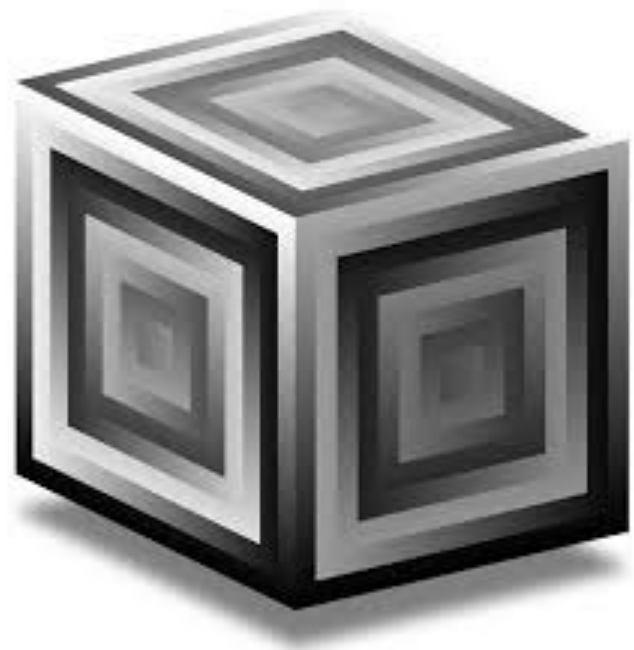
Synth Defs

Topics Addressed

- {}.play
- SynthDefs
- Synth
- Plot Tree
- Nodes/Groups
- Busses
- Order of Execution



{}.play

- We have seen how `.play` method for a function produces sound. But let's unpack what exactly is happening. From the documentation:

> Play a Synth from UGens returned by the function. The function arguments become controls that can be set afterwards.
>
> This works as follows: play wraps the UGens in a SynthDef and sends it to the target. When this asynchronous command is completed, it creates one synth from this definition.

- It turns out that any function called with `.play` gets converted into a SynthDef. The SynthDef gets instantiated on the client-side (sclang) and on the server-side (scsynth). But what is a SynthDef?

SynthDefs

- A SynthDef defines a sound to be played back at a later time. The definition needs to be defined in two places: sclang and scsynth.
  - On the client-side (sclang), we provide a SynthDef a name and a Function which details the UGens used and how they are connected.
  - Once the definition is written, it is converted to OSC (Open Sound Control) messages that are passed along the network (either local or remote) to scsynth where scsynth translates the message into a definition for its purposes.
- Defining a sound is not the same thing as playing a sound in much the same way that defining a function is not the same thing as invoking a function.

The SynthDef on the right is nearly equivalent to the code on the left. All functions with the `.play` method get converted to a SynthDef

Comparison

```
x = {SinOsc.ar(440)}.play;
x.free;
```

- `{}.play` provides a nice shorthand for the SynthDef version on the right to generate sound quickly, often for testing purposes.
- SynthDefs are the preferred way to write sound and what we will be using going forward.

```
(
x = SynthDef.new(\sineWave, {
	var sig = SinOsc.ar(440);
	Out.ar(0, sig);
}).play;
)
x.free;
```

- SynthDefs provide the flexibility of providing a name for the definition.
- Unlike using `{}.play`, SynthDefs need to specify where the signal should be outputted. Much more on this in a bit.
- SynthDefs also provide the flexibility of passing in arguments.

Using .add

- The .play method on a SynthDef actually conflates two separate processes: 1) writing the definition to both the client and server, and 2) invoking an instance of the definition (i.e., a Synth) on the server to play the sound.
- The `.add` method performs the first but does not create the sound. This is actually quite useful, because we generally want to write our definition once but invoke it many times.

```
(
SynthDef(\sineWave, {
	arg freq = 440;
	var sig = SinOsc.ar(freq);
	Out.ar(0, sig);
}).add;
)
```

Here we provide a default argument that we can set with each Synth instance.

