Programming in Supercollider

Author: Iannis Zannos

Introduction

This chapter provides an introduction to the SuperCollider programming language. It explains the mechanisms underlying the interpretation and execution of programs and the programming concepts of SuperCollider, covering as much detail as was possible in the space available. This serves as basis for understanding how to write and debug programs in SuperCollider. The chapter deals with following questions:

- What are the basic concepts underlying the writing and execution of programs?
- What are the fundamental program elements in SuperCollider?
- What are objects, messages, methods and classes and how do they work?
- How are classes of objects defined?
- How are the classes provided in the SuperCollider programming language organized into groups, and what are the characteristics and usage of each group.

Some concepts introduced by chapter 5, such as that of polymorphism, environments and class prototypes, are revisited here in more technical manner, to show they relate to the mechanism of the language as a whole, and to explore further variations and extension possibilities.

The chapter's aim is to convey programming skills without presupposing any previous knowledge of programming languages and compiler technology. Forward references to notions that have not yet been explained are avoided. In case a concept has to be mentioned early on, a brief description is given at first. The dependency of Object Oriented Programming techniques on the structural foundations of Object Oriented systems is so strong, that a fully linear exposition is hardly possible. The plan chosen was to explain the use and necessity of basic features of Object Oriented programming in a bottom-up manner while stepwise introducing the different aspects of programming in general, and to reserve the formal exposition of Class syntax for the end. The defining characteristics of Class Orientation *Encapsulation*, *Inheritance* and *Polymorphism* are thus sketched by practical example in the central parts of the chapter, sections, *Functions* and *Program Flow Control: Design Patterns*. The reason for doing this was to give a practical and therefore more solid understanding of the reasons behind these characteristics. A full account of the language's syntax is given in Appendix (n) *SuperCollider Programing Language*.

The examples in this Chapter also follow an incremental path. In the interest of presenting non-trivial examples, some constructs were used before their formal introduction in the text. These include: Creation of instances with **ClassName(arguments)**. If

statements of the form **if (condition) { true clause } { false clause }**. Use of the **fork** message on functions. Construction of arrays in various forms, including **(start_value ... end_value).** Iteration with **do:**. To introduce the concepts of inheritance and polymorphism, it was necessary to include a small example that employs Class definition code before the explanation of that syntax (see section *Program Flow Control: Design Patterns*). However I believe the explanations to that example and its role justify this exception.

Syntax Elements of the SuperCollider Language

SuperCollider employs syntactic elements from C, C++, Java, Smalltalk and Matlab, creating a style that is both concise and easy to understand for programmers that know one of these common programing languages. The following is not a formal exposition of the syntax rules, but a summary to help the reader in understanding the code of the examples in this book.

Comments

Comments are written as in C++, Java, PHP or similar languages:
- Multiline comments are enclosed between /* and */
- Single line comments start at // and run to the end of the line

Identifiers

Identifiers are sequences of alphanumeric characters and the underscore character _ that do not start with a capital letter. Such a sequence may be one of the following:
- The name of a variable or argument. Variables declared in functions, methods or classes, arguments in functions, or methods. Example:
  { arg freq; /* function code ... */ }
- The name of a message. Message names must correspond to method names.
- The variable and argument declaration keywords **arg** and **var**
- The special keywords **this** and **super**
- The constants **pi**, **inf**, **nan**, **true**, **false**

Literals

Literals as objects whose value is represented directly in the code (rather than computed as a result of sending a message to an object) are:
- Integers (e.g. **-10**, **0**, **123**) and floating point numbers (e.g. **-0.1**, **0.0**, **123.4567**)
- Strings, enclosed in double quotes: **"a string"**
- Symbols, enclosed in single quotes: '**a symbol**', or preceded by \: **\a_Symbol**
- Literal arrays: Immutable arrays declared by prepending the octothorpe #
- Classes: A Class is represented by its name. Class names are like identifiers, but start with a capital letter.

(Note: the SuperCollider help file on literals also considers identifiers to be literals. See Appendix (n) *SuperCollider Programing Language* for more details on the notation of Literals).

Primitives

Primitives are identifiers that start with underscore _. They call code that is compiled in C++ and performs elementary operations of the language, which cannot be coded in SuperCollider.

Grouping Elements

- Parentheses **()** are used to:
  - group expressions in order to specify order of execution: **1 + (2 * 3)**
  - enclose arguments that are accompanying messages: **2.pow(3)**
  - create numerical arrays from "Matlab type" series notation: **(1..5)**
  - create Events from keyword-value pairs: **(freq: 440, amp: 0.1)**
- Brackets **[]** are used to:
  - Create Arrays or other types of collections **[1, 2, 5]**
  - Index into collections for reading or writing of values: **aDictionary[\freq]**, **anArray[0]**
- Braces **{}** are used to:
  - Define functions: **{ arg a, b; a + b }**
  - Define Classes: **Nil { /* class definition code */ }**
  - Define methods: **isNil { ^true }**

Binary Operators

Many arithmetic, logical, stream and other binary operator symbols are used similarly as in C++. For a full account, see Appendix (n) *SuperCollider Programming Language.*

Delimiters

- The dot . is used in the following senses:
  - To attach a message to the receiving object that it is sent to: **123.squared**
  - To separate the decimal part of a floating point number from the integer part: **12.3**
  - To append an *adverb* to a binary operator. (Adverbs are identifiers or Integers that modify the behavior of an operator).
  - Triple dots ... are used as ellipsis to collect multiple arguments into an array, in argument definitions: **{ | ... args |** or in multiple variable assignments: **#a, b ... rest = [1, pi, 10, true, inf];**
  - Double dots are used in notation of arithmetic series: **(1..5)**, **(0, 0.1 .. 10)**

- Comma is used to separate arguments: **f.value(pi, 400)**, elements of Collections: **[pi, \a, 5]**, or variables or arguments in declaration statements: **{ arg a, b;** or in multiple assignment statements: **#freq, amp = #[440, 0.1]**
- Semicolon is used to separate statements. The last statement of a program (function) does not need to end in a semicolon. **a.postln; b = a.squared**
- Pipe signs | | are used to delimit an argument declaration statement: **{ | a, b | a + b }.** This is alternative notation to **arg a, b;**

Special Characters

- **^** marks the statement that it precedes as a return value statement in a method
- * Has two uses:
  - Preceding an argument in a message, it applies the collection's elements as separate arguments.
  - Preceding the name of a method in Class definition code, it indicates that this is a class method.
- # (octothorp) is used as prefix in two cases:
  - Multiple variable assignment: **#freq, amp = [400, 0.1]**
  - Construction of immutable Arrays and closed Functions. **#[1, 5, 11], #{ pi ! 3 }**
- **$** precedes Character instances: **$a, $. $A**
- **~** Tilde before an identifier treats it as an environment variable (see section *Environment Variables*)
- < and > construct accessor methods for variables in classes

Construction of Specific Kinds of Objects, Abbreviations, Various Conventions

- \ before an identifier constructs a Symbol **\symbolic**
- Single quotes enclosing text construct a Symbol **'symbol from any string !'**
- Double quotes enclosing text construct a String **"a string"**
- Braces {} construct Functions
- Brackets **[]** construct Arrays (or other Collections when preceded by collection name)
- Parentheses **()** enclosing keyword-value pairs construct events: **(a: 1, b: 2)**
- At-sign **@** between two numbers constructs a Point: **-5@10**
- At-sign **@** between a collection and a number indexes the number into the collection: **[1, 5, 7]@1** (see variants for types of indexing in Appendix (n))
- Arrow **->** between values constructs an association: **\freq->440**
- Underscore _ by itself in a message statement constructs a function: **_.isPrime**
- element ! n repeats the element n times and collects the result: **_.isPrime ! 12**
- The message **new** can be omitted between a Class name and arguments enclosed in parentheses: **Synth("sine")** is equivalent to **Synth.new("sine")**
- The message **value** can be omitted between a Function and arguments enclosed in parentheses: **foo.(n)** is equivalent to **foo.value(n)**

- Functions as sole arguments need not be enclosed in parentheses: **10.do { "hello".postln }**
- Messages whose name ends in underscore _ can also be written in "variable assigment" format: **aPoint.x_(0)** is equivalent to: **aPoint.x = 0**
- **<** or **>** prepended to a variable name in a variable declaration statement in a Class construct corresponding methods for getting or setting the value of that variable.

Fundamental Elements of Programs

Objects and Classes

The language of SuperCollider implements a powerful method for organizing code, data and programs known as *Object Oriented Programming* (OOP). SuperCollider is a pure OOP language which means that all entities inside a program are some kind of object. This also means that the way these entities are defined is uniform, as are the means for communicating with them.

Objects

There are two main types of objects according to how their internal contents are organized: Objects with a fixed number of internal slots for storing data and objects with a variable number of slots. The generic term for the latter type of object is *collection.* Collections are objects that store a non-predefined number of other objects. A way to create a collection of type Array is to list some objects separated by comma and enclose the whole in brackets **[]**.

Some examples of objects are:

```
1                // the integer number 1
1.234            // the floating-point number 1.234
"hello"          // a String (an array of characters)
\alpha           // a Symbol (a unique identifier)
'alpha 1'        // another notation for a Symbol
$a               // the Character a
100@150          // a Point defined by coordinates x, y
[1, \A, $b]      // an Array containing 3 elements
(a: 1, b: 0.2)   // an Event
{ 10.rand }      // a Function
String           // the Class String
Meta_String      // the Class of Class String
```

Classes

The description of the properties and behavior of an object is called its *Class*, and an object made from such a description is called an *instance* of that Class. A Class can inherit its properties from another Class called its *superclass*. The Class that inherits the

properties is called a *subclass* of the Class from which it inherits them. Names of objects that begin with a capital letter, such as **String** and **Meta_String** above, refer exclusively to Class instances, i.e they represent Classes (see section *Classes and Instances*). A name that starts with capital letter cannot be used as name for a variable, message, or anything else than a class.

A Note on Identifiers and Keywords

An alphanumeric string with or without underscore (_) which is not enclosed in quotes (' or ") neither preceded by \ and starts with a small letter, for example **freq**, **new**, **postln**, **play**, **fork**, **synth** etc., does not represent a specific object, but is an identifier representing a variable (whose contents may change), a message or a keyword (**arg**, **var** etc).

Messages and Methods

To interact with an object, one sends it a *message*. For example, to calculate the square of a number, one sends it the message **squared:**

```
15.squared  // calculate and return the square of 15
```

The object to which a message is sent is called the *receiver*. In response to the message, the receiving object finds and runs the program that is stored in the *method* that has the same name as the message, and returns to the calling program a result, which is called the *return value*. In other words, a method is a program stored under some message name for an object, that can be recalled by sending that object the message's name. *Instance methods* operate on an instance (such as the Integer 1) while *class methods* operate on a class (such as the class Integer).

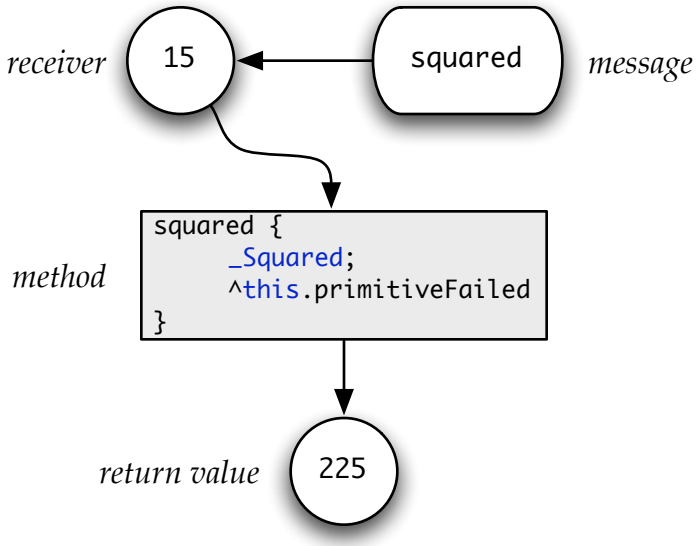


Figure (n): Receiver, message, method, return value

An alternative way of writing a message is in C-style or Java-style function-call form. So the above example can also be written as:

```
squared(15) // calculate and return the square of 15
```

SuperCollider often permits to choose amongst different writing forms for expressing the same thing. It is up to the programmer to decide which form of expression to use. Two main criteria that programers take into account are readability and conciseness.

Chaining Messages

It is possible to write several messages in a row, separated by dots (.), like this:

```
Server.local.boot    // boot the local server
```

Or this:

```
Server.local.quit    // quit the local server
```

When "chaining" messages, each message is sent to the object returned by the previous message (the previous return value). In the examples above, **Server** is the class from which all servers are made. Among other things, it holds by default 2 commonly used servers, the *local server* and the *internal server*, which can be obtained by sending it the messages **local** and **internal** respectively. The objects and actions involved are:

```
Server      // The class Server
// message local sent to Server returns the local server: localhost
Server.local
Server.local.boot // The message boot is sent to the local server
```

Performing Messages

In some cases, the message to be sent to an object may change depending on other conditions. when the message is not know in advance, the messages **perform** and **performList** are used, which permit an object to perform a message passed as argument:

```
Server.local.perform(\boot)   // boot the local server
// boot or quit the local server with 50% probability of either:
Server.local.perform([\boot, \quit].choose)
```

performList permits to pass additional arguments to the message in list form. Thus **Rect.performList(\new, [0, 10, 200, 20])** is equivalent to: **Rect.new(0, 10, 200, 20)** (See also example in section *Customizing the Behavior of Objects with Functions*)

Arguments

The operation of a message often requires the interaction of several objects. For example, raising a number to some power involves two numbers: the base and the exponent. Such additional objects required by an operation are sent to the receiver as *arguments* accompanying the message. Arguments are enclosed in parentheses after the message:

```
5.pow(8) // calculate the 8th power of 5
```

If several arguments are involved, these are separated by commas:

```
// construct an array of 5 elements starting at 10 and incrementing by 10
Array.series(5, 10, 10)
```

The same in "function-call" format:

```
series(Array, 5, 10, 10)
```

If the arguments are provided as one collection containing several objects, they can be separated into individual values by prepending the * sign to the collection:

```
Array.rand(*[5, -10, 10])
```

is equivalent to:

```
Array.rand(5, -10, 10)
```

This can be useful when one wants to provide as arguments some collection that was created in some other part of the program. The next example shows how to construct an Array of random size between 3 and 10, with elements whose values have a random range with 3 as lowest and 10 as highest possible limit.

```
Array.rand(*Array.rand(3, 3, 10))
```

When the only argument to a message is a function, then the parentheses can be omitted:

```
10.do { 10.rand.postln } // function as sole argument to a message
```

*Argument Forms for Implied Messages **at** and **put***

When square brackets are appended to an object, these imply the messages **at** or **put**. Thus **[1, 5, 12][1]** is equivalent to **[1, 5, 12].at(1)** and **()[\a] = pi** is equivalent to **().put(\a, pi)**.

