5 Programming in SuperCollider

Iannis Zannos

5.1 Introduction

This chapter provides an introduction to the SuperCollider programming language from a more technical viewpoint than David Cottle's beginner's tutorial (chapter 1). Some material presented there and elsewhere in this book is also covered here, but in a more methodical and exhaustive manner. Although I try to convey programming skills without presupposing any previous knowledge of programming languages and compiler technology, some of the more advanced programming concepts in this chapter can take a little getting used to. The explanations provided here aim to be complete within the space allowed, but readers new to object-oriented programming may wish to seek out a good general introductory text to accompany their explorations (<http://en.wikipedia.org/wiki/Object-oriented_programming> isn't bad!). Musicians new to computer music may prefer to start earlier in the book and return here only once they have acquired some basic familiarity with the language; for experienced computer musicians new to SuperCollider, or experienced programmers new to audio, this chapter may be the preferred entry route. For everyone, it should provide a useful reference on the SuperCollider language.

Mechanisms underlying the interpretation and execution of programs and the programming concepts of SuperCollider will be explained. This will serve as a basis for understanding how to write and debug effectively in SuperCollider. We will consider the 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?
What are the characteristic techniques of object-oriented programming, and how are they applied in SuperCollider?

SuperCollider employs syntactic elements from C, C++, Java, Smalltalk, and Matlab, creating a style that is both concise and easy to understand for programmers who know one of these common programming languages. A summary of the SuperCollider language syntax is given in the appendix of this book.

5.2 Fundamental Elements of Programs

5.2.1 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. It also means that the way these entities are defined is uniform, as are the means for communicating with them.

5.2.1.1 Objects

Objects are the basic entities that are manipulated within programs. They bundle together data and methods that can act on that data (a musical scale object might store the pitches in the scale and a method to play back those pitches up and down as a sequence of notes in order). In the simplest case they might look like a single number or letter (a character), but they still respond to a number of methods for acting on themselves (return the negative of the current number, return the ASCII key code for the letter) or to more complex methods combining multiple objects of that type (add this number to another to make a third number, combine this letter with another to make a 2-letter word). In practice, there are 2 main types of objects, categorized 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 prove useful for handling big sets of data, such as a library of musical tunings or a mass of partial frequency, amplitude, and phase data for additive synthesis.

Some examples of objects include those shown in figure 5.1. You can see various different objects in the code listing, from simple numbers and letters to more abstract types that will be explained in more detail during the course of this chapter.

5.2.1.2 Classes

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 (e.g., 0, –1, and 50) 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. In-

```
1                   // the Integer number 1
1.234               // the floating-point (Float) number 1.234
$a                  // the character (Char) a
"hello"             // a String (an array of characters)
\alpha              // a Symbol (a unique identifier)
'alpha 1'           // another notation for a Symbol
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
```

Figure 5.1
Objects.

stances are created as literals (which act as primitives of the language, such as 1, –10, $a, \a; more on this below) with one of the shortcut constructor syntax forms (e.g., {}, (), a@b, a->b) or by sending a special message to a class that demands an instance. The most common message for creating instances is `new`, which can be omitted for brevity: `Rect.new(10, 20, 30, 40)` is equivalent to `Rect(10, 20, 30, 40)`, and both create a rectangle with the specified numbers determining its size and position.

A class can inherit properties and behavior from another class, called its *superclass*. A class that inherits properties is a *subclass* of the class from which it inherits. The mechanism of inheritance is central in object-oriented programming for defining a hierarchical family tree of categories that relate to each other. This promotes sharing of common functionality while allowing the specialization of classes for particular tasks. (We'll return to this later.)

5.2.2 Literals

Literals are objects whose value is represented directly in the code (rather than computed as a result of sending a message to an object). Literals in SuperCollider are the following:

Integers (e.g., –10, 0, 123) and floating-point numbers (e.g., –0.1, 0.0, 123.4567), which can be in exponential notation (e.g., 1e4, 1.2e-4); alternative radices up to base 36 (e.g., binary for 13 is 2r1101, and the hexadecimal for 13 is 16rD); or combined with the constant pi (e.g., 2pi, –0.13pi).
Strings, enclosed in double quotes: "a string."
Symbols, enclosed in single quotes ('a symbol') or preceded by \:\a_Symbol.

Literal Arrays: immutable Arrays of literals declared by prepending the number sign #.
Classes: A class is represented by its name. Class names start with a capital letter.
Characters (instances of Char), a single character preceded by the dollar sign $ (e.g., $A, $a); non-printing characters (tab, linefeed, carriage return) and backslash are preceded by a backslash (e.g., $\n, $\t, $\\).
Variables and constants (see also the SuperCollider Help file on Literals and the appendix).

5.2.3 Messages and Methods

To interact with an object, one sends it a *message*. For example, to calculate the square of a number, one sends 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 code that is stored in the *method* which has the same name as the message, then returns a result to the calling program, which is called the *return value*. In other words, a method is a function stored under a message name for an object that can be recalled by sending that object the message's name (see figure 5.2).

Instance methods operate on an instance (such as the integer 1), and *class methods* operate on a class. The most commonly used class method is New, which is used to create new instances from a class.

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

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

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

5.2.3.1 Chaining messages

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

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

Or this:

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

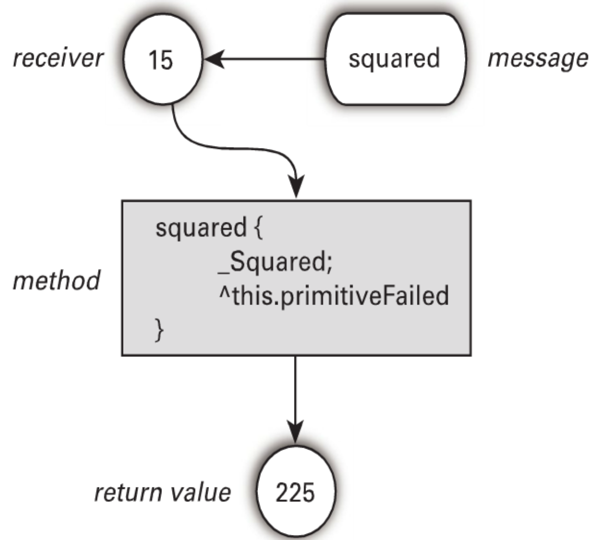

Figure 5.2
Receiver, message, method, return value.

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 the following:

```
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
```

5.2.3.2 Performing messages

In some cases, the message to be sent to an object may change, depending on other conditions. When the message is not known in advance, the messages perform and performList are used, which allow an object to perform a message passed as an 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 one to pass additional arguments to the message in Array form. Thus `Rect.performList(\new, [0, 10, 200, 20])` is equivalent to `Rect.new(0, 10, 200, 20)`. (See also the example in section 5.4.5.)

5.2.4 Arguments

The operation of a message often requires the interaction of several objects. For example, raising a number to some power involves 2 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, they are separated by commas:

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

The same is true in the "function-call" format:

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

If the arguments are provided as 1 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 arguments as a 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 the lowest, and 10 as the highest, possible value.

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

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

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

5.2.4.1 Argument forms for implied messages `at` and `put`

When square brackets are appended to an object, they imply the message at or put (this follows the C or Java syntax for Array indexing). Thus `[1, 5, 12][1]` is equivalent to `[1, 5, 12].at(1)`, and `()[\a] = pi` is equivalent to `().put(\a, pi)`.

