CHAPTER 1

CORE PROGRAMMING CONCEPTS

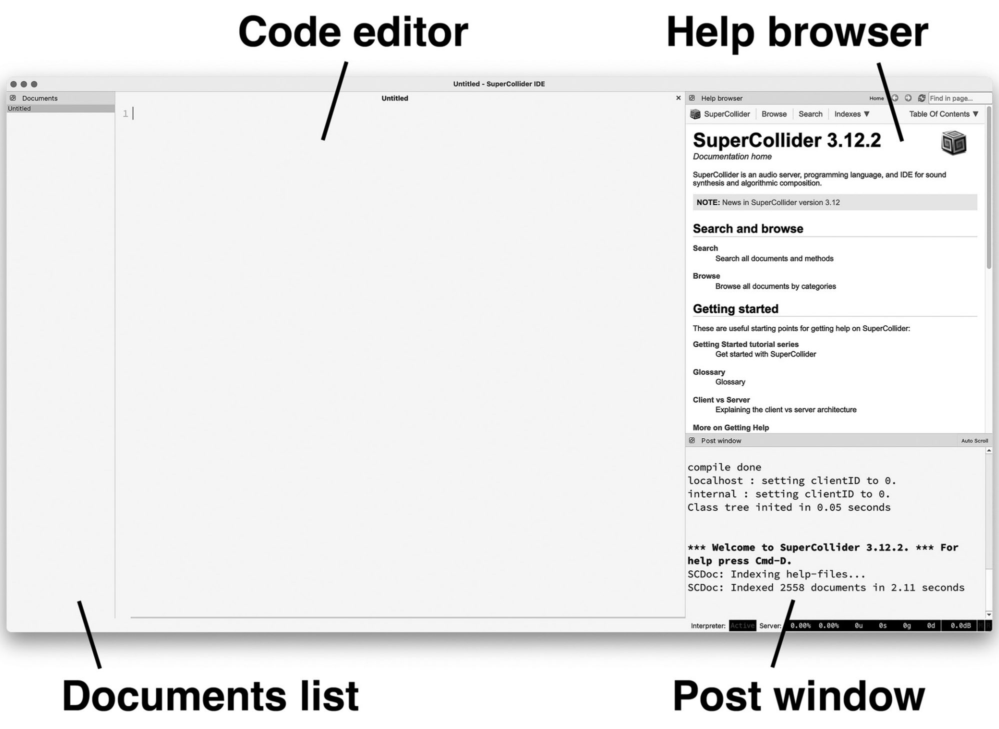
1.1 Overview

This chapter covers the essentials of programming in SuperCollider, such as navigating the environment, getting acquainted with basic terminology, understanding common data types, making simple operations, and writing/running simple programs. The goal is to become familiar with things you'll encounter on an everyday basis and develop a foundational understanding that will support you through the rest of this book and as you embark on your own SC journey.

Keyboard shortcuts play an important role in SC, as they provide quick access to common actions. Throughout this book, the signifier [cmd] refers to the command key on macOS systems, and the control key on Windows and Linux systems.

1.2 A Tour of the Environment

Before we start dealing with code, a good first step is to understand what we see when opening SC for the first time, as depicted in Figure 1.1.



SuperCollider 3.12.2

Documentation home

SuperCollider is an audio server, programming language, and IDE for sound synthesis and algorithmic composition.

NOTE: News in SuperCollider version 3.12

Search and browse

Search
Search all documents and methods

Browse
Browse all documents by categories

Getting started

These are useful starting points for getting help on SuperCollider:

Getting Started tutorial series
Get started with SuperCollider

Glossary
Glossary

Client vs Server
Explaining the client vs server architecture

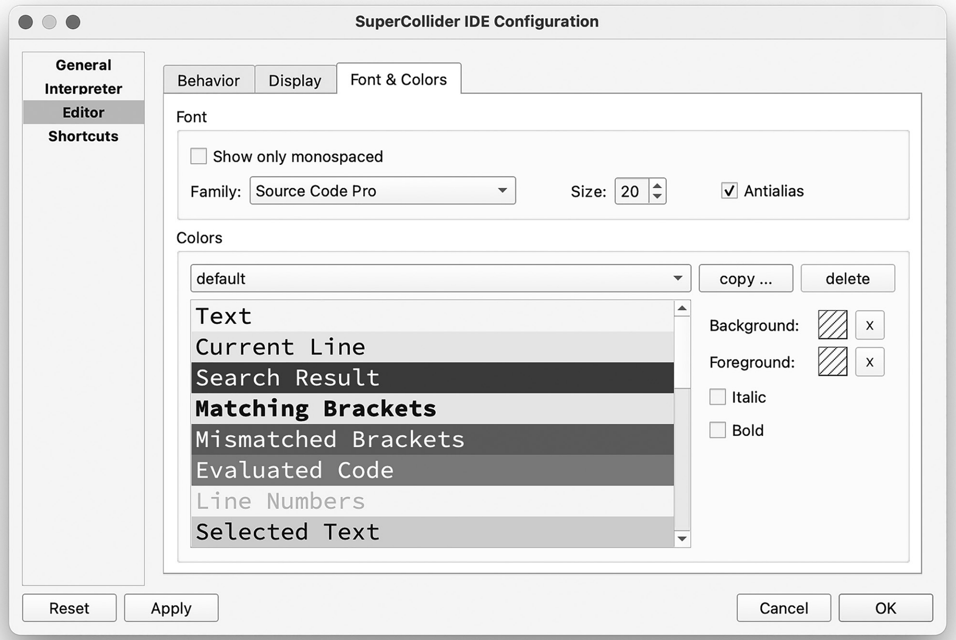
More on Getting Help

```
compile done
localhost : setting clientID to 0.
internal : setting clientID to 0.
Class tree inited in 0.05 seconds


*** Welcome to SuperCollider 3.12.2. *** For
help press Cmd-D.
SCDoc: Indexing help-files...
SCDoc: Indexed 2558 documents in 2.11 seconds
```

FIGURE 1.1 The SuperCollider environment.