Argument Keywords

When calling a function, argument values must be provided in the order in which the arguments were defined (see below, section *Defining Arguments*). However, when one only a few out of many arguments of a function need to be provided, then one can specify those arguments by name in "keyword" form, e.g. if the name of the argument provided is **freq** the call is **foo.value(freq: 400)**. For example the method for **Xline.kr** takes the arguments: **start**, **end**, **dur**, **mul**, **add**, **doneAction**. To provide values only for **start**, **end**, **dur** and **doneAction**, write for example: **Xline.kr(100, 100, 10, doneAction: 2)**. As a result **start**, **end**, and **dur** get the values 100, 1000, and 10, **doneAction** gets the value 2, while **mul** and **add** rely on their default values 1 and 0.

```
// Boot the default server first:
Server.default.boot;
// Then select all lines between the outermost parentheses and run:
(
{
   Resonz.ar(GrayNoise.ar,
      XLine.kr(100, 1000, 10, doneAction: 2),
      XLine.kr(0.5, 0.01, [4, 7], doneAction: 0)
```

```
    )
}.play
)
// further examples:
{ WhiteNoise.ar(EnvGen.kr(Env.perc, timeScale: 3, doneAction: 2)) }.play;
{ WhiteNoise.ar(EnvGen.kr(Env.perc, timeScale: 0.3, doneAction: 2))}.play;
```

Binary Operators

SuperCollider defines the use of operators known from mathematics, logic as well as from other programming languages, such as + (addition), - (subtraction), & (binary and) and others. Since these signs operate between two objects, they are called binary operators. For example, the binary operator ++ joins two Arrays (or related **SequenceableCollection** objects): **[\a, \b] ++ [1, 2, 3]**

Additionally, any message that requires just one argument can be written as binary operator by adding **:** to the name of the message. So **5.pow(8)** can also be written as **5 pow: 8**. Expressions involving the use of binary operators are implicitly translated into message format by the compiler (see section *Byte-Code*).

A list of operators is given in *Appendix (n), SuperCollider Language Overview*, along with other concise syntax elements.

Precedence Rules and Grouping

When one combines several operations in one expression, the final result may depend on the order in which those operations are executed. Compare for example the expression **1 + (2 * 3)**, whose value is 7, to the expression **(1 + 2) * 3**, whose value is 9. The order in which operations are executed is determined by the precedence of operators. The precedence rules in SuperCollider are very simple:

1. Binary operators are evaluated in strict left-to-right order. Thus the expression **1 + 2 * 3** is equivalent to **(1 + 2) * 3** and not to **1 + (2 * 3)**. Another example is:
   ```
   10 * (1..3) addAll: [0.1, 0.2, 0.3] // = [ 10, 20, 30, 0.1, 0.2, 0.3 ]
   ```
   In the example above, 10 is multiplied to the elements of [1, 2, 3] and then the elements of [0.1, 0.2, 0.3] are appended to the result.
2. Message passing as in **receiver.message(arguments)** or as in **collection[index]** has precedence over binary operators. Compare the following to the example above:
   ```
   10 * (1..3).addAll([0.1, 0.2, 0.3]) // = [10, 20, 30, 1, 2, 3 ]
   ```
   Here the elements of [0.1, 0.2, 0.3] were first appended to [1, 2, 3] and then all elements of the resulting new array were multipled by 10.
3. To override the order of precedence, one uses parentheses **()**. For example:
   ```
   1 + 2 * 3 // Left to right order of operator evaluation. Result: 9
   1 + (2 * 3) // Forced the evaluation of * before that of +. result: 7
   ```

Following examples illustrate the different effects of grouping by parenthesis and message passing:

```
((1 + 2).asString).speak   // = speak("3")
"1" ++ "2".speak           // speak("2"), return "12"
("1" ++ "2").speak         // speak("12");
(1.asString ++ 2.asString).speak // speak("12")
("1+2").speak              // speak("1+2")
(1.asString ++ "+2").speak // speak("1+2")
(1 + 2).speak        // error: speak not understood by Integer 3
```

(The message **interpret** can be substituted for **speak** above if **speak** does not function on the computer in question).

Statements

The single-line code examples introduced above normally constitute parts of larger programs that include many lines of code. The smallest standalone elements of code, are called *statements*[1]. When a program contains more than one statement, the individual statements are separated by a semicolon (**;**). The last statement at the end of a program does not need to have a semicolon, since there are no more statements to separate it from.

Below is an example of a program that plays 10 short noise bursts at different time intervals. It contains three main statements. The first statement posts a title for the program: **"Ten Tiny Booms".postln;** The second statement boots the default server so that sounds can be played with it: **Server.default.boot;** The third statement creates a function and sends it the message **fork**. It starts with **{** and continues over several lines until the end of the example: **}.fork**. A function is itself a program, which is defined by enclosing a series of statements in braces {} (see below, section *Programing with Functions*). The message **fork**, when sent to a function, creates a routine, which is a program that can run as an independent process in parallel to other programs. This enables the function contained in the routine to schedule the execution of its statements by using the message **wait**, which tells the process to wait at that statement for as many seconds as specified by the number that **wait** is sent to.

It is important to distinguish between the lines of code text in a Document window as seen by a human programer, and the part of the code that SuperCollider processes as program when the programmer runs a selected portion of that code by hitting the [enter] key. When the [enter] key is hit, SuperCollider does not run the whole code in the window, but only the code that was selected, or the line on which the cursor is currently located. Every time that one runs a piece of code by typing the [enter] key, SuperCollider creates and runs a new program that contains only the selected code. Code that is meant to be run as a whole is usually indicated by enclosing in parentheses, because one can select it easily by double-clicking to the right of an opening parenthesis:

```
(
```

```
// Select all code between the outer parentheses and run.
// Statement 1: post a 'title' for the program
"Ten Tiny Booms".postln;
// Statement 2: boot the default server
Server.default.boot;    // boot the server for playing sounds
// Statement 3: A function played as routine
// The main part of the program is a function, run as Routine with fork:
{                    // Start of function
   // Inside the function is another program consisting of statements
   1.wait;              // wait for the server to boot
   SynthDef(\test, {    // Define a short noise burst sound algorithm
      Out.ar(0,         // send output to the first available output
         PinkNoise.ar(  // create a noisy signal
            // form an amplitude envelope controlling the noisy signal
            EnvGen.kr(Env.perc(0.05, 1, 0.1, -4), doneAction: 2)
         ).dup          // duplicate the burst to left and right channel
      )
   }).send(Server.local);  // send the sound algorithm to the server
   0.1.wait;      // wait for the server to load the sound definition
   10 do: {       // repeat the following 10 times
      Synth(\test);  // play a sound using the sound definition above
      Date.getDate.format("tiny boom at %X").postln;  // post message
   // wait for a random interval between 0.1 and 2.0 seconds
      wait(0.1 exprand: 2.0)
   };             // end of the function that will be repeated 10 times.
   "DONE".postln; // after playing 10 times, post "DONE"
}.fork      // end of function. Fork plays the function as Routine
)
```

Variables

A variable is used to store an object that will be used in other parts of a program. One way to visualize variables is as containers with labels. The name of the variable is the label pointing to the container.

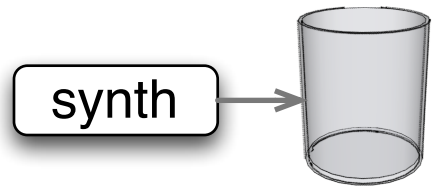


Figure (n) A variable named 'synth'

One creates variables by declaring them with the prefix **var** (*var* is a word reserved for declaring variables and cannot therefore be used in any other sense). Several vari-

ables can be declared in one **var** statement. Variables may only be declared at the beginning of a program:

```
var window; // create a variable named 'window'
// rest of program follows here
```

A variable only exists within the program in which it was created (see also below: *Scope of Variables in Functions*). When a variable is first created it is empty, so its value is represented by the object **nil**. **nil** is the object for no value:

```
(
var window;    // create a variable named 'window'
window.postln; // post the contents of variable 'window' (nil)
)
```

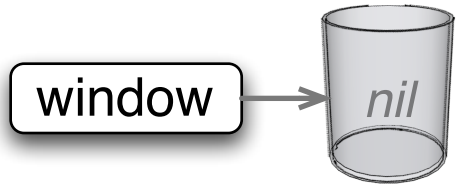


Figure (n) A variable named window with nothing stored

Since, as stated above, every piece of code run separately constitutes a distinct program, one cannot run the lines of a program that use a declared variable separately, but must always run the code as a whole. In the example above, running the line **window.postln;** alone produces following error message, even if one has previously separately run the line **var window;**

- `ERROR: Variable 'window' not defined.`

To store an object in a variable use the assignment sign =. For example, after storing an SCWindow in the variable **window**, one can send it messages to change its state, as well as use it as an argument to other objects:

```
(
// A window with a button that posts: "hello there!"
var window, button;
// create a GUI window and store it in variable window
window = GUI.window.new("OLA!", Rect(200, 200, 120, 120));
// create a button in the window and store it in variable button
button = GUI.button.new(window, Rect(10, 10, 100, 100));
button.states = [["'ALLO"]];  // set one single label for the button
button.action = { "hello there!".postln }; // set the action of the button
window.front;          // show the window
)
```

In the above example, the variable **window** is indispensable to specify which window the button was going to appear in.

Variable initialization

The assignment sign (=) can be used in a declaration statement to initialize the value of a variable:

```
(
var bounds = Rect(10, 20, 30, 50), x = 100, y = 200;
bounds.width.postln; // post the width of a rectangle
bounds.moveTo(x, y); // move the rectangle to a new position
)
```

Multiple Assignment with

A useful shortcut for assigning the values of consecutive values of objects from a collection is the # sign. If the size of the collection is unknown, one can collect any remaining elements in the last variable listed by prepending three dots ... (ellipsis) to it:

```
(
var red, green, blob;
// collect the first element in red, the second in green,
// and the rest in blob:
#red, green ... blob = Array.series(4 rrand: 8, 10, 10);
// rearrange the contents of the variables in a new array:
[blob, red, green]
)
```

Use of Variables

The object stored in a variable remains there until a new assignment statement replaces it with something else. Variables are often also used as temporary placeholders to operate on a changing choice from a set of objects. Here is an example that makes extensive use of variables to alternate at random between two pairs windows and sounds:

```
(
/* Alternately move two windows at random and play a different type of
sound for each of the two windows */

var window1, window2; // the two windows to be displayed
var ctr_x, ctr_y; // coordinates of computer screen's center
#ctr_x, ctr_y = SCWindow.screenBounds.center.asArray;
// create window 1
window1 = SCWindow("1", Rect(ctr_x - 100, ctr_y - 100, 100, 100));
window1.front;    // show the window
window1.view.background = Color.rand(0.5, 0.2, 0.7, 0.9);
// create window 2
window2 = SCWindow("1", Rect(ctr_x + 100, ctr_y + 100, 100, 100));
window2.front;    // show the window
window2.view.background = Color.rand(0.5, 0.9, 0.7, 0.8);
```

```
Server.local.boot;   // boot local Server to play sounds

{  // Create a function that will be played as routine to enable timing
   var sounds_and_windows; // 2 pairs of windows and sounds
   var window, sound;      // the currently selected window and sound
   var countdown = 100;    // count down the number of iterations
   sounds_and_windows = [[window1, \ping], [window2, \gray_perc]];
   1.wait; // wait 1 second for server to boot
   // load the SynthDefs for the 2 types of sounds used:
   SynthDef(\ping, { // SynthDef 1: Short sine sound
      Out.ar(Rand(0,1).round(1),
         SinOsc.ar(Rand(400, 800), 0,
            EnvGen.kr(Env.perc(0.05, 0.3, 0.1, -4), doneAction: 2)
         )
      )
   }).send(Server.local);

   SynthDef(\gray_perc, { // SynthDef 2: Short percussive sound
      Out.ar(Rand(0,1).round(1),
         GrayNoise.ar(
            EnvGen.kr(Env.perc(0.05, 0.3, 0.1, -4), doneAction: 2)
         )
      )
   }).send(Server.local);

   #window, sound = sounds_and_windows.choose; // initialize window, sound
   // Repeat 100 times, randomly selecting different sound-window pairs
   countdown do: {
      countdown = countdown - 1;    // count down number of times left
      // select a different pair of window/sound 40% of the time
      if (0.4.coin) { #window, sound = sounds_and_windows.choose };
      // move the selected window
      window.bounds = window.bounds.moveBy(*[-50, 50].scramble.rand);
      window.name = countdown.asString; // change name of window
      Synth(sound);  // play the selected sound
      0.1.wait       // wait for 1/10 of a second before repeating
   };
   window1.close; window2.close; // close the 2 windows when done
// play with Application Clock to enable graphics in Routine
}.fork(AppClock) // end of function, create routine with fork
)
```

Instance Variables

An instance variable is a variable that is contained in a single object (an instance of a Class). Such a variable is only accessible inside a single object instance – that is, inside

instance methods of an object – unless special code is written to make it accessible to other objects. For example, objects of Class **Point** have two instance variables **x** and **y**, corresponding to the coordinates of a point in 2-dimensional space. These are accessible with the messages **x** and **y**. (See section *Defining Classes*)

```
(
var point = Point(0, pi);
point.x.postln; point.y.postln; point.y == pi;
)
```

Figure (n) shows three instances of Point with their individual instance variables.

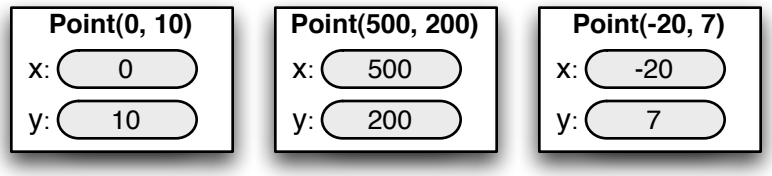


Figure (n) Three instances of Point with their instance variables

Class Variables

A class variable is defined once for the Class it belongs to, and for all its subclasses. It is accessible to class methods and to instance methods of its Class as well of its subclasses. For example, the class variable **allDocuments** of **Document** holds all currently open Document windows. The instance method **prAdd** of Document adds a newly created Document to the class variable **allDocuments**, while the method **closed** removes a Document from **allDocuments** when its window closes. In that way the system keeps track of all open Document windows at any time. One can write **Document.allDocuments do: _.close** to close all Document windows.