5.2.4.2 Argument keywords

When calling a function, argument values must be provided in the order in which the arguments were defined (see section 5.4.4.1). However, when only a few out of many

```
// 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;
```

Figure 5.3
Keyword arguments.

arguments of a function need to be provided, one can specify those arguments by name in "keyword" form; for instance, if the name of the argument provided is freq, the call is `foo.value(freq: 400)`. And the kr method for XLine takes the arguments start, end, dur, mul, add, and 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, respectively; doneAction gets the value 2; and mul and add rely on their default values 1 and 0, respectively. (See figure 5.3.)

5.2.5 Binary Operators

SuperCollider uses signs from mathematics, logic, and other programming languages, such as + (addition), – (subtraction), and & (binary "and"). These are called binary operators because they operate on 2 objects. For example, ++ joins 2 SequenceableCollections: `[\a, \b] ++ [1, 2, 3]`.

Furthermore, any message that requires just 1 argument can be written as a binary operator by adding : to the name of the message. Thus, `5.pow(8)` can also be written as `5 pow: 8`. With this and other syntax shortcuts included in this chapter, there may seem to be a bewildering variety of alternatives available. SuperCollider supports a few different common programming syntaxes, but in vanilla SC, the 'dot' notation would be most common, and with practice you can pick up additional syntax as you

```
((1 + 2).asString).interpret            // = 3
"1" ++ "2". interpret                   // 12: 2 is translated to string by ++
("1" ++ "2").interpret                  // 12
(1.asString ++ 2.asString).interpret    // 12
"1+2". interpret                        // 3
(1.asString ++ "+2"). interpret         // 3
(1 + 2).interpret                       // error: interpret not understood by
Integer 3
```

Figure 5.4
Grouping and Precedence.

code and gain experience. Further details are available in the Syntax Shortcuts Help file.

5.2.6 Precedence Rules and Grouping

When one combines several operations in 1 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, with 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 simple but differ somewhat from those used in mathematics:

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)`.
Message passing, as in receiver.message(arguments) or in collection[index], has precedence over binary operators. Thus, in `10 * (1..3).addAll([0.1, 0.2, 0.3])` the elements of `[0.1, 0.2, 0.3]` are first appended to `[1, 2, 3]`, and then the resulting new Array is multiplied by 10.
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.
```

Figure 5.4 illustrates the effects of grouping by parentheses and message passing.

5.2.7 Statements

The single-line code examples introduced above normally constitute parts of larger programs that include many lines of code. The smallest stand-alone elements of code

```
(
a = 5;
5 do: { a = a + 10; a.postln };
Post << "The value of variable 'a' is now " << a << "\n";
)
```

Figure 5.5
Statements.

are called *statements*.[1] One creates programs by grouping sequences of statements. When a program contains more than 1 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. Figure 5.5 contains 3 statements.

The first statement (`a = 5;`) assigns the value 5 to variable a. The second statement (`5 do: {a = a + 10; a.postln};`) repeats a function 5 times that assigns to a its previous value incremented by 10, and posts the new value of a each time. The third statement (`Post << "The value of variable 'a' is now "<< a << "\n";`) posts the new value of a.

It is important to distinguish between the lines of code text in a Document window as seen by a human programmer, and the part of the code that SuperCollider processes as program when the programmer runs a selected portion of that code. SuperCollider does not run the whole code in the window, but only the code that was selected; or, if no code is selected, the line on which the cursor is currently located. Every time that one runs a piece of code, 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 it in parentheses. This is useful because one can select it easily, typically by double clicking to the right of an opening parenthesis.

5.3 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. (See figure 5.6.)

One creates variables by declaring them with the prefix `var`. Several variables can be declared in one var statement. Variables may be declared only at the beginning of a function (or a selected block of code, which is essentially the same thing; see section 5.4.3).

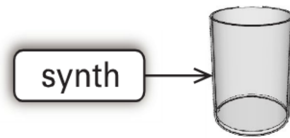

Figure 5.6
Variable as a label pointing to a container.

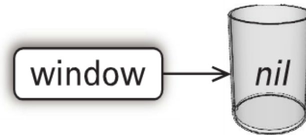

Figure 5.7
`nil` stands for the contents of an empty variable.

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

When a variable is first created, it is empty, so its value is represented by the object nil, which is the object for no value. (See figure 5.7.)

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

One cannot run the lines of a program that use a declared variable separately; one must always run the code as a whole. This is because the variables declared in the beginning of a function disappear from memory as soon as the function that declared them finishes, unless other functions within that function also use them. In figure 5.7, running the line `window.postln;` alone produces the error message `Variable 'window' not defined.`

To store an object in a variable, use the assignment sign =. For example, after storing a Window in the variable window, one can send it messages to change its state, as well as use it as an argument to other objects. (See figure 5.8.)

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

5.3.1 Variable Initialization

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

```
(
// A window with a button that posts: "hello there!"
var window, button;
// create a GUI window and store it in variable window
window = Window.new("OLA!", Rect(200, 200, 120, 120));
// create a button in the window and store it in variable button
button = 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
)
```

Figure 5.8
Variables can store objects that need to be used many times.

```
(
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
)
```

5.3.2 Use of Variables

The object stored in a variable remains there until a new assignment statement replaces it with something else. Variables also are often used as temporary placeholders to operate on a changing choice from a set of objects. Figure 5.9 is an example that makes extensive use of variables to create a chain of upward- and downward-moving runs of short tones.

5.3.3 Instance Variables

An instance variable is a variable that is contained in a single object. Such a variable is accessible only inside instance methods of that object—unless special code is written to make it accessible to other objects. For example, objects of class Point have 2 instance variables, x and y, corresponding to the coordinates of a point in 2-dimensional space. (See figure 5.10.)

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

```
(
// execute this first to boot the server and load the synth definition
Server.default.waitForBoot({
    SynthDef("ping", { | freq = 440 |
        Out.ar(0,
            SinOsc.ar([freq, freq * (4/3)], 0,
                EnvGen.kr(Env.perc(0.05, 0.3, 0.1, -4), doneAction: 2)
            )
        )
    }).add
});
)

(
// execute this next to create the sounds
var countdown = 100;
var note = 50;
var increment_func, decrement_func;
var action;
increment_func = {
    note = note + [2, 5, 7, 12].choose;
    if (note > 100) { action = decrement_func };
};
decrement_func = {
    note = note - [1, 2, 5, 7, 12].choose;
    if (note < 50) { action = increment_func };
};
action = increment_func;
{
    countdown do: {
        Synth("ping", [\freq, note.midicps]);
        action.value;
        0.1.wait;
    }
}.fork;
)
```

Figure 5.9
Variables can point to different objects during a process.

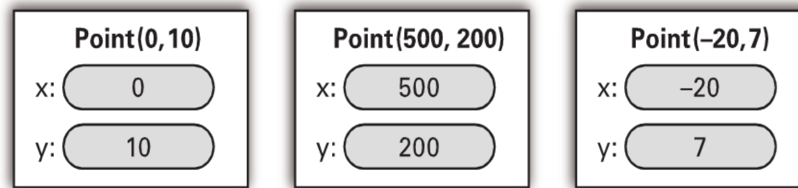

Figure 5.10
Three instances of `Point` with their instance variables.

5.3.4 Class Variables

A class variable is defined once for the class it belongs to and is shared with all its subclasses. It is accessible to class methods and to instance methods of its class and all its subclasses. For example, the class variable `all` of OSCresponder holds all currently active instances of OSCresponder. The instance method add of OSCresponder adds a responder instance to the class variable `all`, and the method remove removes a responder from `all`. In that way the system keeps track of all responders that are active, and checks every OSC message received to see if it matches any of the responders contained in `all`. One can write `OSC.all do: _.remove` to remove all currently active OSCresponders.

5.3.5 Environment Variables

Environment variables are preceded by a tilde (~). For example, ~a = pi. These reference the value of a named variable in the current Environment, a special holding place for data. They do not need to be declared, but are instantly added to the Environment when assigned. An Environment is a kind of Dictionary that represents the set of bindings of values to names; that is, the Environment variables. These bindings differ from those created by normal variable declarations in that they have a less limited scope (though not truly "global" variables in the traditional sense, they can sometimes be treated as such), and they can be modified more easily (see section 5.6.7 for more details).

The relationship between Environment variables and the Environment that contains them can be seen by printing the current Environment. (See figure 5.11.)

5.3.6 Variables with Special Uses

The variables described in this section provide access to objects that are useful or indispensable, but either cannot be accessed by conventional programming within the SuperCollider class system or need to be accessed by all objects in the system.