Synth

- Once a synth has been added, we can play the sound by creating a Synth object, which is an instance of our SynthDef.
- We can also pass along any arguments to the specific instance by providing an optional array of arguments.
- We can stop the sound using the `.free` method and update arguments through `.set`

```
~synth1 = Synth(\sineWave);
~synth1.free;
~synth2 = Synth(\sineWave, [\freq, 300]);
~synth2.set(\freq, 200);
~synth2.free;
```

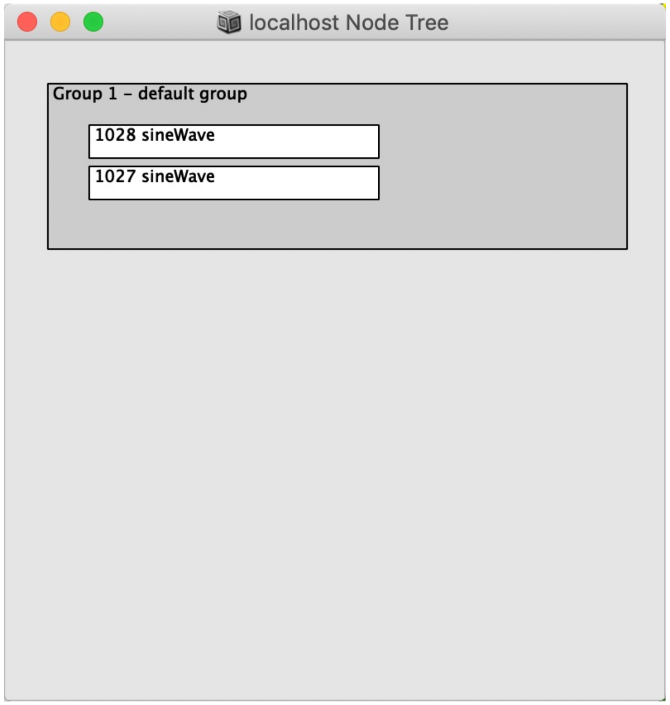
Exercise

Define a SynthDef called `\sqTri` that crossfades between a sine wave and a non-bandlimited triangle wave. Use the UGen `XFade2` to do the crossfade and use a sine oscillator to control the rate of playback between the square and triangle waves. I suggest a frequency of 0.25.

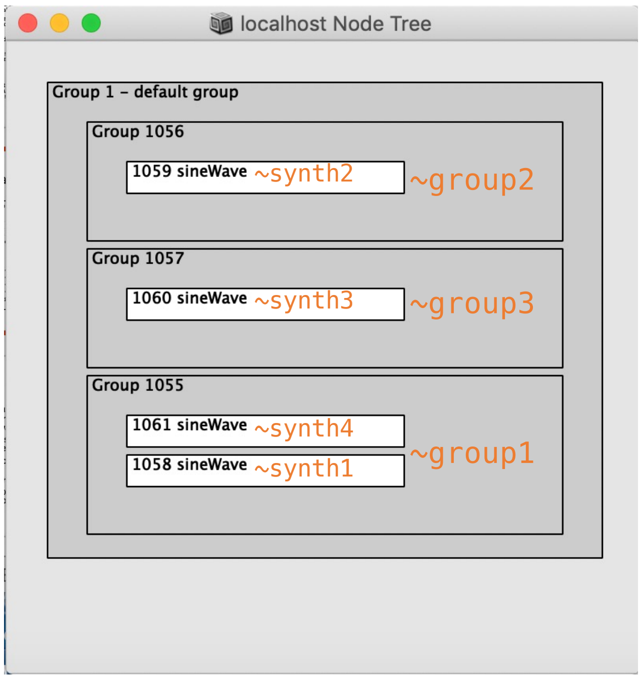
```
(
SynthDef(\sqTri, {
	arg out = 0, freq = 100, amp = 0.2;
	var sine, tri, sig;
	sine = SinOsc.ar(freq);
	tri = LFTri.ar(freq);
	sig = XFade2.ar(sine, tri, SinOsc.kr(0.25));
	Out.ar(out, sig * amp);
}).add;
)
```

Plot Tree

- On the server, allocated synths are organized into groups that can be viewed by calling the `.plotTree` method on the server (usually represented by the variable s).
- By default, all synths are put into the default group. The default group can also contain other groups which can contain any number of synths or groups.
- Groups are useful because you can send the same message to all synths within the group.
- Both groups and synths are called nodes. They both share the parent class Node. Thus, all instance methods for Node will work on groups and synths.
- Calling `.free` on any synth will remove it from the Node Tree. It is important to do so, as this will free up valuable resources on your computer. Freeing is an important concept that extends to many systems in your computer and allows multiple to programs to share resources efficiently.



