the GUI

From MIDI input

From OSC input

Sending MIDI with Pbind

Saving and managing patterns: Pattern libraries

The language

Objects

The SuperCollider application is an object oriented programming language. Object oriented means that the fundamental entities in the SuperCollider Language are objects. Every thing that the programmer deals with is some kind of object. For example, numbers, windows, synthesis elements, and functions are all objects.

Examples of objects are:

```
1               // an Integer
1.234           // a floating point number
"hello"         // a String (an array of characters)
\alpha          // a Symbol (a unique identifier)
'alha 1'        // another notation for a Symbol
$a              // a Character
100@150         // a Point
[1, \A, $b]     // an Array
String          // the Class String
Meta_String     // the Class of the String Class
                // A function is also an object:
{ |x| if ( x > 1) { x * thisFunction.(x - 1) } { x } }
```

☛ **Every thing in SuperCollider is some kind of object**

The reference section *Language Syntax Overview* lists extensively the various kinds of syntax for different types of objects. One can distinguish three kinds of objects based on the way that an object contains other objects:

1. Objects that do not contain other objects. Such are instances of: Integer, Float, Symbol, Boolean.
2. Objects that contain other objects in a fixed number of named instance variables. Such are instances of: Synth, SCWindow, Point, Rect, Function, Routine, Method.
3. Collections, that is, objects which contain a variable number of accessible objects as elements. Such are instances of: Array, List, Dictionary, Set, Bag, Event.

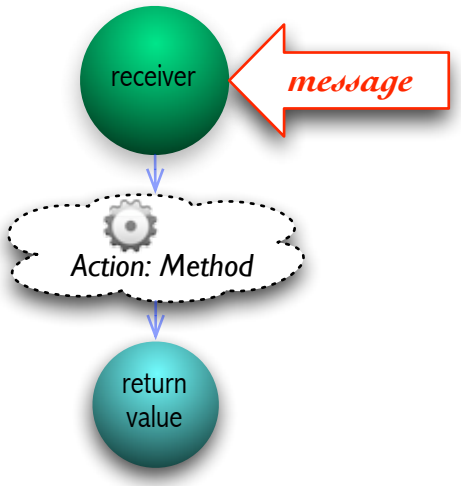
Communication between Objects: Messages

Message, method, return value

Objects communicate by sending messages to each other. To ask something of an object, one sends it a message. The object that receives the message is called the *receiver*. The term *receiver* often appears in error messages or other feedback from the system, so it is important to know its

meaning.

In response to the message, the receiver will perform some action. The specific kind of action performed by a specific kind of object in response to a specific message is called a *method*. After performing the action, the receiver will return some object which constitutes the response to the message, and is called the *return value*.



In SuperCollider code, each message is represented by a name in the form of a string. The most common form of sending a message to an object is usually `<receiver>.<message>`. For example, to calculate the square of the number 2, send the number 2 the message `squared`:

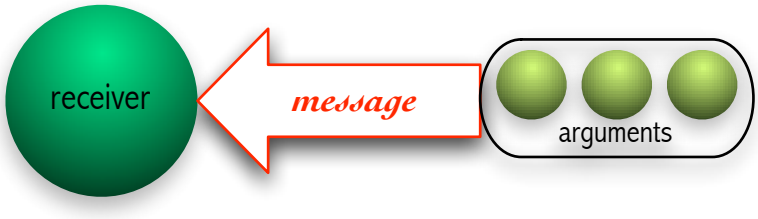
```
2.squared      // get the square of 2
```

Another example in the same form:

```
10.sqrt        // get the square root of 10
```

Messages with arguments

Sometimes, one needs to pass to the receiver some additional information along with the message. For example, to calculate the third power of 2, one needs to send it the message `pow` ("power") and additionally, one needs to specify the number 3 (third power). That additional item is called an *argument* to the message.



Arguments to a message are appended after the message enclosed within parentheses:

```
2.pow(3)  // calculate the 3d power of 2 (2 to the 3)

// scale the point 10@10 by the point 3@5
(10@10).scale(3@5)
```

If a message takes more than one arguments, each argument must be separated from the next one by a comma:

```
// Create an Array of 5 elements ranging at random between 1 and 100:
Array.rand(5, 1, 100)
```

If a message takes only one argument, and that argument is a function, then the parentheses can be omitted, as for example in the message `do`:

```
10.do { |i| Post << "Count: " << i <<", random number: " << 9e9.rand << "\r"
}
```

Declaring arguments

To know what arguments a message expects, we have to look at the definition of the method or function that this message will call. We therefore introduce here the syntax of argument declaration before talking in detail about functions (see section *Storing and executiong programs: Functions* below). Arguments are declared in function or method code by the reserverd word `arg` immediately following the opening braces or by enclosing the arguments in `| |`. In the following example we define a function with 4 arguments and then evaluate it with the message `value`:

```
(
f = { arg x = 0, y = 0, size = 1, proportion = 1; Rect(x, y, size, size *
proportion) };
f.value(10, 20, 100, 0.5);
)
```

Here 10 is the x argument, 20 is y, 100 is size and 0.5 is proportion.

For brevity, we can use `| ... |` instead of `arg ... ;`. The above function can also be written as:

```
f = { | x = 0, y = 0, size = 1, proportion = 1 | Rect(x, y, size, size *
proportion) }
```

Argument order, providing the argument with a keyword

As shown in the previous section, arguments have to be provided in the order in which they are declared in the method or function. If the declaration specifies default values, then we can omit the arguments and they will be provided from the defaults:

```
(
f = { arg x = 10, y = 10, size = 100, proportion = 1;
     Rect(x, y, size, size * proportion)
};
f.value.postln;         // x, y, proportion, size provided by defaults
f.value(50).postln;     // x provided by user, y, proportion, size provided by
defaults
f.value(80, 20).postln;  // x,y provided by user, proportion, size provided
by defaults
)
```

What if in the above example we want to specify our own value for proportion, say 0.5, but leave all the others to their default? In that case, we can skip the argument order by providing the name of the argument that we want to specify as a keyword followed by :. The form is:

```
f.value(proportion: 0.5);
```

Any number of arguments can be given by order, followed by any number of arguments with their keywords. The following is valid:

```
f.value(300, 500, proportion: 0.5);
```

Using a variable number of arguments

There are cases when an object may want to be able to collect *any number of arguments* passed to it by a message. It may not know how many arguments to expect, because the number of arguments may depend on other objects that use those arguments further down, and these objects may be of different kinds expecting different numbers of arguments. In this case, the relevant method or function can collect the arguments in one array by indicating this with an ellipsis (`...`):

```
(
f = { | name = "unknown" ... moreArguments |
     Post << "My name is " << name << " and I was given "
          << moreArguments.size << " extra arguments\r";
     Post << "The extra arguments were: " <<* moreArguments << "\r";
};
f.value("Bob", "hello", "there");
)
```

The opposite can also be the case: We can have a number of objects collected in an array, but want to pass them to an object as separate arguments. This can be achieved by adding the prefix `*` before the array:

```
f.value(*["Jo", "Array", "or", "not", "it", "works"]);
```

```
// Here the * works in any position:
f.value("Jo", *["Array", "or", "not", "it", "works"]);
f.value("Jo", "Array", *["or", "not", "it", "works"]);
```

No limit to the number of arguments

There can be any number of arguments to a message. Following are some compactly coded tests of passing a function variable numbers of arguments. See sections on evaluation of functions and section *List Comprehensions* for background on the syntax and meaning of this code.

```
// Test 1: Count the number of arguments passed to a function. Test this with
// argument lists of sizes 1, 10, 100, 1000, 10000, 100000, 1000000.
all {: { | ... args | args.size }.(*{ 0 }!(10**x)), x<-(0..6) }

// Test 2: For doubting Thomases:
// Instead of counting the number of arguments, calculate their sum
all {: { | ... args | args.sum }.(*{ 2 }!(10**x)), x<-(0..6) }
```

And here is a less compact equivalent to the code of the first example:

```
// Construct the test function and store it in f
f = { | ... args | args.size }   // collect args in array, return size of
array

// evaluate the test function with an array of arguments
f.(*{ 0 } ! 10)         // * passes the array as separate arguments

// Try out with biger arrays:
f.(*{ 0 } ! 100)

f.(*{ 0 } ! 1000)

f.(*{ 0 } ! 10000)

f.(*{ 0 } ! 1000000)   // one million arguments! Delay will be noticeable
```

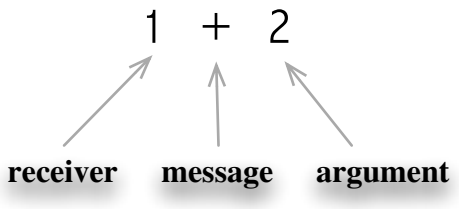
Evaluating the last line above may keep the system occupied for a noticeable time interval (about one second), because it requires evaluating a function one million times and collecting the result of each evaluation in an array which is then passed as argument to *f*.