```
// 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
```

Figure 5.11
`currentEnvironment`.

5.3.6.1 Interpreter variables

The class Interpreter defines 26 instance variables whose names correspond to the lowercase letters a to z. Since all code evaluated at runtime is run by an instance of Interpreter, these variables are accessible within that code without having to be declared. However, this works only when evaluating code from outside of class definitions, that is, with code selected to be run by the Interpreter. The following example can be executed one line at a time (first boot the default server with Server.default .boot).

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

5.3.6.2 Pseudo variables

Pseudo variables are not declared anywhere in the SuperCollider library but are provided by the compiler. They are the following:

`this` represents the object that is running the current method. In runtime code this is always the current instance of Interpreter (see also section 5.4.3.1). Thus one can run `this.dump` to view the contents of the current Interpreter instance, including the variables a–z.
`thisProcess` is the process that is running the current code. It is always an instance of Main. Although 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` is 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` is the innermost function within which the current statement is running. It is indispensable for recursion in functions (see section 5.4.9).
`thisFunctionDef` is 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. Section 5.4.10 briefly discusses its uses.
`thisThread` is the thread running the current code. A thread is a sequence of execution that can run in parallel with other threads and can control the timing of the execution of individual statements in the program. Examples of the use of thisThread are found in the 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 can 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 method new of Object to create a new instance of ListPattern—but again adds some stuff of its own.

5.3.6.3 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` is the Environment being used right now by the running program. This can be changed by the programmer.
`topEnvironment` is the original currentEnvironment of the Interpreter instance that runs programs in the system. It can be accessed independently of currentEnvironment, which changes in response to Environment's use or make methods. (See figure 5.12)
`uniqueMethods` holds a dictionary that stores unique methods of objects. Unique methods are methods that are defined not in a class but only in a single instance. For example:

```
(
~a = "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
)
```

Figure 5.12
`topEnvironment` versus `currentEnvironment`.

```
(
// create 2 windows and store them in variables p, q
#p, q = [100, 400].collect {|i|
    Window(i.asString, Rect(i, i, 200, 200)).front
}
)
// add a unique method to p only
p.addUniqueMethod(\greet, {|w| w.name = "Hello!"});
p.greet;    // p understands 'greet'
q.greet;    // but q does not understand 'greet'
```

`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 the object has changed, one sends the message changed. Details of this technique are explained in section 5.7.7.

5.3.7 Variables versus References

A variable is a container with which one can do only 2 things: store an object and retrieve that object. One cannot store the container itself in another container, so 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. (See figures 5.13 and 5.14.)

To store a container in a variable, one uses a reference object.

```
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
```

```
(
var alpha, beta, gamma;
gamma = alpha;  // storing variable alpha in gamma only stores nil
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 5.13
Variables store only values, not other variables.

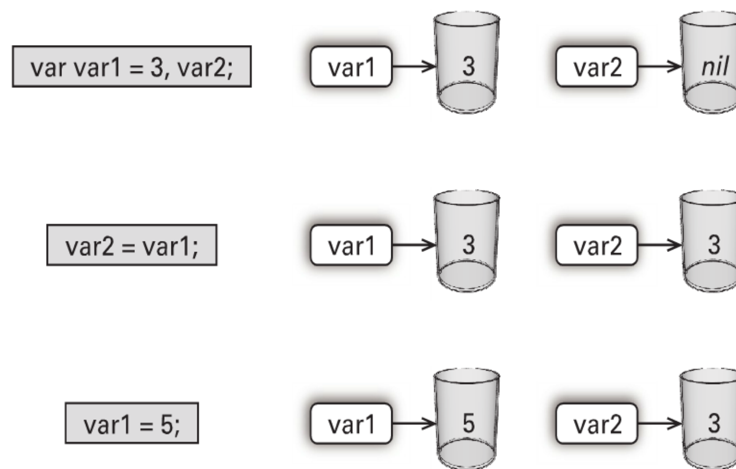

Figure 5.14
Assigning a variable to another variable stores its contents only.

```
aref.value = 10;    // change the value of the reference in aref
cvar.value.postln;     // and retrieve that value from cvar
```

5.4 Functions

A function is an object representing a bit of code that can be evaluated from other code at runtime. In this sense it is like a miniature program. One can “package” a 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 the *definition* of the function. When a program runs a function, it is said to *call* or *evaluate* that function.

To “package” a code into a function, one encloses it in braces {}.

```
{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.

5.4.1 Return Value versus Side Effect

The use of the term “evaluate” here comes from the idea of requesting a value that is computed and returned by the function for further use. The return value of a function is the value of the last statement that is computed in the function. However, in many cases one calls a function not to obtain a final value but to start a process that will result in some change, such as to create sounds or to show graphics on the screen. For example, {10.rand} provides a random number between 0 and 9 as a return value, and {Window.allWindows do: _.close} closes all GUI windows. It is the effect of the latter, rather than the return value, that matters.

This is also true for methods. In the example presented in section 5.2.3.1, `Server.local.boot`, the message `local` is sent to class Server to obtain the object representing the local server as a return value, whereas the message `boot` is sent to the local server in order 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.

5.4.2 Functions as Program Modules

Since functions are objects that can be stored in variables, it is easy to define and store any number of functions (i.e., miniature programs) and run them, whenever required, any number of times. Thus, defining functions and configuring their combinations can be a major part of programming in SuperCollider.

Figure 5.15 illustrates how to call a function that has been stored in a variable in various ways. The function change_freq in the example does 2 things:

It calculates a new frequency for the sound by moving 1 minor tone upward or downward from the previous pitch.
It sets the frequency of the sound to the new pitch.

The code of the function consists of 2 statements:

```
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 = Window("Buttons Archaic", Rect(400, 400, 340, 120));
//  ------------------------- Example 1 -------------------------
button1 = Button(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 = Button(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 = Button(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
)
```

Figure 5.15
Multiple use of a function stored in a variable.

```
{
    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 the variable change_freq and then called in 2 different ways:

It is stored in the action part of a GUI button so that when that button is pressed, it runs the function.
It is called explicitly by a function inside a Routine that sends it the message value. (As noted in chapter 3, Routine has the ability to time the execution of its statements, and therefore can run the function in question at timed intervals.)