Environment Variables

Environment variables are written as identifiers preceded by tilde (~). For example: **~a = pi**. These reference the value of the variable in the current Environment. They do not need to be declared. An Environment represents a set of bindings of values to names, the environment variables. The behavior of an Environment differs from the bindings created by normal variable declarations in that the Environment is an object that holds the bindings independently of a function, and can be modified more easily (see section *Environments* for more details).

The relationship between environment variables and Environment as an object containing these can be seen by printing the current Environment:

```
// run each line separately:
currentEnvironment;  // empty if no environment variables have been set
~alpha = pi;         // set env. variable ~alpha to pi
currentEnvironment;  // see current Environment again: ~alpha is set
```

```
~freq = 800;         // set another environment variable
Server.local.boot;
{ LFNoise0.ar(~freq, 0.1) }.play; // use an environment variable
// setting an environment variable to nil is equivalent to removing it:
~alpha = nil;
currentEnvironment;  // alpha is no longer set
```

Variables with Special Uses

This section deals with a group of variables whose values are given by the system at runtime and change according to the context in which the code is running. These provide access to objects that are useful or indispensable but either cannot be provided by conventional programming within the SuperCollider class system or have been defined as global variables for access by all objects in the system.

Interpreter Variables

The class Interpreter defines 26 instance variables whose names correspond to the letters from a to z. Since all code evaluated from a Document window is run by an instance of Interpreter, these variables are accessible by that code without having to be declared. However this works only when evaluating code from a Document window. The following example can be executed one line at a time on a Document window:

```
Server.default.boot;       // boot the default server
// create a synth and store it in n:
n = { | freq = 400 | LFDNoise1.ar(freq, 0.1) }.play;
n.set(\freq, 1000);  // set the freq parameter of the synth to 1000
n.free;              // free the synth (stop its sound)
```

Pseudo-Variables

Pseudo-variables are not declared anywhere in the SuperCollider library, but are provided by the SuperCollider compiler.

- **this**: This represents object that is running the current method. In code run from a Document window, this is always the current instance of Interpreter (see section *Who Does the Compiling?*). So one can run **this.inspect** on a Document window to view the contents of the current Interpreter instance, including the variables a-z.
- **thisProcess**: The process that is running the current code. It is always an instance of Main. While rarely used, some possible applications are to send the current instance of **Main** messages that affect the entire system, such as **thisProcess.stop** (stop all sounds), or to access the interpreter variables from any part of the system (**thisProcess.interpreter.a** accesses the interpreter variable a).
- **thisMethod**: The method within which the current statement is running. One can use this in debug messages, to print the name of the method where some code is being checked. For example: **[this, this.class, thisMethod.name].postln**.

- **thisFunction**: the innermost function within which the current statement is running. This is indispensable for recursion in functions (see section *Recursion*).
- **thisFunctionDef**: The definition of the innermost function within which the current statement is running. The function definition contains information about the names and default values of arguments and variables, which can be used for example to create a GUI.
- **thisThread**: The thread that is running the current code. A thread is a special type of program that can run in parallel with other threads and can control the timing of the execution of individual statements in the program. Examples of **thisThread** are found in Classes **Pstep** and **Pseg** where it is employed to control the timing of the thread.
- One special case: The keyword **super** redirects the message sent to it to look for a method belonging to the superclass of the object in which the method of the current code is running. This is not a variable at all, because one cannot access its value, but only send it a message. **super** is used to extend a method in a subclass. For example. the class method new of **Pseq** extends the method **new** of its superclass **ListPattern**, which in turn extends the method new of **Object**. This means Pseq's **new** calls **super.new** – thereby calling method **new** of **ListPattern** – but adds some statements of its own. In turn, **ListPattern** also calls **super.new** – thereby calling new of **Object** to create a new instance of **ListPattern** – but also adds some stuff of its own.

Class Variables of Object

The following variables are class variables of Class **Object**. Since all objects are instances of some subclass of Object, they have access to these variables, and thus these variables are automatically accessible everywhere.

- **currentEnvironment**: the current environment is the environment wherein is run the program that contains a call to **currentEnvironment**.
- **topEnvironment:** The top environment is the original currentEnvironment of the Intepreter instance that runs programs in the system. It can be accessed independently of **currentEnvironment**, which changes in response to **use** or **make** messages.

```
(
~q = "TOP";                    // store "TOP" in ~a, top environment
(a: "INNER") use: { // run function in environment with ~a = "INNER"
   currentEnvironment.postln; // show the current environment
   topEnvironment.postln;     // show the top environment (different!)
   ~a.postln                  // show ~a's value in current environment
};
~a;                            // show ~a's value in top environment
)
```

- **uniqueMethods**: Holds a dictionary that stores unique methods of objects. Unique methods are methods that are not defined in a Class, but only in a single instance. For example:

```
// add a unique method to the object 1:
1.addUniqueMethod(\greet, { { "hello there".postln; } ! 34 });
// now 1 understands the message "greet":
1.greet;
// but 2 or any other number does not understand the message "greet":
2.greet  // ERROR: Message 'greet' not understood.
```

- **dependantsDictionary:** Holds a dictionary that stores *dependants* of objects. A dependant of an object **o** is any object that needs to be notified when **o** changes in some way. To notify the dependants of an object that it has changed, one sends it the message **changed**. Details of this technique are explained in section *The Observer Pattern.*

Variables Versus Symbols

Variable names are not to be confused with symbols referring to synth parameters or elsewhere. Variable names cannot contain backslash \ or quote ', and their value is independent from their name. Given a Synth object stored in variable a: **a = Synth.new(\test)**, to set the parameter of the synth one writes: **a.set(\freq, 800)**. The following are incorrect:

```
a.set(freq = 800);   // not a way to set parameter 'freq' in a synth
a.set(freq, 800);    // variable freq is not the same as symbol 'freq'
```

Variables Versus References

Variable is a container with which one can do two things only: Store an object and retrieve that object. One cannot store the container itself in another container, i.e. it is not possible to store a variable **x** *itself* in another variable **y**. As the following example shows, what is stored is the *content* of variable **x.** When the content of variable **x** is changed, the previous content still remains in variable **y**:

```
var alpha, beta, gamma;
gamma = alpha; // storing variable alpha in gamma has no effect:
alpha = 10;    // store 10 in alpha ...
gamma.postln;  // but the value of gamma remains unchanged
alpha = beta;  // so one cannot use gamma as 'joker'
beta = 20;     // to switch between variables alpha and beta.
gamma.postln;  // gamma is still nil.
```

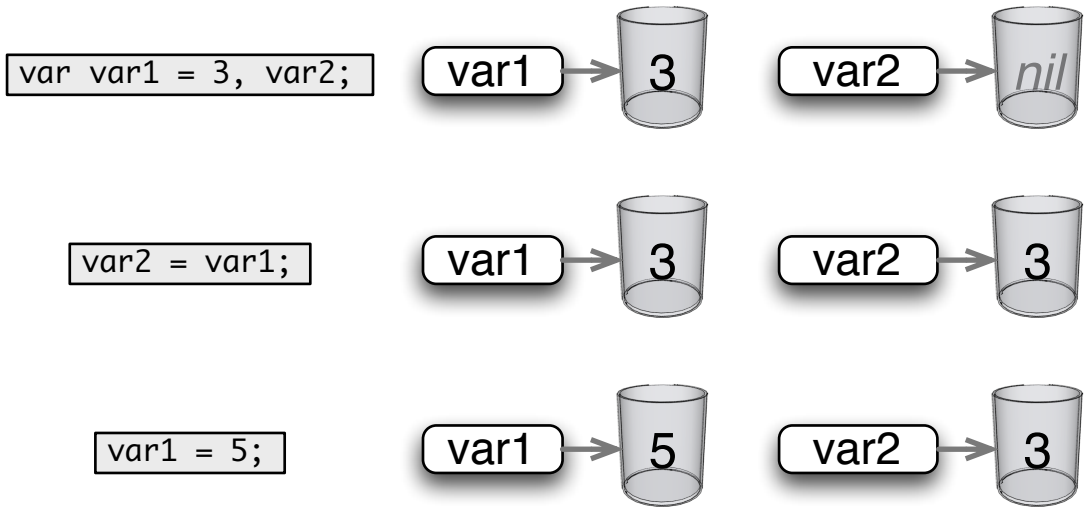


Figure (n): Assigning a variable to another variable stores its contents only

To store a container in a variable, one uses a reference object: **Ref**:

```
var aref, cvar;
aref = Ref.new; // first create the reference and store it in aref
cvar = aref;   // then store the contents of aref in cvar
aref.value = 10;  // change the value of the reference in aref
cvar.value.postln;   // and retrieve that value from cvar
```

Functions

A function is a program which can be run by other programs. One can "package" some code that does something useful inside a function and then run that function wherever one wants to do that thing, instead of writing out the same code in different places. The code that creates a function is called *definition* of the function. When a program runs a function it is said that it *calls* that function.

To "package" some code into a function, one encloses it in braces {}, like this:

```
{ 1 + 1 } // a function that adds 1 to 1
```

This creates a function object, or in other words *defines* a function. To run the function, one sends it the message **value**:

```
{ 1 + 1 }.value // evaluate { 1 + 1 }.
```

This is called *function evaluation.* The above outputs the result (2) to the post window when run by pressing the [enter key].

Return Value vs. Side Effect

The term "evaluate" comes from the idea of requesting a value that is computed and returned by the program for further use (*evaluate*: to determine or fix the value of[2]). The return value of a function is the value of the last statement that is computed in the func-

tion. However, in many cases, one does not run the program to obtain a final value, but to start a process that will result in some change, such as create sounds or show graphics on the screen. For example, the function **{ 10.rand }** is used to obtain a random number between 0 and 9 as a value, while the function **{ Document.allDocuments do: _.close }** closes all windows (it is certainly the effect of the latter rather than the return value that matters).

In the example of section *Chaining Messages* above, **Server.local.boot**, the message **local** is sent to class **Server** to obtain the object representing the local server as return value, while the message boot is sent to the local server to boot it. In the first case (message **local**) it is the return value of the operation that is of further use, while in the second case (message **boot**) it is the effect of the boot operation that matters.

Implicit vs. Explicit Function Evaluation

Every time that a piece of code is run in a Document window by pressing the [enter] key, SuperCollider turns the code selected into a function and evaluates it. For example the program: **GUI.window.new** is actually translated by SuperCollider into the equivalent of: **{ GUI.window.new }.value**. The **{ }.value** part is implicitly provided by SuperCollider for convenience. However, the message **value** is useful for running functions that have been previously defined and stored in other parts of a program. The next example illustrates this.

Functions as Program Modules

In SuperCollider, all programs are functions. Consequently running programs consists in running functions. Since functions are objects that can be stored in variables, it is easy to define and store any number of functions (i.e. programs), and run these whenever required, any number of times. Thus, a major part of programing in SuperCollider amounts to defining functions and configuring their combinations.

The example below illustrates how to call a function that has been stored in a variable in various ways. The main function of the example does two things:

1. It calculates a new frequency for the sound by moving one minor tone upwards or downwards from the previous the pitch.
2. It sets the frequency of the sound to the new pitch.

   The program of the function consists of two statements:

```
{
   freq = freq * [0.9, 0.9.reciprocal].choose; // change freq value
   synth.set(\freq, freq);    // set synth's frequency to new value
}
```

This function is stored in variable **change_freq** and then called in two different ways:

1. It is stored in the **action** part of a GUI button so that when that button is pressed, it runs the function.

2. It is called explicitly by a function inside a Routine that sends it the message **value** (a Routine has the ability to time the execution of its statements, and can therefore run the function in question at timed intervals).

```
Server.default.boot; // (boot Server before running example)
(
// Define a function and call it in different contexts
var synth;        // Synth creating the sound that is changed
var freq = 220;   // frequency of the sound
var change_freq;  // function that changes the frequency of the sound
var window;    // window holding buttons for changing the sound
var button1, button2, button3; // buttons changing the sound

// Create a synth that plays the sound to be controlled:
synth = { | freq = 220 | LFTri.ar([freq, freq * 2.01], 0, 0.1) }.play;
// Create frequency changing function and store it in variable change_freq
change_freq = {   // start of function definition
      freq = freq * [0.9, 0.9.reciprocal].choose; // change freq value
      synth.set(\freq, freq);    // set synth's frequency to new value
};                // end of function definition

// Create 3 buttons that call the example function in various ways
window = GUI.window.new("Buttons Archaic", Rect(400, 400, 340, 120));
//  ------------------------ Example 1 ------------------------
button1 = GUI.button.new(window, Rect(10, 10, 100, 100));
button1.states = [["I"]]; // set the label of button1
// button1 calls the function each time that it is pressed
button1.action = change_freq; // make button1 change freq once
//  ------------------------ Example 2 ------------------------
button2 = GUI.button.new(window, Rect(120, 10, 100, 100));
button2.states = [["III"]];
// Button2 creates a routine that calls the example function 3 times
button2.action = {         // make button2 change freq 3 times
   { 3 do: { change_freq.value; 0.4.wait } }.fork; // play as routine
};
//  ------------------------ Example 3 ------------------------
button3 = GUI.button.new(window, Rect(230, 10, 100, 100));
button3.states = [["VIII"]];
button3.action = {         // like example 2, but 8 times
   { 8 do: { change_freq.value; 0.1.wait } }.fork; // play as routine
};
// use large size font for all buttons:
[button1, button2, button3] do: _.font_(Font("Times", 32));
// stop the sound when the window closes:
window.onClose = { synth.free };
window.front; // show the window
```

)

Scope of Variables in Functions, Global Variables

As mentioned in section *Variables* above, variables are only accessible within the program – i.e. the function – that defines them. However, if a function *mother_func* creates another function *child_func*, then *child_func* has access to the variables created within *mother_func*. This is useful when several functions want to share data, as shown by previous section's example: The variable **freq** is defined in the top-level program, which takes here the place of *mother_func*, while the function stored in variable **change_freq** is the *child_func* that has access to this variable. The function **change_freq** can therefore both *read* (access) the value of variable **freq** and set (*write*) it whenever it is called. Note that this program calls the function at several different points in the code, and each call has access to the same variable, **freq**.

Global Variables

In the above program the variable **freq** is a *global variable*. Variables declared at the beginning of a top-level program are global variables with regard to any functions that this program defines, because they are accessible by all such functions.

Section *Function Closures* below takes this technique one step further by creating multiple separate copies of *child_func*, each of which is evaluated repeatedly and operates on its own set of variables.

Compilation and Evaluation: The Details

To compile a program means to translate it from a form of code that specifies the structure of the program (usually human readable form) into a form that can be executed by the computer. The form used internally by SuperCollider is called *byte-code*, because every elementary instruction is represented by one or more bytes. The present section deals with compilation that is done whenever the user runs a piece of code in a Document Window. Additionally, there is a second type of compilation, which happens when compiling the entire Class system the (see section *Compiling the Class Library*). Both types result in byte-code. Their difference is that the first compiles just one piece of text within an already existing system of classes and objects, wheres the second compiles all classes, thereby re-creating from scratch the entire system within which SuperCollider programs run.