The centerpiece of the SC environment is the code editor, a large blank area that serves as your workspace for creating, editing, and executing code. Like a web browser, multiple files can be open simultaneously, selectable using the tab system that appears above the editor. Code files can be saved and loaded as they would be in any text editing or word processing software. SC code files use the file extension "scd" (short for SuperCollider document). The code editor is surrounded by three docklets: the post window, the help browser, and the documents list. The post window is where SC displays information for the user. Most often, this is the result of evaluated code, but may also include error messages and warnings. The help browser provides access to documentation files that detail how SC works. The documents list displays names of files that are currently open, offering a handy navigation option if many documents are open at once. Altogether, these and other features constitute the SC Integrated Development Environment, commonly known as the IDE. The IDE serves as a front-end interface for the user, meant to enhance workflow by providing things like keyboard shortcuts, text colorization, auto-completion, and more. The preferences panel, which is shown in Figure 1.2, is where some of these customizations can be made. The preferences panel is accessible in the "SuperCollider" drop-down menu on macOS, and the "Edit" drop-down menu on Windows/Linux.



Show only monospaced

Family: Source Code Pro

Size: 20

✓ Antialias

default

Text
Current Line
Search Result
Matching Brackets
Mismatched Brackets
Evaluated Code
Line Numbers
Selected Text

FIGURE 1.2 The SuperCollider preferences panel.

The IDE is one of three components that make up the SC environment. In addition, there is the SC language (often called "the language" or "the client") and the SC audio server (often called "the server"). The technical names for these components are *sclang* and *scsynth*, respectively, and despite being integral parts of the environment, they remain largely invisible to the user.

The language is home to a library of objects, which are the basic building blocks of any program. It also houses the interpreter, a core component that interprets and executes code, translating it into action. The interpreter automatically starts when the IDE is launched, and a small status box in the lower-right corner of the IDE displays whether the interpreter is active. SC is an interpreted language, rather than a compiled language. In simple terms, this means that code can be executed selectively and interactively, on a line-by-line basis, rather than having to write the entirety of a program before running it. This is especially useful because SC is regularly used for real-time creative audio applications. In some musical scenarios (improvisation, for example), we won't know all the details in advance, so it's convenient to be able to interact with code "in the moment." This dynamism is featured in examples throughout this book (many of which are broken into separate chunks of code) and is on full display in the last chapter of this book, which discusses live coding.

The audio server is the program that handles calculation, generation, and processing of audio signals. Colloquially, it's the "engine" that powers SC's sound capabilities. It cannot be overstated that the server is fully detached and fundamentally independent from the language and IDE. The language and the server communicate over a network using a client-server model. For example, to play some audio, we first evaluate some code in the language. The language, acting as a client on the network, sends a request to the server, which responds by producing sound. In a sense, this setup is like checking email on a personal computer. An email client gives the impression that your messages are on your own machine, but in reality, you're a client user on a network, sending requests to download information stored on a remote server.

Often, the language and server are running on the same computer, and this "network" is more of an abstract concept than a true network of separate, interconnected devices. However, the client-server design facilitates options involving separate, networked machines. It's possible for an instance of the language on one computer to communicate with an audio server running on a different computer. It's equally possible for multiple clients to communicate with a single server (a common configuration for laptop ensembles), or for one language-side user to communicate with multiple servers. Messages passed between the language and server rely on the Open Sound Control (OSC) protocol, therefore the server can be controlled by any OSC-compliant software.[1] Most modern programming languages and audiovisual environments are OSC-compliant, such as Java, Python, Max/MSP, and Processing, to name a few. Even some digital audio workstations can send and receive OSC. However, using an alternative client doesn't typically provide substantial benefits over the SC language, due to its high-level objects that encapsulate and automatically generate OSC messages. In addition, the client-server divide improves stability; if the client application crashes, the server can play on without interruption while the client is rebooted.

Unlike the interpreter, the server does not automatically start when the IDE is launched and must be manually booted before we can start working with sound, which is the focus of the next chapter. For now, our focus is on the language and the IDE.

1.3 An Object-Oriented View of the World

The SC language is an object-oriented language. Though it's helpful to understand some core principles of object-oriented programming (OOP), a deep dive is not necessary from a musical perspective. This section conveys some essentials of OOP, relying primarily on analogies from the real world, avoiding a more theoretical approach.

1.3.1 OBJECTS AND CLASSES

Our world is full of *objects*, such as plants, animals, buildings, and vehicles. These objects exist within a hierarchy: some objects, like houses, supermarkets, and stadiums, are all members of the same category, or *class* of objects, called buildings. Applying some programming terminology to this example, buildings would be considered the parent class (or superclass) of houses, supermarkets, stadiums, and many others. Similarly, houses, supermarkets, and stadiums are each a child class (or subclass) of buildings. This hierarchy might extend further in either direction: we might have several subclasses of houses (cottages, mansions, etc.), while buildings, vehicles, plants, and animals might be subclasses of an even more all-encompassing superclass named "things." This organizational structure is tree-like, with a few large, broad branches near one end, and many smaller and more specific branches toward the other end.