5.4.3 Compilation and Evaluation: The Details

SuperCollider undergoes a 3-step process every time that it executes code entered in a work space window: First, it *compiles* the code of the program and creates a function that can be evaluated. Second, SuperCollider *evaluates* that function. Finally, SuperCollider prints out the result of the evaluation in the *post* window.

Figure 5.16 shows what happens when one runs the code 3 + 5. The equivalent of the entire compilation plus evaluation process can be expressed by the code "3 + 5". interpret.

5.4.3.1 Who does the compiling?

In SuperCollider, even the top-level processes of interaction with the user are defined in terms of objects inside the system. An easy way to see what happens is to cause an error and look at the error message. For example, evaluate: 1.error. The bottom line shows the beginning of the compilation 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 text input is an instance of class Main, and that it delegates the interpretation to an instance of Interpreter, calling the method interpretPrintCmdLine.

5.4.3.2 Byte code: Looking at the compiled form of a function

The compilation process consists in successively replacing the SuperCollider code of the program with pieces of byte code and data in the computer's memory. The

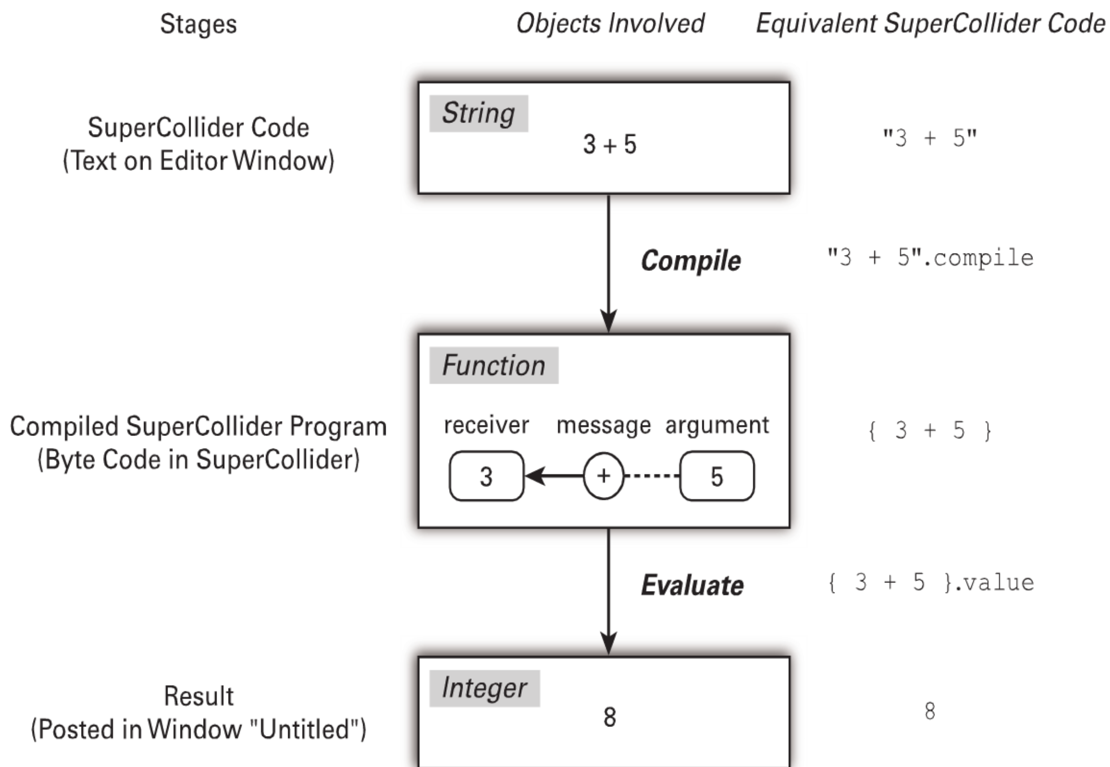

Figure 5.16
Compiling and Evaluating Code.

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. 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`. Thus, to display the byte code of the above example 3 + 5, evaluate this line:

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

This first sends 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 5.17 explains the resulting printout.

5.4.4 Functions with Arguments

Functions can have inputs for receiving data from the context that calls them. The inputs, if any, are defined at the beginning of the function, before any variables, and

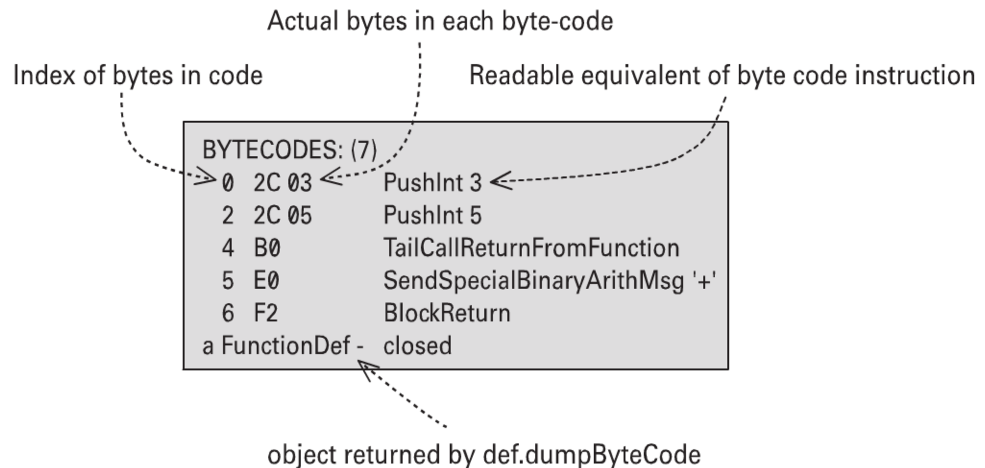

Figure 5.17
Bytecode of Function `{ 3 + 5 }`.

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 be evaluated 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 2 numbers, a and b:

```
(
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 the numbers 2 and 3as arguments:
sum2.value(2, 3);       // the returned value is 5
)
```

See the Functions Help file for a depiction of the metaphorical "inputs" and "output" of a function.

5.4.4.1 Defining arguments

In SuperCollider, arguments are defined by prepending the declaration keyword arg or by enclosing them in vertical bars ||. (See figure 5.18.)

Three dots (. . . argName) before the final (or only) argument name in the argument list can be used to collect any number of provided arguments into 1 array passed as a single argument to the function. (See figure 5.19.)

```
(
// 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);
)
```

Figure 5.18
Simple function with arguments.

```
(
// 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'
)
```

Figure 5.19
Using ... for undefined number of arguments.

5.4.4.2 Default argument values

The default values of arguments can be included in argument definitions, in the same manner as variables. A default value is used only if no value was provided for the argument when the function was called. (See figure 5.20.)

Since functions are objects in SuperCollider—or, more exactly, "*first-class objects*"[2]—their behavior can easily be extended to include other things besides running them with the message value. The following sections describe common ways of using functions.

5.4.5 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

```
(
var w_func;
w_func = { arg message = "warning!", bounds = Rect(200, 500, 500, 100);
    var window;
    window = Window("message window", bounds).front;
    TextView(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)));
)
```

Figure 5.20
Using and overriding default values of arguments.

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

Figure 5.21
Performing messages chosen by index.

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 a mouse click.

Example 1. The action of the button chooses between 2 messages to perform on the default Server, depending on the value (state) of the button. (See figure 5.21.)
Example 2. The action chooses between 2 functions, depending on the state of the button. (See figure 5.22.)

