SuperCollider in Athens [notes01]

MusicN [1] languages are based on the Unit Generators (UGen's), an invention by Max Matthews, and they can be described very well in a Smalltalk-like languages. SuperCollider name is describing the process of "colliding" a real-time sound synthesis engine and high-level garbage collected language. SuperCollider combines a purely object-oriented Smalltalk-like language and features from functional programming languages with a C-like syntax.

Smalltalk [2] is an object-oriented, dynamically typed, reflective programming language, and its main body is based on a large number of predefined classes. Classes are global and starts with an uppercase, and they can call code written in C++ which in SuperCollider could refer to a primitive or a UGen.

1 Basic elements of SC language

- *Classes*: Begins with an uppercase. (eg. `Class`, `Array`)
- *Methods*: Begins with a lowercase. (eg. `.post; .mirror; neg(1);`)
- *Variables*: No need to predefine the type of a variable. Three types of variables: interpreter, local and environmental, as follows.
  - Every lowercase single letter. (eg. `a; d=9; f="{func}";`)
  - Use of `var` identifier. (`var nameOfVariable; var rho=0.05;`)
  - Environmental variables starting with a tidle `~` and lowercase. (eg. `~a=[ ]; ~theta=pi; ~ego = Ego.me;`)
- *Arguments*: They start with a lowercase and they follow the special keyword `arg`, or just included in `| |`. (eg. `arg a=0, beta;` or `|alpha=1, b=2|`)
- *Symbols*: Symbols have a unique representation and they start with a backslash `\`, or included in `' '` quotes. (eg. `\alphabeta; 'gamma';`).
- *Functions*: A function consists within curly brackets `{ }`. (eg. `{|a| a.not;}`)
- *Collections*: Many types of collections. (eg. `[a,1,2]; Set[[2][1]]; List[0,1]`)
- *Strings*: Alphanumeric sequences in double-quotes `" "`. (eg. `"just a word";`)

[1] `http://en.wikipedia.org/wiki/MUSIC-N`
[2] `http://en.wikipedia.org/wiki/Smalltalk`

1.1 Code by example

Execute each line as follows.

```
1; // see post window
1+1;
// lowercase single letters are used as global variables
a;
a=3;
b=5;
c = a+b;
//
c.class;
c.neg; // .neg is a unary operator (ckeckout Implementations of)
/* after mark "SimpleNumber:neg" and press cmd+J, this redirects
   you to the method description in SimpleNumber class. */
c;
c.mod(b) // .mod or % is a binary operator, just as +, -, *, /.
//:-- Function
f.class;
{ }.class;
f = { arg a; a.exp };
f.class;
f.(3);
//:-- Array (see help ArrayedCollection, SequenceableCollection)
[1,2,3,4]*4;
(1..10) // dynamic creation
(1,4..38) // same with step
[1,1,1]+[1,0,0]
[1,1,1]++[1,0,0]
[0,1]+++[2,3]
a=[]; b=[];
a = [a, n, \array, ~of, ~things];
b = [\and_, a, s]; // remember s is for localhost
b.reverse
b
b.at(2)
b.size
b[0] ++ b[2]
//:-- String (see help)
"a usual string"
"a usual string".post
"a".compile.value  // .value is not a method for String class
"a".compile.class
a.class
```

1.2 Classes

`Object` is the root class of all other classes. All objects are indirect instances of class Object. A class name starts with an uppercase and may not involve blank spaces.

The class `Collection` is a *subclass* of `Object`, and it is a *superclass* of `Set`, `Array`, `Dictionary`, among others. Dictionary class is able to do associations between objects.

```
Class.subclasses
Object.superclasses
UGen.dumpClassSubtree;
ArrayedCollection.dumpFullInterface
Environment.superclasses
Routine.superclasses
Post<<*Pattern.subclasses
Post<<*UGen.allSubclasses
Array.dumpMethodList
List.browse // .browse will open a gui class browser
```

A subclass is likely to inherit variables and methods from its superclasses. A class description may involves a superclass, class variables, instance variables, class methods and instance methods. None of these are mandatory, try to create an `Empty { }` class. Save the class file using the *ClassName.sc* file format into the following directory:

```
Platform.userConfigDir +/+ "/Extensions"
```

Now recompile the SC class library and inspect your empty class `Empty.inspect;`.