SuperCollider undergoes a three step process every time that it executes code entered from a workspace window: First SuperCollider *compiles* the code of the program, The result of this process is a function that can be evaluated (i.e. run). Second, SuperCollider *evaluates* the function (i.e. the program) created. Finally, SuperCollider prints out the result of the evaluated function in the Untitled window. Consider following program:

```
3 + 5
```

When one selects the above code and presses [enter], SuperCollider does the following three things: (a) it *compiles* the code contained in the string **"3 + 5"** into the function **{ 3 + 5 }**, (b) it *evaluates* that function, and (c) it posts the result (the number 8) on the post window. In the first step, compilation, SuperCollider inputs the code entered by the programer as a string (text) and translates it into a program as a function object. In the second step, SuperCollider and evaluates the function to obtain its *return value*, which can be any object. The return value of a function is the value of the last statement executed by that function. In this example the return value is the number 8.

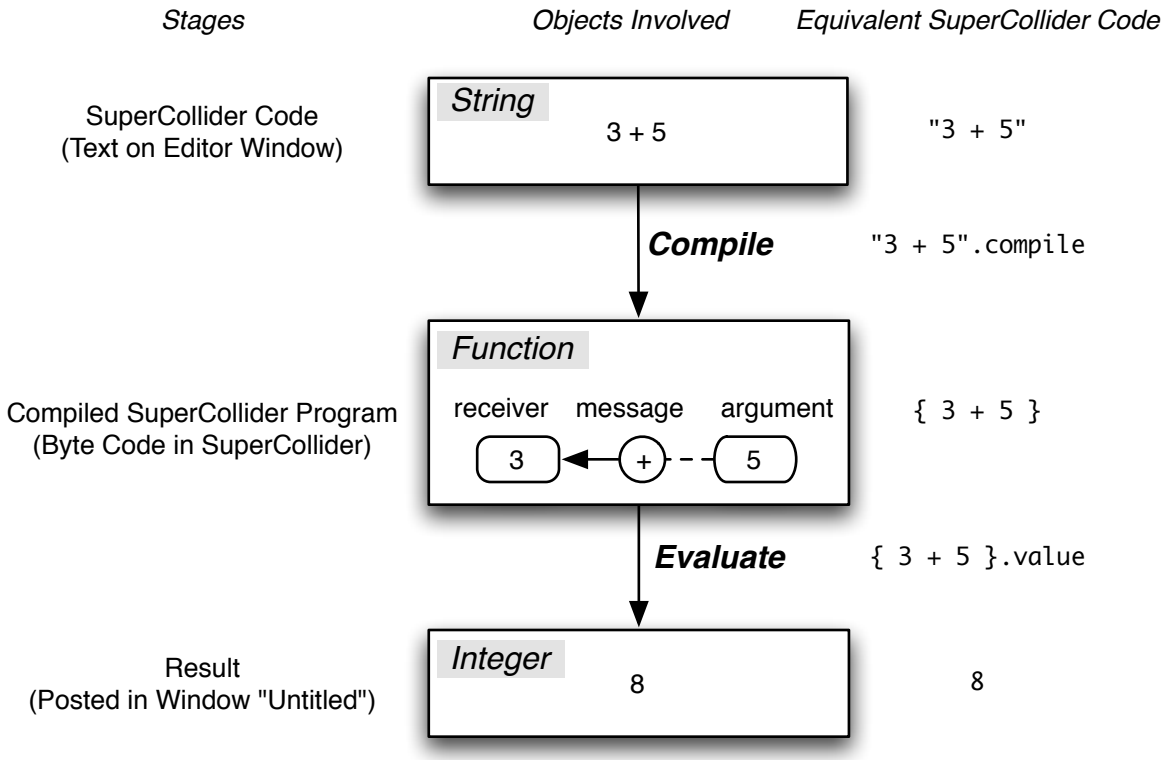


Figure (n) Compiling and Evaluating Code

Figure (n) shows in detail what happens when one runs the code **3 + 5**. The left column shows the three stages of the evaluation process: Text entered by the user, compiled code inside SuperCollider and the final result obtained by the evaluation. The middle column shows the types of objects that are involved in the three stages of the process: (a) The text that was entered, (b) the function that resulted from compiling the text, and (c) the number that resulted from evaluating the function. The right column shows the SuperCollider program code that represents the objects involved at each step. **"3 + 5"** represents the string of text entered by the user. **"3 + 5".compile** is equivalent to the compilation process that turns the string of text into a function. **{ 3 + 5 }** is equivalent to the function that results from the compilation. **{ 3 + 5 }.value** is equivalent to evaluat-

ing the function. The equivalent of the entire compilation plus evaluation process can be expressed by the code: **"3 + 5".interpret**.

Who Does the Compiling?

SuperCollider is constructed to be as transparent as possible, which means that even the top-level processes of interaction with the user such as interpreting a piece of code in a Document Window are defined in terms of objects inside the SuperCollider system. An easy way to look what happens when pressing [enter] is to cause an error and look at the error message. Evaluate for example: **1.error**. The bottom line of the error message shows the top of the hierarchy of messages that start the compilation-interpretation process:

```
Process:interpretPrintCmdLine   14A562F0
      arg this = <instance of Main>
```

Immediately above that is the next method call:

```
Interpreter:interpretPrintCmdLine   15055D00
      arg this = <instance of Interpreter>
```

This shows that the top-level object responsible for compiling and interpreting input from a Document window is an instance of Class **Main**, and that this delegates the interpretation to an instance of **Interpreter**, method **interpretPrintCmdLine**. The code for these methods can be inspected in the source code of classes **Main** and **Interpreter**, as explained in Section *Inspecting Code and Objects*.

Byte-Code: Looking at the Compiled Form of a Function

The compilation process consists in successively replacing the SuperCollider code of the program by pieces of byte-code and data in the computer's memory. The compiler's task is first to parse, i.e. understand the program structure contained in the code, and then to translate that exact structure – including data and instructions – into byte-code. In the above case, the human-readable form of number "3" is translated to a 4-byte representation of the integer 3, likewise the human readable form of the number "5" is translated into a 4-byte representation of the number 5 and the operator "+" is translated into the instruction code for adding one number to the other. To be precise, in the case of SuperCollider, the operator "+" is translated into a message that is sent to the first number, 3, with the second number, 5, as argument. While expressions in SuperCollider code may take many different forms, such as **receiver.message(arguments)** or **message(receiver, arguments)** or **message operator: argument**, their internal structure in compiled byte-code invariably takes a form equivalent to sending an object a message, optionally provided with additional data as arguments.

To display the actual byte-code of a compiled SuperCollider program, one sends the definition of the function representing the program the message **dumpByteCodes**. To obtain the definition of the function, one sends it the message **def**. So, to display the byte-code of the above example **3 + 5**, evaluate this line:

```
{ 3 + 5 }.def.dumpByteCodes
```

This sends first the message **def** to the function to obtain its definition, and then the message **dumpByteCodes** to the definition, to print out the byte-code. Figure n shows the result and explains the different parts of the print-out.

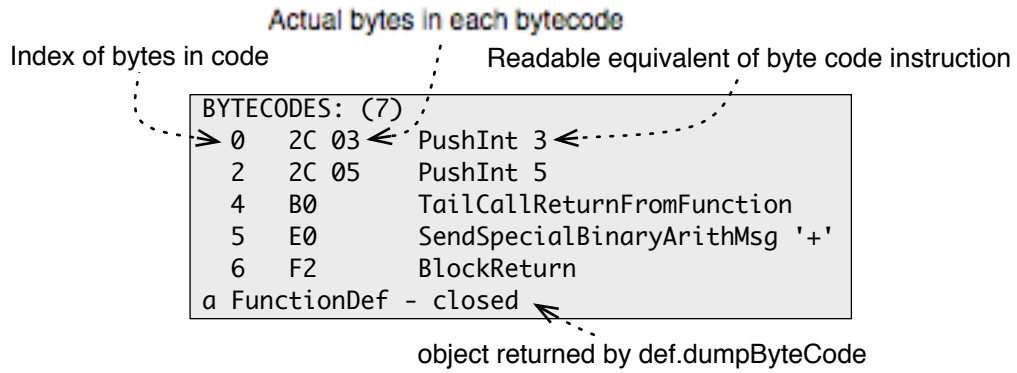


Figure (n) Bytecode of Function { 3 + 5 }

Functions with Arguments

A function can be described as a program module that can have inputs for receiving data from the program that calls them and an output for sending data back to the program. The inputs, if any, are defined right at the beginning of the function, before any variables, and are called *arguments*. Arguments are variables of a function whose values can be set by the program that calls it. When a function needs to run with different sets of data each time, it defines as many arguments as there are data items required. The program can then give data to a function by appending them as arguments in the value message. Here is how to define and call a function that computes and returns the sum of two numbers **a** and **b**:

```
// 1. The code without comments:
(
var sum2;
sum = { arg a, b; a + b }
sum.value(2, 3);
)

// 2. The code with comments:
(
var sum2; // define variable to store the function;
// define the function and store it in variable sum2:
sum2 = { arg a, b;   // start of function definition, arguments a, b
   // the body of the function (the program), is here
   a + b             // compute and return the function of a and b
};                   // end of function definition
// call the function giving it as arguments the numbers 2 and 3:
```

```
sum2.value(2, 3);    // the returned value is 5
)
```

Figure (n) (from the Functions help file) shows graphic representation of four examples of functions that correspond to the four possible combinations of input and output: no input and no output, inputs but no output, no inputs but output, both inputs and output (for the sake of analogy, it is considered that a function with a return value of nil has "no output").

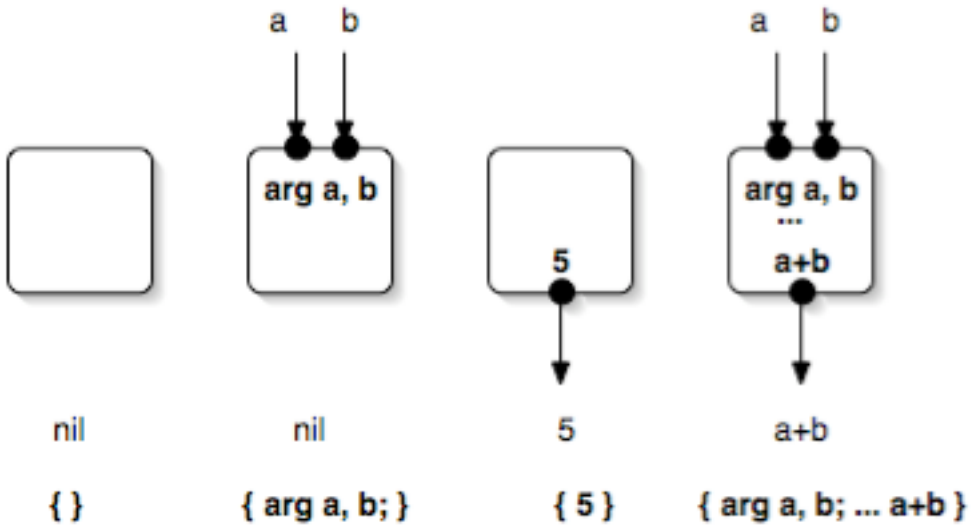


Figure (n): Functions as modules with input and output.

Defining Arguments

In SuperCollider, arguments are defined by prepending the declaration keyword **arg** or by enclosing them in vertical bars | |:

```
(
// a function that calculates the square of the mean of two numbers
var sq_mean;
sq_mean = { arg a, b;   // arguments a, b defined in arg statement form
   (a + b / 2).squared;
};
// calculate the square of the mean of 3 and 1:
sq_mean.value(3, 1);
)
```

Three dots (*..., ellipsis*) can be used to collect any number of provided arguments into one array passed as a single argument to the function. The message **value** can be omitted when running a function with arguments: **foo.value(5)** can be written as: **foo.(5)**:

```
(
// a function that calculates the square of the mean of any numbers
var sq_mean_all;
```

```
sq_mean_all = { | ... numbers | // using ellipsis and | | argument form
   (numbers.sum / numbers.size).squared;
};
// calculate the square of the mean of [1, 3, 5, -7]:
sq_mean_all.(1, 3, 5, -7); // short form: omit message 'value'
)
```

Default Argument Values

The default values of arguments can be included in argument definitions, in the same manner as for variables. A default value is used only if no value was provided for the argument at the call of the function:

```
(
var w_func;
w_func = { arg message = "warning!", bounds = Rect(200, 500, 500, 100);
   var window;
   window = GUI.window.new("message window", bounds).front;
   GUI.textView.new(window, window.view.bounds.insetBy(10, 10))
      .string = message;
};
 // provide text, use default bounds
w_func.(String.new.addAll(Array.new.addAll(" Major news! ").pyramid(7)));
)
```

Function Closures, Instances and Encapsulation

Section *Scope of Variables in Functions* demonstrated the uses of sharing a variable. This section illustrates a further use of the scope of variables inside functions. The example below defines a *mother_func* stored as **counter_maker** which in turn creates and returns a *child_func*. Each time that **counter_maker** is run, it creates a new instance of its *child_func*. Furthermore, it also creates copies of its own variables, in this case the argument-variable **max_count** and the variable **current_count**, which are accessible only to its own child-function.

```
(
// a function that creates a function that counts to any number
var counter_maker;
var window, button1, button2; // gui for testing the function

// the function that makes the counting function
counter_maker = { | max_count |
   // current_count is used by the function created below
   // to store the number of times that it has run
   var current_count = 0;
   {  // start of definition of the counting function
      if (current_count == max_count) {
         format("finished counting to %", max_count).postln;
```

```
            max_count;           // return max count for eventual use
        }{
            current_count = current_count + 1; // increment count
            format("counting % of %", current_count, max_count).postln;
            current_count        // return current count for eventual use
        }
    }  // end of definition of the counting function
};

//  ----- Test application for the counter_maker function -----
// window displaying 2 buttons counting to different numbers
window = GUI.window.new("Counters", Rect(400, 400, 200, 80));
// make a button for triggering the counting:
button1 = GUI.button.new(window, Rect(10, 10, 180, 20));
button1.states = [["counting to 10"]]; // labels for button1
// make a function that counts to 10 and store it as action in button1
button1.action = counter_maker.(10);
button2 = GUI.button.new(window, Rect(10, 40, 180, 20));
button2.states = [["counting to 5"]];  // labels for button2
// make a function that counts to 5 and store it as action in button2
button2.action = counter_maker.(5);
window.front;                          // show the window
)
```

The set of variables created by a function *f* and made available to functions created within that function *f* is called a function's *closure*[3]. So from one mother-function one can create multiple closures, where each closure has its own set of variables and functions and each function in that closure can run multiple times. In this way, one can construct programs that make programs that work on their own copies of data. In the present example, the function stored in **counter_maker** is run once with a **max_count** argument value of 10 and once with **max_count** argument value of 5. Consequently, it creates the first time a function that counts to 10 and the second time one that counts to 5.