Like the real world, the SC language is full of objects that exist within a similar class hierarchy, called the *class tree*. Classes of objects in SC include things like Integers, Floats, Strings, Arrays, Functions, Buttons, Sliders, and much more. When represented with code, a class name always begins with a capital letter, which is how SC distinguishes them. If some of these terms are foreign to you, don't worry. We'll introduce them properly later on.

1.3.2 METHODS AND INHERITANCE

Certain types of interactions make sense for certain types of objects. For example, it's reasonable to ask whether an animal is a mammal, but nonsense when applied to a vehicle. On the other hand, some types of queries make sense for all objects, like asking whether something is alive. The point is that a specific type of object "knows" how to respond to certain types of inquiries, to which it responds in a meaningful way. Such queries are called *methods* or *messages*.[2] For example, an integer knows how to respond to being "squared," (multiplied by itself). To a graphical slider, this method is meaningless. Further still, there are some methods that have meaning for both integers and sliders, such as one that asks, "are you an integer?"

If a method is defined for a class, all its subclasses automatically inherit that method and respond to it in a similar manner. It's also possible for the same method to produce different results when applied to different objects (this ability is called *polymorphism*). Consider the verb "to file." Semantically, what this means depends on the object being filed. To file a piece of paper means to organize it in a filing cabinet. To file a piece of metal, on the other hand, means to smooth it by grinding it down. Many examples of polymorphism exist within SC. The "play" method, for instance, is defined for many types of objects, but produces various results that may or may not involve sound production.

1.3.3 CLASSES VS. INSTANCES

There is an important distinction between a tangible object, and the abstract idea of that object. To illustrate, consider a house, compared to a blueprint for a house. The blueprint represents a house and provides a set of instructions for creating one. But it's not the same thing as the house itself. You can't live in a blueprint!

In this example, the blueprint functions similarly to the *class* of houses, while a physical house would be considered an *instance* of that class. You could, for example, use one blueprint to construct many houses. It might even be possible to construct several different houses using the same blueprint, by making a few tweaks to the instructions before each build. Generally, a class can conjure into existence a tangible version of what it represents, but it's usually these instances that we are interested in working with.

1.4 Writing, Understanding, and Evaluating Code

As we begin translating these abstract concepts into concrete code, bear in mind that writing computer code requires precision. Even the most seemingly minor details, like capitalization and spelling, are important. The interpreter will never try to "guess" what you mean if your code is vague or nonsensical. On the contrary, it will attempt to do exactly what you instruct and report errors if it can't comply.

Type the following into the code editor, exactly as it appears:

```
4.squared;
```

Then, making sure your mouse cursor is placed somewhere on this line, press [shift]+[return], which tells the interpreter to execute the current line. You'll see a brief flash of color over the code, and the number 16 will appear in the post window. If this is your first time using SC, congratulations! You've just run your first program. Let's take a step back and dissect what just happened.

1.4.1 A SINGLE LINE OF CODE, DECONSTRUCTED

The expression **`4.squared`** contains a member of the integer class and the "squared" method, which is defined for integers. We say that four is the *receiver* of the method. The period, in this context, is the symbol that applies the method to the receiver. This "receiver-dot-method" construction is commonly used to perform operations with objects, but there is a syntax alternative that involves placing the receiver in an enclosure of parentheses, immediately following the method:

```
squared(4);
```

Why choose one style over the other? This is ultimately dictated by personal preference and usually governed by readability. For example, an English speaker would probably choose **`4.squared`** instead of **`squared(4)`**, because it mimics how the phrase would be spoken.

The parentheses used to contain the receiver are one type of *enclosure* in SC. Others include [square brackets], {curly braces}, "double quotes," and 'single quotes.' Each type of enclosure has its own significance, and some have multiple uses. We'll address enclosures in more detail as they arise, but for now it's important to recognize than an enclosure always involves a pair of symbols: one to begin the enclosure, and another to end it. Sometimes, an enclosure contains only a few characters, while others might contain thousands of lines. If an opening bracket is missing its partner, your code won't run properly, and you'll likely see an error message.

The semicolon is the statement terminator. This symbol tells the interpreter where one expression ends and, possibly, where another begins. Semicolons are the code equivalent of periods and question marks in a novel, which help our brain parse the writing into discrete sentences. In SC, every code statement should always end with a semicolon. Omitting a semicolon is fine when evaluating only a single expression, but omitting a semicolon in a multi-line situation will usually produce an error.

In the case of either **`4.squared`** or **`squared(4)`**, the result is the same. On evaluation, the interpreter *returns a value* of 16. Returning a value is not the same thing as printing it in the post window. In fact, the interpreter always posts the value of the last evaluated line, which makes this behavior more of a byproduct of code evaluation than anything else. Returning a value is a deeper, more fundamental concept, which means the interpreter has taken your code and digested it, and is passing the result back to you. We can think of the expression **`4.squared`** or **`squared(4)`** as being an equivalent representation of the number 16. As such, we can treat the entire expression as a new receiver and chain additional methods to apply additional operations. For instance, Code Example 1.1 shows how we can take the reciprocal (i.e., one divided by the receiver) of the square of a number, demonstrated using both syntax styles:

CODE EXAMPLE 1.1: COMBINING MULTIPLE METHODS INTO A SINGLE STATEMENT, USING TWO DIFFERENT SYNTAX STYLES.

```
4.squared.reciprocal;

reciprocal(squared(4));
```

Methods chained using the "receiver-dot-method" style are applied from left to right. When using the "method(receiver)" syntax, the flow progresses outward from the innermost set of parentheses.