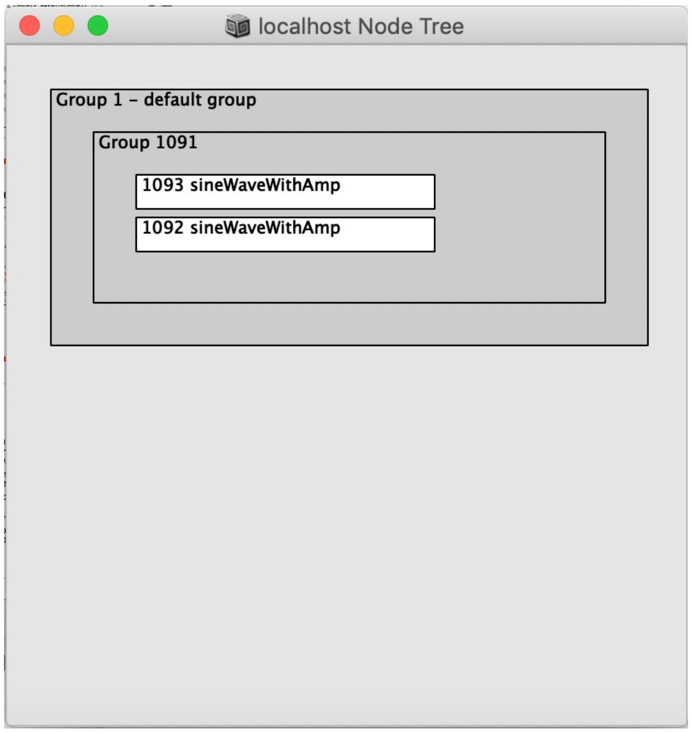
Creating Groups



ORDER OF GROUPS AND SYNTHS MATTER. We shall see why soon.

```
s.plotTree; // Visualize the plot tree
(
~group1 = Group(s.defaultGroup); // Create a group inside
                                 // the default group
~group2 = Group(s); // Does the same things as above.
~group3 = Group(~group1, \addBefore); // The action
                                       // addBefore specifies
                                      // to place above Group 3
)

(
// Add to group1
~synth1 = Synth(\sineWave, [\freq, 200], ~group1);
// Add to group2
~synth2 = Synth(\sineWave, [\freq, 300], ~group2);
// Add to group3
~synth3 = Synth(\sineWave, [\freq, 400], ~group3);
// Add to group of synth1 but before it
~synth4 = Synth(\sineWave, [\freq, 500], ~synth1, \addBefore);
)
```

Power of Groups




```
(
SynthDef(\sineWaveWithAmp, {
	arg freq = 440, amp = 0.5;
	var sig = SinOsc.ar(freq, 0, amp);
	Out.ar(0, sig);
}).add;
)

~group1 = Group(s);
~synth1 = Synth(\sineWaveWithAmp, target: ~group1);
~synth2 = Synth(\sineWaveWithAmp, [\freq, 880], ~group1);

// Send the set message to all nodes in the group!
~group1.set(\amp, 0.1);
```

Busses

- The UGen `Out` specifies a bus index as its first argument. What exactly is a bus? A bus is simply a way to route information. Busses are used in computer hardware and many other electronics, including analog mixers.
- Busses can be used to send signals internally between SuperCollider synths or between hardware inputs/outputs.
- At startup, SuperCollider allocates 1024 busses for audio rate transmission of data as well as control rate busses.
  - Note these defaults can be changed. See ServerOptions.
- The UGen `Out` can be used to send signals to hardware busses to output sounds to your computer speakers or to other SynthDefs on the server.
- Busses can be either control rate or audio rate depending upon the data being sent.

Busses as Handles

- Busses are handles (abstract references to computer resources). In this case, the reference is an integer and the resource is the input/output devices.
  - Other kinds of handles include file descriptors, network sockets, PIDs… etc.
- All SynthDefs need to know where to send their data and therefore must have an `Out` UGen.
  - `Out` takes two arguments: an integer index representing a bus and an array of signals or single signal to output.
  - `{}.play` actually generates an `Out` UGen but this is abstracted away from the user as a convenience.