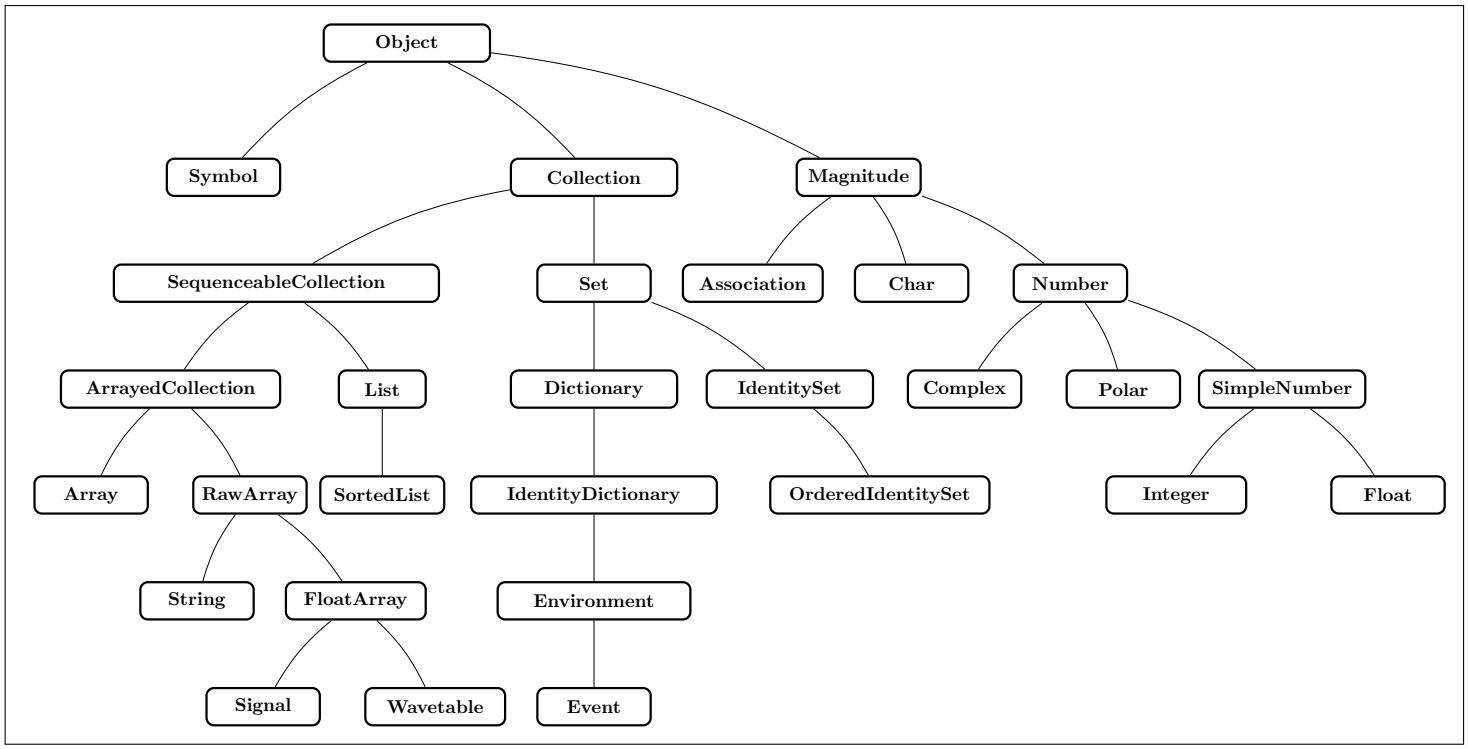

Figure 1: A class tree (see Class Tree help file).

1.3 Methods

A method is called each time a message is sent to the receiver (eg. `2.sin`). We call "receiver" the object that the message is sent to: `receiver.method(argument)`. Also chains of messages is a common practice in SC, as a related product of polymorphism. (eg. `pi.sin.round;`).

Precedence is from left to right.

```
[1,2,3,4].reverse.mirror
reverse(mirror([1,2,3,4]))
```

A method could be a class method or an instance method, where the later operates on an instance of a class and the former operates on a class.

A return value is introduced with a "hat" `^`. It is easy enough to write a simple method as an extension to an existing class, if you follow the previous path [1.2] for saving your *methodName.sc* file.

```
+ Object { addone { ^(this.value + 1) } }
```

Special keyword `this`, refers to the receiver (ie. the object which receives the message). `.addone` method actually adds one to the receiver, extending the Object class.