5.4.6 Functions as Arguments in Messages for Asynchronous Communication

Asynchronous communication happens when a program requests an action from the system but cannot determine when that action will be completed. For example, it

```
(
var window, button;
window = Window("Server Button", Rect(400, 400, 200, 200));
button = Button(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].value;
};
window.front;
)
```

Figure 5.22
Evaluating functions chosen by index.

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 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 the following.

5.4.6.1 Asynchronous communication with a Server

The system asks for an action to happen on a server, for example, to load a sound file into a buffer (Buffer.read). Since the time it will take 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. Figure 5.23 demonstrates that the action passed as an argument to the read method is executed *after* the statement following Buffer.read.

5.4.6.2 Dialogue windows

Dialogue windows that demand input from the user employ an action argument to determine what to do when input is provided. This prevents the system from waiting indefinitely for the user. (Boot the server first with Server.default.boot.)

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

```
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;
)
```

Figure 5.23
Asynchronous communication with the server.

```
});
format("continuing at: %", Main.elapsedTime).postln;
)
```

5.4.7 Iterating Functions

Iteration is the technique of repeating the same function a number of times. It 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 5.6.3).

5.4.7.1 Iterating a specified 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 an Array:
{10.rand * 3}!5
for: Iterate between a minimum and a maximum integer value:
30.for(35, {|i| i.postln});
forBy: Iterate between 2 values, using a definable step:
2.0.forBy(10, 1.5, {|i| i.postln})
```

5.4.7.2 Iterating while a condition is true

The message while will repeatedly evaluate a function as long as a test function returns true: `{[true, false].choose}.while({"was true".postln;})`. It is usually coded like this:

```
Server.default.boot;  // do this first
(                       // then the rest of the program
var window, routine;
window = Window("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;
)
```

Figure 5.24
Illustrates 'loop' and the use of Event—(key: value).play—to play notes.

```
(
var sum = 0;
while {sum = sum + exprand(0.1, 3); sum <10} {sum.postln}
)
```

5.4.7.3 Infinite (indefinite) loop

loop repeats a function until the process that contains the loop statement is stopped. It can be used only within a process that stops or pauses between statements; otherwise, it will hang the system with an infinite loop. (See figure 5.24.)

5.4.8 Partial Application: Shortcut Syntax for Small Functions

It is possible to construct functions that apply arguments to a single message call by using the underscore character _ as a placeholder for an argument. For example, instead of writing {arg x; x.isPrime}, one can write _.isPrime. If more than 1 _ is included, then each _ takes the place of a subsequent argument in the function. Examples are shown in figure 5.25.

5.4.9 Recursion

Recursion is a special form of iteration in which a function calls itself from 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 is not declared anywhere in the SuperCollider class library. See section 5.3.6.2.) The value of thisFunction is always the function inside which thisFunction is accessed. Figures

```
_.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));
```

Figure 5.25
Partial application.

```
(
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.(10).postln;    // 10 factorial: 3628800
)
```

Figure 5.26
Iterative factorial.

5.26 and 5.27 show the difference in implementing the algorithm for computing the factorial of a number iteratively and using recursion. The recursive algorithm is shorter.

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. An example is shown in figure 5.28.

5.4.10 Inspecting the Structure of a Function

A particular feature of functions is the ability to access a function's parts, which define its structure. An example is the following:

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

```
// Define the factorial function and store it in variable f:
f = { | x | if ( x > 1) { x * thisFunction.value(x - 1) } { x } };
 f.value(10);          // 10 factorial: 3628800
```

Figure 5.27
Recursive factorial.

```
(
/* 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
)
```

Figure 5.28
Recursion over a tree of unknown structure.

The source code of a function is stored only if that function is *closed*, that is, if it does not access the variables of an enclosing function. A function's def variable contains a FunctionDef object that also contains the names of the arguments and variables of the function and their default values. These are used by the SynthDef class, for example, to compile a function into a UGen graph and then into a SynthDef that can be used to create synths on the Server.

5.4.11 Scope of Variables in Functions

As mentioned in section 5.3, variables are accessible only within the context (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. Thus, in figure 5.15, the variable freq is defined in the (implicit) top-level function, and the

function stored in change_freq is a *child_func* that has access to this variable. The function change_freq can therefore both *read* (access) the value of the variable freq and set (*write*) it whenever it is called. 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]

It is in fact possible to view a closure as a simple and limited form of an object, where the variables defined in the top-level function of the closure serve as instance variables. This technique has been used in *Pyrite,* an "external object" that provides a programming language inside *Max*. Pyrite was created by James McCartney in 1994 and is one of the direct precursors of SuperCollider. The following 2 examples show how to model the behavior of the Counter class shown in section 5.5 without writing a class definition.

5.4.11.1 Modeling instances through functions that create other functions

Figure 5.29 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*. 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.

So from 1 mother function one can create multiple closures, each closure having its own set of variables and functions, and each function in that closure is able to run multiple times. In this way, one can construct programs that make smaller 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 a max_count argument value of 5. Consequently, the first time it creates a function that counts to 10 and the second time, one that counts to 5.

The effect of this technique is similar to defining instance variables, and the child functions that have access to these variables are similar to instances of a class that has access to these variables. Figure 5.30 shows how closures with their own variables are generated from a function.

5.4.11.2 Functions in Events as methods

This section extends the example of section 5.4.11.1 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. An Event is an object that can hold values associated to named keys. Instead of having a fixed, predefined number of instance variables, an Event can hold any number of key-value associations that 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 = Window("Counters", Rect(400, 400, 200, 80));
// make a button for triggering the counting:
button1 = Button(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 = Button(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
)
```

Figure 5.29
A function that creates functions that count.

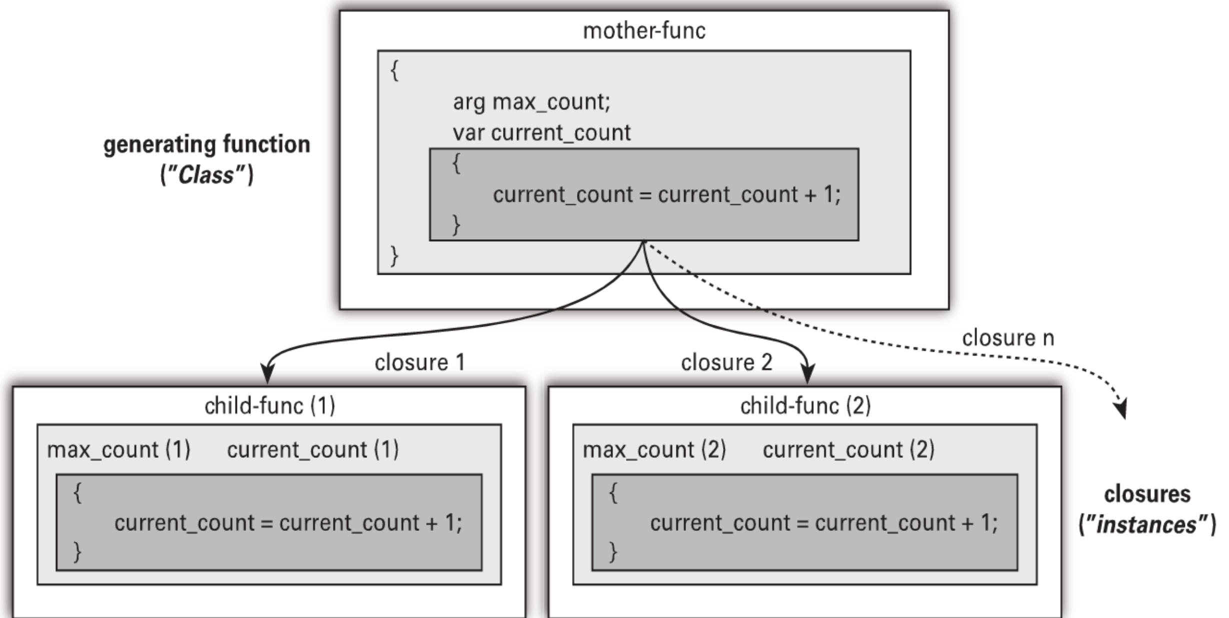

Figure 5.30
Functions created by functions as models of instances.