Another form of messages: Binary operators

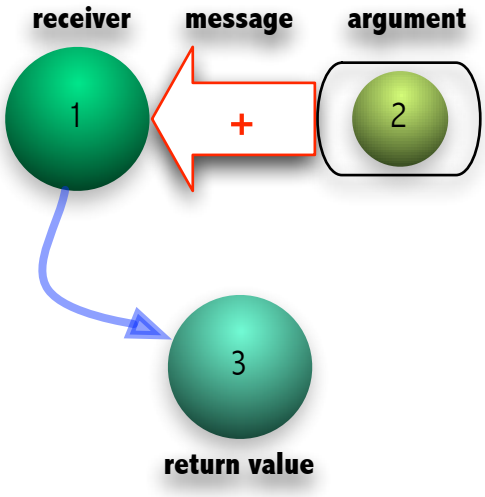
Expressions such as the one introduced above in the section *Evaluating one line of code* are actually also messages:

```
1 + 2
```

means to send the number 1 the message "+" with the argument 2.



The action performed by the system can be represented thus:



Messages that take the form such as + above are called *binary operators*.

Binary operator form for messages with one argument

Any message with an alphanumeric name that takes only one argument can be written in the form of a binary argument by appending the character ":" to the message name. Thus:

```
2.pow(5)
```

can also be written as:

```
2 pow: 5
```

Changing the properties of objects with messages

A message can be used to change some property of an object rather than just to calculate and return a result:

```
// Set the default server to be the local server
Server.default_(Server.local)
```

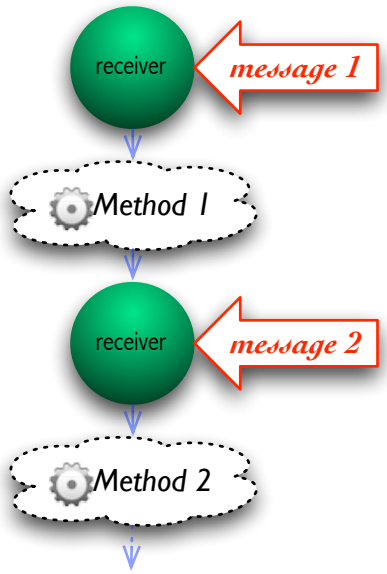
Messages with appended underscore like `default_` in the above example usually have the effect of storing some object in some property (which in most cases coincides with an *instance variable*) of

some object. An alternative form for messages that end with an underscore uses the assignment sign "=". Thus the above example can also be written as:

```
Server.default = Server.local
```

Chaining several messages in one statement

As shown in the previous section, there are many cases in which the purpose of a message is to cause some change inside the receiver, rather than to obtain some new object to work with. In these cases, the method corresponding to the message will return the receiver, so that new messages may be "appended" after the first message to the same object.



Since every message has as a result an object returned by the receiver, one can send to the result of one message another message, by appending one more dot-message after the first message. One can thus create chains of messages of any length.

```
SCWindow.new; // Create a window.
              // The window will be invisible until brought to the front

SCWindow.new.front // Create a window and make it visible
```

(Use the mouse to close the window just created before continuing)

```
SCWindow.new.front.view     // get the main view of the window just created
```

(Use the mouse to close the window just created before continuing)

```
// get the main view of the window and set its background color to red:
SCWindow.new.front.view.background_(Color.red)
```

(Use the mouse to close the window just created before continuing)

When chaining messages, it is essential to know which object is returned by each message in turn. If for example one wants to set the name of the window by chaining a new message to the above statement, the new message must be addressed to the window, not to the view. Since the `view` message returns the main view of the window, one must place the message that sets the name before getting the view, not after it.

```
// wrong: name_ message sent to the view is not understood and results in
error
SCWindow.new.front.view.background_(Color.red).name_("a new window")

// correct
SCWindow.new.front.name_("a new window").view.background_(Color.red)
```

Error: "Message not understood"

The example of wrong order in the chaining of messages above causes an error message "Message 'name_' not understood. Such a message will occur every time that an object does not understand a message. Let's try this again:

```
SCWindow.new.front.view.background_(Color.red).name_("a new window")
```

Here is the entire error message posted by the system:

```
ERROR: Message 'name_' not understood.
RECEIVER:
Instance of SCTopView {    (06AD45D0, gc=98, fmt=00, flg=00, set=04)
  instance variables [11]
    dataptr : RawPointer 657B420
    parent : nil
    action : nil
    background : instance of Color (06A98B10, size=4, set=2)
    keyDownAction : nil
    keyUpAction : nil
    keyTyped : nil
    beginDragAction : nil
    onClose : nil
    children : nil
    decorator : nil
}
ARGS:
Instance of Array {    (069C8660, gc=98, fmt=01, flg=00, set=01)
  indexed slots [1]
      0 : "a new window"
}
CALL STACK:
     DoesNotUnderstandError-reportError   074D6430
          arg this = <instance of DoesNotUnderstandError>
     Nil-handleError   07CB9A90
          arg this = nil
          arg error = <instance of DoesNotUnderstandError>
```

```
Object-throw   06B16650
     arg this = <instance of DoesNotUnderstandError>
Object-doesNotUnderstand   06A19B00
     arg this = <instance of SCTopView>
     arg selector = 'name_'
     arg args = [*1]
Interpreter-interpretPrintCmdLine   06CA2EA0
     arg this = <instance of Interpreter>
     var res = nil
     var func = <instance of Function>
Process-interpretPrintCmdLine   07D11A10
     arg this = <instance of Main>
```

Immediately below the first line of the error message (ERROR: ...) follows a printout of the receiver of the message that caused the error and of the contents of that receiver. Here, the receiver is "An instance of SCTopView". ...

Computation order, precedence rules and grouping

(Under development:

- Precedence inside message arguments
- Precedence of (.) messages over binary operators
- Left-to-right Precedence in binary operator expressions .

The arguments of a message are computed first, then they are sent to the receiver with the message. Thus, the last object computed in a nested message statement is the leftmost receiver.

Summary

☛ To communicate with an object, one sends it a message

☛ An object that receives a message is called the receiver

☛ The action that object executes in response to a message is called the method

☛ In response to a message, the receiver always returns some object

☛ Additional objects passed to the receiver together with the message are called arguments

Storing and accessing objects: Variables

The previous section has shown how objects are produced as the result of sending messages to other objects. But what if one needs to store some of the resulting objects somewhere so as to access them at some later place in the code or during an interactive development session? Storing objects and making them accessible for retrieval is the job of *variables*. The present section explains how this is done.

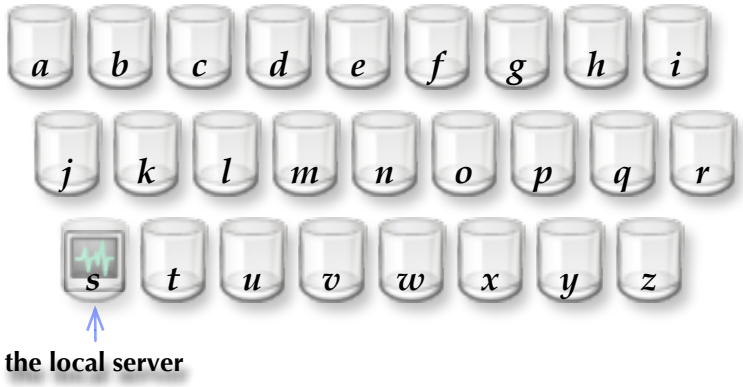
The present section introduces a special kind of variables called *interpreter variables*. These variables are always available when evaluating code interactively in workspace windows. Moreover, their contents are preserved across evaluations, that is, if some object is stored in an interpreter variable by evaluating some code, then it will still be there in that same variable when that variable is used in evaluating some other code, or in evaluating the same code again. Future sections will introduce other types of variables that whose lifetime and availability across code is different than that of interpreter variables.

Variables: Named containers for storing objects

A *variable* is a named container where one can store any object in order to retrieve it at some later point. They are the only means for the programmer to put objects somewhere so that they can be accessed and used again at some later point. Consider for example a variable named *x* or a variable named *freq*. This can be thought of as two containers that can be accessed by their names:

The 26 "interpreter variables"

For convenience, the SuperCollider language system provides 26 permanent variables, named after the 26 lower-case letters of the alphabet (*a*, *b*, *c*, ... *z*). These are called *interpreter variables*. Such variables are often encountered in example- or help-files, and can be freely used when experimenting with code interactively in a workspace window.



The 26 interpreter variables

As shown above, the 26 interpreter variables are initially empty, except for the variable *s*, which stores an object representing the local server. This is useful to access the local server for sending it commands in order to boot, quit, or to play sounds with it.

