Introduction to SuperCollider workshop, Notam 2019

Program for the day

1. An overview: What is SuperCollider and what can you do with it?
2. The design and architecture of SuperCollider
3. Language basics: syntax, variables and expressions
4. Functions
5. Learning resources: How to proceed from here

Who am I?

- Mads Kjeldgaard
- Composer & developer
- Work at NOTAM

Contact info

Me

- website: madskjeldgaard.dk
- github: github.com/madskjeldgaard
- email: mail@madskjeldgaard.dk

Notam

- notam02.no: notam02.no
- Instagram: @notam02
- Twitter: @notam02
- Facebook: Notam02

SuperCollider meetups in Oslo

- Monthly SuperCollider meeetups at NOTAM
- Superduper friendly and fun (+ often has cake)
- Next one is September 9th, 2019
- Not in Oslo? Start your own meetup group!

See the
SCOslo
group for more info

What is SuperCollider?

SuperCollider is a platform for audio synthesis and algorithmic composition, used by musicians, artists, and researchers working with sound

It is free and open source software available for Windows, macOS, and Linux.

Why SuperCollider?

- Open source and free
- 20+ years of development
- Efficient, robust and stable
- Incredibly flexible
- Cross platform
- Unique design concepts and features
- Text based -> fast
- Big community

What is SuperCollider used for?

- Composition
- Sound synthesis
- Live coding
- Improvisation
- Networked performances
- Installation
- Dance / theater work
- Immersive sound

Examples

- Roosna and Flak (Dance performance)
- Verdensteatret (Theater performance)
- Renick Bell (Livecoding)
- Streifenjunko (Improvised music)

Design

Short history of SuperCollider

SC was designed by James McCartney as closed source proprietary software

Version 1 came out in 1996 based on a Max
object
called Pyrite. Cost 250$+shipping and could only run on PowerMacs.

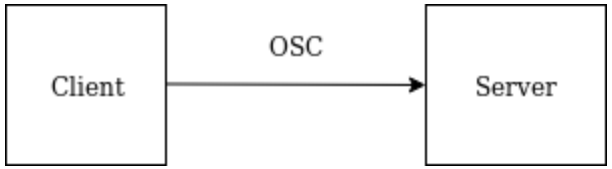
Became free open source software in 2002 and is now cross platform.

Overview

When you download SuperCollider, you get an application that consists of 3 separate programs:

1. The IDE, a smart text editor
2. The SuperCollider language (sclang)
3. The SuperCollider sound server (scsynth)

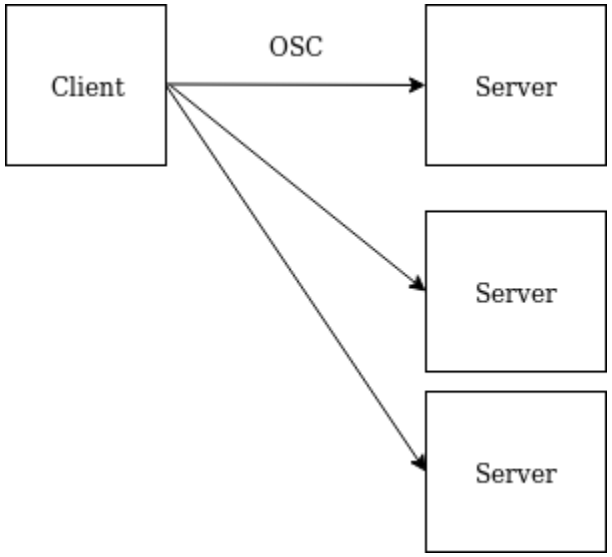
Architecture



The client (language and interpreter) communicates with the server (signal processing)