Unlike **`squared`** or **`reciprocal`**, some methods require additional information to return a value. For instance, the **`pow`** method raises its receiver to the power of some exponent, which the user must provide. Code Example 1.2 shows an expression that returns three to the power of four, using both syntax styles. In this context, we say that four is an *argument* of the pow method. Arguments are a far-reaching concept that we'll explore later in this chapter. For now, think of an argument as an input value needed for some process to work, such as a method call. Attempts to use **`pow`** without providing the exponent will produce an error. Note that when multiple items appear in an argument enclosure, they must be separated with commas.

CODE EXAMPLE 1.2: A CODE EXPRESSION INVOLVING A RECEIVER, METHOD, AND ARGUMENT, USING TWO DIFFERENT SYNTAX STYLES.

```
3.pow(4);

pow(3, 4);
```

1.4.2 MULTIPLE LINES OF CODE, DECONSTRUCTED

Suppose we continued chaining additional methods to one of the expressions in Code Example 1.1. After several dozen methods, the line would become quite long. It might run over onto a new line and would also become increasingly difficult to read and understand. For this reason, we frequently break up a series of operations into multiple statements, written on separate lines. To do this correctly, we need some way to "capture" a value, so that it can be referenced in subsequent steps.

A *variable* is a named container than can hold any type of object. One way to create a variable is to declare it using a **`var`** statement. A variable name must begin with a lowercase letter. Following this letter, the name can include uppercase/lowercase letters, numbers, and/or underscores. Table 1.1 contains examples of valid and invalid variable names.

TABLE 1.1 Examples of valid and invalid variable names.

Valid	Invalid
`num`	`Num`
`myValue`	`my#$%&Value`
`sample_04`	`sample-04`

Variable names should be short, but meaningful, striking a balance between brevity and readability. Once a variable is declared, an object can be assigned to a variable using the equals symbol. A typical sequence involves declaring a variable, assigning a value, and then repeatedly modifying the value and assigning each new result to the same variable, overwriting the older assignment in the process. This initially looks strange from a "mathematical" perspective, but the equals symbol doesn't mean numerical equality in this case; instead, it denotes a storage action. In Code Example 1.3, the expression **`num = num.squared`** could be translated into English as the following command: "Square the value currently stored in the variable named 'num,' and store the result in the same variable, overwriting the old value."

To evaluate a multi-line block, we need to learn a new technique and a new keyboard shortcut. The code in Code Example 1.3 has been wrapped in an enclosure of parentheses, each on its own line above and below. Here, we're already seeing that a parenthetical enclosure serves multiple purposes. In this case, it delineates a multi-line block: a modular unit that can be passed to the interpreter with a single keystroke. Instead of pressing [shift]+[return], place the cursor anywhere inside the enclosure and press [cmd]+[return].

CODE EXAMPLE 1.3: A MULTI-LINE BLOCK OF CODE.

```
(
var num;
num = 4;
num = num.squared;
num = num.reciprocal;
)
```

TIP.RAND(); EVALUATING CODE

Using the correct technique and keystroke for evaluating code can be a common point of confusion for new users. Even if your code is written correctly, misevaluation will produce errors that suggest otherwise, because it causes the interpreter to "see" your code incorrectly. Remember that there are two keyboard shortcuts for evaluating code: [shift]+[return] and [cmd]+[return]. Their behavior depends on the following conditions:

- If any code is highlighted:
  - either [cmd]+[return] or [shift]+[return] will evaluate the highlighted code.
- If no code is selected, and the cursor is inside a multi-line block enclosure:
  - [cmd]+[return] will evaluate the entire block.
  - [shift]+[return] will evaluate the line on which the cursor is placed.
- If no code is selected, and the cursor is not inside a multi-line block enclosure:
  - either [cmd]+[return] or [shift]+[return] will evaluate the line on which the cursor is placed.

Also note that in this context, the presence of return characters (rather than semicolons) determines what constitutes a "line" of code. For example, despite containing multiple statements, the following example is considered a single line, so either [cmd]+[return] or [shift]+[return] will evaluate it:

```
var num; num = 4; num = num.squared; num = num.reciprocal;
```

Conversely, the following example is considered to occupy multiple lines, despite containing only one statement. Thus, [shift]+[return] will not work unless the entire example is highlighted, and [cmd]+[return] will not work unless contained in a parenthetical enclosure.

```
4
.squared;
```

Variables declared using a **`var`** statement are *local variables*; they are local to the evaluated code of which they are a part. This means that once evaluation is complete, variables created in this way will no longer exist. If you later attempt to evaluate **`num`** by itself, you'll find that it not only has lost its value assignment, but also produces an error indicating that the variable is undefined. Local variables are transient. They are useful for context-specific cases where there's no need to retain the variable beyond its initial scope.

