INSTITUTE OF SONOLOGY

Royal Conservatoire The Hague

Routines

Programming and Music
http://www.bjarni-gunnarsson.net

Functions

Functions

Functions contain code that can be used later or elsewhere in a program.

The code creating a function is called the **definition** of the function

When a function runs its said to be **called** or **evaluated**.

Functions are enclosed in **curly brackets { }**

Functions

A function can be split to three parts.

Argument declarations, if any, follow the first open bracket.

Variable declarations follow argument declarations.

The function body follows the variables and specifies its behavior.

Functions

A function call has the following form:
«function_name»(«arguments»)

Functions can **return values** or **cause side-effects**, by manipulating external values.

A function **returns** the value of the last statement it executes.

An empty function returns the value nil.

A function **executes** when it receives the value message.

Functions

In **SuperCollider**, functions follow a 3 step lifecycle:

1. They are compiled
2. Then evaluated
3. Their evaluation result is returned

The compilation process first parses the function code and then translates it to byte code stored in the computer memory.

The translated byte code is accessible in the language if needed for example in the case of optimization.

Arguments

Functions can have **arguments** that are set each time the function is called. These define the inputs of the function and will can be varied.

Function arguments come at the <u>beginning of the function</u>, before any variables are declared.

Function arguments are **declared** either with vertical bars || or the arg keyword.

If the number of arguments is unknown one can use ... in front of a name that will compile all provided arguments to a list variable corresponding to that name.

Streams & Routines

Abstract classes

In many object oriented languages the **abstract type** is used for classes that cannot be instantiated themselves. Classes that inherit from abstract classes can be instantiated and add or implement the required functionality.

Abstract classes can be used to define **behaviour** or **properties** used by their children but they can also be empty and used purely for design purposes.

Abstract classes are useful for defining a **protocol** for of the classes children. If one is programming for a variety of classes that inherit an abstract one, trusting that they implement the protocol can be done and therefor one does not need to know the details of how they work.

AbstractFunction

SuperCollider does not have language structures for enforcing abstract classes but they can be created using normal classes and **AbstractFunction** is an example of such a class.

AbstractFunction is the parent class of many important classes such as **Function**, **Pattern**, **Stream** and **UGen**.

AbstractFunction responds to **messages** that represent mathematical functions. It does not provide concrete implementations but a structure for doing so for example by calling the correct operator classed for a certain message. The subclasses then override the ones they need to define functionality for.

Stream inherits from AbstractFunction and is itself also an abstract class.

Stream

Stream is the base class for objects generating streams. It is an abstract class and should therefor not be instantiated directly.

A stream is a **sequence of values** where each value is fetched by calling the .next message on the stream. Getting one value at a time instead of retrieving them all is called lazy evaluation. **Lazy evaluation** only returns a value when the value is actually needed.

Streams can both be infinite and finite. To reset a sequence of values produced by a stream the reset method should be called.

In SuperCollider the base class Object defines the **.next** simply by returning itself. This means all objects can respond as streams to the next message.

Stream

FuncStream is a useful class that inherits from Stream and allows to execute functions each time next and reset are called.

Routine is another subclass of stream that allows to return (or yield) multiple times from its function and resuming from the middle when asked for the next value. A routine must be reset when it reaches the end of its function.

When called with the **play** message the yield points in a routine should be floating point numbers that indicate how long it should wait until it continues execution. This forms the basics of all scheduling that uses streams.

Streams

Routine and **Task** are subclasses of **Stream**.

A stream uses a **lazy evaluation** where an expression is only evaluated if the result is required.

Using a stream means obtaining <u>one value at a time</u> with a lazy evaluation using the .**next** message.

A stream sequence can be finite but also infinite.

Routines

A **Routine** runs a **Function** and allows it to be **suspended** in the middle and be resumed again where it left off. This functionality is supported by the Routine's superclass Thread.

A Routine is **started** the first time -next is called, which will run the Function from the beginning. It is suspended when it "yields".

A Routine can be **stopped** a before its Function returns using -stop.

Routine inherits from **Stream**, and thus shares its ability to be combined using math operations and "filtered".

Patterns

A **pattern** is an object that responds to the **asStream** message by returning a stream. Patterns describe the behaviour of a stream and a pattern can result in more than one returned stream.

Using patterns means giving abstract descriptions of streams that will be generated later. A pattern never runs but only generates stream using the asStream method.

Patterns are a powerful tool for scheduling events and making algorithmic music. There are more than 150 different classes that work with patterns in Supercollider.

Routines

Routine is a function which can pause during execution (return in the middle) and later continue.

The message **yield** or **wait** (interchangeable) is used to wait some number of seconds (using SystemClock) or number of beats (using TempoClock).

By **waiting** for a specific amount of time event scheduling can take place.

Tasks

Task is a **pauseable** process. It is implemented by wrapping a PauseStream around a Routine.

Most of its methods (start, stop, reset) are inherited from **PauseStream**.

Tasks are not 100% interchangeable with Routines. **Condition** does not work properly inside of a Task.

Exercises

```
(
    NF(\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;
)


(
    NF(\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;
}
```

```
(
    NF(\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(FMOsc.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. Implement a scheduling process that will play a random note on a fixed interval such as every two seconds.

2. Implement a timed procedure that prints the characters of a string on the post window one by one until the input string appears.

3. Implement a scheduling process that plays two lists of pitches first in sequence and then in parallel.

4. Implement a sequencing of pitches where both pitch and duration increase with time.

Exercises

5. Implement a sequencing process that plays a sequence of a few notes in three layers. Normal, octave down and octave up.

6. Implement a sequencing process that takes a list of pitches and durations and plays them. First normally and then in reverse.