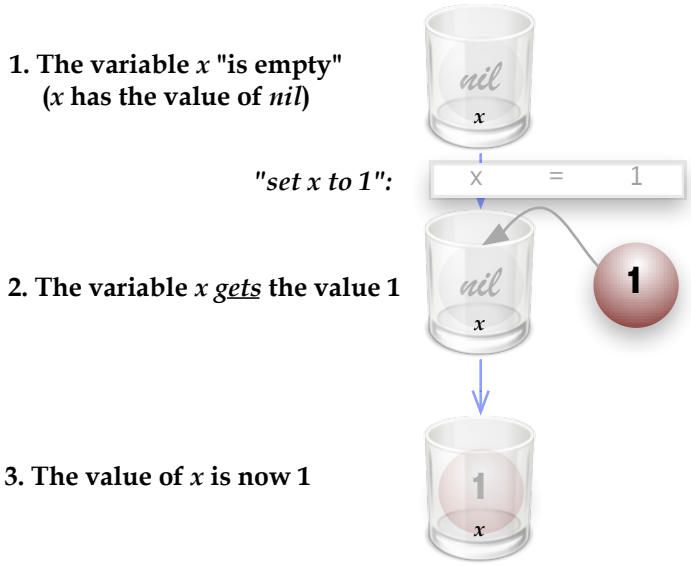
The interpreter variables are only available when evaluating code entered in workspace windows interactively. They should not be used inside class definition code. They are called interpreter variables because they are defined as instance variables of the Interpreter Class, which is the class responsible for evaluating code entered interactively via workspace windows.

Storing an object in a variable

Putting something in a variable container is called *setting the value of a variable* or *assigning a value to a variable*. To assign a value to a variable one uses the assignment operator `=`. For example, to store the number 1 in the variable x, evaluate the expression:

```
x = 1
```

Following figure illustrates what happens when evaluating `x = 1` (provided that initially *x* contains nothing):



After evaluating the above statement, the variable *x* will contain the number `1`. To find out what is contained in a variable one *gets the value of the variable* as explained in the next section.

Assigning a value to a variable replaces the object that was previously stored in that variable.

The previous object may be lost (no longer accessible to the user) if a new value is stored in the variable.

To collect many values together, special kinds of objects are used, which are called *Collections*. These will be introduced in the following section.

Variable types (SuperCollider does not have them)

The *type* of a variable denotes the kinds of objects that can be stored in it. In SuperCollider, variables have no type. This means that technically one can store any kind of object in any variable. Programming languages that restrain the kind of object that can be stored in a variable according to its type are called *typed* languages, but SuperCollider is not one of them.

Retrieving an object from a variable

When the system encounters a piece of code that consists of alphanumeric characters and starts with a small letter and is not a message, it looks to see whether there is a variable named after that piece of code. If the variable exists, the system returns the value that is stored in that variable. This is called *getting*, or *accessing* the *value of the variable.* For example, evaluate the single line of code that just accesses variable *s*:

```
s
```

The Post Window will show the result of accessing the value of the variable *s*, which, as explained in the previous section, should by default be the local server, printed here simply as: `"a Server"`. One can inspect the contents of that server by sending it the message `inspect`, This will open an inspector window:

```
s.inspect
```

In the example from the previous section, after setting the value of variable x to `1` with `x = 1`, one can look at the value of *x* in the same way as *s* above.

If the system cannot find a variable under a given name, then it issues an error message. Try for example accessing the non-existent variable `whatIsThis123`:

```
whatIsThis123
```

The resulting error will be:

```
· ERROR: Variable 'whatIsThis123' not defined.
   in file 'selected text'
   line 1 char 0 :
  ·whatIsThis123

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

Since the system did not find any object to return in its attempt to evaluate the code, it returns the object that stands for "nothing", that is the object *nil*. As can be expected, the system will also return nil if a variable exists but does not contain anything. Try for example accessing the value of any one of the remaining 25 interpreter variables other than *s*, before storing anything into it:

```
y
```

A variable containing nothing (*nil*)

Examples of interpreter variable usage

```
x = 1;

50 do: { x = x * (x + 1) }

// result:
1048004806
```

Three kinds of variables: Interpreter, Declared, Environment

Collecting many objects in a collection

Setting many variables at once: Multiple assignment

Constants

Constants are named containers that hold a fixed value. In other words, while the value of a variable can be changed, the value of a constant cannot be changed. The constants of SuperCollider are:

`nil`: The object standing for nothing - i.e. for the contents of a variable that has nothing stored in it (an empty variable). nil is also what a stream returns when trying to access an element beyond its end.

circumference of a circle

`true`: The boolean value of true

`false`: The boolean value of false

`pi`: floating point approximation of the value of π, i.e. the ratio of circumference to diameter of a circle, approximately 3.1415926535898.

`inf`: The floating point number representing infinity

nan: "Not A Number": The pseudo-constant representing the result of a numeric calculation that does not result in a valid number. This results mostly when dividing 0 by 0. Try: `0/0`. Another example: `-inf.rrand(inf).` *nan* is a pseudo-constant because it

cannot be used in program code, it is only created and printed by the system as the result of a calculation.

Storing many objects in one: Collections

Array

List

Bag

Set

Dictionary

Environment

Functions (temporary, alternative version)

The subsections of this version are to be merged with those of the sections that follow it ...

Defining functions

Evaluating functions

Defining arguments

Passing arguments

Storing and executing programs: Functions

(Under development:) A function is a special kind of object that stores a program. The program is stored in the form of a sequence of statements.

Declaration and Scope of Variables

Declaring variables

To create a variable (a new "container") under a new name, one has to *declare* the variable. This reserves space for a new container and makes the name of the variable known to the system so that it can be accessed. A statement that creates new variables is called *variable declaration statement* and starts with the keyword `var`. For example, the following statement declares the variable *frequency* and subsequently sets its value to 400. Finally, it adds 40 to the current value of frequency and returns the result:

```
(
var frequency;     // declare variable "frequency"
frequency = 400;   // set the value of frequency to 400
frequency + 40;    // add 400 to the value of frequency
)
```

New variables can only be created at the beginning of an independent block of code such as at the very start of the entire code evaluated by the user in a workspace window, and at the beginning of a function, method or class definition.

Also, the above example must always be evaluated as one entire block of code. If you try to evaluate lines 2 or 3 alone, an error will occur. This is because variables declared with the `var` statement are always created anew at the beginning of evaluating some function (block of code) and disappear as soon as the function has been evaluated.

☛ **Variables must be declared at the beginning of a block of code**

☛ **Declared variables only exist within the block of code where they were declared.**

The lifetime of variables

Rewrite and expand:

The contents of *x* will remain available in the system for further evaluations of the same code or of other code, until they are replaced by setting *x* to another value. (Note however that other kinds of variables "disappear" as soon as the code is evaluated, as will be explained in the section *Declaring variables*).

The scope of variables

They are accessible only to the elements that are within this block of code, including code of all other sub-blocks that may be defined within the block of code. This limitation eliminates the danger of overwriting a variable's value from some foreign part of code when that code happens to use the same variable name for a different purpose. The set of elements that have access to a

variable is called the *scope* of that variable. The principle of limited scope of variables makes it easier to keep program modules independent from each other, thereby allowing a safer programing style.

Environment Variables

"Pseudo-Variables" (or "System-Variables")

There is a small number of variables that are provided by the system and are accessible at all times. These contain objects that can be essential for certain tasks, but would be inconvenient to track manually. They may be called system-variables or pseudo-variables, because they cannot be declared by the user neither can they be set by the user. These are:

this: used inside the code of a method, enables the method to access the object that is executing this method.
super: used inside the code of a method, makes the object that
thisFunction: Used inside a function, accesses that function itself. This enables recursion (see example below).
thisMethod: Used inside a method, accesses that method itself.
currentEnvironment: Accesses the current environment
thisProcess: Accesses the *Main* process that is responsible for startup, shutdown and other top-level commands.

Examples using system-variables outside of Class definition code are:

1. Accessing the Main process

```
thisProcess.stop;  // Stops all sounds and routines
```

2. Defining a function that accesses itself (recursion).