If we need several local variables, they can be combined into a single declaration. They can also be given value assignments during declaration. Code Example 1.4 declares three variables and provides initial assignments to the first two. We square the first variable, take the reciprocal of the second, and return the sum, which is 49.2.

CODE EXAMPLE 1.4: A MULTI-LINE CODE BLOCK THAT DECLARES MULTIPLE LOCAL VARIABLES AND PROVIDES DEFAULT ASSIGNMENTS.

```
(
var thingA = 7, thingB = 5, result;
thingA = thingA.squared;
thingB = thingB.reciprocal;
result = thingA + thingB;
)
```

TIP.RAND(); NEGATIVE NUMBERS AS DEFAULT VARIABLE ASSIGNMENTS

If a variable is given a negative default value during declaration, there's a potential pitfall. The negative value must either be in parentheses or have a space between it and the preceding equals symbol. If the equals symbol and the minus symbol are directly adjacent, SC will mistakenly interpret the pair of symbols as an undefined operation. This same rule applies to a declaration of arguments, introduced in the next section. Of the following three expressions, the first two are valid, but the third will fail.

```
var num = -2;

var num =(-2);

var num =-2;
```

What if we want to retain a variable, to be used again in the future as part of a separate code evaluation? We can use an *environment variable*, created by preceding the variable name with a tilde character (~). Alternatively, we can use one of the twenty-six lowercase alphabetic characters, which are reserved as *interpreter variables*. Both environment and interpreter variables can be used without a local declaration, and both behave with global scope, that is, they will retain their value (even across multiple code documents) as long as the interpreter remains active. Interpreter variables are of limited use, since there are only twenty-six of them and they cannot have longer, more meaningful names. But they are convenient for bypassing declaration when sketching out an idea or quickly testing something. Environment variables are generally more useful because we can customize their names. The two blocks of code in Code Example 1.5 are each equivalent to the code in Code Example 1.3, but in either case, the variable will retain its assignment after evaluation is complete.

CODE EXAMPLE 1.5: USAGE OF INTERPRETER AND ENVIRONMENT VARIABLES (RESPECTIVELY).

```
(
n = 4;
n = n.squared;
n = n.reciprocal;
)

(
~num = 4;
~num = ~num.squared;
~num = ~num.reciprocal;
)
```

1.4.3 POSTING A VALUE

The **`postln`** method has the effect of printing its receiver to the post window, followed by a new line. This method simply returns its receiver, so it is "neutral" in the sense that it does not modify the object to which it applies. This method is useful for visualizing values partway through a multi-line block, perhaps to check for correctness. On evaluating the code in Code Example 1.6, we will see the value 16 appear in the post window (the result of **`postln`**), followed by 0.0625 (the result of the interpreter posting the result of the last evaluated line).

CODE EXAMPLE 1.6: USE OF THE `postln` METHOD.

```
(
~num = 4;
~num = ~num.squared.postln;
~num = ~num.reciprocal;
)
```

1.4.4 COMMENTS

A comment is text that the interpreter ignores. Comments are typically included to provide some information for a human reader. The imagined reader might be someone else, like a friend or collaborator, but it also might be you! Leaving notes for yourself is a good way to jog your memory if you take a long break from a project and don't quite remember where you left off.

There are two ways to designate text as a comment. For short comments, preceding text with two forward slashes will "comment out" the text until the next return character. To create larger comments that include multiple lines of text, we enclose the text in another type of enclosure, beginning with a forward slash–asterisk, and ending with an asterisk–forward slash. These styles are depicted in Code Example 1.7.

CODE EXAMPLE 1.7: USAGE OF TWO DIFFERENT COMMENT STYLES.

```
(
/*
this is a multi-line comment, which might
be used at the top of your code in order
to explain some features of your program.
*/
var num = 4; // a single-line comment: declare a variable
num = num.squared;
num = num.reciprocal;
)
```

1.4.5 WHITESPACE

Whitespace refers to the use of "empty space" characters, like spaces and new lines. Generally, SC is indifferent to whitespace. For example, both of the following statements are considered syntactically valid and will produce the same result:

```
4.squared+2;

4 . squared + 2;
```

The use of whitespace is ultimately a matter of taste, but a consensus among programmers seems to be that there is a balance to be struck. Using too little whitespace cramps your code and doesn't provide enough room for it to "breathe." Too much whitespace, and code spreads out to the point of being similarly difficult to read. The whitespace choices made throughout this book are based on personal preference and years of studying other programmers' code.