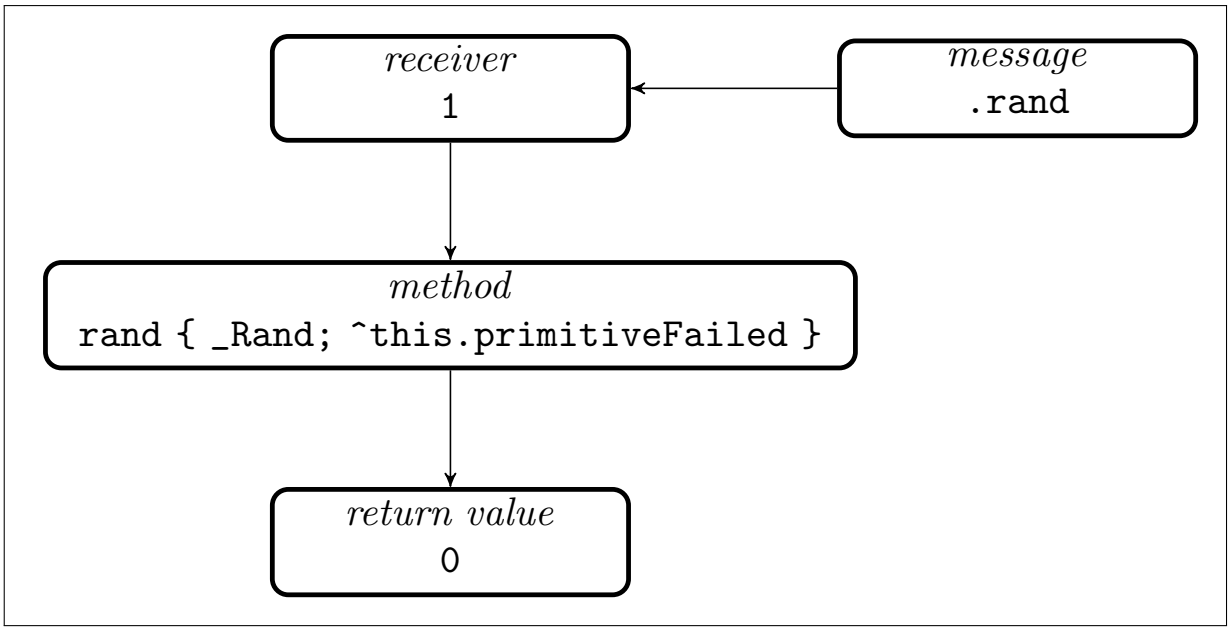

Figure 2: Method flow chart. Diagram adopted from *The SuperCollider Book*, Chap. 5, *Programming in SuperCollider*, by I. Zannos.

1.4 Symbols

A `Symbol` have unique representation in language. May refers to the name of an argument, or to the name of a synth definition (see `Symbol`, `ControlName`, `NamedControl` help files).

1.5 Functions

A `Function` is any valuable code included in curly brackets `{ }`. A function may have arguments and variables. Functions usually include control structures, or they have a significant functionality into bigger structures, such as classes, methods, UGen's and more. The return value of a function is accessible by sending the `.value` message to the receiver (ie. the name of the function).

```
h = { 5.sqrt };
h.value;
h;
g = { |a| a**h.value } // ** === .pow()
g.value(2)
g.(2) // an alternative to set a value to a func
g.(a: 2)
```

1.6 Strings

A `String` may also refers to the name of a synth definition. Strings are alphanumeric sequences within quotes `" "`.

1.7 Collections

There are many types of Collections including `List`, `Array`, `Dictionary`, `Set`, `Bag`, etc. `Array` is a subclass of `ArrayedCollection`. Arrays may contain any type of object, although they are fixed size collections. For more extendable arrays see `List`. Immutable -or literal- arrays start with a hashtag `#`. In literal arrays names are interpreted as symbols. It is a cheaper for cpu to manage literal arrays.

```
Array.with(2, \three, "four");
Array.geom(100,1e5,2).mirror.plot;
[1,2,3,4].put([0,1].choose, nil);
#[foo, bar];
List[1, 2, 3, 4].collect({ arg item, i; item + 10 });
(1..4) collect: [\a, \b, _]
```

2 Messages and Polymorphism

Messages is the standard way to interact with objects. A message always returns a result. The kind of result depends on the kind of message. In the default case, the return value is the receiver itself.

In SuperCollider there are unary, binary and keyword messages. Polymorphism is the ability of different classes to respond to a message in different ways. Check out the related classes for `.abs` and `.compile`.