Bus Organization

- There are two types of busses: control rate and audio rate busses
  - Control Rate busses pass along control rate information from UGens
    - By default SuperCollider allocates 16384 busses for control rate data. This can be checked by running the code `s.options.numControlBusChannels` (assumes that the server is stored in the variable s).
  - Audio Rate busses pass along audio information and are separated into three distinct sections. The total number of audio busses can be found with `s.options.numAudioBusChannels`. By default, that number is 1024.
    - Output devices: bus indices from 0 to `s.options.numOutputBusChannels – 1` are reserved for hardware output devices. The user will generally configure the number of output bus channels at the start of sc file and set the input/output device. If none is provide, the system's default input/output device will be used.
    - Input devices: bus indices starting at `s.options.numOutputBusChannels` for all `s.options.numInputBusChannels` are reserved for hardware input devices. The input device need not match the output device.
    - The remaining busses are 'private' busses intended for internal routing between synths on the server.
    - Example, suppose that we have a two output audio device and a two input audio device and the number of channels has been set properly for both, then bus indices 0-1 would be output, 2-3 would be input and 4-1023 would be private.

Simple Example

```
(
SynthDef(\sineWaveWithAmp, {
	arg freq = 440, amp = 0.5;
	var sig = SinOsc.ar(freq, 0, amp);
	Out.ar(0, sig);
}).add;
)
```

Bus handle of 0 typically means that it will flow to the first output channel on the output device (usually the left speaker)

The signal to be passed into a bus.

Multichannel Busses

- Most audio signals are stereo (2-channel) to accommodate left and right speakers.
- There is no notion of a multichannel bus. Instead audio signals with multiple channels are sent through adjacent bus indices. Each bus supports only a mono signal (i.e., 1 channel).
- `Out` is deceptive in the sense that the user only provides a single integer for a bus index regardless of how many channels constitute the signal. Because signals are adjacent, any array of signals is laid out in contiguous order.

Stereo Signal

```
(
SynthDef(\sineWaveWithAmp, {
	arg freq = 440, amp = 0.5;
	var sig = SinOsc.ar(freq, 0, amp);
	Out.ar(0, [sig, sig]);
}).add;
)
```

Bus handle of 0 is the starting bus index of any n-channel signal.

Multichannel signal. The first sig will be passed through bus index 0 and the second will be passed through bus index 1. Here, left and right speakers.

Connecting Synths

- Synths are connected with three classes: `Out`, In, and `Bus`
- To connect synths, we must use the 'private' busses. The documentation specifically indicates **not** to use hardware input/output busses to connect synths.
- The In class accepts two arguments: an integer index for a bus handle and the number of channels of the incoming signal
  - Note that this integer index represents the first bus index of a n-channel signal
- Bus provides a means to select the first available private busses and assign them names. In general, you should use this class to assign busses.

In/Out example

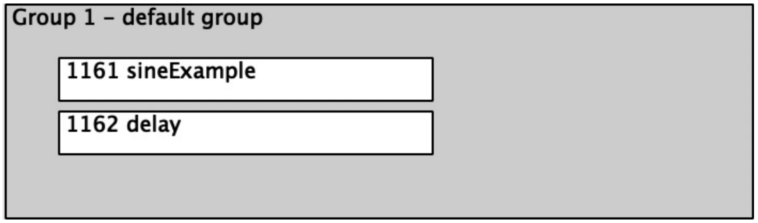
```
SynthDef(\sineExample, {
    arg out = 0, freq = 440, amp = 0.2, ampFreq = 0.5;
    var volume = SinOsc.kr(ampFreq, 3 * pi / 2, amp/2, amp/2);
    Out.ar(out, [Saw.ar(freq, mul: volume),
                 Saw.ar(freq + 1, mul: volume)]);
}).add;

SynthDef(\delay, {
    arg out = 0, in;
    var sig, delaySig;
    sig = In.ar(in, 2);
    delaySig = DelayN.ar(sig, 0.6, 0.6);
    Out.ar(out, sig + delaySig);
}).add;
```

Detuning can “widen” the sound.