1.5 Getting Help

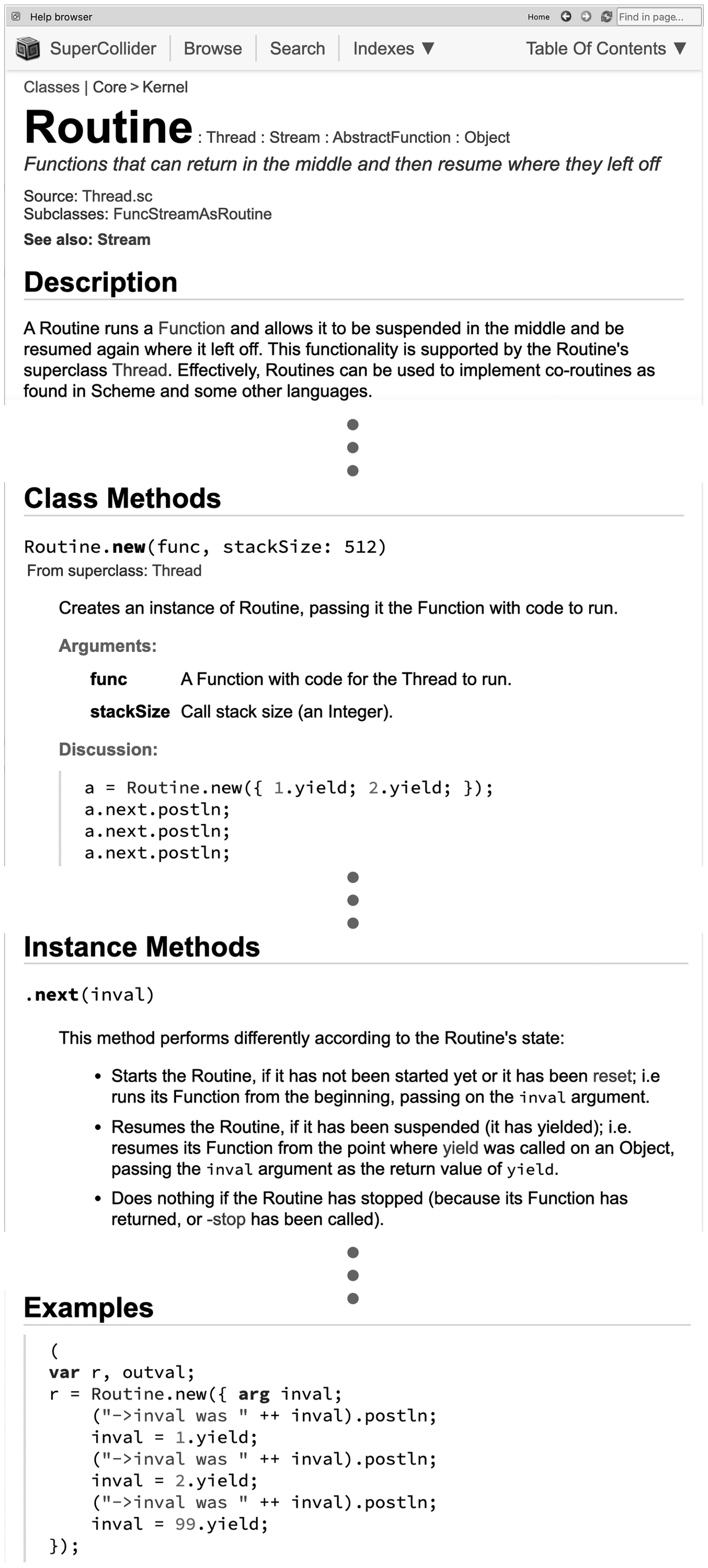
The IDE includes resources for getting help on a number of topics. This section focuses on accessing and navigating these resources.

1.5.1 THE HELP BROWSER

The help browser is the primary resource for getting help within the IDE. It provides access to all of SC's documentation, which includes overviews, reference files, guides, a few tutorials, and individual help files for every object.[3] It includes a search feature, as well as browse option for exploring the documentation by category. It's a good idea to spend some time browsing to get a sense of what's there. Keep in mind that the documentation is not structured as a comprehensive, logically sequenced tutorial (if it were, there'd be less need for books such as this); rather, it tends to be more useful as a quick reference for checking some specific detail.

There are two keystrokes for quickly accessing a page in the help browser: [cmd]+[d] will perform a search for the class or method indicated by the current location of the mouse cursor, and [shift]+[cmd]+[d] invokes a pop-up search field for querying a specific term. If there is an exact class match for your query (capitalization matters), that class's help file will appear. If not, the browser will display a list of results that it thinks are good matches. Try it yourself! Type "Integer" into the editor, and press [cmd]+[d] to bring up the associated help file.

The structure of a class help file appears in Figure 1.3. In general, these files each begin with the class name, along with its superclass lineage, a summary of the class, and a handful of related classes and topics. Following this, the help file includes a detailed description, class methods, instance methods, and often some examples that can be run directly in the browser and/or copied into the text editor.



Classes | Core > Kernel

Routine : Thread : Stream : AbstractFunction : Object

Functions that can return in the middle and then resume where they left off

Source: Thread.sc
Subclasses: FuncStreamAsRoutine

See also: Stream

Description

A Routine runs a Function and allows it to be suspended in the middle and be resumed again where it left off. This functionality is supported by the Routine's superclass Thread. Effectively, Routines can be used to implement co-routines as found in Scheme and some other languages.

●
●
●

Class Methods

```
Routine.new(func, stackSize: 512)
```

From superclass: Thread

Creates an instance of Routine, passing it the Function with code to run.

Arguments:

func	A Function with code for the Thread to run.
stackSize	Call stack size (an Integer).

Discussion:

```
a = Routine.new({ 1.yield; 2.yield; });
a.next.postln;
a.next.postln;
a.next.postln;
```

●
●
●

Instance Methods

```
.next(inval)
```

This method performs differently according to the Routine's state:

- Starts the Routine, if it has not been started yet or it has been reset; i.e runs its Function from the beginning, passing on the `inval` argument.
- Resumes the Routine, if it has been suspended (it has yielded); i.e. resumes its Function from the point where yield was called on an Object, passing the `inval` argument as the return value of `yield`.
- Does nothing if the Routine has stopped (because its Function has returned, or -stop has been called).