3 Keywords and symbolic notations

SC has a lot of syntax shortcuts and alternative paths to write your program. Checkout `Symbolic Notations` help file.

```
//special keywords
[true,false,nil]
super
this
inf // inf is floating point infinity
// pseudo-variables
[thisFunction,thisProcess,thisThread]
currentEnvironment
//
"string"     // a string
'symbol'     // a symbol
\symbol // also a symbol
$d  // Char, a single character
//compare
1>2
1<2
1<=1
1>=1
==  // equal
!=  // not equal
=== // indentical (\symbol==='symbol')
!== // not indentical ("symbol"!="symbol")
\symbol === 'symbol'    // symbols are indentical
"string" === "string"   // strings not
//logical
&&  // logical AND
||  // logical OR
xor
or
and // true.and(false)
not // true.not; not(true);
<<  // left shift; ie. 2r10 << 2
//arrays
++  // [1,1] ++ [0,1]
+++ // [1,1] +++ [0,1]
//set a number in another base
2r111       // binary numbers starting with #r
16rA        // hex
//sugar syntaxes
1!5         // same as .dup()
2@3         // create a Point
```

4 Control Structures

Few examples for writing an if statement, a case statement, a do loop and a while loop. See `Control Structures` help file.

```
//if (expr, trueFunc, falseFunc);
if(10.rand>5, { "true" }, { "false" });
if(10.isPrime) { "true" } { "false" }; // alternative syntax

//do (collection, function)
//or collection.do(function)
10.do{ arg i; i.postln; };
(2,4..20) do: { |item, i| "item= ".post; item.postln; "i= ".post;
   i.postln};
do(10, { |i| i.post; });     // all these are alternative syntaxes

//case
(
i = 10.rand;
case
    { i<3 } { "small" }
    { (i>3)&&(i<7) } { "medium" }
    { i>7 } { "big" };
)

//while
(
i=0;
while( { i<5 }, { i=i+1; i.post; } );
)
```

5 UGen

The UGen class provide language side representation of the unit generators [3] available on the server. Their language description is similar to class description, although their are actually defined as plug-ins, written in C++ code. A unit generator can generate or modify audio signals. They are capable to input/output floating point data, in audio-rate (`*ar`), control-rate (`*kr`) and constant-rate (`*ir`). All calculations take place on the server.

```
play{ Blip.ar(9, mul:0.1) };
play{ WhiteNoise.kr(1).rand2.poll };
play{ SinOsc.ar(300,0,LFPulse.kr(1)) };
play{ Blip.ar(9 * LFNoise0.kr([9,8]).abs, mul:0.3) };
```

[3] http://en.wikipedia.org/wiki/Unit_generator

```
{ Pan2.ar(Blip.ar(9, mul:0.1), LFNoise0.kr) }.play;
{ SinOsc.ar(500) * EnvGen.ar(Env.perc,doneAction:2) }.play;
// plotting ugens
{ Sweep.ar(Dust.kr(9), 1) }.plot(duration:1);
{ Sweep.ar(Impulse.kr(9), 1) }.plot(1);
{ LFNoise0.kr }.plot(0.1);
```

5.1 Pseudo UGen

A pseudo-UGen looks like a real UGen , but instead its description is a part of sc-lang (client-side). Make a percussive `SinOsc` using a linear envelope.

```
SinLine {
    *ar { arg freq=440, phase=0.0, amp=0.2, dur=0.1;
        ^SinOsc.ar(freq,phase,amp)*Line.ar(1,0,dur,doneAction:2)
    }
}
```

Save your ugen under the name `SinLine.sc` in your `"Extensions"` directory. Recompile and run the pseudo-ugen.

```
{ SinLine.ar }.play;
{ SinLine.ar(1000) }.play;
{ SinLine.ar(dur:1.0) }.play;
{ SinLine.ar(phase: SinOsc.kr(9), dur:2.0) }.play;
fork{10.do{ wait(0.1); {SinLine.ar()}.play; }}
fork{(10,20..100).do{ |item, i| {SinLine.ar(item*i)}.play; 0.08.
   wait; }}
```

6 SynthDef

A synth definition is responsible to describe the signal flow, each time you send the .play message to a Function (`{}.play`). You may write a SynthDef to your hard disc, play it immediately or you can send it to the server after the compilation of SCClassLibrary.

The name of the SynthDef should be a symbol or a string. Following to the name there is a comma and then you define the function in which you describe the synth definition.

```
// play it immediately
SynthDef(\nameOfSynth, { Out.ar(0, SinOsc.ar(400,0,0.1)) }).play;
// send it to the server
SynthDef(\nameOf, { arg freq=440; Out.ar(0, SinOsc.ar(freq,0,0.1))
    }).add;
Synth(\nameOf, [freq: 300]);
// write it to HD ( to listen to it recompile ! )
SynthDef("synthInHD", { Out.ar(0, LFPar.ar(400)/9) }).writeDefFile
Synth("synthInHD");
```