Delay the sound by 0.6 seconds and add it to the original sound. More on delays soon!

- Here the sine SynthDef can be played normally just as we have done before. It’s a stereo signal (2-channel). But now we will redirect its output to the delay synth.
- The delay synth needs an `in` argument to specify which starting bus handle to listen on. Note that the variable sig now constitutes a stereo signal.
- Note that we have arbitrarily chosen bus handle 4 to pass our information.

```
a = Synth(\sineExample, [\out, 4]);
r = Synth(\delay, [\in, 4], a, \addAfter);
s.plotTree;
a.free;
r.free;
```



Why is choosing bus handle 4 bad?

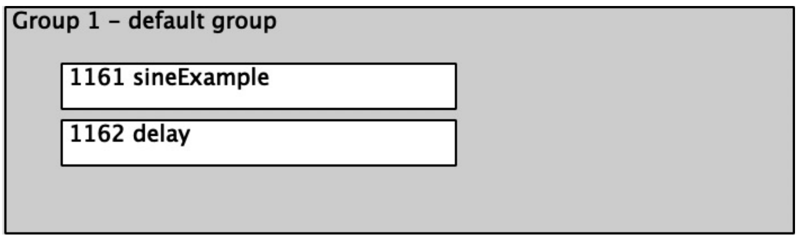
Using Bus

```
(
~delayBus = Bus.audio(s, 2);
"Bus index is ".post;
~delayBus.index.postln;
a = Synth(\sineExample, [\out, ~delayBus]);
d = Synth(\delay, [\in, ~delayBus], a, \addAfter);
)

(
a.free;
d.free;
~delayBus.free;
)
```

- Here we forgo using a hardwired bus handle of 4 and allow sclang to select the first available bus number for us.
- Notice that we have chosen to create an audio bus and that we will be using a 2-channel signal
- In our Synths we will specify the Bus we have chosen as our input and output, respectively.
- Notice also that we free the bus when we are done. Freeing releases the bus handles associated with the bus so they can be used for transferring other information. If your code uses many busses, you could potentially run out of available busses.

Order of Execution

```
a = Synth(\sineExample, [\out, 4]);
r = Synth(\delay, [\in, 4], a, \addAfter);
```



- In the previous slide, we deliberately added the reverb synth **after** the sine synth. It turns out we **must** do this.
- UGens get processed in a top down order by synth on the server. If the delay gets processed before the sine wave does, then no sound will be produced.
- Here, we ensure that the delay gets added after the sine example by specific the sine example as a target and an action of “addAfter”.
- Order of execution matters only for those synths that use `In`
- Groups are useful for solving this issue. Placing sounds in one group and effects in another group ensures a proper order of execution.

```
a = Synth(\sineExample, [\out, 4]);
r = Synth(\delay, [\in, 4], a, \addBefore);
```

This will produce no sound!

More on Order of Execution

- Audio content is generally delivered to DAC or received by an audio application (like SuperCollider) in block sizes called frames (simply another word for a buffer or array of audio data)
  - Frames are important so that the operating system is not taxed with expensive I/O operations for every sample of audio data.
  - The downside is that latency can occur during recording as there is delay due to the time it takes to complete the data for a frame and send it to the application. 64/128 samples for a frame is typical for recording purposes.
- SuperCollider adopts a similar approach, creating blocks of audio data that gets sent to the driver in charge of processing audio data
  - Drivers are low level programs that are used to interface with hardware components in the computer.

More Complicated Example

```
(
~delayBus = Bus.audio(s, 2);
~synths = List[];
[440, 523.25, 659.25, 783.99].do({
	arg freq, index;
	var synth = Synth(\sineExample, [
		\out, ~delayBus,
		\freq, freq,
		\amp, 0.1,
		\ampPhase, index * pi/2,
		\ampFreq, 0.3 - (0.05 * index)
	]);
	~synths.add(synth);
});
~synths.add(Synth(\delay, [\in, ~delayBus], addAction: \addToTail));
)
```

Plays a swelling Am7 chord

Free resources on server →

```
~synths.do({arg synth; synth.free});
~delayBus.free;
```