●
●
●

Examples

```
(
var r, outval;
r = Routine.new({ arg inval;
    ("->inval was " ++ inval).postln;
    inval = 1.yield;
    ("->inval was " ++ inval).postln;
    inval = 2.yield;
    ("->inval was " ++ inval).postln;
    inval = 99.yield;
});
```

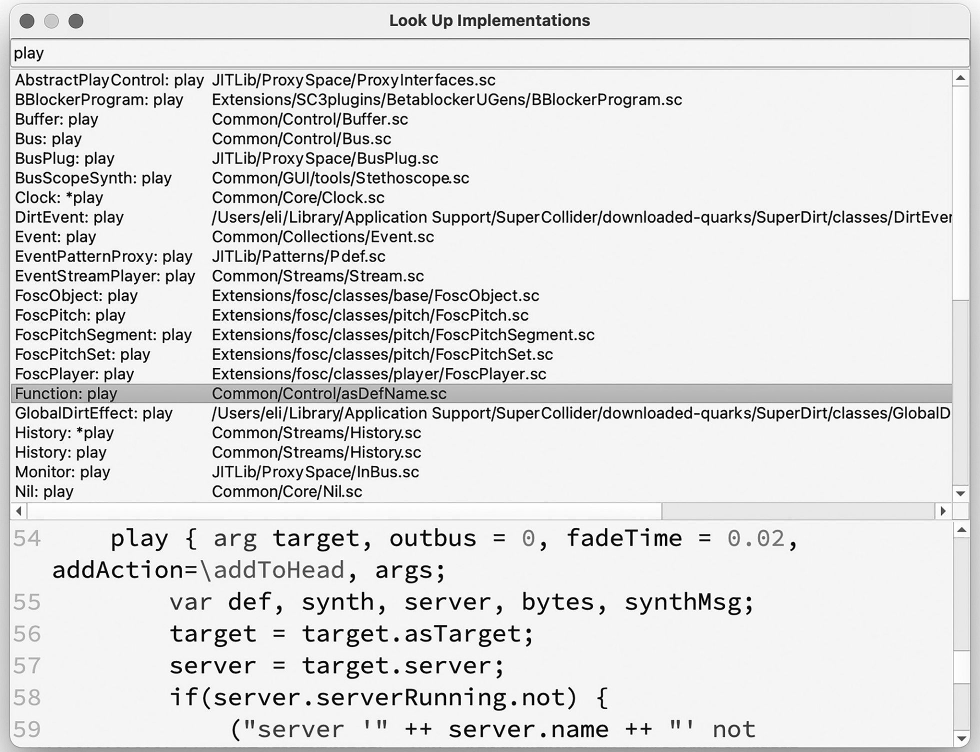
FIGURE 1.3 The structure of a class help file. Vertical ellipses represent material omitted for clarity and conciseness.

1.5.2 SEARCHING FOR IMPLEMENTATIONS

The help browser is the primary resource for new users, but as you get more comfortable with SC, you may want to look directly at the source code for a class or method. Additionally, because SC is an open-source project in active development, there are cases where the content of a help file may not be completely consistent with the behavior of the object it describes. The source code, on the other hand, lets us see exactly how something works.

There are two keystrokes for looking up implementations of class or method: [cmd]+[i] will perform a search for all the source code implementations of the class or method indicated by the current location of the mouse cursor, and [shift]+[cmd]+[i] will bring up a search bar. When you enter your search term, a list of source code files in which that term appears will populate a list (see Figure 1.4). Selecting one will open the source file and automatically scroll to the relevant spot. Just be sure not to edit anything!

To try this yourself, press [shift]+[cmd]+[i], type "play" in the search bar, and press enter to see a list of classes that implement this method. Click one of these items and press enter to open the source code for that class. If the source code looks totally incomprehensible to you, don't worry! Being able to read these files isn't essential, but as you get more comfortable with SC, you may find yourself digging around in these files more often.


Look Up Implementations

play

```
AbstractPlayControl: play  JITLib/ProxySpace/ProxyInterfaces.sc
BBlockerProgram: play      Extensions/SC3plugins/BetablockerUGens/BBlockerProgram.sc
Buffer: play               Common/Control/Buffer.sc
Bus: play                  Common/Control/Bus.sc
BusPlug: play              JITLib/ProxySpace/BusPlug.sc
BusScopeSynth: play        Common/GUI/tools/Stethoscope.sc
Clock: *play               Common/Core/Clock.sc
DirtEvent: play            /Users/eli/Library/Application Support/SuperCollider/downloaded-quarks/SuperDirt/classes/DirtEver
Event: play                Common/Collections/Event.sc
EventPatternProxy: play    JITLib/Patterns/Pdef.sc
EventStreamPlayer: play    Common/Streams/Stream.sc
FoscObject: play           Extensions/fosc/classes/base/FoscObject.sc
FoscPitch: play            Extensions/fosc/classes/pitch/FoscPitch.sc
FoscPitchSegment: play     Extensions/fosc/classes/pitch/FoscPitchSegment.sc
FoscPitchSet: play         Extensions/fosc/classes/pitch/FoscPitchSet.sc
FoscPlayer: play           Extensions/fosc/classes/player/FoscPlayer.sc
Function: play             Common/Control/asDefName.sc
GlobalDirtEffect: play     /Users/eli/Library/Application Support/SuperCollider/downloaded-quarks/SuperDirt/classes/GlobalD
History: *play             Common/Streams/History.sc
History: play              Common/Streams/History.sc
Monitor: play              JITLib/ProxySpace/InBus.sc
Nil: play                  Common/Core/Nil.sc
```

```
    play { arg target, outbus = 0, fadeTime = 0.02,
  addAction=\addToHead, args;
        var def, synth, server, bytes, synthMsg;
        target = target.asTarget;
        server = target.server;
        if(server.serverRunning.not) {
            ("server '" ++ server.name ++ "' not
```

FIGURE 1.4 The search window for looking up implementations of methods.

