INSTITUTE OF SONOLOGY

Royal Conservatoire The Hague

Syntax

Programming and Music

http://www.bjarni-gunnarsson.net

Programs and Programming

A **program** consists of a set of **instructions**.

A computer executes these instructions to complete tasks set out for the program.

New computer operations can by composed by defining **combinations** of old operations.

Defining **new operations** and **combining** them to do useful things is what programming is about.

Programs and Programming

Programming is usually aimed at finding ways to **solve a specific problem**.

This includes **analysis** of the problem, **implementation** of the solution and **testing** of the correctness of the program.

Programs are created using **programming languages**.

Programming languages

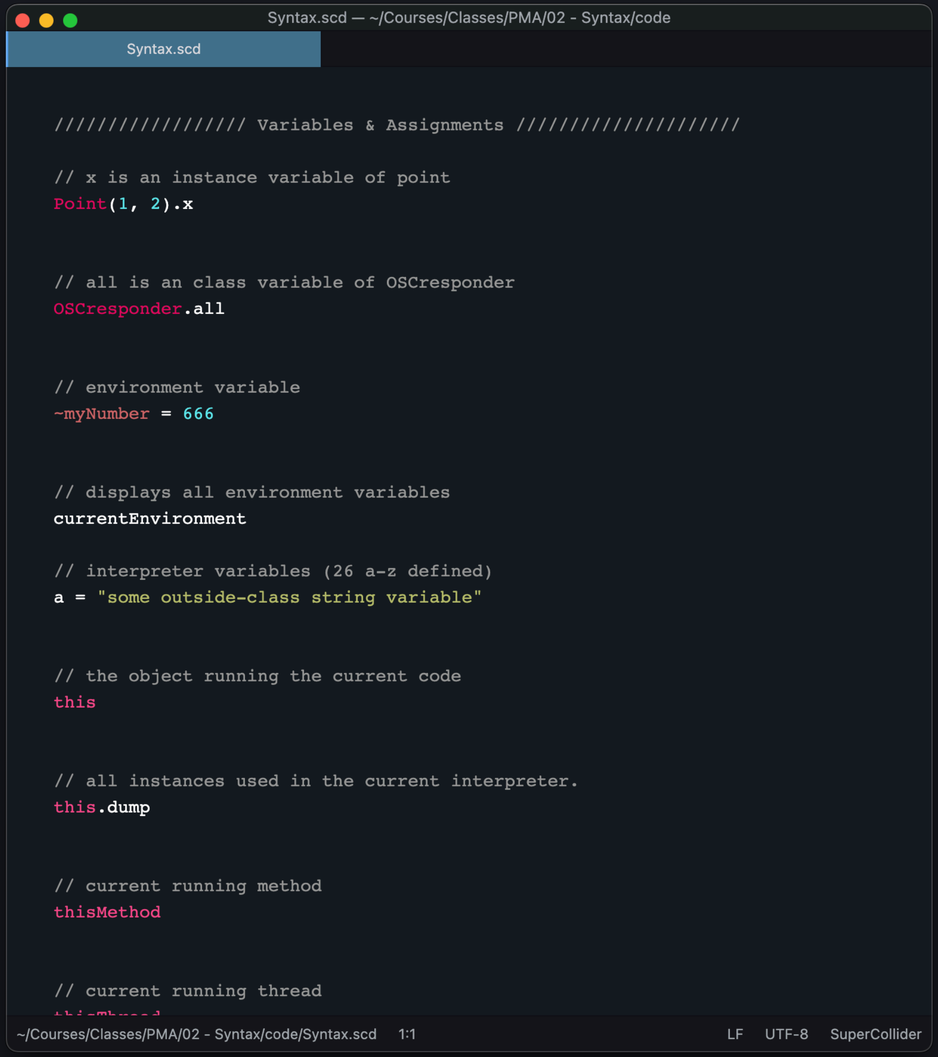
A **programming language** is a formally constructed language used to specify instructions to a computer.

It presents an abstract model of **computation** where issues that are not relevant to the **problem domain** of the program are excluded.

The better a language is in providing **useful abstractions**, the less one has to worry about the language and can focus more on the task to be accomplished.

Programming languages

SuperCollider is a **dynamically typed**, **single-inheritance**, **garbage-collected** and **object-oriented** language created with musical applications in mind.