```
// Define a function that calculates factorial recursively:
f = { | x | if (x > 1) { x * thisFunction.(x - 1) } { x } };
// f now calculates the factorial of x:
f.(3).postln;      // factorial of 3
f.(5).postln;      // factorial of 5
f.(10).postln;     // factorial of 10


);
```

An example using

Other types of containers: Containers inside objects

Summary

- A variable is a named container that can hold exactly one object.
- The object stored in a variable is called the value of that variable.
- An empty variable has a value of nil
- An object created from a class is called an instance of that class
- A class defines the structure (properties, i.e. variables) and behavior (i.e. methods) of its instances.
- A class may inherit the structure of another class, called its superclass
- A class also has a class, called its Meta-Class. It thus has class variables and class methods.
- Class variables are shared between the class, its subclasses, and all their instances.
- Those parts of methods that contain precompiled machine code are called primitives

Control structures

if
?
??
case
switch

Iteration

do
collect
select
reject
detect
for
forBy
loop
while
all {: } (list comprehensions)

Routines

Templates for creating Objects: Classes

Classes and instances

Class is like a template or factory that creates objects. The objects created by a class are called *instances* of that class. Usually, one can create any number of instances from one class.

Many of the examples up until now have involved simple objects such as integer or floating-point numbers.

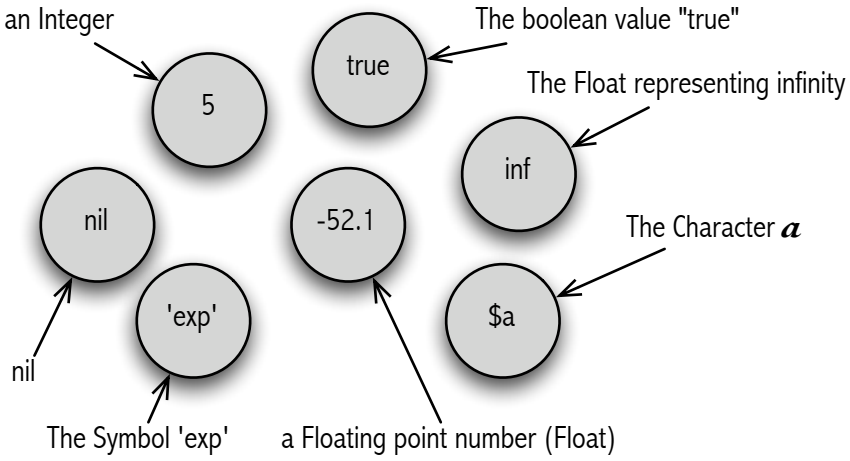
(examples of instances)

```
Array.newClear(2)       // make a new array with 2 empty slots

Array.newClear(10)      // make a new array with 10 empty slots

// Make a new window named "window 1" within the rectangle 200, 200, 300, 100
and bring it to the front
SCWindow.new("window 1", Rect.new(200, 200, 300, 100)).front

// Make a new window named "window 2" within the rectangle 300, 350, 100, 100
and bring it to the front
SCWindow.new("window 2", Rect.new(300, 350, 300, 100)).front
```

Programming in SuperCollider means extending the existing system either (a) by the execution of pieces of code that create *Objects* and perform computations with them, or (b) by modifying or extending the definitions of *Classes*.

Storing objects inside other objects: Instance variables

Certain "simple" objects such as numbers, boolean values, characters do not hold any other object but themselves. Some examples of such objects are illustrated graphically below:



Other kinds of object may hold other objects "inside themselves". If an object contains other objects it may do so in two ways:

1.An object may contain a fixed number of named "slots", where each slot can store an object. Such named slots are called *instance variables*, since they are variables private to the instance that contains them. For example, every instance of the class Point that represents the x and y coordinates a point on a plane such as the computer screen always holds the instance variables x and y. Different points will have different values for x and y.

Examples of such objects are:

Model
Point
Rect
Association
Synth

[Figure]

2. An object may contain a variable number of slots without name. Such

As an object oriented principle, an object can only access the objects that it contains. This principle is called *encapsulation* of data. Naturally, there are many cases where one may need to know the value of an object *x* contained in the instance variable of another object *y*. In order to find this out, one may enquire that value from object *y* by sending it a message. The next section, *Messages: Communication between Objects*, explains how messages work.

Methods

= Named functions belonging to classes

Performing (evaluating) methods

Primitives

The method for "new" and its variants

The structure of classes

Inheritance

The SCClassLibrary

The class library is represented by a collection of files residing in the folder SCClassLibrary. The code in these files defines the Classes of the System, that is, the collection of templates from which objects are created. The SCClassLibrary is compiled when the SuperCollider application starts.

There are many hundreds of classes in SuperCollider. These define objects for dealing with following kinds of tasks:

- Numerical computation (Integers, Floating point numbers) and numerical calculation
- Handling of Text (Characters, Strings, Symbols)
- Handling collections of objects (Arrays, Lists, Dictionaries, Sets etc.)
- Representing and communicating with the server engine and all objects it contains (see list of server objects below)
- Creating and using Graphic User Interface objects: windows, dialog boxes, sliders, text and numeric elements
- Defining and using computational algorithms, i.e. sequences of program statements: Functions, Methods
- Timing: Defining the relative time interval after which a function or statement should be executed: Clocks, Routines, Tasks etc.
- Definition of algorithmically generated musical structures: Patterns, Streams etc.
- Communication with other devices and programs: MIDI, OSC, HID

Later sections show how to write ones own classes and recompile the library.

Writing classes

Replace the following text which has been used in an earlier section with a text on writing classes and compiling ...

~~The class library is represented by a collection of files residing in the folder SCClassLibrary. The~~

code in these files defines the Classes of the System, that is, the collection of templates from which objects are created. The SCClassLibrary is compiled when the SuperCollider application starts. New Classes can be defined by the user. In order for any changes in the class definition code to take effect, one must compile the entire library anew (keyboard shortcut: Command-K). This rebuilds the entire system from scratch. Any objects created by evaluating code in workspace windows or by previous program runs are erased.

There are many hundreds of classes in SuperCollider. These define objects for dealing with following kinds of tasks:

- Numerical computation (Integers, Floating point numbers) and numerical calculation
- Handling of Text (Characters, Strings, Symbols)
- Handling collections of objects (Arrays, Lists, Dictionaries, Sets etc.)
- Representing and communicating with the server engine and all objects it contains (see list of server objects below)
- Creating and using Graphic User Interface objects: windows, dialog boxes, sliders, text and numeric elements
- Defining and using computational algorithms, i.e. sequences of program statements: Functions, Methods
- Timing: Defining the relative time interval after which a function or statement should be executed: Clocks, Routines, Tasks etc.
- Definition of algorithmically generated musical structures: Patterns, Streams etc.
- Communication with other devices and programs: MIDI, OSC, HID

Finding information about classes and objects

Summary

- ☛ Objects are created from templates called "Classes"
- ☛ An object created from a class is called an instance of that class
- ☛ A class defines the structure (properties, i.e. variables) and behavior (i.e. methods) of its instances.
- ☛ A class may inherit the structure of another class, called its superclass
- ☛ A class also has a class, called its Meta-Class. It thus has class variables and class methods.
- ☛ Class variables are shared between the class, its subclasses, and all their instances.
- ☛ Those parts of methods that contain precompiled machine code are called primitives

Reference

Anatomy of a synthdef

(Synthdefs in detail. Some topics to expand:)
- Arguments to the synth function create controls
- Extra arguments to the synthdef creation function set the type of each control

---

Following are too detailed for now: (Mention them? Rather not?)
- Controls can be created also explicitly inside the synthdef
- Only those signals that are explicitly output via Out.kr or Out.kr are audible
- One can have many Out UGens in one Synthdef.

. How synthdefs are compiled, How to read and to snoop into synthdefs, inspecting SynthDefs, SynthDescs,

Language Syntax Overview

Following is a terse account of SC Language syntax by example.

Comments: Making your code readable

Comments are useful for explaining to humans what the code does. The compiler ignores them.

Everything to the right of "//" on one line counts as a comment, i.e. it is ignored by the compiler. Example:

```
var rpt; // Comment: rpt holds the number of repetitions of the pattern
```

Long comments are enclosed in /**/. This can also be used to comment out out blocks of code for debugging. Example:

```
/*   Everything between /* and */
     will be ignored by the compiler!
     Nesting of /* is allowed as long as it is matched by
     the closing */
     ALL THIS IS JUST A COMMENT!
     USE COMMENTS TO MAKE YOUR CODE READABLE!!!
*/
```

Objects: Modeling the world

Objects are the basic elements of the Language. Anything that the user can interact with in the system is eventually some kind of object. Basic forms of commonly used objects are:

Numbers

Numbers are:

Integers: 0 1 -1 123456 -98765

Floating point numbers (digits must exist on either side of dot): 0.0, -1.01, 1.0, 1234.01234, -987.4321

Floating point numbers in exponential notation (E or e denotes raising multiplying the number by a power of 10):
1.2345E2 is 123.45, 12.345e2 is 1234.4, 12.345e-3 is 0.12345

Floating point numbers with decimal "alterations":
b means -0.1, s means +0.1, up to 4 alterations accepted. Examples:
1b, 1s, 3b, 5ss, 10bbb, 7ssss,

Some special numbers:
- `pi` (circumference of circle with diameter 1: 3.1415926535898...),
- `inf` (infinite),
- `nan` (not a number, result of 0/0, only created by the system)

Characters

Characters are notated with the $ sign as prefix: `$a $b $A $Z $. $ $#`
Some special characters can be encoded by preceding them with backslash "\":

Next line:	`$\n`
Form Feed:	`$\f`
Tab	`$\t`
Return	`$\r`
Space (the space after $ counts!):	`^$`

Characters with ascii code beyond 127 or unprintable and with no escape "\" equivalent can only be encoded indirectly by converting their ascii number to a character:

Bullet: `165.asAscii`

The above character "Bullet" also has a convenient "memo" method:

```
Char.bullet   // is equivalent to: 165.asAscii
```

Strings

Strings are enclosed within double quotes " ":

```
"This is a string"
```

Special characters \n, \f, \t, \r (see Characters) can be included:

```
"This is a string with line feed: \n"
```

Strings can be concatenated with ++:

```
"This is a number: " ++ pi.asString;
```

The class Post can be used to concatenate strings with other objects on the Post Window:

```
Post << "Some array elements ... " <<* (_.rand ! 10) << "\r"
```

Elements of strings are characters and can be accessed like those of arrays.

