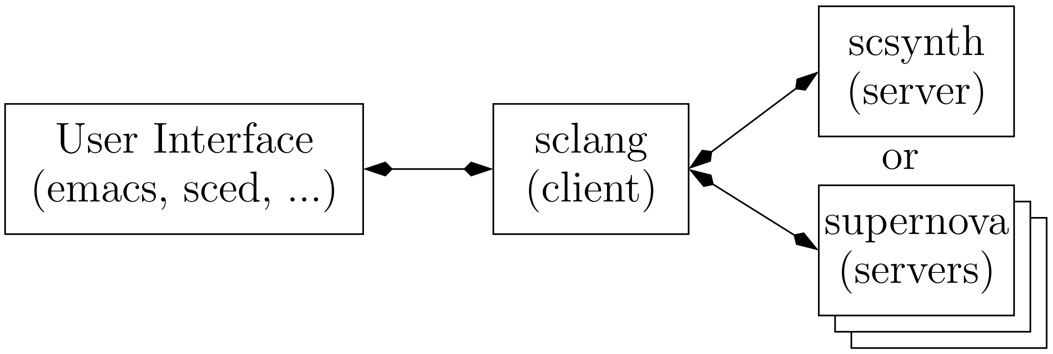
A Bottom-Up Structural Introduction to SuperCollider (Under Development)

Julius O. Smith III

Center for Computer Research in Music and Acoustics (CCRMA)
Department of Music, Stanford University, Stanford, California 94305 USA

jos at ccrma.stanford.edu



Abstract

SuperCollider is a software environment for computer music consisting of a synthesis server (scsynth), language client (sclang), and an extensive set of facilities for musical sound synthesis and event processing. This tutorial introduces SuperCollider in a bottom-up fashion, starting with the synthesis server, then the client-side language, and finally application examples in the spectral audio domain.

1 Outline

```
* A Bottom-Up Structural Introduction to SuperCollider
  - General plan: Bottom-up coverage (no fwd refs) incorporating existing doc by reference
    + Main job is to cover prereqs for next doc
    + If GPL is chosen, can copy and trim existing doc to elim fwd refs
  - scsynth
    block diagram
    Server Architecture
    Server-Command-Reference
    Bus
  UGen
  Node
  SynthDef
  Synth
  Group
  Buffer
- sclang
  Intro to object oriented languages
    Small Talk
    Objective C
    ...
  Overview of SC language: Objects, messages, functions, ...
  Syntax is a blend of object-oriented and functional style
    Examples...
    Note: SC has a lot of "syntactic sugar", i.e., there are various
    alternative ways to specify the same thing.  I personally agree with
    such design, since if it is possible to specify precisely the same
    operation in a more compact form, one ought to be able to write it that
    way, especially if it then becomes easier to read.
    Example: <boolean stuff>
  Maybe mention Ruby
- Follow optimally ordered examples /l/sc/tests.sc
- NRT synthesis first (only need OSC, Array, ...)
- Online tutorials, do examples, look up class docs, more tutorials, SC book
- Numerically ordered SC Class documentation, reworded with illustrations and discussion,
  and with all examples using only things introduced so far (no fwd refs)
  Routine
  Score (calls Routine)
- Review:
  scsynth block diagram
  sclang class hierarchy
- Application Examples
  FFT Overview helpfile
```

2 SuperCollider Architecture

Figure 1 illustrates the general software architecture used when working with SuperCollider.

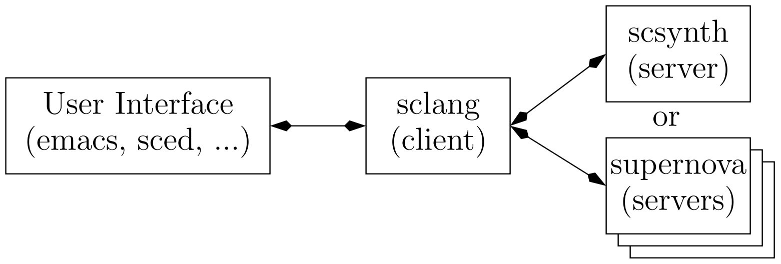


Figure 1: SuperCollider usage architecture

3 SuperCollider Server

The SuperCollider Server (scsynth) is a real-time audio server *not* written in the SuperCollider language (sclang) but instead receives commands from the sclang to configure running processing modules called *unit generators*. In one of the oldest traditions of computer dating back to the 1950s, unit generators are combined into instruments called *synth defs* in SuperCollider.