similarly to the named instance variables of an Object. In the current example the Event contains 3 keys—'count1,' 'reset_count,' and 'max_count'—whose values are bound to functions that operate on the variables of the counter_maker closure. These functions assigned as values to keys act the way an instance method would in a normal class definition. Thus, an Event made by counter_maker is the model of an object with 2 variables and 3 methods. The code in the example of figure 5.31 is hardly any bigger than the previous version, despite the addition of 2 features.

The above example can be seen as a rudimentary class definition constructed without employing the regular class definition system of SuperCollider. 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 to model a class Counter with 2 class methods.

The syntax for running the functions stored in an Event is the same as that of a method call, (receiver.message), the only difference being that the first argument passed to a function in an Event is the Event itself. There is a catch, however. If one stores a function in an Event 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;
```

For more information on this approach, see chapter 8.

```
(
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 = Window("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 |
        Button(window, Rect(0, 0, 190, 20))
            .states_([["Counting to: " ++ counter.max_count.asString]])
            .action = { counter.count1 };
        Button(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
)
```

Figure 5.31
Functions stored in events as instance methods.

5.5 Program Flow Control and Design Patterns

Control structures are structures that permit you to choose the evaluation of a function depending on a condition. That is, a function is evaluated only if the value of a test condition is true. There are variants involving 1 or more functions. (For alternative syntax forms, see the Help file Syntax-Shortcuts.)

5.5.1 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}
```

5.5.2 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.

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

5.5.3 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—whereas the case statement uses a series of independent functions as tests.

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

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

5.5.4 Other Control Techniques: Behavior Patterns

Selecting among alternatives for 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 et al. 1994; Beck, 1996). Typically, techniques of this category would fall under the group *Behavior Patterns*. Examples of such patterns are *Chain of Responsibility*, *Command*, *Iterator*, *Mediator*, *Observer,* and *State*. Beck (1996) classifies Behavior Patterns into 2 major categories. Under "Method" he lists patterns that are based on the organization of an algorithm inside methods. Under "Message" he classifies patterns that use message passing to create algorithms. These patterns can be very small but equally powerful. An example is the *Choosing Message* pattern (Beck, 1996, pp. 45–47). Instead of choosing among 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, consider an object that represents an entry in a list of publications, and that responds to the message responsible by returning some object that represents the name of the person who is responsible for the object. 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 to redefine it. (See section 5.2.1.2 for syntax of methods and class extensions.)

The power of this technique is, first, that the number of choices can easily be extended 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 1 place. In other words, complexity is reduced—or, rather, broken down into pieces in an elegant way—by delegating responsibility for different parts of the algorithm to different classes. Thus, algorithms are organized by the combination of a number of method calls, which split the algorithm into pieces and delegate the responsibility for different parts of the algorithm to different classes. As a result, methods tend to contain very little code, often just a single line. Although 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 programming.

5.6 Collections

Collections are objects that hold a variable number of other objects. For example, figure 5.32 shows a program that adds a new number to a sequence each time the user clicks on a button, and then plays the sequence as a "melody."

The above example builds a sequence of notes by adding a new random integer between 0 and 15 each time. Note that adding an element to nil creates an array with the added element (i.e., nil add: 1 results in [1]).

```
Server.default.boot;    // boot the server first;
(
var degrees, window, button;
window = Window("melodies?", Rect(400, 400, 200, 200));
button = Button(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;
)
```

Figure 5.32
Building an Array with add.

The subclass tree of Collection is extensive (Collection.dumpClassSubtree) and is summarized in the Help file Collections. Collections can be classified into 3 types according to the way in which their elements are accessed:

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 include Array, List, Interval, Range, Array2D, Signal, Wavetable, and String. 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. Other messages for access exist—wrapAt, clipAt, foldAt—that modify invalid index numbers so they always return some element. There are collections that hold only a specific kind of object, such as Char (String), Symbol (SymbolArray), Float (Signal, Wavetable).
Collections whose elements are accessed by using a symbol, or another object, as index. For example (a: 1, b: 2)[\a] returns 1. Such collections are Dictionary, IdentityDictionary, MultiLevelIdentityDictionary, Library (a global, nested MultiLevelIdentityDictionary that can be accessed by series of objects as indices), Environment, and Event. All such collections are made up of Association objects, which 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 lookup by key.
Collections whose elements are accessed by searching for a match to a condition. For example, Set[1, 2, 3, 4, 5] select: (_ > 2). These are Set and Bag.

5.6.1 Creating Collections

The generic rule for creating a collection is to enclose its elements in brackets [], separating each element by a 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 values 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 section 5.6.8).
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 such as sine or Chebyshev 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.

5.6.2 Binary Operators on Collections

Most binary operators on collections can work both between 2 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 2 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
```

5.6.3 Iterating over Collections

The following messages iterate over each element of a collection with a function:

do(function): Evaluate function over each element, return the receiver. (1..5) do: _.postln
collect(function): Evaluate function over each element, return the collected results of each evaluation. (1..5) collect: _.sqrt.
pairsDo(function): Iterate over adjacent pairs of elements of a collection.
inject(function): Iterate passing the result of each iteration to the next one as an argument:
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}
```

5.6.4 Searching in Collections

The 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 matching of regular expressions on a string.

5.6.5 Restructuring Collections

A full account of the structure-manipulation features of the SuperCollider language would require a chapter of its own. For full details the reader is referred to the Help files of the various Collection classes (and those of their superclasses, such as Collection and SequenceableCollection). Here are a few examples of some of the more commonly used methods:

reverse: Reverse the order of the elements. (1..5).reverse.
flop: Turn rows into columns in a 2-dimensional collection. [[1, 2], [\a, \b]].flop.
scramble: Rearrange the elements in random order. (1..5).scramble.
clump(n): Create subcollections 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 discussed in the Help files J Concepts in SC and List Comprehensions.

5.6.6 IdentityDictionary

IdentityDictionary is a dictionary that retrieves its values by looking for a key identical to a given index. “Identical” means that the key should be the same object as the index. For example, the 2 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 with the same characters are always stored as 1 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.

However:

```
a = IdentityDictionary[\foo->1];
a[\foo]; // Returns 1
```

Searching for a matching object by identity is much faster than searching by 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. Though this is a fast process, it is still considerably more expensive in computing cycles than accessing a "real" variable!

IdentityDictionary defines 2 instance variables: proto and parent. These are used by the classes Environment and Event to provide a default environment when needed (see section 5.6.8). The parent scheme makes it possible to build hierarchies of parent events in a way similar to that for class hierarchies.