```
a = "alpha";
a.postln;
a[2] = $o;
```

Symbols

Symbols are unique names made out of strings. They cannot be modified or accessed as arrays. They are written with the prefix "\": `\aSymbol`, or if they contain non-alphanumeric characters or start with a space, they must be enclosed in quotes: `'another Symbol!'`

Symbols can be concatenated:

```
\Synth ++ \Def;
```

But to access a Symbols string one must convert it first:

```
\Synth.asString[2];
```

One can modify the string of a symbol and convert it back to obtain a new symbol:

```
\Synth.asString.put(0, $C).asSymbol;
```

Collections

Collections are enclosed in square brackets ([]).
Each element of a collection is separated from the next one by a comma (,)
Examples:

```
["this", "is an Array", 1, 2, 4]     // [...] creates Arrays
List["This is a", \List]    // Classname[...] creates collection of class
Classname
Dictionary[$a->"Dictionary", \this->\is] // Dictionary objects require
Associations
(3..-1)                // shorthand for arithmetic progressions
(1, 1.1..3.02)         // shorthand for arithmetic progressions with variable
step
(freq: 400, amp: 0.1)  // special notation for creating Events
```

Functions

Functions are enclosed in braces: `{ arg a, b;  (a + b) / 2 }`

Classes

Class names always start with capital letter: `Class Object Boolean Number Server UGen Function Routine Clock LFNoise2 K2A`

Immutable objects

Immutable objects are objects whose internal structure cannot be changed once they have been created. To illustrate the difference between immutable and non-immutable objects, compare the Array [1, 2, 3] with the immutable Array #[1, 2, 3]:

```
a = [1,2,3];        // a changeable Array
```

Now try to change the contents of a:

```
a[1] = 10;          // store 10 into the second element of a
// now print the array in a, to observe that it has changed:
a                   // the result is:
[1, 10, 3]

b = #[1,2,3];       // an immutable Array
```

Now try to change the contents of b:

```
b[1] = 10;          // try storing 10 into the second element of b
```

This will cause an error. The system posts:

```
ERROR: Primitive '_BasicPut' failed.
Attempted write to immutable object.
```
... etc.

Immutable objects are:
- All integer and floating point numbers.
- `true`, `false`, `nil`, `pi`, `inf`
- All symbols
- All characters
- Arrays whose definition was preceded by the # sign, such as: `#[1, 2, 3]`. These may contain only other immutable objects and / or strings.
- Closed functions. Closed functions are functions whose definition is preceded by the # sign, such as: `#{ |x| x.sqrt + 1 }`

If you want to change an immutable Array or String, make a copy of it and work with that copy:

```
a = #[1, 2, 3];         // a gets an immutable Array as value
a = a.copy.put(1, 10)   // copy a, change the copy, and put the copy back in a
a[2] = 20;              // the copy in a can now be freely changed
```

Messages and their arguments: Communicating with objects

Messages are used to communicate with objects:
Sending message "rand" to object "10": `10.rand`
A less used alternative form for the above is: `rand(10)`

Arguments

Arguments are used to pass additional objects to the receiver of a message:
Sending a message with an additional argument: `10.rrand(20)`
A less used alternative form for the above is: `rrand(10, 20)`
Sending a message with several additional arguments: `Array.series(10, 2, 0.1)`
Keyword syntax accesses arguments by name: `Server.internal.scope(zoom: 8)`
Omitted message "new:" `Dictionary(5)`
Chaining messages: `SCWindow("transparent?").alpha_(0.7).front`

Alternative syntax forms for messages with one argument

1. Binary operator ending in ":". 1.rrand(5) can also be written as:
   1 rrand: 5
2. Omitting the () when the argument is a function. 5.do({|i| i .post}) can also be written as:
   5.do {|i| i.post}

Other syntax forms for messages and arguments

Message first:

```
cos(pi);          // this means: pi.cos;
if (10.rand > 5, { "bigger" }, { "smaller" })
```

Binary operators: + - * / ** % == === < > <= >= != !== & | << >> +>> ++ <<< <<* <<<* <! ! ? ??

```
1 + 2,   2 * pi,    { 1.0.rand } ! 5, 10 rrand: 100
```

Many binary operators accept an adverb, i.e. second argument - limited to integer and symbol types:

```
Array.iota(3,3) +.2 [100, 200, 300]
```

Unary operators: ` ~:

`100 means create a Ref holding the integer 100 as value.
~ex means currentEnvironment.at(\ex)

Index access of a Collection's elements with [].

```
(1, 1.1..3.02)[3]
```

Arrays can be used to access elements of a collection by index, creating a (reordered) sub-collection:

```
(1, 1.21..3.02)[(3.rand .. 5.rand) * 2]
(10..1)[10.rand] = \changed          // [] can also be used to write into a collection!
```

Moving blocks out of argument lists:

```
if (x<3) {\abc} {\def};
while { a < b } { a = a * 2 };
```

Statements: Building blocks for programs

Statements represent the syntactically complete "sentences" in a program. Each statement must be separated from the next one by ";". The program returns the value of the last statement:

```
(
     (1..2 + 5.rand).postln;
     (10 + 1).postln;
     (1..2 + 5.rand) + (5 * 10.rand) / 2);
)
```

Statements can be included in lists of arguments or of elements in an array:

...

Omitting the semicolon of the last statement in a block

The last semicolon in a sequence of statements that is terminated by a closing parenthesis) or bracket } is optional and may be omitted. Example:

```
3 do: {
     (1..2 + 5.rand).postln;
     (10 + 1).postln;
     (1..2 + 5.rand) + (5 * 10.rand) / 2) // semicolon omitted
}
```

Embedding sequences of statements in element lists

As explained in the help-file "*Expression-Sequence*" under the Language subfolder of SuperCollider Help folder: "*A sequence of expressions separated by semicolons and optionally terminated by a semicolon are a single expression whose value is the value of the last expression. Such a sequence may be used anywhere that a normal expression may be used.*" This means that statemens can be embedded in the listing of elements in a collection or in a list of arguments.

```
max(a = 3.sqrt; b = a * 2; b + 5,  10);   // computes the maximum of b+5 and
10
```

Note: Forgetting the element that follows such a sequence, or mistakenly using a semicolon instead of a comma will cause results other than those intended, because the result of a statement closed by a semicolon will not be included in the indented list:

```
[a = 3.sqrt; b = a * 2; b + 5,  10]
// is different from:
[a = 3.sqrt; b = a * 2; b + 5;  10]
```

Dangerous unforseeable results may occur if a semicolon mistake happens in an argument list. Here, `max` always returns the last argument regardless of the value of the first argument:

```
max(a = 3.sqrt; b = a * 2; b + 500;  100;);
```

Variables: Containers for storing objects

Variables are named memory locations for storing data. They must be declared at the beginning of a function or program. There are 3 exceptions of variables that do not need to be declared: (1) global Interpreter variables a-z, (2) global Environment variables preceded by tilde: ~anEnvironmentVar, (3) system or pseudo-variables - see below.

Variable declaration

```
var freq;               // declare single variable x.
// Multiple var declarations on consecutive lines are accepted:
var x, y;               // several variables declared on one line.
var isRunning = false;  // provide a default value at declaration time
```

Variable assignment

Assigning a value to a variable:

```
a  = 1.0;
```

Multiple assignment - each variable gets the next value frome the array:

```
#freq, amp, pan = [400, 0.1, -0.5];
```

Ellipsis - last variable gets the rest of the array:

```
#freq, amp ... someMoreParams = [400, 0.1, -0.5, 101.3, pi];
```

Interpreter variables

System variables (pseudo-variables)

System variables, are variables that are declared by the system - not the user, and contain certain objects whose access may be useful in the context of some computation.