Syntax.scd — ~/Courses/Classes/PMA/02 - Syntax/code
Syntax.scd
~/Courses/Classes/PMA/02 - Syntax/code/Syntax.scd 1:1
LF UTF-8 SuperCollider

```
////////////////// Variables & Assignments /////////////////////

// x is an instance variable of point
Point(1, 2).x


// all is an class variable of OSCresponder
OSCresponder.all


// environment variable
~myNumber = 666


// displays all environment variables
currentEnvironment

// interpreter variables (26 a-z defined)
a = "some outside-class string variable"


// the object running the current code
this


// all instances used in the current interpreter.
this.dump


// current running method
thisMethod


// current running thread
```

Programming with SuperCollider

Objects

SuperCollider is a pure **object-oriented** programming (OOP) language, meaning all entities inside the program are some kind of **objects**.

Objects are the basic entities of the language, they bundle data and and methods that act on that data.

Two basic types of objects exist, objects with a **fixed** slot of data and object with a **dynamic** slot of data (collections).

Objects belong to a **class** which is a of blueprint of the object, it describes its attributes and methods.

Objects belonging to a class are called **instances** of that class.

Objects

A class can **inherit properties** and **methods** from another class (its superclass) and then becomes its subclass.

To **interact** with an object one sends it a **message**.

The **receiver** is the one receiving the message. It looks up its own **implementation** corresponding to the message and then produces a return value executing that implementation.

There exist **instance methods** and **class methods** (such as new).

Instance methods are more common in SuperCollider.

Arguments

Messages to objects come with **arguments**, these are either instances of other objects or literals.

If there are several arguments provided, they are separated by commas.

Default values for arguments can be set so that one does not have to set all arguments each time a message is passed.

The **argument keyword** can be used to target a specific argument in the list of all possible ones.

Objects

```
p = Point.new(1, 2)

e = Env([0,1], [1])

a = [1,2, "this is an array"]

r = Rect(2, 4, 6, 8)
r.top
r.moveTo(10, 12)

"reverse it and convert to upper".toUpper.reverse

m = Scale.major()
m.degrees()
```

Arguments

```
{ SinOsc.ar }.play

{ SinOsc.ar(200, 0, 1, 0) }.play

{ SinOsc.ar(freq: 200, mul: 0.1) }.play

Array.series(10, 5, 2)

Array.series(*[10, 5, 2])

10.do({'programming'.postln})

10.do{'programming'.postln}
```

Expressions and Statements

An **expression** consists of values and operators for example: 1 + 2.

Expression are evaluated for making calculations and **producing a value** that is a result of the evaluation.

A **statement** is the smallest standalone element expressing an action to be carried out.

Programs are created by sequences of one or more statements.

Statements in SuperCollider are separated by **semicolons** ;

Expressions and Statements

```
2 * 4

"sono" ++ "logy"

["composing", "music"].choose

x = [1, 2, 3, 4].rotate(1);

if(1.0.rand >= 0.5,
{"0.5 or higher"}, {"lower than 0.5"})
```

Variables

A **variable** is a **storage location** and an associated **identifier** containing a value used in a program.

Variable names usually consist of letters, digits, and the underscore symbol.

Variables usually contain **values** or **result of evaluated expressions**.

Variables can be thought of as **boxes with labels**.

Variables

Several **variables** can be created in **one statement**.

Variables must be **declared** at the beginning of a function.

An **empty** variable has the value **nil**.

Variable Types

Different kind of variables exist in SuperCollider such as:

* **Global variables** (available everywhere)

* **Function variables** (available within a function)

* **Class variables** (available to a class)

* **Instance variables** (available with an instance of a class)

* **Pseudo variables** (provided by the compiler, this or thisProcess)

***Reference** variables also exist which reference a variable container.*

Assignments

Single assignment

The value of an expression on the right hand side is assigned to a variable on the left hand side. <variable> = <an expression>

Multiple assignment

Assigns the elements of a Collection which is the result
of an expression on the right hand side, to a list of variables on the left hand side. <list of variables> = <expression>

Series assignment to a list

A syntax for doing assignments to a range of values in an ArrayedCollection or List. <variable> = (<start>, <step> .. <end>)

Assignments

```
Point(1, 2).x

OSCresponder.all

~myNumber = 666

this

c = 2 + 4;

# a, b, c = [1, 2, 3, 4, 5, 6];

a = (0..10);
```

Operators

An **operator** is a program element that is applied to one or more **operands** in an expression or statement.

Operators that take one operand, such as the inversion operator (neg) are referred to as **unary** operators.

Operators that take two operands, such as arithmetic operators (+,-,*,/), are referred to as **binary** operators.

Operators

Operator **precedence** is determined by **order** and **parentheses**.

SuperCollider supports **operator overloading**.

Operators can thus be applied to a variety of different objects for example: Numbers, Ugens and Collections.

Comments

To describe code it is helpful to write **comments** so that when one reads it again, all explanation and detail is available and easy to grasp.