[To be written - for now, and in addition, see The SuperCollider Book (MIT Press, 2011) for in-depth discussion of scsynth.]

4 SuperCollider Language

There are many tutorials on the SuperCollider language, and plenty of documentation as well. This tutorial attempts to differ from the others (at the time of starting it in 2011) in that it is organized *bottom-up*, *i.e.*, with minimized forward references. As a result, we do not start out with nice interesting code examples containing unexplained constructs. Instead we assume you have already read some of the main tutorials, have run a number of examples, and are now ready to make a bottom-up pass on learning SuperCollider.

5 Learning SuperCollider

There are different types of learning that people tend to use in personal mixtures []. This section describes what works for me. You are free to try my approach or develop your own system. In the current world, deep and rapid self-education is becoming an increasingly important skill—you definitely want to be good at that.

In my experience, the following approach works well:

1. Initial Overview (get the "lay of the land")

2. Try some examples (*e.g.*, following the existing tutorials and help files)

3. Engage in "backwards learning" (discussed below)

4. Read a bottom-up treatment
5. Refer often to the help files in practice, drilling down as needed for reference

...

5.1 Object Oriented Languages

In an Object Oriented language such as Smalltalk, Objective C, C++, or SuperCollider, objects may be viewed as "characters" in a book or play. When we read a novel, we expend considerable time near the beginning getting to know the characters. At first, their behavior may appear somewhat random, but after we get to know them—after the characters are fully "developed"—their behavior becomes natural and *predictable*. One measure of the quality of a computer language is its *predictability*. That is, after you learn the basics of a good language, you should be able to guess how to express concise, powerful constructs that you never explicitly read about, and be right. This means, in part, knowing the characters well and how they are organized into teams—you can correctly guess "who" is in charge of specific tasks, and how one should work with them.

"Behavior" in an object-oriented (OO) langage results from *sending messages to objects*. An OO language usually also supports other, more traditional software elements, such as assignment, function evaluation, and so on. However, it is the message-passing among objects that is the object-oriented behavior in the language. In our play analogy, the characters receive "messages" sent to them like spoken dialogue. Breaking from our play analogy, objects do not typically respond to a message with another message, as in true dialogue (although any number of messages may be emitted to various objects in response to a received message). Instead, messages tend to be one-way, delivered by executing a line of code. If there is any return value from the object in response to the message, it is usually a returned object reference.

5.2 Example: The Array Object in SuperCollider

In SuperCollider, we can create and populate an *array* of objects as follows (from '//' to the end of the line is a *comment*):

```
a = Array.new(3); // 3=size, a = [] (empty Array)
a.add(1); // Now a is the Array [1]
a.add(2); // Now a is the Array [1,2]
a.add(3); // Now a is the Array [1,2,3]
```

The first line assigns the reference returned by `Array.new(3)` to the variable `a`. `Array` is a "class name" which stands for the *factory object* which creates instances of the class. When the `new` message is sent to a factory object, an instance of the class is created and returned as an object reference. Here we assigned that object reference to the variable `a`. In this case, our `new` message had a single argument, which is the initial size desired for the array.

SuperCollider is full of "syntactic sugar," meaning relatively concise and convenient ways of expressing things. For example, we can create `a` as above in the following shorter form:

```
a = [1,2,3];
```

In SuperCollider, square brackets around any list of object-references denotes an array:

```
a = [1,\foo,"bar baz"]; // Array of an integer, symbol, and string
```

In this example, 1 becomes an instance of Integer,
`foo` becomes an instance of Symbol, and `''bar baz''` becomes and instance of String. To check this, you can ask any object its class:

```
1.class          // => Integer
\foo.class       // => Symbol
"bar baz".class  // => String
```

6 Modifying Objects

In the obove Array example, suppose we

```
b = a.add(4); // Now a is [1,2,3] while b is [1,2,3,4]

a = [1,2,3]; // Array of three Integers
a.class // => Integer
b = a.add(4); // Now BOTH a AND b are [1,2,3,4]
```

Perhaps what you meant:

```
a = [1,2,3]; // Array of three Integers
b = a.deepCopy; // b is now a copy of a
b.add(4);    // [1,2,3,4]
a;           // [1,2,3]
```

The first line is a simple *assignment* to the variable `a`. The right-hand side instantiates an Array object initialized to [1,2,3].

This example illustrates two messages to the object `a` (an Array). The first message, `class`, results in the class name Integer being printed in the SuperCollider “post window”. The second message, `add`, has an *argument* 4, so the message to `a` is “add the Integer 4 to yourself,” resulting in `a` becoming the longer Array [1,2,3,4].