```
this      // the current object within which a method is being called
super     // forces messages to use a method from the superclass of this,
          // (or recursively, up to Object)
```

Following special variables hold what their name says they do:

```
currentEnvironment thisMethod thisFunction thisProcess
```

Environment variables

Constants: Some useful objects

Constants are a small number of reserved variables whose value never changes. These are:

`nil` ("nothing": value of a variable that does not contain any object)
`true` (The boolean value true, i.e. the result of a boolean expression that is true)

`false` (opposite of true)
`pi` (ratio of circumference to diameter in a circle)
`inf` (floating point number representing infinity)
`nan` ("not a number", result of the expression 0/0).
"nan" cannot be used as part of user code, it is only created by the system

Functions: Programs that can be repeated whenever needed

Functions are blocks of code enclosed in {}. They are objects that can be evaluated at any time, any number of times.

```
// create a function and store it in variable f
f =  {  "this is a function".postln; }
10.do { f.value }                // evaluate the function in f, 10 times.
// shorthand for repeating the evaluation of functions:
{ "this is a function } ! 10
```

Function arguments are used to pass additional data to a function from the outside:

```
// calculate the mean of x and y:
f = { arg x, y; (x + y) / 2  };  // define the function
f.(0, 10);                       // calculate the mean of 0 and 10
```

Function definitions preceded by # create closed functions. These are stored inline, which means they can be used as initialization values to variables, but they do not have access to a scope (context) outside of themselves:

```
// An open var can use
(
{     var scopeVar, open;

}
)
```

Note: perform and performList are equivalent forms to value and valueArray for performing a method:

```
Number.perform(\dumpClassSubtree);
```

Function evaluation

Functions can be evaluated by sending the messages `value` or `valueArray`. As a shortcut, the message value can be omitted:

```
(    var foo;
     foo = { arg x, y = 100;  (1 / (x + y)).postln; };
     // full form: message value
     foo.value(2, 3);                // arguments provided individually
```

```
    // Short form for foo.value(2, 3): value omitted:
    foo.(2, 3);
    // missing arguments get any defaults existing in function definition
    foo.(50);
    // arguments provided in an array
    foo.valueArray([4, 6]);
    // Also accepted:
    foo.valueArray(10, [11, 12]);    // rest of arguments in array
)
```

Note: `perform` and `performList` are equivalent forms to value and valueArray for performing a method:

```
    Number.perform(\dumpClassSubtree);
```

Function argument syntax and usage

Function Arguments are declared just after the opening bracket:

```
    // initialization of argument values works as for variables
    { arg arg1, arg2 = 10;
```

As shown above, defaults can be provided for arguments just as for variables.
Short form of argument declaration:

```
    { | x, y |  is equivalent to { arg x, y;
```

Ellipsis in arguments is used just as in multiple assignment for variables:

```
    // return array holding additional arguments passed to the function
    { arg x ... restArgs;
```

To pass the collected args as separate args to another method call, use **args* :

```
    f = { arg foo ... restArgs;
         foo.value(*restArgs);
    };
    f.value({ | x, y | x@y }, 100, 200);
```

The value of arguments can be changed inside the function code by assignment, just as that of variables:

```
f = { arg bounds;
    // provide default rectangle if not given as argument:
    bounds = bounds ?? { Rect(400, 400, 200, 200) };
    SCWindow("test", bounds).front;
};
f.(Rect(500,500,300,300));  // bounds provided by user
f.value;                    // bounds provided by defaults
```

Partial application: Function construction shortcut with _

Function construction shortcut:

```
    _.squared // construct function { |x| x.squared }
```

Tests (see ! iteration operator below under Loops):

```
    (0,0.1..1) do: _.postln; // this is equivalent to:
    [0,0.1,0.2,0.3,0.4,0.5,0.6,0.7,0.8,0.9,1].do({ arg x; x.postln;});
    // other examples:
    _.squared ! 10;
    _.sqrt ! 10;
    _.linlin(0,9,0,0.5) ! 10;
```

(Here possibly some of the following examples from the Partial Application help file:

```
(1..8).collect([\a, \b, _]);
(1..8).collect {|x| [\a, \b, x] };

(1..8).collect((a:_));
(1..8).collect {|x| (a:x) };

(1..8).collect(Polar(_, pi));
(1..8).collect {|x| Polar(x, pi) };

(1..8).collect((1.._));
(1..8).collect {|x| (1..x) };

f = (a:_, b:_); // f is a two argument function
g = f.(_, 5);   // g is a partial application of f
g.(7);          // get the answer

// equivalent to this:
f = {|x, y| (a:x, b:y) }
g = {|z| f.(z, 5) };
g.value(7);
)
More examples:
     Array.iota(4,5) collect: _[2..]
     Array.iota(3,4) * 10 collect: { |x| Rect(_,_,_,_).(*x) }
```

Iteration: Repeating a function over the elements of a collection

Following messages accept a function as argument, and evaluate that function for every element in the receiver (or, in the case of detect, until the function returns true): `do, collect, select, reject, detect.`

Examples:

```
(
var bar;
bar = (0, pi .. 6 * pi).postln; // make a collection for testing the
examples.
bar.do({|p| [p, p.cos].postln; }); // do: evaluate the function, return the
receiver
// do also works with an integer as receiver - see also repeating a function
10.do({|p| [p, p.squared].postln; });
bar.collect({|p| p.sin; }).postln;   // collect results of iteration and
return them
// select only items that make the function return true:
bar.select({|p| p.cos > 0; }).postln;
// select only items that make the function return false:
```

```
bar.reject({|p| p.cos > 0; }).postln;
// return the first item for which the function returns true:
bar.detect({|p| p.cos < 0; });
)
```

Loops: Repeating a function a number of times

Loops: Repeating a function a number of times, or while a condition is true.

Note: Here we use the "simple" syntax without parentheses around the function argument:

```
// evaluate counting from 0 to n:
4.do { |i| (i.asString++"...").post; };

// repeat with argument ranging from start value to end value
for (1, 5, { |i| (i.asString++"...").post; });
1.for (5) { |i| (i.asString++"...").post; };   // alternative form

// repeat from start to end value, incrementing by step value:
0.0.forBy(0.2, 0.05, { |i| (i.asString++"! ").post; });

// Collect result of evaluation of a function n times
{ 10.rand }.dup(5);
{ 10.rand } ! 5;            // alternative form for dup

// while: this condition is true      ... keep repeating this function
while { (true!3).add(false).choose } { "still going strong ... ".post };

// infinite loop (only useable inside Routine with wait!)
loop { 10.rand.postln; 1.wait }
```

Conditionals: Choosing between alternatives (if, ?, ??, !?, switch, case)

Conditionals enable one to choose whether to do something depending on some condition. There are four such constructs, which operate on different kinds of objects :

if operates on ...
?, !? and ?? operate on ...
switch operates on
case operates on

if-statements

If the receiver is true, evaluate a function. Otherwise, if provided, evaluate another function.

```
if ( [true, false].choose, { "pinpon!" }, { "buuu!" });
// alternative form for the above:
if ([true,false].choose) { "pinpon!" } { "buuu!" };
```

?, !?, ??

Return the first argument if not nil, otherwise return the second argument ("?"), or the evaluation of the second argument (??):

```
(
var thisVar;
thisVar = [nil, 100].choose;
// evaluate and return { ... } only if thisVar is nil:
thisVar ?? { "thisVar needs to be set".postln; };
// if the value of thisVar is not nil then return it, else return 101.
thisVar ? 101;
)
```

switch

case

List comprehensions

A list comprehension is a routine that enumerates all combinations between elements in a number of lists and optionally selects only those combinations which satisfy certain conditions. It also introduces its own "micro-syntax" rules which contains features that can only be used inside the definition of a comprehension. The combination of enumeration and rule application create a very powerful tool for expressing compactly combinatorially complex algorithms.

basic syntax ...

Examples ...

Classes: Templates for making objects

Classes describe the properties (class variables and instance variables) and behavior (methods) of objects. An object created from a class is called an *instance* of that class.

An instance is created from a class by sending the message `"new"`, to that class. Other instance creation messages with different effects are `basicNew` and `newCopyArgs`.

Class definitions must be contained in files ending in `.sc` (or `sc.rtf`) and contained in the folder SCClassLibrary in order to be compiled.

Use Command-K to compile the library.

Classes cannot be compiled from a workspace window

Superclass

Class variables

Class methods

Instance variables

Instance methods

Class definition example

```
/*
     Following defines a Class named TestClass which will be a
     subclass of SuperTestClass.
     SuperTestClass is assumed to be defined somewhere else.
     You must put this code in a file with name ending in .sc
     inside the SCClassLibrary folder and recompile the library with Command-
K
     for the class definition to be compiled.
     You cannot evaluate Class definition code by
     entering it in a workspace window.
*/
TestClass : SuperTestClass { // start of class definition. Names the class
     // and optionally its superclass. Default superclass is Object.
     classvar all;      // declare a class variable
     var myvar;         // declare an instance variable
     // some more variables to illustrate shorthands for creating
     // "getter" and "setter" methods:
     var <readable;     // automatically create method 'readable' for
               // getting the contents of variable readable ("getter")
     var >writable = 1; // automatically create method 'writable_'
     // for setting the value of variable writable ("setter").
     // 1 is set as default value

     var <>read_and_writable;    // create both getter and setter
          // methods for read_and_writable