After you send the SynthDef to the server, you have access into it using a Synth. Synth is complitely different class from SynthDef.

```
SynthDef("nameOf", { arg out, freq=440, amp=0.1;
    var signal, env;
    signal = LFTri.ar(freq,0,amp);
    env = EnvGen.ar(Env.perc, doneAction:2);
    Out.ar(out, Pan2.ar(env * signal));
    }).add;
Synth("nameOf");
Synth("nameOf", [freq: 120, amp:0.4]);
//
6.do{ |item, i| Synth(\nameOf, [freq: 50*i]) };
fork{10.do{ |i| wait(1/i.addone); Synth(\nameOf, [freq: 30*i.
   addone]) }};

// do loops can be implememt into synthdef's
SynthDef(\multiOf, { arg out, freq=440, amp=0.1;
    var signal, env;
    15.do{ |i| signal = Blip.ar(freq/i.addone,i,amp); };
    env = EnvGen.ar(Env.perc, doneAction:2);
    Out.ar(out, Pan2.ar(env * signal));
    }).add;
Synth(\multiOf, [freq: rrand(1000,1e4)]);

// implement a multiple delay
SynthDef(\multiFX, { arg out, freq=440, ffreq=440, amp=1.0;
    var signal, env, fx;
    signal = PMOsc.ar(freq,ffreq,mul:amp);
    env = EnvGen.ar(Env([0,1,0],[0.1,0.5]), doneAction:2);
    5.do{ fx = AllpassN.ar(signal) };
    Out.ar(out, Pan2.ar(env * fx));
    }).add;
Synth(\multiFX, [freq: rand(1400), ffreq: rrand(1050,1400)]);

// make multiple SynthDef's using a function and a do loop
g = { |i|
SynthDef("linen" ++ i, { |freq, att=0.01, sus=1, rel=0.1, amp=0.5|
    var env = EnvGen.kr(Env.linen(att, sus/(i+1), rel),
       doneAction:2);
    Out.ar(0, Formant.ar(freq, freq/i.addone, freq*2, amp*env))
        }).add;
};
// call g func 10 times
10.do(g);
fork{10.do{ |i| Synth("linen" ++ i, [freq: 110*i.addone]);  (0.2).
   wait; }};
```

7 Related help files

Highlight "ClassName" and press *cmd+J* , or Menu > Lang > Open Class Def. For methods, highlight "methodName" and press *cmd+Y*, or Menu > Lang > Implementations

```
"Shortcuts".openHelpFile;
["Class Tree", "Collections"].choose.openHelpFile;
["Operators", "Symbolic Notations"].choose.openHelpFile;
"Control Structures".openHelpFile;
"UGen done-actions".openHelpFile;
```

```
> Classes
    Object
    Class
    Collection
    Array
    ArrayedCollection
    SequenceableCollection
    Function
    Env
    SynthDef
    Synth
    UGen
        Line
        EnvGen
        Pan2
        Blip
        LFPulse
        WhiteNoise
        Dust
        LFNoise0
        Sweep
> Methods
    .plot
    .browse
    .inspect
    .compile
    .superclasses
    .subclasses
    .dumpMethodList
    .dumpFullInterface
    .def.dumpByteCodes
    .reverse
    .mirror
    .size
    .value
    .writeDefFile
    .isPrime
```

8 Exercises

- Write a function which gets an integer value and prints all the prime numbers up to the input value.
- Do the same and write a method (use `^` and `this` which refers to receiver).
- Try to print in post window the synthdef's folder path.
- Make a percussive SynthDef using `Line`, `LFTri`.
- Make a percussive SynthDef using `EnvGen`.
- Make a percussive pseudo-ugen using `LFPulse`. (hint: use `.lag` or `Lag`)
- Make a percussive pseudo-ugen using `Sweep`.
- Make a percussive method (eg. `.envperc`) for UGen's.

References:

Fedora Documentation. n.d. *SuperCollider*, Chap. 11 in *Musicians Guide*. `http://docs.fedoraproject.org/en-US/Fedora/14/html/Musicians_Guide/chap-Musicians_Guide-SuperCollider.html`

Sharp, A. and H. McGraw. 1997. *Smalltalk by Example: the Developer's Guide*. `http://stephane.ducasse.free.fr/FreeBooks/ByExample/`

SuperCollider. n.d. *SuperCollider Help*, Documentation in *Resources*. `http://doc.sccode.org/`

Roads, C. 1996. *Intoduction to Digital Sound Synthesis*. Chap.3, in *The Computer Music Tutorial*. Cambridge, MA: MIT Press.

Wilson, S., Cottle, D. and N. Collins, eds. 2011. *The SuperCollider Book*. Cambridge, MA: MIT Press.

Current notes by Yorgos Diapoulis `[ ydiapoulis@gmail.com ]`

November 13, 2012