The effect of this technique is similar to defining private variables inside objects, called *instance variables*, and the child-functions that have access to these variables are similar to *instances* of a Class that have access to these variables. Section *Modeling Classes with Functions and Events* extends this example to add multiple named functions operating on these variables. These named functions are called *methods*. The property of each function (instance) having exclusive access to its own variables is called *encapsulation.* Figure (n) shows how closures with their own variables are generated from a function.

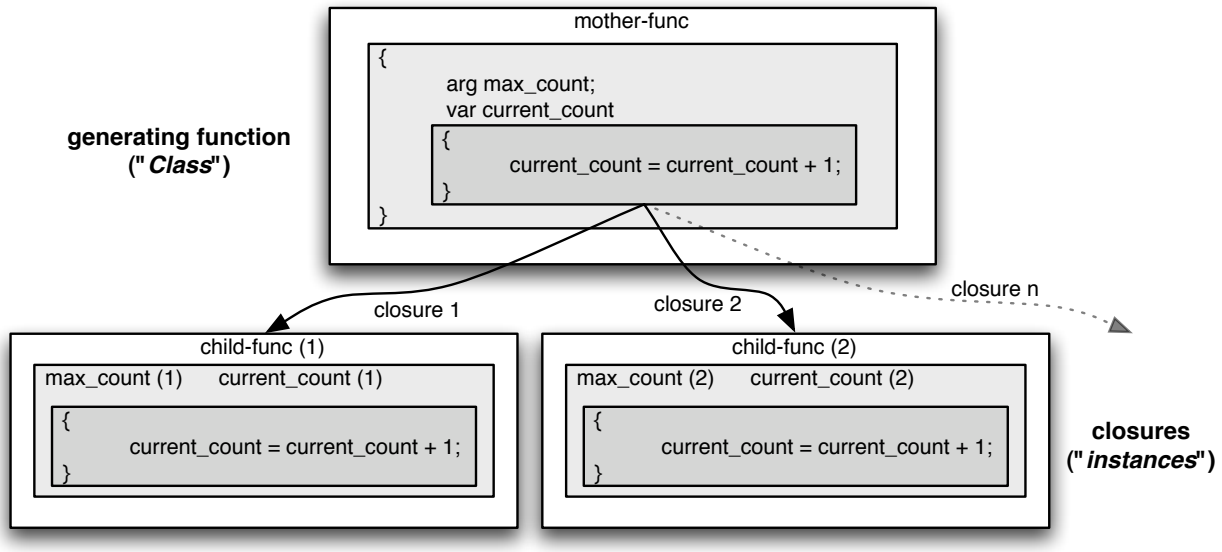


Figure (n) Closures Created by a Function and their Variable Scopes

Since functions are objects in SuperCollider – or more exactly, *"first-class objects"*[4] – their behavior can be easily extended to include other things besides running them with the message **value.** Following sections describe common ways of using functions – some of which have already figured in previous examples.

Customizing the Behavior of Objects with Functions

Several classes of objects that deal with user interface, or with interactive features that should be easily set by the programmer, store functions in variables. Such functions in variables define how an object should react to certain messages. For example, buttons or other GUI widgets use the variable **action** to store the function that should be called when the user activates the widget by mouse-click.

Example 1: The action of the button chooses between two messages to perform on the default Server, depending on the value (state) of the button:

```
(
var window, button;
window = GUI.window.new("Server Button", Rect(400, 400, 200, 200));
button = GUI.button.new(window, Rect(5, 5, 190, 190));
button.states = [["boot"], ["quit"]];
button.action = { |me| Server.default perform: [\quit, \boot][me.value] };
window.front;
)
```

Example 2: The action chooses between two functions depending on the state of the button:

```
(
var window, button;
window = GUI.window.new("Server Button", Rect(400, 400, 200, 200));
```

```
button = GUI.button.new(window, Rect(5, 5, 190, 190));
button.states = [["boot"], ["quit"]];
button.action = { | me |
   [{ "QUITTING THE DEFAULT SERVER".postln;
      Server.default.quit;
   },{ "BOOTING THE DEFAULT SERVER".postln;
      Server.default.boot;
   }][me.value]
};
window.front;
)
```

Functions as Arguments in Messages for Asynchronous Communication

Asynchronous communication happens when a program requests some action from the system, but cannot determine when that action will be completed. For example, it may ask for a file to be loaded or to be printed, but the time required for this to finish is unknown. In such a situation, it would be disruptive to pause the execution of the program while it is waiting for the action to complete. Instead, the program delegates the processing of the answer expected from the action to an independent process – represented by a function – that waits in the background. Two common cases are:

Asynchronous Communication with a Server

The system asks for an action to happen on a Server, as for example to load a sound file into a buffer (**Buffer.read**). Since the time it takes the Server to load the file is not known in advance, a function is given to **read** as argument, which is executed when the server completes loading the buffer. Following example demonstrates that the action passed as argument to the **read** is executed *after* the statement following **Buffer.read**.

```
Server.default.boot // boot default server before running example
(
var buffer;
buffer = Buffer.read(path: "sounds/a11wlk01.wav",
   action: { | buffer |
      format("loaded % at: %", buffer, Main.elapsedTime).postln;
   });
format("Reached this after 'Buffer.read' at: %", Main.elapsedTime).postln;
buffer;
)
```

Dialog Windows

Dialog windows that demand input from the user use an action argument to determine what to do when input is provided. This prevents the system for waiting indefinitely for the user.

```
Server.default.boot // boot default server before running example
```

```
(
Buffer.loadDialog(action: { | buffer |
   format("loaded % at: %", buffer, Main.elapsedTime).postln;
});
format("continuing at: %", Main.elapsedTime).postln;
)
```

Iterating Functions

Iteration is the technique of repeating the same function a number of times. Iteration may be run for a prescribed number of times (**anInteger.do(aFunction)**), an unlimited number of times (**loop(aFunction)**), while a certain condition is true (**while**), or over the elements of a Collection (see section *Iterating in Collections*). For brevity, individual techniques are explained here directly by example.

Iteration for a Number of Times

do: Iterate n number of times, pass the count as argument:
```
10 do: { | i | [i, i.squared, i.isPrime].postln }
```
!: Iterate n number of times, pass the count as argument, collect results in array:
```
{ 10.rand * 3 } ! 5
```
for: Iterate between a minimum and a maximum integer value:
```
30.for(35, { | i | i.postln });
```
forBy: Iterate between two values using a definable step.
```
-2.0.forBy(10, 1.5, { | i | i.postln })
```

Iteration While a Condition is True

The message **while** will repeat evaluating a function argument as long the receiver function returns **true**: **{ [true, false].choose }.while({ "was true".postln; })**. It is usually coded like this:
```
(
var sum = 0;
while { sum = 0.1 exprand: 3 + sum; sum < 10 } { sum.postln }
)
```

Infinite (Indefinite) Loop

loop repeats a function until the process which contains the loop statement is stopped. It can only be used within a process that stops or pauses between statements, otherwise it will crash the system with an infinite loop.
```
Server.default.boot;  // do this first
(                    // then the rest of the program
var window, routine;
window = GUI.window.new("close me to stop").front;
window.onClose = { routine.stop };
routine = {
```

```
    loop {
        (degree: -10 + 30.xrand, dur: 0.05, amp: 0.1.rand).play;
        0.05.rand.wait;
    }
}.fork;
)
```

Partial Application: Shortcut Syntax for Small Functions

It is possible to construct functions that apply arguments to one single message call by using the nameless shortcut _ as placeholder for an argument. For example, instead of writing: **{ arg x; x.isPrime }** one can write **_.isPrime**. If more than one _ is included, then each _ takes the place of a subsequent argument in the function. Examples:

```
_.isPrime ! 10
_.squared ! 10
_@_.(30, 40) // equivalent to: { | a, b | Point(a, b) }.value(30, 40)
Array.rand(12, 0, 1000).clump(4) collect: Rect(*_)
(1..8).collect([\a, \b, _]);
(a: _, b: _, c: _, d: _, e: _).(*Array.rand(5, 0, 100));
```

Recursion

Recursion is a special form of iteration where a function calls itself inside its own code. To do this, the function refers to itself via the pseudo-variable **thisFunction**. (A pseudo-variable is a variable that is created and set by the system and that is not declared anywhere in the SuperCollider Class library. See section *Pseudo-variables* below). The value of **thisFunction** is always the function inside which **thisFunction** is accessed. The following two examples show the difference in implementing the algorithm for computing the factorial of a number iteratively and using recursion. The recursive algorithm is shorter.

```
(
var iterative_factorial;
iterative_factorial = { | n |
    var factorial = 1;   // initialize factorial as factorial of 1
    // calculate factorial n times, updating its value each time
    n do: { | i | factorial = factorial * (i + 1) };
    factorial;  // return the final value of factorial;
};

iterative_factorial.(3).postln;  // 3 factorial: 6
iterative_factorial.(10).postln; // 10 factorial: 3628800
)
```

Recursive factorial definition:

```
// Define the factorial function and store it in variable f:
f = { | x | if ( x > 1) { x * thisFunction.value(x - 1) } { x } };
```

```
f.value(3);          // 3 factorial: 6
f.value(10);         // 10 factorial: 3628800
```

Conciseness is not the only reason for using recursion. There are cases when only a recursive algorithm can be used. Such cases occur when one does not know in advance the structure and size of the data to be explored by the algorithm. For example:

```
(
/* a function that recursively prints all folders and files
   found in a path and its subfolders */
{ | path |
   // store function here for use inside the if's {}:
   var thisFunc = thisFunction;
   format("====== now exploring: %", path).postln;
   // for all items in the path:
   path.pathMatch do: { | p |
      // if the item is a folder, run this function on its contents
      // otherwise print the file found
      if (p.last == $/) { thisFunc.(p ++ "*") }{ p.postln }
   }
}.("*") // run function on home path of SuperCollider
)
```

Timing the Execution of Functions and of Statements Within Functions

Scheduling the Execution of a Function with defer

Sending the message **defer** to a function postpones the evaluation of that function by the time interval specified in the argument (in seconds). For example:

```
(
var foo = { Date.getDate.format("function at %X").postln };
foo.value;        // evaluate f now
foo.defer(3);     // evaluate f 3 seconds later
)
```

Following example shows repeated use of defer to create a sequence of events that are all scheduled with reference to one common time point:

```
Server.default.boot;    // boot the default server first
// When server is booted, send it the SynthDef for the example:
(
// define and send the algorithm for playing the notes
SynthDef("p", { | f = 440 |
      Out.ar(0, Resonz.ar(
         WhiteNoise.ar(EnvGen.kr(Env.perc(0.01, 0.99), doneAction: 2)),
         [f, f * 1.2].scramble, 0.01)
      )
}).send(Server.default);
)
// Each line below is a separate example to be run on its own
```

```
// play 50 notes at intervals of 0.1 seconds
50 do: { | i | { Synth('p', [\f, 200 rrand: 2000]) }.defer(i/10) };
// 100 notes at uniformly distributed random time points (0.1 - 10 sec.)
100 do: { |i| { Synth('p', [\f, 200 rrand: 2000]) }.defer(0.1 rrand: 10)};
// 100 notes at exponentially increasing random time intervals (0.1 - 10)
100 do: {|i| {Synth(\p,[\f,200 rrand: 2000])}.defer(0.1 exprand: 10)};
// 100 notes at exponentially decreasing random time intervals 0.1 - 10
100 do:{|i|{Synth(\p,[\f,200 rrand: 2000])}.defer(10-(0.1 exprand: 9.9))};
```

Routines

Ordinary functions run their statements in one go, without delay between statements. Any musical structures running in real-time, however, must specify time intervals between the execution of parts of code, or else pause execution until a signal has been received from the environment. These capabilities are introduced by Class **Routine**. Inside a Routine one can employ two messages to control timing: **wait** and **yield**. **wait** pauses the Routine for the number of seconds or time beats given by the receiver: 1.wait pauses for one time unit. **yield** is accepted by any object and pauses until the Routine is sent the message **next**.

The message **fork** creates a Routine from a Function and runs it:

```
{ { | i | i.post; 0.25.wait; } ! 10 }.fork
```

If a Routine contains statements that operate on GUI items, with OSC or with MIDI, then it should be "forked" or "played" with an AppClock: **{ }.fork(AppClock)**. (See *Clocks*)

Message **play** starts playing a Routine, **stop** stops it. A Routine that has been stopped must be sent the message **reset** in order to start playing it again:

```
r = Routine({ inf do: { | i | i.post; 0.1.wait } });  // create a routine
r.play;  // start playing the routine
r.stop;  // stop the routine
r.reset; // to restart the routine, one must reset it first
r.play   // now start is possible again
```

The message **yield** is used in a Routine to create temporal structures that advance stepwise in response to **next** messages from other objects:

```
Server.default.boot;
(
r = {
   var synth;
   "send 'next' to start".postln;
   synth = { SinOsc.ar(LFNoise0.kr(10, 1000, 1100), 0, 0.1) }.play;
   "send 'next' to stop".yield;
   synth.free;
   "done".postln;
};
)
r.next; // send next to r to step through it.
```

It is possible to construct structures of arbitrary complexity with yield and wait. One application is to define sections or stages in the execution of a piece that can be stepped through with a button or via a signal from some external device. In such a case, it is convenient to organize the sections of the piece in separate code modules, such as functions, events, or instances of a user-defined class. For example here is the skeleton of a "piece" with three "stages":

```
Server.local.boot; // boot the server first to enable sound
(
f = { // stage 1
   var synth;
   "starting phase 1".postln;
   synth = { SinOsc.ar(LFNoise0.kr(10, 1000, 1100), 0, 0.1) }.play;
   synth.yield;
   synth.free;
   "phase 1 finished. Send 'next' to continue to phase 2".postln.yield;
};
e = { // stage 2
   var pattern;
   "starting phase 2".postln;
   pattern = Pbind(\degree, Pwhite(-10, 10, inf),  \dur, 0.1).play;
   nil.yield;
   pattern.stop;
   "phase 1 finished. Send 'next' to continue to phase 2".postln.yield;
};
g = { // stage 3
   var synth;
   "starting phase 3".postln;
   synth = { LFNoise0.ar(LFNoise0.kr(10, 1000, 1100), 0.1) }.play;
   synth.yield;
   synth.free;
   "phase 3 finished. Send 'next' to continue to end".postln.yield;
};
r = { // the whole piece as one routine
   f.value;
   e.value;
   g.value;
   "piece finished".postln;
   w.close;
}.fork;
w = GUI.window.new("phases of a piece", Rect(200, 200, 250, 250));
b = GUI.button.new(w, w.view.bounds.insetBy(10, 10));
b.states = [["Press twice to advance to next phase"]];
b.action = { r.next };
w.front;
)
```

Patterns and Streams

A limitation of the above approach with routines is that two or more modules that contain **yield** messages cannot run in parallel, because their yield statements will be interleaved in sequential manner by the routine. This limitation is overcome by the class Pattern in combination with Stream. A Pattern is a template that creates a Stream operating on a Routine. A Stream is an object that generates values in response to the message "next". When a Stream no longer has a value to generate, it returns **nil**. A Stream created by a Pattern wraps itself around the Routine of the Pattern in such a way as to permit both the nesting of patterns within patterns and the parallel execution of patterns (when referring to patterns in this way it is customary to mean the streams generated by these patterns which are in a way like "instances" of these patterns). Chapter 11 of the present book covers patterns in depth.