     *initClass { // define a class method called initClass
          // "*" indicates it is a class method
          // if a method initClass is defined  in a class,
          // it is always executed when the
          // library is compiled, before anything else is run.
          all = Set.new;     // initialize the value of class variable "all"
     }    // end of method initClass

     jump { arg x, y;   // define instance method called jump
          // provide defaults if arguments have not been given
          x = x ?? { 100.rand };
          y = y ?? { x ** 2 };
          myvar = x * y;          // do some computation with the arguments,
```

```
                                   // assign result to myvar
          ^myvar + writable;       // do some more computation,
                                   // return the result.
     //   ^ means return the value of the above statement.
     }    // end of instance method jump

     /* define binary operator & with optional adverb. Usage example:
          TestClass.new &.3 1.2
          The adverb can only pass objects of type Integer or Symbol
     */
     & { | argument, adverb |
          ^(writable + argument) * adverb
     }
     // example of a method that will be redefined by code in another file.
     overwrittenMethEx {
          // return an array containing all and myvar:
          ^[all, myvar]
     }
}         // end of class definition.
```

Extending or modifying a class that has been defined in another file

To add new methods or overwrite existing methods for a Class that has been defined in another file, write class definition code as if you were writing a new class, but:

- add the sign + before the class name
- do not add the superclass after the class name
- additional class variable or instance variable declarations are not permitted

```
+ TestClass { // + signifies that following are additions to an existing
method
     /* one can add new methods, or overwrite existing methods in a class
        but one cannot add instance or class variables */
     // provide a new definition overwriting any previous existing definition
     // of this method:
     overwrittenMethEx {
          "Hello, this method has been modified to just post this
message".postln;
     }
}    // end of additions or modifications to existing TestClass
```

Syntax Elements Reference

This section serves as a compact reference for:

(a) The use of separators ((), {}, []) and certain special characters (*, ;, ., ~ etc.)
(b) Reserverd keywords and programming constructs such as do, loop etc.

It is not an overview of the language and how it works. It does not contain a formal account of the syntax. For an overview of the language see section *Language Syntax Overview*.

Typographic conventions for this section

Italics indicate placeholder elements that will be replaced by some valid expression in the code. Single letters (n, m, o) indicate numeric expressions, placeholders ending in *name* indicate user-defined symbolic names, placeholders ending in *l* indicate lists of elements separated by commata (",").

n.`do { }`	*n* can be any integer numeric value or expression such as `10, 10.rand` etc.
<*varname*>	*varname* can be any valid variable name, such as `freq`.
(*argl*)	*argl* is a list of one or more arguments separated by commata: (1, \freq, \lin, true, x)

Messages

. An object separated by a message indicates sending the message to the object.
object.mname or *object.mname*(*argl*): Send message *mname* to *object* with optional arguments
- Note 1: distinguish from *n.m* where n and m are digit sequences: Create floating point number value
- Note 2: *operator.adverb* syntax is used with Array operators (see Arrays), and can also be defined for use in methods with operator-names in user-defined classes. see for example method `SequenceableCollection-+`

Grouping and precedence

() Parentheses indicate grouping of operator precedence in expressions.
- Alternative usage for convenience: Enclose blocks of code between () to select for entering.
- See also below under ArraySyntax for use as series constructor.
- Also encloses pairs of symbols-values in construction of events

{} Braces define a function. In class definition code they enclose a class or (*methodname* { }) a method

[] Square brackets create a collection of elements - or access an element in a collection
- See also below under ArraySyntax for use as indexing or range indicator.

Separators

Separators

, Separate elements in a collection
Rules: ...

; Separate statements
Rules: ...

Constants and special variables

Special numeric constants or mathematical values: inf, nan, pi
Special constant: nil,

Boolean constants: true, false
Special variables: this, super, thisProcess, thisMethod, thisFunction

Declaration and assignment of variables and arguments

= *varname* = *expression* : Assign value of *expression* to variable *varname*
Multiple Assignment #var1, var2, var3 ... rest = anArray; // assign elements of anArray to variables
... (Ellipsis)
In argument declaration:
arg ... argname; argname receives an array with the values of all arguments passed to the function.
In multiple variable assignment with #:
#var1name ... var2name = anArray;
var1name receives first element of *anArray*, *var2name* receives the rest of the elements in *anArray*
* ***argArrayName* is symmetrical to arg ... *argArrayName* : It passes all values collected in argArrayName as separate arguments in a method call or function evaluation:
changed{|what ... moreArgs| dp.do { |i| i.update(this, what, *moreArgs) }}

Abbreviations and alternative syntax forms

Instance creation abbreviation: *Classname*(*argl*) means: *Classname*.new(*argl*)
.() abbreviation for 'value': *expr*.(*argl*) means: *expr*.value(*argl*)
| | argument definition abbreviation: | *argl* | is equivalent to: arg *argl*;
~ environment variable access abbreviation:
~*varname* is equivalent to: currentEnvironment.at(*varname*)
~*varname* = *expr* is equivalent to: currentEnvironment.put(*varname*, *expr*)
-> Association constructor: *key*->*value* means: Association(*key*, *value*)
@ Point constructor: x@y means: Point(x, y)
? *expr1* ? *expr2* means: if (*expr1*.notNil) { *expr1* } { *expr2* }
?? *expr1* ?? *expr2* means: if (*expr1*.notNil) { *expr1* } { *expr2.value* }
!? ...
: binary operator construction: *obj mname*: *arg* means: *obj.mname*(*arg*)
` Reference constructor: `(*expr*) means: Ref(*expr*), `*value* means: Ref(*value*)
! Abbreviation for dup:. *expr* ! *n* means: *expr* dup: *n* which is equivalent to:
(0..*n*-1) collect: { |i| *expr*.value(i) }
_.*message* abbreviation for { |x| x.*message* }. Arguments also accepted: _.*message*(*argl*)

Class definition elements

: *Classname* : *Superclassname* { *class code* } define new class *Classname* as subclass of *SuperClassname*
- Default superclass is Object.
- Do not confuse with : as binary operator constructor and in Event definition, see below.
* ***methodname* Define class method *methodname*
^ ^*statement* Return the value of *statement* as this method's result i.e. value.

< var <*varname* create method named *varname* that returns the value of variable *varname*
> var >*varname* create method named *varname_* that sets the value of variable *varname*
+ + *Classname* { *class code* } is used to add variables or methods, overwrite methods in class *Classname*

Array operators, Array constructors, operator adverbs

Syntax:	Meaning:
array@*index*	*array*.at(*index*)
array@@*index*	*array*.wrapAt(*index*)
array|@|*index*	*array*.clipAt(*index*)
array@|@*index*	*array*.foldAt(*index*)
array[*index*]	*array*.at(*index*)
array[*indexarray*]	*indexarray* collect: { |i| *array*@i }
array[..*n*]	*array* copyFromStart: *n*
array[*n*..]	*array* copyToEnd: *n*
array[*n*..m]	*array*.copyRange(*n*, *m*)
(*n*..*m*)	Construct arithmetic series from *n* to *m* with increment 1 (or -1 if *m* is smaller than *n*)
(*n*, *m*..*o*)	Construct arithmetic series from *n* to *o* with increment the same as between *n* and *m*
.f, .s, .x, .t	Binary operator adverbs: **f**olded, **s**horter, **x** = cross, **t**able (see J_concepts_in_SC)
.n	Operator depth (see explanation in J_concepts_in_SC)
`object ++ object`	concatenation
+++	Laminate (see explanation in J_concepts_in_SC)

Syntax for Events

(*name1*: *expr1*, *name2*: *expr2*) means: Event[*name1->expr1*, *name2->expr2*]
(for any number of name-symbols and expression pairs)

Arithmetic operators

*, +, -, /, **, %

Binary (bit) arithmetic operators

&
|
<<
>>
+>>

Boolean operators

&& (not inlined), and: (inlined)
|| (not inlined), or: (inlined)

Comparison operators

>
>=
<
<=

`object == object`	equivalence
`object === object`	identity
`object != object`	not equal to
`object !== object`	not identical to

Stream operators

(From the Language help file, SymbolicNotations:)

`stream << object` represent the object as a string and add to the stream

A common usage is with the Post class, to write output to the post window.