SuperCollider supports **single** and **multiline comments.**

```
// single line comment

/*
    multi
    line
    comment
*/
```

Operators and Comments

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

((1 + 2).asString)

1 + (2 * 2)

0.444.neg

(1.0.rand).min(0.8)

// single line comment

/*
    multi-line
    comment
*/
```

Literals

Every value in SuperCollider has a specific **object type**.

A type determines what operations can be applied to the value.

SuperCollider is dynamically typed so variable types are usually determined during **run-time**.

Variables having a direct direct syntactic representation are named **literals**.

Literals

The following literals exist:

* **Integers**: 8, -1, 666

* **Floats**: 0.25, -25.89

* **Strings**, “Hello Sonology”

* **Symbols**, ‘lecture’

* **Characters**, $a

* **Special**, true, false, nil

* **Literal Arrays**, #[1, 2, ’abc’, "def", 4]

Type introspection

Where the type of an object is unknown **type introspection** can be used to determine the type or which operations it supports.

This is important to do when objects with an unknown type are handled since the program can use the introspection to reason how to treat a specific object instance.

In SuperCollider it is possible to **query** an object about its class, its methods and if it is an instance of a specific class.

Literals and Types

```
a = 1.2
a.class

s = "don't use s, it's for the server!"
s.class

a = #["array", "that", "can't" "be", "changed"]

Array.dumpInterface

a = 'something'
b = "anything"

a.class
b.dump
a.isKindOf(Symbol)
```

Next Steps

Download the code for the class and try to get a solid understanding of the topics covered so far.

Execute some expressions and try to store the results in variables.

Experiment with different ways of assigning variable values.

Try to compose arithmetics using operators and values.

Experiment with operator precedence using parenthesis.

```
(
    Ndef(\iop, {|freq=78, mul=1.0, add=0.0|
        var noise = LFNoise1.ar(0.001).range(freq, freq + (freq * 0.1));
        var osc = SinOsc.ar([noise, noise * 1.04, noise * 1.02, noise * 1.08],0,0.2);
        var out = DFM1.ar(osc,freq*4,SinOsc.kr(0.01).range(0.92,1.05),1,0,0.005,0.7);
        HPF.ar(out, 40)
    }).play;
)


(
    Ndef(\dsc, {|freq = 1080|
        HPF.ar(
            BBandStop.ar(Saw.ar(LFNoise1.ar([19,12]).range(freq,freq*2), 0.2).excess(
            SinOsc.ar( [freq + 6, freq + 4, freq + 2, freq + 8])),
            LFNoise1.ar([12,14,10]).range(100,900),
            SinOsc.ar(20).range(9,11)
        ), 80)
        ;
    }).play;
)
```

```
var <>pindex, <>cindex;

initialize {
    if(pindex.isNil, { pindex = 1000 });
    if(cindex.isNil, { cindex = 2000 });
}

clearProcessSlots {
    pindex = 1000;
    (this.pindex - 1000).do{|i| this[this.pindex+i] = nil; }
}

clearOrInit {|clear=true|
    if(clear == true, { this.clearProcessSlots() }, { this.initialize() });
}

transform {|process, index|
    if(index.isNil && pindex.isNil, {
        this.initialize();
    });

    pindex = pindex + 1;
    this[pindex] = \filter -> process;
}

control {|process, index|
    var i = index;

    if(i.isNil, {
        this.initialize();
        cindex = cindex + 1;
        i = cindex;
    });

    this[i] = \pset -> process;
}
```

Exercises

```
(
    Ndef(\depfm, {|freqMin=5, freqMax=20, mul=20, add=80, rate=0.5, modFreq=2100, index=0.3, amp=0.2|
        var trig, seq, freq;
        trig = Dust.kr(rate);
        seq = Dwhite(freqMin, freqMax, inf).midicps;
        freq = Demand.kr(trig, 0, seq);
        HPF.ar(PMOsc.ar(LFCub.kr([freq, freq/2, freq/3, freq/4], 0, mul, add),
        LFNoise1.ar(0.3).range(modFreq,modFreq*2), index) * amp, 50)
    }).play;
)
```

Exercises

1. Create a string: *“Programming and Music”*. The string should be composed by adding to gather **three variables** where each one contains one of the words.

2. Create an array using **Array.series** that goes from 1 to 10 and then from 10 to 1. Finally multiply the array by 2.

3. Calculate a **multiplication** (for example 2 times 4) and **print** the result to the post window. Store the result in a variable.

4. Create an **array** that contains the **date** of **today**, the **date** of **tomorrow** and the number of lessons you have each day.