This happens over the network using Open Sound Control

Multiple servers



This modular / networked design means one client can control many servers

Consequences of this modular design

Each of SuperCollider's components are replacable

IDE <---> Atom, Vim, or Visual Studio

language <---> Python, CLisp, Javascript

server <---> Max/MSP, Ableton Live, Reaper

Extending SuperCollider

The functionality of SuperCollider can be extended using external packages

These are called Quarks and can be installed using SuperCollider itself

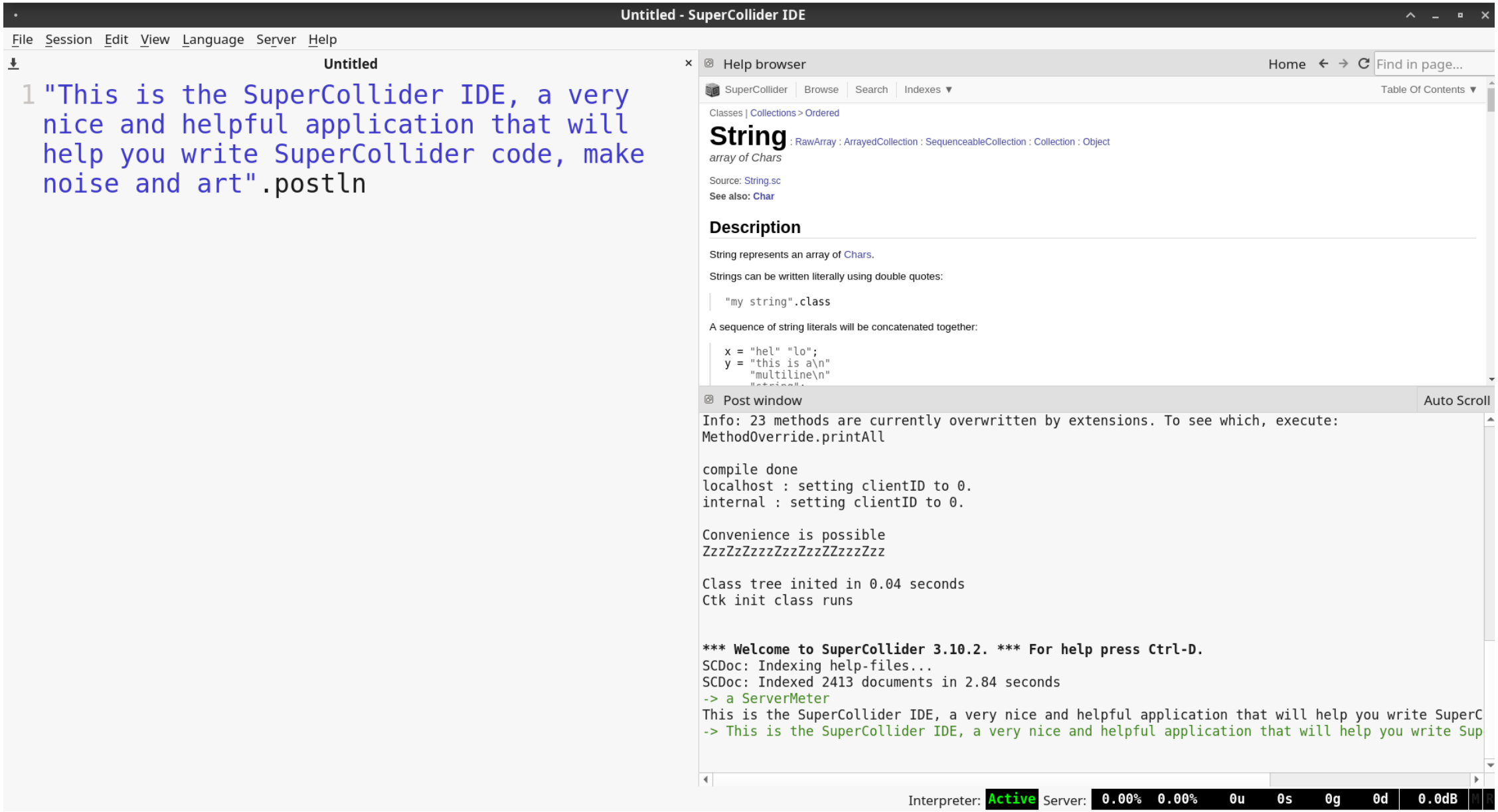
```
// Install packages via GUI (does not contain all packages)
Quarks.gui;

// Install package outside of gui using URL
Quarks.install("https://github.com/madskjeldgaard/KModules");
```

SC Plugins

SC3 Plugins is a
collection of user contributed code, mostly for making sound

The plugins are quite essential (and of varying quality / maintenance)

IDE


Untitled - SuperCollider IDE
File Session Edit View Language Server Help
Untitled
1 "This is the SuperCollider IDE, a very nice and helpful application that will help you write SuperCollider code, make noise and art".postln
Help browser
Home
Find in page...
SuperCollider Browse Search Indexes ▼
Table Of Contents ▼
Classes | Collections > Ordered
String : RawArray : ArrayedCollection : SequenceableCollection : Collection : Object
array of Chars
Source: String.sc
See also: Char
Description
String represents an array of Chars.
Strings can be written literally using double quotes:
"my string".class
A sequence of string literals will be concatenated together:
x = "hel" "lo";
y = "this is a\n"
"multiline\n"
Post window
Auto Scroll
Info: 23 methods are currently overwritten by extensions. To see which, execute:
MethodOverride.printAll
compile done
localhost : setting clientID to 0.
internal : setting clientID to 0.
Convenience is possible
ZzzZzZzzzZzzZzzZZzzzZzz
Class tree inited in 0.04 seconds
Ctk init class runs
*** Welcome to SuperCollider 3.10.2. *** For help press Ctrl-D.
SCDoc: Indexing help-files...
SCDoc: Indexed 2413 documents in 2.84 seconds
-> a ServerMeter
This is the SuperCollider IDE, a very nice and helpful application that will help you write SuperC
-> This is the SuperCollider IDE, a very nice and helpful application that will help you write Sup
Interpreter: Active Server: 0.00% 0.00% 0u 0s 0g 0d 0.0dB