5.6.7 Environment

An Environment is an IdentityDictionary that can evaluate functions which contain environment variables (see section 5.5). To make an Environment from a function, use the message make:

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

This is not just a convenient notation; it also allows one 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}
```

The message use evaluates a function within an Environment.

```
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)})
```

Note that the function must be explicitly evaluated with valueEnvir for this to work. Therefore, the following is not the right way to supply arguments with use:

```
(a: 1, b: 2).use({|a, b| a + b})
```

valueEnvir in normal code text outside of use draws on the currentEnvironment:

```
~a = 3; ~b = 5;
{|a, b| a + b}.valueEnvir
```

Patterns are a specific extension library within SuperCollider that is very useful for musical scheduling, and they rest on the Environment mechanisms to manipulate musical data. (They actually use Event, a subclass of Environment, which we will introduce next.) Patterns exploit the ability to supply values for arguments from an Environment with valueEnvir when playing instruments that are defined as functions.

5.6.8 Event

Event is a subclass of Environment with several additional features. An Event itself is playable:

```
(degree: 2, dur: 3).play
```

Event stores several prototype Events in its class variables that embody default musical event types (a class variable is globally available to the instances of its 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 a parent Event as its Environment, and overrides only those items of the parent 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:

```
Server.default.boot;   // boot the server first. Run each following line
separately:
(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,...)
```

5.7 Working with 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 or in the platform-specific extension folder, in files with the extension .sc. 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.

5.7.1 Encapsulation, Inheritance, Polymorphism

The 3 defining principles of object-oriented languages are Encapsulation, Inheritance, and Polymorphism. Encapsulation means that data inside an object are accessible only to methods that belong to that object. This protects the data of the object from external changes, thereby aiding in creating consistent and safe programs. In Polymorphism, the same message can correspond to different behaviors according to the class of the object that receives it. In section 5.5.4, an entry of class Film responds differently to the message responsible than an entry of class EditedBook does. 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 (see also section 5.7.4). Together, these 3 features are responsible for the capabilities of object-oriented languages.

5.7.2 Compiling the SuperCollider Class Library

In contrast to code executed from a window holding SuperCollider code, which can be run at any time, changes made in class definition code take effect only after recompiling the SuperCollider Class Library. (See the Shortcuts Help file for platform-specific info on how to do this.) Compiling the library rebuilds all classes and resets the entire memory of the system.

5.7.3 Defining a Class

The structure of a class is defined by its variables and its methods. Additionally, a class may define class variables, constants, and class methods.

As noted above, class variables are accessible by the class itself as well as by all instances, whereas instance variables are accessible only inside methods of the instance in question. Constants are like class variables, except that 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, and instance methods to instances of that class. For example, in Window.new("test," Rect(500, 500, 100, 100)).front the class method new is addressed to the class Window and returns an appropriate window for the platform on which SuperCollider is running, and the method front is addressed to the instance created by method New.

A class may inherit variables and methods from another class, which is called its *superclass*. Inheritance works upward over a chain of 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 5.33).

As seen in figure 5.33, the code that defines a class has 2 major characteristics in common 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 2 sections: variable declarations and 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 is summarized below.

The name of the class is indicated at the start of the definition. If the class has a superclass other than Object, it is indicated like this:

```
Integer: SimpleNumber {// define Integer as subclass of SimpleNumber
```

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 Server has a class variable set that stores all servers known to the system. One can quit these servers with Server.set do: _.quit. Class Char has several const statements declaring special characters.

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;}
}
```

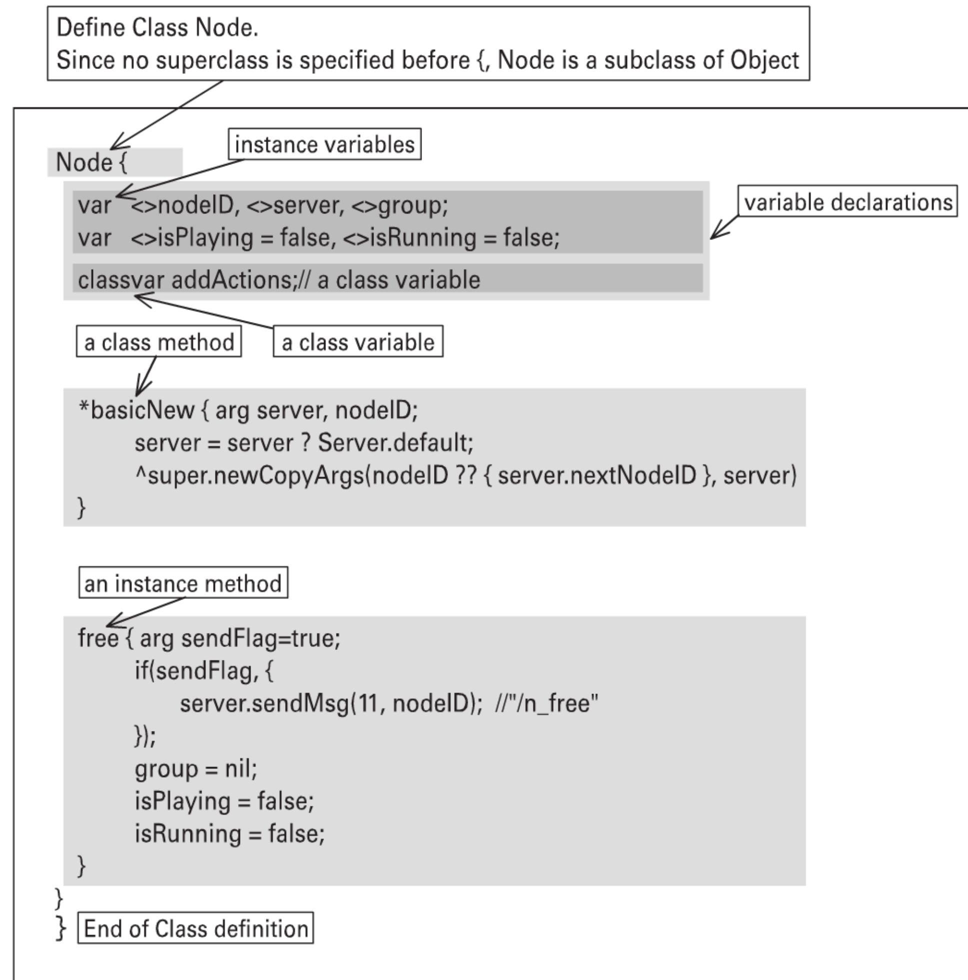

Figure 5.33
Summary of Class Definition Parts (Excerpt from Definition of Class Node).

The declaration of any variables of a class is followed by 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.

The sign * before a method's name creates a class method.

```
*new {arg x=0, y=0; ^super.newCopyArgs(x, y);} // (from class Point)
```

The default return value of an instance 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 will be returned. The method freq {^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. 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;
}
```

Identifiers starting with underscore (_) inside methods call *primitives*, that is, computations that are done by compiled code in the system, and whose code can be seen only in the underlying 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
}
```

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.