Clocks

Clocks are used to control timing in Routines and **defer** statements. There are three applicable subclasses of Clock:

- **SystemClock** is the most accurate clock.
- **TempoClock** allows the timing of processes according to musical tempi. Several instances of TempoClock can be used to time processes that run in parallel with independent tempi.
- **AppClock** is less accurate than **SysemClock** or **TempoClock** because it runs at a priority level that allows calls to the application framework of SuperCollider. The advantage of this is that processes timed with AppClock can run statements that involve the application framework.

The message *defer* uses the AppClock, and can therefore be wrapped around a statement in a Routine that runs under SystemClock or TempoClock in order to enable it to perform an application framework operation. So one can use either **{ { GUI.window.new.front }.defer }.fork** or **{ GUI.window.new.front }.fork(AppClock)**, depending on the accuracy requirements of the routine application.

Inspecting the Structure of a Function

A particular feature of functions as first class objects is the ability to access a function's parts, which define its structure. For example:

```
var foo;
foo = { | a = 1, b = 2| a.pow(b) };
foo.def.sourceCode.postln; // print sourceCode
```

The source code of a function is stored only if that function is a closed one, that is, if it has not been defined inside another function and shares its variables. A functions def variable is a FunctionDef object which also contains the names of the arguments and

variables of the function as well as the default values defined for these. This is used by SynthDef to compile a function into a UGen graph, that is an that runs on the Server.

Program Flow Control and Design Patterns

Control structures are structures that permit to choose the evaluation of a function depending on a condition. That is, a function *f* is evaluated only if the value of a condition *c* is true. There are variants involving one or more functions. (For alternative syntax forms see help file *Syntax-Shortcuts*.)

if Statements

Run a function only if a condition is true:

```
if ([true, false].choose) { "was true".postln }
```

Run a function if a condition is true, otherwise run another function:

```
if ([true, false].choose) { "was true".postln } { "was false".postln }
```

case Statements

A case statement is a sequence of function pairs of the form "condition-action". The condition functions are evaluated in sequence, until one of them returns true. Then the action function is evaluated and the rest of the pairs are ignored. One can add a single default action function at the end of the pairs sequence, which will be executed if none of the condition functions returns **true**.

```
(
var i, x, z;
z = [0, 1, inf];
i = z.choose;
x = case
   { i == 0 }   { \no }
   { i == 1 } { \yes }
   { \infinity };
x.postln;
)
```

switch Statements

A switch statement matches a given value to a series of alternatives by checking for equality. If a match is found, the function corresponding to that match is evaluated. The form of the switch statement is similar to that of the case statement. The difference is that the switch statement uses a fixed test – that of equality with a given value – while the case statement uses a series of independent functions as tests.

```
(
switch ([0, 1, inf].choose,
   0, { \no },
```

```
    1, { \yes },
    { \infinity };
)
)
```

Other Control Techniques, Behavior Patterns

Selecting among alternatives or directing the execution flow of a program is not limited to the statements above. There are many techniques addressing this topic, some of which are also known as *Design Patterns* (Gamma 1995, Beck 1996). Typically, techniques of this category would fall under the group of *Behavior Patterns*. Examples of such patterns are: *Chain of Responsibility*, *Command*, *Iterator*, *Mediator*, *Observer* and *State*. The observer pattern is discussed in section *The Observer Pattern* below. Examples of some techniques have already been given above. Kent Beck (1996) classifies Behavior Patterns into two major categories: Under section "Method" he lists patterns that are based on the organization of an algorithm inside methods. Under "Message" patterns he classifies patterns which use message passing to create algorithms. These patterns can be very small but equally powerful. An example is the *Choosing Message* pattern (Beck 1996: 45-47). Instead of choosing amongst a number of alternatives with an **if** statement or a **switch** statement, one delegates the choice to the methods of the possible objects involved. For example, assume that one wants an object that represents an entry in a list of publications to respond to the message **responsible** by returning some object that represents the name of the person that is responsible for the object. Now for film publications, the responsible is the producer, for edited books it is the editor, for single author books it is the author. The Choosing Message pattern says that instead of writing:

```
responsible { | entry |
   case
      { entry.isKindOf(Film) } { ^entry.producer }
      { entry.isKindOf(EditedBook) } { ^entry.editor }
      { ^entry.author }    // in all other cases, return the author
}
```

One writes:

```
responsible { | entry | ^entry.responsible }
```

and then codes the different reactions to responsible in the classes of the objects that are involved:

```
// add method 'responsible' in 3 previously defined classes:
+ Publication { responsible { ^author } }
+ Film { responsible { ^producer } }
+ EditedBook { responsible { ^editor } }
```

In this example, Publication is the default class for entries and gives the default method; all other classes for entries are subclasses of Publication. Only those classes which deviate from the default **responsible** method need redefine it. (See section *Class* below for syntax of methods and Class extensions.)

The power of this technique is first that the number of choices can be extended easily by creating new classes, and second that the method **responsible** for each Class can be as complex as needed, without resulting in a huge case statement that aggregates all the choices for "responsible" in one place. In other words, complexity is reduced – or rather broken down to pieces in an elegant way – by delegating responsibility for different parts of the algorithm to different classes. Thus, as a general tendency, algorithms are organized by the combination of a number of method calls, which split up the algorithm into pieces and possibly delegate the responsibility of each piece to different objects. As a result methods tend to contain very little code, often just a single line. While this may seem confusing at the first encounter, it gets clearer as one becomes familiar with the style of code that pervades good Object Oriented Programing.

Encapsulation, Inheritance, Polymorphism

The three defining properties of Object Oriented Languages have now been introduced: Section *Function Closures, Instances and Encapsulation* introduced encapsulation, that is the creation of variables that are private to a single object instance. The above example has shown a use for *polymorphism* as well as for *inheritance*: Polymorphism says that the same message can correspond to different behaviors according to the Class of the object that receives it. In this case, an entry of Class **Film** responds differently to the message responsible than an entry of Class **EditedBook**. Inheritance on the other hand entails that any subclass of **Publication** that does not define its own method **responsible** will use the method as defined in **Publication** instead. Together, these three features are responsible for the capabilities of Object Oriented Languages.

Collections

Collections are objects that hold a variable number of other objects. For example, here is a program that adds a new number to a sequence each time that the user clicks on a button, and then plays the sequence as a "melody":

```
Server.default.boot; // boot the server first;
(
var degrees, window, button;
window = GUI.window.new("melodies?", Rect(400, 400, 200, 200));
button = GUI.button.new(window, window.view.bounds.insetBy(10, 10));
button.states = [["click me to add a note"]];
button.action = {
   degrees = degrees add: 0.rrand(15);
   Pbind(\degree, Pseq(degrees), \dur, Prand([0.1, 0.2, 0.4], inf)).play;
};
window.front;
)
```

The above example builds a sequence of notes by adding a new random integer between 0 and 15 each time. It exploits the feature that adding an element to nil builds an array with the added element: **nil add: 1** creates **[1]**, to build a sequence starting from **nil**, that is from the value of the uninitialized variable **degrees** in the beginning.

The subclass tree of **Collection** is extensive (**Collection.dumpClassSubtree**), and is summarized in help file *Collections*. Collections can be classified into three kinds according to the way in which their elements are accessed:

1. Collections whose elements are accessed by numeric index. For example, [0, 5, 9].at(0) accesses the first element of the array [0, 5, 9], and [0, 5, 9].put(1, \hello) puts the symbol \hello into the second position of array [0, 5, 9]. Such collections are: **Array**, **List**, **Interval**, **Range**, **Array2D**, **Signal**, **Wavetable, String** and others. Numeric indexes in SuperCollider start at 0, that is 0 refers to the first element in a collection. Accessing an element at an index past the size of the collection returns nil. There are however alternative messages for access – **wrapAt**, **clipAt**, **foldAt** – which modify invalid index numbers to always return some element. A subcategory is formed by collections that hold only a specific kind of object, such as Char (String), Symbol (SymbolArray), Float (Signal, Wavetable).
2. Collections whose elements are accessed by using a symbol, or by another object, as index. For example **(a: 1, b: 2)[\a]** returns **1**. Such collections are: **Dictionary**, **IdentityDictionary**, **Library** (a nested dictionary that can be accessed by series of objects as index), **Environment** and **Event**. All such collections are made up of Association objects, that are pairs that associate a key to a value, and are written as **key->value**. Although it is possible to look up such pairs both by key and by value, Dictionaries are optimized for look-up by key.
3. Collections whose elements are accessed by searching for a match to a condition. For example: **Set[1, 2, 3, 4, 5] select: (_ > 2)**. These Collections are **Set** and **Bag**.

Creating Collections

The generic rule for creating a Collection is to enclose its elements in brackets [], separating each element by comma. If the class of a collection is other than Array, it is indicated before the brackets: **List[1, 2, 3]; LinkedList[1, 2, 3]; Signal[1, 2, 3]; Dictionary[\a->1, 2->pi, \c->'alpha']; Set[1, 2, 3]**.

Additionally there are several alternative techniques for notating and generating specific types of collections:

- An arithmetic series can be abbreviated by giving the beginning and end value, and optionally the step between subsequent values: **(1..5); (1, 1.2 .. 5);**
- An event can be written as a pair of parentheses enclosing a list of the associations of the event written as keyword-value pairs: **(a: 1, b: 2)**
- Environments and Events can be created from functions with the message **make** (see *Environment* below).

- There are several messages for constructing numerical Arrays algorithmically. For example: **Array.series(5, 3, 1.5); Array.geom(3, 4, 5); Array.rand(5, -10, 10);**
- Wavetables and Signals are raw arrays of floating point numbers that can be created from functions a such as sine or Chebychev polynomials or window shapes such as Welch.
- The class **Harmonics** constructs Arrays that can be used as wavetables for playing sounds with the **UGen Osc** and its relatives.

Binary Operators on Collections

Most binary operators on collections can work both between two collections of any sizes and between a collection and a non-collection object: (0..6) < (3..0); (0..6) + (3..0); 10 * (1..3); (2..5) + 0.1. One can append an adverb to a binary operator to specify the manner in which the elements of two collections are paired for the operation. For example:

```
[10, 20, 30, 40, 50] + [1, 2, 3] // default: shorter array wraps
[10, 20, 30, 40, 50] +.s [1, 2, 3] // s = short. operate on shorter array
[10, 20, 30, 40, 50] +.f [1, 2, 3] // f = fold. Use folded indexing
```

Iterating in Collections

Following messages iterate a function over each element of a collection:

- **do(foo)**: Evaluate function over each element, return the receiver. **(1..5) do: _.postln**
- **collect(foo)**: Evaluate function over each element, return the collected results of each evaluation. (1..5) collect: _.sqrt
- pairsDo(foo):
- inject(foo):
- keysDo, keysValuesDo, associationsDo, pairsDo, keysValuesChange: These work on Dictionaries as follows:

```
(a: 10, b: 20) keysDo: { | key, index | [key, index].postln }
(a: 10, b: 20) keysValuesDo: { | k, v, i | [k, v, i].postln }
(a: 10, b: 20) associationsDo: { | assoc, index | [assoc, index].postln }
(a: 10, b: 20) pairsDo: { | k, v, i | [k, v, i].postln }
(a: 10, b: 20) keysValuesChange: { | key, value, index | value + index }
```

Searching in Collections

Following messages search for matches and return either a subset or a single element from a Collection:

- **select(foo)**: Return those elements for which foo returns true. **(1..5) select: (_ > 2)**
- **reject(foo)**: Return those elements for which foo returns false. **(1..5) reject: (_ > 2)**
- **detect(foo)**: Return the first element for which foo returns true:
  "asdfg" detect: { | c | c.ascii > 100 }
- **indexOf(obj)**: Return the index of the first element that matches obj:
  "asdfg" indexOf: $f

- **includes(obj)**: Return true if the receiver includes obj in its elements. **"asdfg" includes: $f**
- **matchRegexp(string, start, end)**: Perform POSIX style matching of regular expressions on a String

Restructuring Collections

A full account of the structure-manipulation features of the SuperCollider language would require a chapter of its own. Here are some examples:

- **reverse**: Reverse the order of the elements. **(1..5).reverse**
- **flop**: Turn rows into columns in a two-dimensional collection. **[[1, 2][\a, \b]].flop**
- **scramble**: Rearrange the elements in random order. **(1..5).scramble**
- **clump(n)**: Create sub-collections of size n. **(1..10).clump(3)**
- **stutter(n)**: Repeat each element n times. **(1..5).stutter(3)**
- **pyramid(n)**, where 1 <= n <= 10: Rearrange in quasi repetitive patterns. **(1..5).pyramid(5)**
- **sort(foo)**: Sort using foo as sorting function. Default sorts in ascending order: **"asdfg".sort**. Descending order is specified like this: **"asdfg" sort: { | a, b | a > b }**

Further powerful restructuring, combinatorial and search capabilities are provided by J Concepts in SC and by List Comprehensions (see related help files).

IdentityDictionary

IdentityDictionary is a Dictionary that retrieves its values by looking for a key identical to the given index. Identical means that the key should be the same object as the index. For example, the two strings "hello" and "hello" are equal but not identical:

```
"hello" == "hello"; // true: the two strings are equal
"hello" === "hello"; // false: the two strings are not identical
```

By contrast, symbols that are written the same characters are always stored as one object by the compiler, and are therefore identical: **\hello === \hello** returns **true**. Thus:

```
a = IdentityDictionary["foo"->1]; // store 1 under the "foo" as key
a["foo"]; // nil! The second "foo" is not identical to the first one
// but:
a = IdentityDictionary[\foo->1];
a[\foo]; // Returns 1
```

The search for a matching object by identity is much faster than that for equality. Therefore, an IdentityDictionary is optimized for speed. It serves as superclass for Environment, which is the basis for defining environment variables. Accessing an environment variable thus means looking it up by identity match. While this is a fast process, it is still considerably more expensive in computing cycles than accessing a "real" variable!