Important keyboard shortcuts

- Open help file for thing under cursor: **Ctrl/cmd + d**
- Evaluate code block: **Ctrl/cmd + enter**
- Stop all running code: **Ctrl/cmd + .**
- Start audio server: **Ctrl/cmd + b**
- Recompile: **Ctrl/cmd + shift + l**
- Clear post window: **Ctrl/cmd + shift + p**

The IDE as a calculator

SuperCollider is an interpreted language

This means we can "live code" it without waiting for it to compile

A good example of this is using it as a calculator

Autocompletion

Start typing and see a menu pop up with suggestions (and help files)

The status line

Shows information about system usage

Right click to see server options + volume slider

Syntax, strings and variables

Hello world

Use `.postln` to post something to the post window (important when debugging):

```
"Hello world".postln
```

An important point on numbers in SC

As opposed to mathematical convention: there is no hierarchy between operators

If you pick up a calculator and type `2+2*10` the result is probably `=22`

Because normally there is an implicit parenthesis here: `2+(2*10)`.

This isn't the case in SuperCollider:

```
2+2*10
-> 40
```

Using brackets to create mathematical hiararchy

SC looks at the first part (`2+2`) and calculates it, then multiplies it (`*10`).

Therefore: Always use parenthesis when you need mathematical hierarchy:

```
2+(2*10)
-> 22
```

Syntax

Like with any other programming language, correct syntax is important.

When you get it wrong, the interpreter will give you an error (and thus help you solve your problem)

If for example I wanted to write `9.cubed` but accidentally wrote `9cubed` and evaluated it, I would get the following error

```
RECEIVER: nil
ERROR: syntax error, unexpected NAME, expecting $end
  in interpreted text
  line 1 char 6:
  9cubed
   ^^^^^
-----------------------------------
ERROR: Command line parse failed
-> nil
```

Brackets / parenthesis

`()` encapsulates a block of code that is supposed to be executed together `;` is used to mark the end of a statement

An example of a block:

```
(
a = 111+222+333;
b = 444+555+666;
c = 777+888+999;
)

a; // -> 666
b; // -> 1665
c; // -> 2664
```

Expressions

The end of an expression is marked by a semicolon `;`

SC will interpret everything up until the semicolon as one expression

Example: Two expressions

```
"hello".postln; "how are you?".postln;
```

This results in the following in the post window:

```
hello
how are you?
-> how are you?
```

Receiver notation

A way of executing a function (message) on an object (receiver)

```
Receiver.message(argument)
```

or

```
message(Receiver, argument)
```

Receiver notation examples:

`100.rand` same thing as `rand(100)`

`"hello".postln` same thing as `postln("hello")`

`0.123.round(0.1)` same thing as `round(0.123, 0.1)`

The interpreter doesn't care about line breaks

```
(
a = [1,2,3,4];
)
```

Is the same as this:

```
(
a = [
    1,
    2,
    3,
    4
    ];
)
```

As long as you use semicolons at the end of your expressions

Comments

`//` can be used as single line comments:

`// This comment is a one line comment` Or at the end of a line:

`10+10; // This comment is at the end of a line`

`/* */` is used for multiline comments. Everything between these is treated as a comment.

```
/*
Roses are red
Violets are blue
SuperCollider is cool
and so are you
*/
```

Strings

A string is marked by double quotes: `"This is a string";`

It is now a String object:

```
"This is a string".class
-> String
```

String concatenation

A common string operation is the concatenation of strings

This is done using the `++` operator:

```
"One" ++ "Two" ++ "Three";
-> OneTwoThree
```

Symbols

A symbol can be written by surrounding characters by single quotes (may include whitespace):

```
'foo bar'
```

Or by a preceding backslash (then it may not include whitespace):

```
\foo
```

Why symbols

From the Symbol help file: "A symbol, like a String, is a sequence of characters.