If the class method *initClass is defined, then it will be run right after the system is compiled. It is used to initialize any data needed. To indicate that a class needs to be initialized *before* the present initClass is run, one includes the following code in the definition of initClass:

```
Class.initClassTree(NameOfClassToBeInitialized).
```

A class is usually defined in 1 file. If the same class name is found in definitions in 2 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
}
```

5.7.4 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

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

Figure 5.34
Classes of classes.

one class, and by defining subclasses to differentiate the properties and behaviors of objects that have a more specialized character. For example, the class Integer inherits the properties of the class SimpleNumber. SimpleNumber is called the *superclass* of Integer, and Integer is called a *subclass* of SimpleNumber. Float, the class describing floating-point numbers such as 0.1, is also a subclass of SimpleNumber. Classes are thus organized into a family tree. The following expression prints out the complete SuperCollider class tree: Object.dumpClassSubtree.

5.7.5 Metaclasses

Since all entities in SuperCollider are objects, classes are themselves objects. Each class is the sole instance of its "*metaclass*." For example, the class of Integer is Meta_Integer, and consequently Integer is the only instance of the class Meta_Integer. All metaclasses are instances of Class. The following examples trace the successive classes of objects starting from the Integer 1 and going up to Class as the class of all Metaclasses.

The cycle Class-Meta_Class-Class in figure 5.34 shows the end of the Class relationship tree. Since the class of Class is Meta_Class and Meta_Class is also a Class, those 2 classes are the only objects that are instances of one another:

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

Class methods are equivalent to instance methods of the class's Metaclass. For instance, the class method *new of Server is an instance method of Meta_Server. (See figure 5.35.)

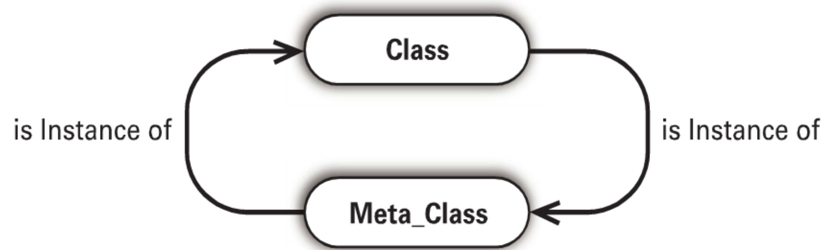

Figure 5.35
`Class` and `Meta_Class` are mutually instances of each other.

5.7.6 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, and so on. Any subclass can override this default behavior in its own code, as well as extend it by defining new variables and methods. The tree formed by Object and its subclasses thus describes all classes in the SuperCollider system.

5.7.7 Notifying Objects of Changes: Observer and Adapter/Controller Patterns

This section shows how to convert the class model from earlier in the chapter into a *real* class. The Observer design pattern implemented in class Object allows one to attach objects (called dependants) to any object in such a way that they are updated when that object notifies itself with the message changed and an optional list of arguments. This results in the message update (along with the arguments) being sent to each of the dependants. It is the responsibility of the dependants to know how to respond correctly to an update message, and the changing object can remain ignorant of the kind and number of its dependants. Thus it is possible to attach a sound, a GUI element, or any other object or process to another object and make it respond to changes of that object in any manner. Crucially, this can happen without having to modify the class definition of the changing object to deal with the specifics of the objects being notified. This technique is similar to the design pattern known as Model-View-Controller (MVC) (see chapters 9 and 10), and serves as its basis. The goal of this pattern is to separate data or processes (the model) from their display (views) and from the control mechanisms, so as to permit multiple displays across different media and platforms.

```
Counter {
    // variables: maximum count, current count
    var <>max_count, <>current_count = 1;
    // 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
    reset {
        current_count = 1;
        this.changed(\reset);
    }
}
```

Figure 5.36
Counter Class.

The present example adds auditory displays and a GUI display 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 shown in figure 5.36.

This must be placed in a file Counter.sc in the SCClassLibrary folder and compiled with [command-K]. After that, boot the server and add the SynthDefs for the sounds (see figure 5.37).

Next, create 5 counters and store them in ~counters:

```
~counters = (6, 11 .. 26) collect: Counter.new(_);
```

Now create a sound adapter to follow changes in any counter it is added to (see figure 5.38).

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

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

SynthDef("wham", {
    Out.ar(0, BrownNoise.ar(
        EnvGen.kr(Env.perc(level: 0.1), doneAction: 2)
    ))
}).add;
)
```

Figure 5.37
SynthDefs for the Counter model example.

```
(
~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]
            )
        }
    )
};
)
```

Figure 5.38
A dependant that plays sounds.

sense it is similar to an Adapter pattern. (An Adapter pattern translates between incompatible interfaces.)

This works because class Function defines the method update as a synonym for value, thus conveniently allowing it to work as a dependant in a straightforward manner:

```
update {|obj, what ... args| this.value(obj, what, *args)}
```

Attach the sound adapter to all 5 counters:

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

Then start a routine that increments the counters at 0.25-second intervals:

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

The routine can be stopped with ~count.stop. But before doing that, let's add GUI displays for the counters. (See figure 5.39.)

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

```
~make_display.(~counters[0]);
```

```
(
~make_display = { | counter |
    var window, label, adapter, stagger;
    window = Window(
        "counting to " ++ counter.max_count.asString,
        Rect(stagger = UniqueID.next % 20 * 20 + 400, stagger, 200, 50)
    );
    label = StaticText(window, window.view.bounds.insetBy(10, 10));
    adapter = { | counter, what, 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
};
)
```

Figure 5.39
A dependant that displays the count.

Or all of them at once:

```
~counters do: ~make_display.(_);
```

The Observer pattern is considered so important that it is implemented in class Object and is thus available to all objects.

5.8 Conclusion

The present chapter has attempted to describe the programming language of SuperCollider and its capabilities in as much detail as possible in the given space. It also has presented some techniques of programming that may serve as an introduction to advanced programming. Many other techniques exist. A great many of these are described in print and on the Web in publications that deal with design patterns for programming. Kent Beck's *Smalltalk Best Practice Patterns* (Beck, 1996) is recommended as a basic manual of good style and because the patterns it describes are as powerful as they are small. Gamma et al. (1994) is considered a standard book on patterns. Beck (2000) and Fowler et al. (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 on the Web site of this book as supplementary material to the present chapter) makes extensive use of the Observer pattern and defines a class Script that enables one to code algorithms for performance in prototypes that 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. Contributions by musicians and programmers, through suggestions and bug reports to the sc-dev 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. Even though SuperCollider has already gained considerable 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 programming and sound. The features and capabilities of the SuperCollider programming 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 how the trend for coding for performance or composition as a musically creative activity matures in practice. Yet whatever the future may bring, the particular marriage of toolmaking and music making that SuperCollider embodies so successfully will mark it as an exceptional

achievement, and hopefully will give birth to further original ideas and amazing sounds.

References

Beck, K. 1996. *Smalltalk Best Practice Patterns*. Upper Saddle River, NJ: Prentice Hall.

Beck, K. 2000. *eXtreme Programming eXplained: Embrace Change*. Reading, MA: Addison-Wesley.

Burstall, R. 2000. "Christopher Strachey—Understanding Programming Languages." *Higher-Order and Symbolic Computation,* 13(1/2): 52.

Fowler, M., K. Beck, J. Brant, W. Opdyke, and D. Roberts. 1999. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley Object Technology series. Reading, MA: Addison-Wesley.

Gamma, E., R. Helm, R. Johnson, and J. Vlissides. 1994. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional Computing series. Boston: Addison-Wesley.

Strachey, C. 2000. "Fundamental Concepts in Programming Languages." *Higher-Order and Symbolic Computation,* 13(1/2): 11–49.

Notes

1. Two relevant definitions of statements are "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 1 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 from a statement group in C or C++.

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

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."