IdentityDictionary defines two instance variables: proto and parent. These are used by the classes Environment and Event to provide a default environment when needed

(see section *Event*). The parent scheme makes it possible to build hierarchies of parent events in a similar way as Class hierarchies.

Environment

An Environment is an IdentityDictionary that can evaluate functions which contain environment variables (notated with ~). The use of environment variables has been introduced in chapter 5. This section explains how to create environments from functions and how to run functions explicitly in an Environment.

To create an environment from a function, use the message make:

```
Environment make: { ~a = 10; ~b = 1 + pi * 7.rand; }
```

This is not just convenient for notation, it also permits to compute variables that are dependent on the value of variables previously created in the environment:

```
Environment make: { ~a = pi + 10.rand ; ~b = ~a pow: 5 }
```

To evaluate a function in an environment use the message **use**.

```
Environment make: { ~c = 3 } use: { ~a = 2 pow: 10.rand; ~c + a }
```

Environment.use(f) evaluates f in an empty environment:

```
Environment use: { ~a = 10; ~b = 1 + pi * 7.rand; ~c }
```

Additionally, an Environment can supply values from its variables to the arguments of a function that is evaluated in it with the message **valueEnvir**. Only values for those arguments that are not provided by **valueEnvir** are supplied:

```
(a: 1, b: 2).use({ ~a + ~b });      // using environment variables
// Supplying arguments to a function from the environment with valueEnvir
(a: 1, b: 2).use({ { | a, b | a + b }.valueEnvir(3) })
// Not the correct way to supply arguments with use:
(a: 1, b: 2).use({ | a, b | a + b })
// valueEnvir in a Document window uses the currentEnvironment:
~a = 3; ~b = 5;
{ | a, b | a + b }.valueEnvir(3)
```

Patterns employ the ability of to supply values for arguments from an environment with **valueEnvir** when playing instruments that are defined as functions.

Event

Event is a subclass of Environment with several additional features developed for playing Patterns. An event itself is playable: **(degree: 2, dur: 3).play.**

Event stores several prototype events in its class variables that contain default event types for playing patterns (a class variable is globally available to the instances of is Class and its subclasses). These events define a complete musical environment, covering aspects such as tuning, scales, legato, chords and chord strumming, midi and playing with different instruments. To play, an event receives or selects itself a parent event as environment, and overrides only those items of the event that deviate from the default settings. For example, the parent event of **(degree: 5)** is nil before playing: (degree:

5).parent. To run **(degree: 5).play**, the event sets its own parent event, which can be printed by: **(degree: 5).play.parent.asCompileString.** The parameters of this environment also compute and set the final parameters that are needed to play the event. In the present example, these are **freq, amp** and **sustain**, as can be seen in the resulting event in the post window:

```
// Run each line separately
Server.default.boot;
(degree: 5).parent; // the parent before playing is nil
(degree: 5).play.parent.asCompileString; // The parent has been set
(degree: 5).play; // event, becomes ('degree': 5, 'freq': 440, ...)
```

Playing Patterns with Events

To achieve maximum flexibility, the final synthesis parameters are defined as functions of higher-level musical parameters. It is possible to specify a parameter at any one of several levels of musical concepts. For example, pitch can be provided at the levels of degree, midi-note or frequency, so to play a note of a = 440 Hz one may write **(degree: 5).play;** or **(midinote: 69).play;** or **(freq: 440).play;** The resulting network of parameter functions is quite extensive (currently, over 400 lines of code for the default parent event). Thus to play, a pattern overrides specific parameters of the default parent environment at each event according to the specification of the pattern. The key pattern for playing patterns is **Pbind** (see chapter 11). Here is an pattern that plays a jingle in 7-tone equal temperament:

```
Server.default.boot;    // boot the server first
f = Pseq([Pbind(
   \degree, Pseq([Pseq([Pseq((-10..20)), Pwhite(10, 20, 10)], 2),
      Pn(\pause, 4), { rrand(-10, 10) } ! 5, Pn(\pause, 3)]),
   \dur, 0.1
), (degree: { rrand(-10, 20) } ! 10, dur: 3)]).play(SystemClock,
   (stepsPerOctave: 7, scale: #[0, 2, 3, 5, 6]));
```

Modeling Classes with Functions and Events

This section extends the counter example of section *Function Closures* to add a further feature: The ability of each counter to reset itself. It also shows a more flexible technique for creating a graphical user interface: instead of a fixed number of counter items, a function is defined that can generate a GUI for any number of counters, whose maximum counts are given as arguments to the function. Instead of a function, the **counter_maker** in this example returns an event. The event contains three environment variables **count1**, **reset_count** and **max_count** bound to functions that operate on the variables of the counter_maker closure. These three functions bound to variable names are examples of the way instance methods work in Classes. Thus, an event made by counter_maker is the model of an object with two variables and three methods. The

code this example is hardly any bigger than the previous version, despite the addition of 2 features.

```
(
var counter_maker;      // creator of counters
var make_counters_gui;  // function making counters + a gui
/* a function that creates an event that counts to any number,
   and resets: */
counter_maker = { | max_count |
   var current_count = 0;
   (  // the counter object is an event with 3 functions:
      count1: // function 1: increment count (stored as count1)
      {  // start of definition of the counting function
         if (current_count == max_count) {
            format("finished counting to %", max_count).postln;
         }{
            current_count = current_count + 1; // increment count
            format("counting % of %", current_count, max_count).postln;
         }
      }, // end of definition of the counting function
      reset_count: { // function 2: reset count (stored as reset_count)
         format("resetting % counter", max_count).postln;
         current_count = 0
      },
      max_count: { max_count } // function 3: return value of max_count
   )
};
// Function that makes several counters and a GUI to control them
make_counters_gui = { | ... counts |
   var window, counter;
   window = GUI.window.new("Counters",
         Rect(400, 400, 200, 50 * counts.size + 10));
   // enable automatic placement of new items in window:
   window.view.decorator = FlowLayout(window.view.bounds, 5@5, 5@5);
   counts collect: counter_maker.(_) do: { | counter |
      GUI.button.new(window, Rect(0, 0, 190, 20))
         .states_([["Counting to: " ++ counter.max_count.asString]])
         .action = { counter.count1 };
      GUI.button.new(window, Rect(0, 0, 190, 20))
         .states_([["Reset"]])
         .action = { counter.reset_count };
   };
   window.front;
};
make_counters_gui.(5, 10, 27); // example use of the GUI test function
)
```

The above example can be seen as a rudimentary class definition constructed without employing the regular Class definition system of SuperCollider. The **counter_maker** function is like a Class "Counter" that has two "instance variables": **max_count** and **current_count**. The event that it creates contains three "methods", that are the functions stored in **count1**, **reset_count** and **max_count**.

It is left to the reader to extend the example in one further step, by storing the functions of **counter_maker** and **make_counters_gui** in an event, which will then play the role of a Class **Counter** with two class methods: one for creating instances (**counter_maker**) and one of creating a window with buttons operating on instances (**make_counters_gui**).

The syntax for running the functions stored in an event is the same as a method call done by sending a message to a receiver (**receiver.message**), the difference being only the first argument passed to a function in an event is always the event itself. This is not only convenient, but also indicates the potential of events to act as class-prototypes . There is one catch: If one stores in an Event a function under the name of an instance method that is defined in the Class **Event**, then that method will be run, instead of the function stored by the user. So for example one cannot use a function stored in an event under **reset**: **(reset: { "this is never called".postln; }).reset**;

Classes

Classes are the heart of the SuperCollider system, because they define the structure and behavior of all objects. All Class definitions are contained in the folder SCClassLibrary, in files that end in .sc or .sc.rtf. By studying these definitions one can understand the function of any part of the system in depth. By writing one's own classes or modifying existing classes, one can extend the functionality of the system.

Compiling the SuperCollider Class Library

In contrast to code executed from a Document Window, which can be run at any time, changes made in Class definition code take effect only after compiling the SuperCollider Class Library. This is done from the menu item Lang->Compile Library (keyboard shortcut: [command]-K). Compiling the library rebuilds all classes and resets the entire memory of the system.

Classes and Instances

A Class describes the attributes and behavior that are common to a group of objects. All objects belonging to a class are called *instances* of that class. For example, all integer numbers such as 0, -1, 50 etc are instances of Class **Integer**. All integers are able to perform arithmetic operations on other numbers, therefore the Class **Integer** describes – among other things – how integers perform arithmetic operations. Instances are created

as literals (for example, 1, -10, $a, \a), with one of the constructor syntax forms ({}, (), a@b, a->b) or by sending a message to a Class that demands an instance. The most common message for creating instances is new, and can therefore be omitted:

Rect.new(10, 20, 30, 40) is equivalent to: **Rect(10, 20, 30, 40)**

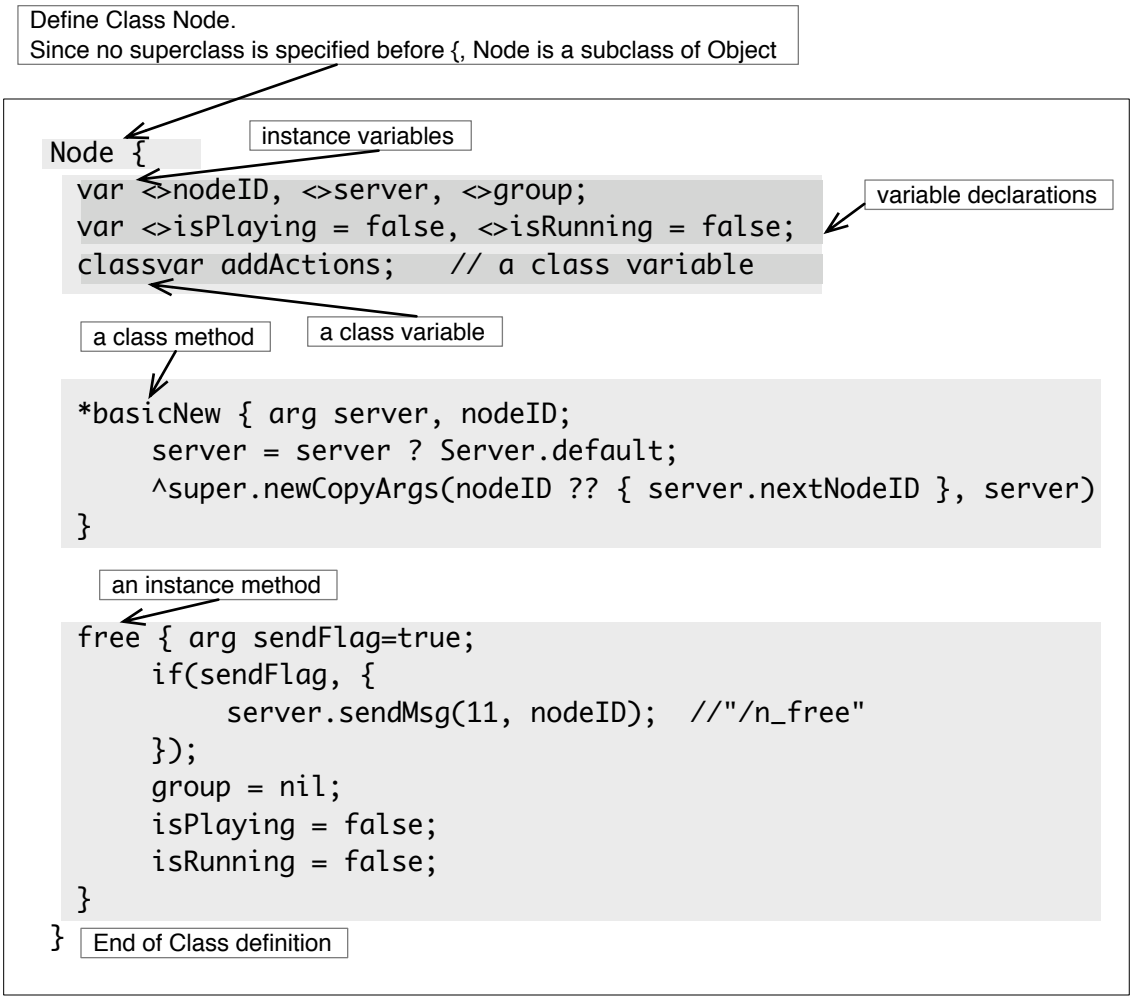
Defining a Class

The structure of a class is defined by its variables and its methods. Section *Modeling Classes with Functions and Events* has given an example of how instance variables and methods work, without making use of a class definition. Additionally, a Class may define class variables, constants and class methods.

Class variables are accessible by the Class itself as well as by all instances, while instance variables are only accessible inside methods of the instance in question. Constants are like class variables, except their values are set at the definition statement, and cannot be changed subsequently. For example, the Class **Char** defines several constants that hold the unprintable characters for new line, form feed, tab and space as well as the character comma.

Class methods are addressed to the class; instance methods to instances of that class. For example

A class may inherit variables and methods from another class, which is called its *superclass*. Inheritance works upwards over several superclasses, and always up to the superclass of all classes: **Object**. Before explaining the role and syntax of each element in detail, here is an example showing the main parts (Figure n):




```
Node {
    var <>nodeID, <>server, <>group;
    var <>isPlaying = false, <>isRunning = false;
    classvar addActions;   // a class variable

    *basicNew { arg server, nodeID;
        server = server ? Server.default;
        ^super.newCopyArgs(nodeID ?? { server.nextNodeID }, server)
    }

    free { arg sendFlag=true;
        if(sendFlag, {
            server.sendMsg(11, nodeID);  //"/n_free"
        });
        group = nil;
        isPlaying = false;
        isRunning = false;
    }
}
```

Figure (n) Summary of Class Definition Parts (Excerpt from Definition of Class Node)

As seen in figure (n) the code that defines a Class has two major characteristics in common with with that of a Function: It is enclosed in {} and it starts with variable declarations followed by program code. The code of a Class definition is organized in two sections: (a) Variable declarations and (b) method definitions. (No program statements may be included in the definition of a Class other than those contained in variable declarations and methods). Class syntax can be summarized in the following points:

1. The name of the Class is prepended at the start of the definition. If the Class has a superclass other than Object then it is written like this:
   `Integer : SimpleNumber { // define Integer as subclass of SimpleNumber`
2. In addition to **var** statements that declare instance variables, there can also be **classvar** statements that declare class variables and **const** statements that create constants. For example, Class **Document** has a class variable **allDocuments** that stores

all Document windows. One can close these with: **Document.allDocuments do: _.close.** Class **Char** has several **const** statements declaring special characters.