Unlike strings, two symbols with exactly the same characters will be the exact same object."

Symbols are most often used to name things (like synthesizers, parameters or patterns)

Tip: Use symbols to name things, use strings for input and output.

Variables

A variable is a container that you can store data in:

```
var niceNumber = 123456789;
```

Variable names

Variable names must be written with a lowercase first letter.

Like this: `var thisWorks` and not like this: `var ThisDoesNotWork`

Reserved keywords

Another limitation in naming variables: Reserved keywords

These are words used to identify specific things in SC: `nil`, `var`, `arg`, `false`, `true`

Example:

```
var var
-> nil
ERROR: syntax error, unexpected VAR,
    expecting NAME or WHILE
  in interpreted text
  line 1 char 7:
  var var
      ^^^
```

Local variables

Local to a block of code

Must be initialized at the top of the block

Environment variables

"Global" in scope, can be accessed throughout the environment

Don't need a `var` keyword in front of them when declared

Can be initiliazed at any point in the program

Writing environment variables

1. The letters a-z: `a = [1,2,3,4,5,6]`
2. The tilde (~) prefix `~array = [1,2,3,4,5,6]`

Demonstration of variable scope

```
(
// local variable
var array = [1,2,3];

// This works:
array.postln;
)

// This returns a "not defined" error:
array.postln;
```

Syntactic sugar

SC allows the user to write code in different styles using different types of syntax.

The helpfiles Syntax
Shortcuts and
Symbolic
Notation can be
a big help when this becomes confusing

Functions

What is a function?

A function is a reusable encapsulation of functionality

Lets you reuse and call it elsewhere in your code

Repetitive code can often be simplified with functions

Functions

The core of the function is contained in curly brackets: `{}`

We declare a function like this. Note: This does not evaluate or activate the function yet:

```
{2+2}
-> a Function
```

A function is evaluated by sending it the `.value` message:

```
{2+2}.value
-> 4
```

Syntactic sugar

Tip: `.value` can be omitted by just adding `.()` like so: `{arg x, y; x+y}.(x:2, y:7)`, although `.value` is usually clearer

Function arguments

Functions can take arguments (data) as input and do something with them.

Arguments must be declared in the beginning of the function.

To pass values to the arguments, open a parenthesis after `.value`

Here we have named the argument `x`

```
{arg x; 2+x}.value(x: 8)
-> 10
```

Alternatively, the argument name can be omitted (but then you have to know the order of arguments):

```
{arg x, y; x+y}.value(2, 8)
-> 10
```

Named

You can call arguments by their names:

```
{arg x, y; x+y}.value(x:2, y:8)
-> 10
```

Mixing named and unnamed arguments

You can mix named and unnamed arguments but you must call the unnamed arguments at the end of the list

correct way:

```
{arg x=2, y; x+y}.value(2, y:8)
-> 10
```

incorrect way:

```
{arg x=2, y; x+y}.value(x:2, 8)
ERROR: syntax error, unexpected INTEGER, expecting ')' in interpreted text
```

Alternative argument syntax

Instead of writing `arg argname1, argname2` you can put the arguments inside pipe symbols:

```
f = {|x, y| x+y}
```

Argument default values

You can set the initial value of an argument when declaring it:

```
f = {|x=1, y=4| x+y}
```

Declaring multiple arguments or variables in one go

You can choose between declaring like this:

```
arg argument1, argument2, argument3;
```

Or like this:

```
arg argument1;
arg argument2;
arg argument3;
```

The same goes for local variables

Functions can be put in variables and reused

```
f = {arg x, y; x + y};
f.value(2,1000); // = 1002
f.value(9,22); // = 31
```

Function returns

All blocks of code in SC return the result of the last statement (in both `()` and `{}`)

This is useful for doing further computations

```
f = {arg x, y; x + y};
a = f.value(2,1000); // = 1002
b = f.value(9,22); // = 31
a+b; // = 1033
```

Learning resources

Videos

Tutorials by Eli Fieldsteel covering a range of subjects: SuperCollider Tutorials

Books

E-books

- A gentle introduction to SuperCollider
- Thor Magnussons Scoring Sound

Paper books

- Introduction to SuperCollider, Andrea Valle
- The SuperCollider Book

Community

- scsynth.org
- sccode.org
- Slack
- Lurk
- Mailing list
- Telegram
- Telegram ES
- Facebook

Awesome SuperCollider

A curated list of SuperCollider stuff

Find inspiration and (a lot more) more resources here:

Awesome
Supercollider

Learning to code: Advice

- Practice 5 minutes every day
- Set yourself goals: Make (small) projects
- Use the community