```
Post << "Here is a random number: " << 20.rand << ".\n";
Here is a random number: 13.
```

`stream <<* collection` add each item of the collection to the stream

```
Post << [0, 1, 2, 3]
[ 0, 1, 2, 3 ]

Post <<* [0, 1, 2, 3]
0, 1, 2, 3
```

`stream <<< object` add the object's compile string to the stream

```
Post <<< "a string"
"a string"
```

`stream <<<* collection` add each item's compile string to the stream

Set operators

(From the Language help file, SymbolicNotations:)

```
set & set
```
intersection of two sets

```
set | set
```
union of two sets

```
setA - setB
```
difference of sets (elements of setA not found in setB)

```
set -- set
```
symmetric difference

(setA -- setB) == ((setA - setB) | (setB - setA))

```
a = Set[2, 3, 4, 5, 6, 7];
b = Set[5, 6, 7, 8, 9];

a - b
Set[ 2, 4, 3 ]

b - a
Set[ 8, 9 ]

((a-b) | (b-a))
Set[ 2, 9, 3, 4, 8 ]

a -- b
Set[ 2, 9, 3, 4, 8 ]
```

Geometry operators

```
(From the Language help file, SymbolicNotations:)
number @ number
```
x @ y returns Point(x, y)

```
point @ point
```
Point(left, top) @ Point(right, bottom)
returns Rect(left, top, right-left, bottom-top)

```
ugen @ ugen
```
create a Point with 2 UGens

```
rect & rect
```
intersection of two rectangles

```
rect | rect
```
union of two rectangles (returns a Rect whose boundaries exactly encompass both Rects)

Conditionals: *if*, *switch* and *case*

if (cond) { truefunc }
if (cond) { truefunc } { falsefunc }
Alternative forms:
(cond).if({ truefunc }, { falsefunc })
(see also above ?, ??)

switch ...

case ...

Loops

`loop { }`	Loop indefinitely over function { }. (ONLY POSSIBLE IN ROUTINES!)
	Alternative: `loop({ })`
`n do: { }`	Repeat function { } for n times, with argument counting from 0 to n - 1
	Alternatives: `n.do({ }), n.do { } do(n, { })`
`for (n, m, { })`	Repeat { } with argument counting from n to m
`forBy (n, i, m, { })`	Repeat { } with argument counting from n to m and increment i.
`while { cond } { f }`	Repeat function { f } while condition { cond } evaluates to true.

(see also: ! and dup: above)

List comprehensions

all {: }

UGen List and Maps

UGen Category Maps

Signal I/O

Bus I/O

Input	Output
In	Out
InTrig	SharedOut
InFeedback	XOut
LagIn	LocalOut
LocalIn	OffsetOut
SharedIn	ReplaceOut
	OutputProxy
	Silent

Buffer I/O

BufRd
Tap
BufWr
PlayBuf
RecordBuf
TGrains
ScopeOut

Disk I/O

DiskIn
DiskOut

Multichannel

Pan2
LinPan2
Pan4
PanAz

PanB2
PanB
BiPanB2
DecodeB2
Rotate2
Balance2

Periodic Oscillators

Sinusoid

SinOsc
FSinOsc
SinOscFB
PSinGrain
Klang

Var. Shapes

Pulse
Saw
SyncSaw
VarSaw
Blip
Formant

Low Freq.

LFPulse
LFSaw
LFTri
LFCub
LFPar
Vibrato

Wavetable

Osc
OscN
VOsc
VOsc3
COsc

Aperiodic Oscillators

Fixed Spectrum

WhiteNoise
PinkNoise
BrownNoise
GrayNoise
ClipNoise

Variable Spectrum

LFNoise0
LFNoise1
LFNoise2
LFClipNoise
LFDNoise0
LFDNoise1
LFDNoise3
LFDClipNoise

Chaotic

Gendy1
Gendy2
Gendy3
Crackle
Logistic
MantissaMask
Hasher
Latoocarfian
NoahNoise

Impulse generators

Regular

Impulse

Random

Dust
Dust2

Conditional

PulseDivider

Delays

Shared Buffers

AllpassC
AllpassL
AllpassN
DelayN
DelayC
DelayL
Delay1
Delay2

No Buffers

BufCombN
BufCombC
BufCombL
BufAllpassC
BufAllpassL
BufDelayN
BufDelayC
BufDelayL
CombN
CombC
CombL

Filters

Basic

TwoPole
TwoZero
OnePole
OneZero
SOS
FOS
MidEQ
Median

Hi/Lo/Band Pass

HPF
HPZ1
HPZ2
LPF
LPZ1
LPZ2
BPF
BPZ2
APF

Resonant

BRF
RLPF
RHPF
BRZ2
Resonz
Ringz
Klank
Formlet

Control Shapes

EnvGen	Line
XLine	Linen
Phasor	Sweep

User Device Input

KeyState	MouseButton
MouseX	MouseY

Signal Selection

Gate	Select
TWChoose	MostChange
LeastChange	

Non-Spectral Transforms

Amplitude

Mapping
Index Shaper (*s)
WrapIndex DegreeToKey

Adaptive
Normalizer Limiter
Compander CompanderD
LeakDC

Modulo
Wrap Clip
Fold

Over time
Decay Decay2
Integrator TDelay

Change-Rate
Slew Ramp Lag Lag2 Lag3

Frequency
PitchShift

Signal-Rate
A2K K2A

Scale
MulAdd LinLin LinExp

Feature Extraction, Measurement
Pitch
Amplitude
DetectSilence
PulseCount
Latch
LastValue
RunningSum
Peak
PeakFollower
ZeroCrossing
Slope
Trapezoid
InRect
InRange
Timer
Schmidt
PV_JensenAndersen
PV_HainsworthFoote

Synth Inputs
Control LagControl
TrigControl

Physical Models
Ball TBall Spring

Image Ops
ImageWarp XY

Spectral Transforms
PV_BinScramble
PV_MagFreeze
PV_Diffuser
PV_RandWipe
PV_RectComb
PV_RectComb2
PV_RandComb
PV_MagMul
PV_Add PV_Mul
PV_Min PV_Max
PV_CopyPhase
PV_BinWipe
PV_BrickWall
PV_PhaseShift
PV_MagSquared
PV_PhaseShift270
PV_PhaseShift90
PV_MagNoise
PV_BinShift
PV_MagShift
PV_MagSmear
PV_MagAbove
PV_LocalMax
PV_MagClip
PV_MagBelow
IFFT FFT
Convolution
Convolution2
PV_ConformalMap

Discrete Value Generators

Triggered
TExpRand
TIRand
TRand
RandSeed
SendTrig
ToggleFF
Stepper
SetResetFF
Trig1
Trig
TWindex
CoinGate

Demand
Dbrown
Dibrown
Dwhite
Diwhite
Dswitch1
Dxrand
Drand
Dser
Dseq
Dgeom
Dseries
Demand

Constant (I-Rate)
IRand
Rand
RandID
ExpRand
NRand
LinRand

Info UGens

Buffer Info
BufChannels
BufDur
BufSamples
BufFrames
BufRateScale
BufSampleRate

Audio Engine
NumRunningSynths NumBuffers
NumAudioBuses NumControlBuses
NumInputBuses NumOutputBuses
SampleRate ControlRate
SampleDur RadiansPerSample

Node Control
Free Pause FreeSelf PauseSelf
PauseSelfWhenDone FreeSelfWhenDone
Done

UGen Operators (*)
neg, reciprocal, bitNot, abs, asFloat, asInteger, ceil, floor, frac, sign, squared, cubed, sqrt, exp, midicps, cpsmidi, midiratio, ratiomidi, ampdb, dbamp, octcps, cpsoct, log, log2, log10, sin, cos, tan, asin, acos, atan, sinh, cosh, tanh, rand, rand2, linrand, bilinrand, sum3rand, distort, softclip, coin, even, odd, rectWindow, hanWindow, welWindow, triWindow, scurve, ramp, isPositive, isNegative, isStrictlyPositive, rho, theta, rotate, dist, , +, -, *, /, div, mod, pow, min, max, , < , <=, > , >=, , bitAnd, bitOr, bitXor, lcm, gcd, round, roundUp, trunc, atan2, hypot, hypotApx, leftShift, rightShift, unsignedRightShift, ring1, ring2, ring3, ring4, difsqr, sumsqr, sqrsum, sqrdif, absdif, thresh, amclip, scaleneg, clip2, fold2, wrap2, excess, firstArg, rrand, exprand, @ real, imag, | |, &&, xor, not, clip, wrap, fold, blend, , linlin, linexp, explin, expexp

(*) Note: Not all of these operators work with all UGens - I must yet filter out some of these which do not work with UGens. (IZ)

Signal I/O

Periodic Oscillators

Aperiodic Oscillators

Impulse Generators

Delays

Filters

Control Shapes

User Device Input

Signal Selection

Non-Spectral Transforms

Feature Extraction, Measurement

Spectral Transforms

Multichannel

Discrete Value Generators

Synth Input Definition

Node Control

Physical Models

Info UGens

Image Operations

Summary and examples

Further reading