```
(
  Server.internal.boot;
  Server.default = Server.internal;
  s = Server.default;
)
```

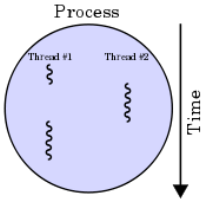
## Terminology

In the vocabulary of computer science, a **process** is a instance of a computer program that is being executed by the operating system. Processes are independent, have considerable state information and have separate addresses spaces. They generally do not share memory or other resources. Of course, the word 'process' is also used in its more general meaning.

A **task** is a plan for accomplishing something that is contained within program's address space. 'Tasks' are often associated with reatime processing (as opposed to processes). A Task can run indefinitely until it is halted.

A **thread** is one of several *concurrently* running tasks. In general, a thread is contained inside of a process and the different threads for the process share memory and other resources.

In general, computers perform one operation at a time. The concept of threads must be supported by the operating system and are generally implemented by time-division multiplexing whereby the processor is switched between different threads. This can create the illusion that more than one thing is happening at the same time.




## Stream

**Stream** is another term that is used in computer science. It is used in several different ways. SuperCollider implements one of these meaning in the class Stream which represents a so-called 'lazy' sequence of values that are obtained incrementally. Stream is an abstract class that is not used directly and it is the base class for classes that define streams. The next value in the sequence is obtained by sending the message **next** to the stream object. A Stream can be restarted with a **reset** message. A stream can be of finite or infinite length. When a finite length stream has reached the end, it returns nil.

Then too, the class Object defines **next** to return the object itself. Thus every object can be viewed as a stream that most simply streams itself. In addition to the default stream behavior implemented by Object, the class **Stream** provides functionality such as math operations on streams and filtering of streams.

In SuperCollider, Streams are primarily used for handling text and for generating music.

### Stream Inheritance Tree

```
Stream
    |
    |-------------------------|----------------------|
Thread              FuncStream      PauseStream
    |                                   |          |
    |                                   |          |
Routine - - - - - - - - - - - - - - - - ->Task   EventStreamPlayer
```

One useful subclass of Stream is the class **FuncStream** which allows the user to provide functions to execute in response to **next** and **reset**. Here is a FuncStream that represents an infinite random sequence:

```
(
    var a;
    a = FuncStream.new({ #[1, 2, 3, 4].choose });
    5.do({ a.next.postln; });          // print 5 values from the stream
)
```

Another useful subclass of Stream that we have already discussed is **Routine** which is a special kind of function

that can *act like a Stream and which inherits the property of being a thread*. Routines are functions that can return a value from the middle and then be resumed from that point when called again. The **yield** message returns a value from the Routine. The next time the Routine is called, it begins by returning from the **yield** and continues from that point.

Routines also know about the SystemClock and may contain timing instruction and run independently within your main program. SimpleNumber provides the **wait**, **waitUntil** and **sleep** methods in support of Routine.

**Task** is like a Routine except that it is pauseable. It is implemented by wrapping a PauseStream around a Routine thereby including Routine's properties and methods. (There are many synthesis examples in SuperCollider in which Routines and Tasks are virtually interchangable.) Most of it's particular methods (**start, stop, pause, resume, reset**) are inherited from PauseStream.

```
t = Task({
    50.do({ arg i;
        i.squared.postln;
        0.5.wait
    });
});

t.start;
t.stop;
t.start;
t.pause;
t.resume;
t.reset;
t.stop;
```

## Routine

The first argument (and usually the only argument) to a routine is a function.

```
// template for a routine
Routine({
    ".... code within curly braces is a function .... "
});
```

The **.yield** message applied to an expression in the function in a Routine returns a value.

```
r = Routine({ "hello, world".yield.postln });
```

To evaluate a routine, send a **.next** message. It will "hand over" the value of the expression to which the .yield message is attached

```
r.next;
```

Evaluate (again)

```
r.next;
```

The routine above returns nil when its evaluated a second time. This is because once a routine "yields" and if there's no additional code after the .yield message, the routine is finished, over, and done - unless it receives a reset message. Then it can start over again.

```
r.next;     // returns nil
r.reset;    // reset the routine
r.next;     // it works!

(
r = Routine({
    "hello, world".yield;
    "what a world".yield;
    "i am a world".yield;
});
)
```

The first three .next messages return a string. The fourth .next message returns nil.

```
r.next; // returns a string
r.next; // returns a string
```

```
r.next; // returns a string
r.next; // returns nil
```

Reset the routine.

```
r.reset;

r.next;
r.next;
r.next;
r.next;
```

You can use a .do message in a routine to make a loop.

```
(
r = Routine({
    var array;
    array = [ "hello, world", "what a world", "i am a world" ];
    // the loop
    3.do({ array.choose.yield })
});
)
```

Evaluate the routine one more time than the loop in the routine allows.

```
4.do({ r.next.postln });
```

The routine returned three strings followed by nil.