3. The special signs < and > prepended to a variable name in a variable declaration statement construct corresponding methods for getting or setting the value of that variable:
```
var <freq; // constructs method: freq { ^freq };
var >freq: // constructs method: freq_ { | argFreq | freq = argFreq }
```
For example, the Class definition **Thing { var <>x; }** is equivalent to:
```
Thing { var x;
   x { ^x }
   x_ { arg z; x = z; }
}
```
4. After the declaration of any variables of a Class follow the definitions of its methods. A method is defined by the name of the method followed by the definition of the function that is executed by that method.
5. The sign * before a method's name creates a class method:
```
*new { arg x=0, y=0; ^super.newCopyArgs(x, y); } // (from Class Point)
```
6. The default return value of a method is the instance that is executing that method (the receiver of the message that triggered the method). To return a different value, one writes the sign ^ before the statement whose value must be returned. **freq { ^freq }:** the method **freq** returns the value of the variable **freq**. The sign ^ also has the effect of "returning" from the function of the method, which means any further statements will not be executed after it. This effect can be useful:
```
count1 {
   if (current_count >= max_count) { ^current_count };
   // the next statement is executed only if current_count > max_count:
   ^current_count = current_count + 1;
}
```
7. Identifiers starting with underscore (_) inside methods call *primitives*, that is, computations that are done by compiled code in the system, and whose code can only be seen in the C++ source code of the SuperCollider application. A primitive returns a value if it can be called successfully. Otherwise, execution continues to the next statement of the method's code.
```
   *newCopyArgs { arg ... args; // (from class Object)
      _BasicNewCopyArgsToInstVars
      ^this.primitiveFailed
   }
```
8. Three special keywords can be used in methods: **this** refers to the object that is running the method (the *receiver*). **thisMethod** refers to the method that is running. **super** followed by a message looks up and evaluates the method of the message in the superclass of the instance that is running the method.

9. If the Class method ***initClass** is defined, then it will be run right after the system is compiled. This is used to initialize any data needed. To indicate that a Class needs to be initialized *before* the present initClass is run, one writes in the code of initClass: **Class.initClassTree(NameOfClassToBeInitialized)**.
10. A Class is usually defined in one file. If the same Class name is found in definitions in two or more files, then the compiler issues the message: **duplicate Class found:** followed by the name of the duplicate Class. However, one can extend or modify a Class by adding or overwriting methods in a separate file. The syntax for adding methods to an existing class is:

```
+ Function {   // + indicates this extends an existing Class
   // the code of any methods comes here
   update { | ... args |      // method update
      this.valueArray(args);
   }     // other methods can follow here
}
```

Inheritance

A Class may inherit the properties of another Class. This principle of inheritance helps organize program code by grouping common shared properties of objects in one class, and by defining subclasses to differentiate the properties and behavior of objects that have more specialized character. For example, the Class **Integer** inherits the properties of the Class **SimpleNumber**. **SimpleNumber** is called the *superclass* of **Integer**, while **Integer** is called a *subclass* of **SimpleNumber**. **Float**, the Class describing floating-point number such as 0.1, is also a subclass of **SimpleNumber**. Classes are thus organized into families with a tree-like structure. The following expression prints out the complete SuperCollider Class tree: **Object.dumpClassSubtree**.

Meta-Classes

Since all entities in SuperCollider are objects, classes are themselves objects. Each class is the sole instance of its "*meta-class*". For example, the class of Integer is Meta_Integer, and consequently Integer is the only instance of the class Meta_Integer. All meta-classes are instances of Class. Following examples trace the successive classes of objects starting from the Integer 1 and up to Class as the Class of all Meta-Classes:

```
1.class     // the class of Integer 1: Integer
1.class.class  // the Class of the Class of Integer 1: Meta_Integer
// the Class of the Class of the Class of Integer 1:
1.class.class.class                 // Class
// the Class of the Class of the Class of the Class of Integer 1
1.class.class.class.class           // Meta_Class
// the Class of the Class of the Class of the Class of the Class of 1
1.class.class.class.class.class     // Class
```

The cycle Class-Meta_Class-Class in the example above shows the end of the Class-relationship tree. Since the Class of Class is Meta_Class and Meta_Class is also a Class, those two Classes are the only objects that are instances of each other:

```
Class.class          // the Class of Class is Meta_Class
Meta_Class.class     // the Class of Meta_Class is Class
```

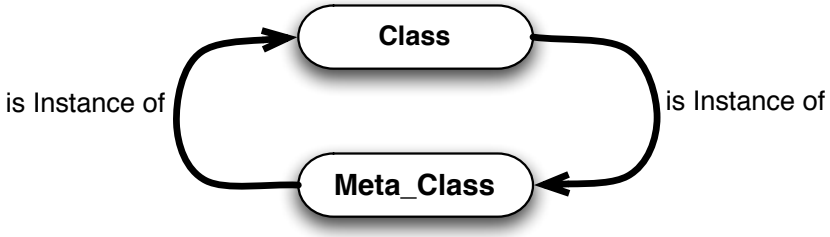


*Figure (n): **Class** and **Meta_Class** are mutually instances of each other.*

Class methods are equivalent to instance methods of the Meta-Class. For instance, the class method ***new** of **Server** is actually an instance method of **Meta_Server**.

The SuperCollider Class Tree

At the top of the class hierarchy of SuperCollider is the class Object. This means all other classes inherit from class Object as its subclasses, and consequently all objects in SuperCollider share the characteristics and behavior defined in class Object. Object defines such global behaviors as how to create an instance, how an object should react to a message that is not understood, how to print the representation of an object as text, etc. Any subclass can override this default behavior in its own code, in addition to extending it by defining new variables and methods. The tree formed by Object and its subclasses thus describes all classes in the SuperCollider system. Here is an overview of some common classes, classified in terms of their use:

Basic Infrastructure:

- Object, Class, Main, Interpreter, Nil, Boolean, Platform, Date

File I/O:

- File, UnixFile, Pipe, SoundFile, Directory, PathName

Magnitudes (Numbers and Characters):

- Integer, Float, Polar, Complex, Char

Collections

- Collections Indexable by Number: Array, List, Signal, Wavetable, String, Interval, Range, Harmonics (creates wavetables)
- Collections Indexable by Symbol or other object: Dictionary, Library, Environment, Event

- Non-indexable Collections: Set, Bag

Computational Processes

- Function, Thread, Routine,

Server and Sound Synthesis Objects

- Server: Server, ServerOptions
- Allocation of Resources on Server: NodeIDAllocator PowerOfTwoAllocator LRUNumberAllocator StackNumberAllocator RingNumberAllocator ContiguousBlock ContiguousBlockAllocator
- Monitoring the Server: NodeWatcher
- Data and Signal Flow on Server: Buffer, Bus
- Synthesis Processes on Server: SynthDef, Node, Group, Synth, all UGens

Pattern and (Sound) Event Structure Generators

- Stream, Pattern, PauseStream,

Graphical User Interface

- Windows: SCWindow, Document, SCFreqScope, ScopeView, Stethoscope, Inspector
- Views in Windows: SCView, SCButton, SCNumberBox SCSlider, SC2DSlider, SC2DTabletSlider, SCRangeSlider, SCStaticText, SCKnob, SCPopUpMenu, SCTextField, SCTextView SCTabletView SCUserView, SCMultiSliderView, SCEnvelopeView, SCMovieView SCDragSink SCDragBoth SCDragSource, SCScope, SCFreqScope, SCSoundFileView, SCListView, SCQuartzComposerView

Communication and Interfaces to External Devices

- OSC: OSCresponder, OSCresponderNode
- MIDI: MIDIIn, MIDIOut, MIDIClient MIDIResponder, NoteOnResponder, NoteOffResponder, CCResponder, TouchResponder
- HID: HIDDevice HIDInfo HIDDeviceElement HIDDeviceService

Notifying Objects of Changes: Observer and Adapter / Controller Patterns

This section shows how to convert the class model of section *Modeling Classes with Functions and Events* into a *real* Class. It furthermore makes that counter a subclass of Model. The Observer design pattern implemented by Model allows one to attach objects to any object so that these get updated when that object notifies itself with the message **changed**. Thus it is possible to attach sounds, GUI elements or any other object or process to an object and make it respond to changes of that object in any manner, without having to change the Class definition of the object. This technique is similar to the de-

sign pattern known as Model-View-Controller (MVC). The goal of this pattern is to separate data or processes (the model) from their display (views) and from the control mechanisms, to permit multiple displays across different media and platforms.

The present example adds auditory displays as well as a GUI displays that respond to counter changes. These displays are completely independent from each other and from the counter both in code and in functionality, in the sense that one can attach a display or remove it from any counter at any moment, and that one can attach any number of displays to one counter.

The definition of the Counter class is:

```
Counter : Model {
   // variables: maximum count, current count
   var <>max_count, <>current_count = 0;
   // class method for creating a new instance
   *new { | max_count = 10 |
      ^super.new.max_count_(max_count)
   }
   // if maximum count not reached, increment count by 1
   count1 {
      if (current_count == max_count) {
         this.changed(\max_reached)
      }{
         current_count = current_count + 1;
         this.changed(\count, current_count);
      }
   }
   // reset count to 0
   reset {
      current_count = 0;
      this.changed(\reset);
   }
}
```

This must be placed in a file Counter.sc in the SCClassLibrary folder, and compiled with [command-K]. After that, boot the server and load the SynthDefs for the sounds:

```
Server.default.boot;
(
SynthDef("ping", { | freq = 440 |
   Out.ar(0,
      SinOsc.ar(freq, 0,
         EnvGen.kr(Env.perc(level: 0.1), doneAction: 2)
   ))
}).send(Server.default);

SynthDef("wham", {
   Out.ar(0, BrownNoise.ar(
```

```
        EnvGen.kr(Env.perc(level: 0.1), doneAction: 2)
    ))
}).send(Server.default);
)
```

Next create five counters and store them in ~counters:

```
~counters = (5, 10 .. 25) collect: Counter.new(_);
```

Now create a sound-adapter to follow changes in any counter it is added to:

```
(
~sound_adapter =  { | counter, what, count |
    switch (what,
        \reset, { Synth("wham"); },
        \max_reached, { counter.reset },
        \count, { Synth("ping",
            [\freq, count.postln * 10 + counter.max_count * 20]
            )
        }
    )
};
)
```

The sound_adapter function receives **update** messages from a counter object and translates them to actions according to the further arguments of the message. In this sense it is similar to an Adapter pattern. It can also be compared to a Controller pattern in that it responds to event notifications from the system.

Attach the sound-adapter to all five counters:

```
~counters do: _.addDependant(~sound_adapter);
```

And start a routine that increments the counters at 1/4 second intervals:

```
~count = { loop { ~counters do: _.count1; 0.25.wait } }.fork;
```

The routine can be stopped with **~count.stop**. But before doing that, lets add GUI displays for the counters:

```
(
~make_display = { | counter |
    var window,label, adapter;
    window = GUI.window.new(
        "counting to " ++ counter.max_count.asString,
        Rect(400, 400, 200, 50)
    );
    label = GUI.staticText.new(window, window.view.bounds.insetBy(10, 10));
    adapter = { | counter, what, count |
        switch (what,
            \reset, { { label.string = "0"}.defer },
            \count, {
                { label.string = counter.current_count.asString }.defer
            }
        )
```

```
    };
    counter addDependant: adapter;
    /* remove the adapter when window closes to prevent error in
       updating non-existent views: */
    window.onClose = { counter removeDependant: adapter };
    window.front
};
)
```

Now one can make displays for any of the counters at any time:

```
~make_display.(~counters[0]);
~make_display.(~counters[2]); // etc.
```

The Observer pattern is considered so important that it is enabled for all Objects – not just Model subclasses. The present example refers to the Model class for the sake of explanation, as its code implements the Observer pattern succinctly.

Conclusion: Further Reading, Programing Techniques, Libraries

The present chapter has attempted to describe the programing language of SuperCollider and its capabilities in as much detail as possible in the given space. It also introduced some techniques of programing that may serve as an introduction to advanced programing. Many other techniques exist. A great deal of these are described in print and on the web in publications that deal with design patterns for programing. Kent Beck's *Smalltalk Best Practice Patterns* (Beck 1996) is recommendable as basic manual of good style, and because the patterns it describes are as powerful as they are small. Gamma (1995) is considered a standard book on patterns. Beck (2000) and Fowler and Beck (1999) deal with more advanced techniques of coding.

The SuperCollider class library itself is a good source for learning more about programming techniques. The GUI class implements the Factory pattern. The Lilt library (included in the DVD of this book) makes extensive use of the Observer pattern and defines a class Script that enables one to code algorithms for performance in prototypes which create their own GUIs.

SuperCollider as an open source project depends on the active participation of members of the community to continue developing as one of the most advanced environments for sound synthesis around. Contributions by musicians and programers, through suggestions and bug-reports to the sc-devel mailing list, through quarks in the quark repository, or through proposals for inclusion in the SCClassLibrary itself, are vital for the further development of this environment. At this stage, while SuperCollider has already gained a considerable amount of popularity, there is still much room for growth. One of the most attractive aspects of this environment is that it is equally a tool for music making, experimentation, research and learning about programing and sound. The features and capabilities of the SuperCollider programing language outlined

in the present chapter can serve as a springboard for projects that will further expand its capabilities and user base. It remains to be seen whether the trend for coding as a musically creative activity matures to become widely accepted musical practice. Yet whatever the future may bring, the particular marriage of tool-making and music-making that SuperCollider embodies so successfully will mark it as an exceptional achievement, and hopefully give birth to further original ideas and amazing sounds.

Endnotes

[1] Two relevant definitions of statements are: "Computer Science An elementary instruction in a programming language." (http://www.thefreedictionary.com/statement) and: "A statement is a block of code that does something. An assignment statement assigns a value to a variable. A for statement performs a loop. In C, C++ and C# Statements can be grouped together as one statement using curly brackets" (http://cplus.about.com/od/glossar1/g/statementdefn.htm). In SuperCollider, statements enclosed in {} create a function object, which is different than a statement group in C or C++.

[2] Definition from Merriam-Webster's Online Dictionary

[3] Wikipedia writes about closures:

> "In computer science, a closure is a function that is evaluated in an environment containing one or more bound variables. When called, the function can access these variables. The explicit use of closures is associated with functional programming and with languages such as ML and Lisp. Constructs such as objects in other languages can also be modeled with closures."

[4] When a program can construct functions while it is running and store these as objects in variables, it is said that it treats functions as "first class objects" (Burstall 2000).