## Routines and Clocks

Clocks are used to create automated, algorithmic scheduling. Here is the fundamental way in which it is done:

**Example:**

```
(
    var r;
    r = Routine.new({
        10.do({ arg a;
            a.postln;
            1.wait;
        });
        0.5.wait;
        "done".postln;
    });

    SystemClock.play(r);  // Here the Routine is passed to the SystemClock
)
```

Alternatively, we can send the Routine a .play message:

```
    r.play
```

which will implement:

```
    SystemClock.play(r)
```

When a routine receives a .play message, control (of the routine) is redirected to a clock. The clock uses the receiver of the .wait message as a unit of time to schedule ("play") the routine.

It is usually a good idea to write:

```
    r.reset.play;                // reset the routine before playing it
```

## Clocks

SuperCollider has three clocks, each of which has a help file.

```
SystemClock                // the most accurate
```

```
AppClock            // for use with GUIs
TempoClock          // to schedule in beats
```

We will just examine the SystemClock.

## SystemClock
**superclass: Clock**

This is an example of a Class that is never instanced as a object because it is responsible for managing central resources. It is an example of the programming pattern known as a **Singleton**.

***schedAbs(time,task)**

```
SystemClock.schedAbs( (thisThread.seconds + 4.0).round(1.0),{ arg time;
    ("the time is exactly " ++ time.asString
        ++ " seconds since starting SuperCollider").postln;
});
```

***sched(delta,task)**
the float you return specifies the delta to resched the function for

```
SystemClock.sched(0.0,{ arg time;
    time.postln;
    rrand(0.1,2.0)
});
```

returning nil will stop the task from being rescheduled

```
SystemClock.sched(2.0,{
    "2.0 seconds later".postln;
    nil
});
```

***clear**
clear the SystemClock's scheduler to stop it

```
SystemClock.clear
```

## Scheduling Synths with Routines

Enclose synths within routines. It's often the case that the synthdef used by the synth in routines should have an envelope with a doneAction parameter set to 2 (to deallocate the memory needed for the synth after its envelope has finished playing).

```
(
// DEFINE A SYNTHDEF
    SynthDef( "Ringing", { arg freq;                // frequency is the only argument
        var out;
        out = RLPF.ar(
            LFSaw.ar( freq, mul: EnvGen.kr( Env.perc, levelScale: 0.3, doneAction: 2 )),
            LFNoise1.kr(1, 36, 110).midicps,
            0.1
        );
        4.do({ out = AllpassN.ar(out, 0.05, [0.05.rand, 0.05.rand], 4) });
        Out.ar( 0, out );
    }).send(s);
)

(
    // two routines used
    var stream, dur;
    dur = 1/6;
    // this routine produces a stream of pitches
    stream = Routine.new({
        loop({                                          // loop indefinitely !!!
            if (0.5.coin, {  // check SimpleNumber
                // run of fifths:
                24.yield;       // C1
                31.yield;       // G1
                36.yield;       // C2
                43.yield;       // G2
                48.yield;       // C3
                55.yield;       // G3
            });
            rrand(2,5).do({
                // varying arpeggio
                60.yield;                   // C4
                #[63,65].choose.yield;      // Eb, F  Use 'choose' with arrays for random selection
                67.yield;                   // G
                #[70,72,74].choose.yield;  // Bb, C5, D
            });
            // random high melody
            rrand(3,9).do({  #[74,75,77,79,81].choose.yield });  // D5, Eb, F, G, A
        });
    });
```

```
	// this rountine can be played
	Routine({
		loop({
			Synth( "Ringing", [ \freq, stream.next.midicps ] );
			dur.wait;	// synonym for yield, used by .play to schedule next occurence
		})
	}).play
)
```

## Routines and Arguments

SC Routines can't take arguments like a Function.  (You can use the valueArray method to take in arguments that are packed in an Array.) The way to pass arguments is to wrap the Routine within a function.

```
(
SynthDef("fm2", {
	arg freq = 440, carPartial = 1, modPartial = 1, index = 3, mul = 0.2, ts = 1;
	var mod;
	var car;
	mod = SinOsc.ar(freq * modPartial, 0, freq * index * LFNoise1.kr(5.reciprocal).abs);
	car = SinOsc.ar((freq * carPartial) + mod, 0, mul);
	Out.ar(0,car * EnvGen.kr(Env.sine(1), doneAction: 2, timeScale: ts) )
}).load(s);
)

(
	var sequence, rout;

	sequence = { arg notes, mod;
		rout = Routine({
			notes.value.postln;
			mod.value.postln;
			notes.do({
				Synth("fm2", [\freq, 400.rrand(1200),\modPartial, mod, \ts,2.5 ]);
				0.5.wait;

			})
		});
		rout.reset.play;
	};

	sequence.value(20, 7);
)
```