1.5.3 THE CLASS BROWSER

The class browser is a graphical tool for browsing the class tree. The **`browse`** method can be applied to any class name (e.g. **`Array.browse`**), which invokes the browser and displays information for that class. The class browser, shown in Figure 1.5, displays a class's superclass, subclasses, methods, and arguments. Buttons near the top of the browser can be used for navigating backward and forward (like a web browser), as well as navigating to a class's superclass, or bringing up source code files. You can also navigate to a subclass by double-clicking it in the list in the bottom-left corner.

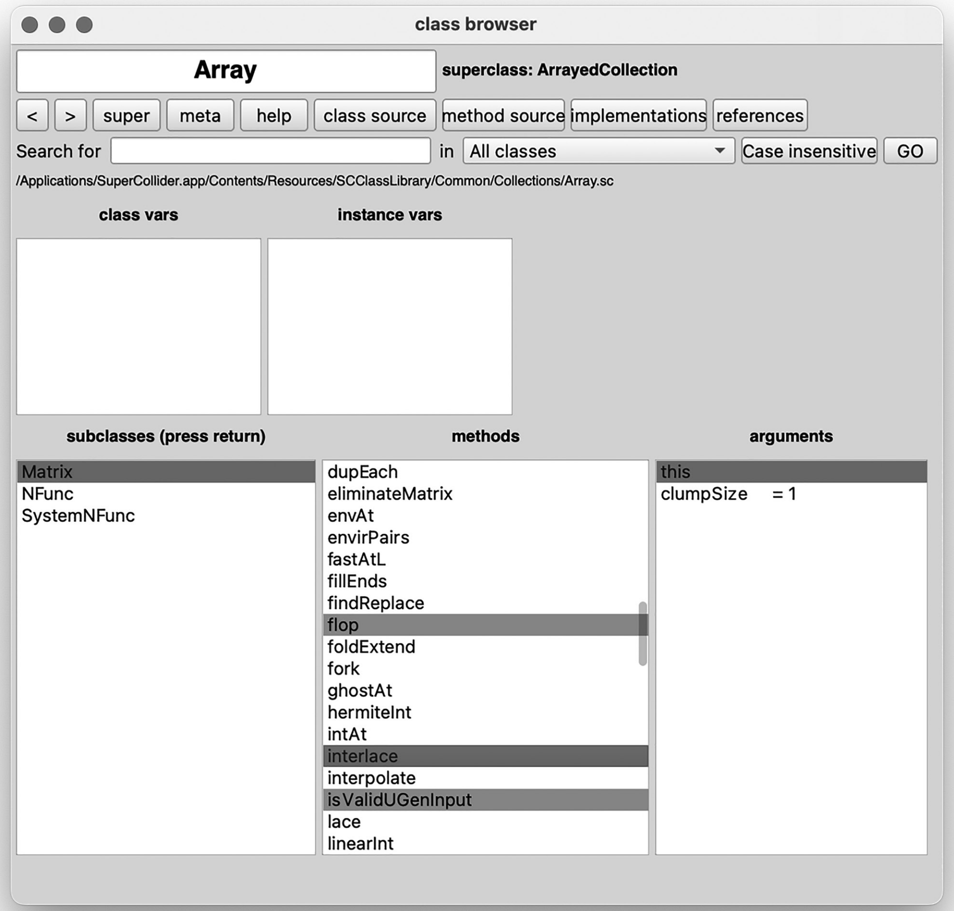


FIGURE 1.5 The class browser.

1.6 A Tour of Classes and Methods

With a basic understanding of writing and evaluating code behind us, we're ready to take a more detailed look at some commonly used classes and methods, to get a better sense of the operations we're likely to regularly encounter.

1.6.1 INTEGERS AND FLOATS

The **`Integer`** and **`Float`** classes represent numerical values. We've already encountered integers, which are whole numbers that can be positive, negative, or zero. A float, on the other hand, is a number that includes a decimal point, like 9.02 and -315.67. Even the number 1.0 is a float, despite being quantitatively equal to one. Some programming languages are quite particular about the distinction between integers and floats, balking at certain operations that attempt to combine them, but SC is flexible, and will typically convert automatically as needed. Integers and floats are closely related on the class tree and therefore share many methods, but there are a few methods which apply to one and not the other.

We'll begin our tour by introducing the **`class`** method, which returns the class of its receiver. This method is not exclusive to integers and floats; in fact, every object knows how to respond to it! However, it's relevant here in showing the classes to which different numbers belong.

```
4.class; // -> Integer

4.0.class; // -> Float
```

There are many methods that perform mathematical operations with numbers. Those that perform an operation involving two values are called *binary operators*, while those that perform an operation on a single receiver, like taking the square root of a number, are called *unary operators*. Most operations exist as method calls (e.g., **`squared`**), but some, like addition and multiplication, are so commonly used that they have a direct symbolic representation (imagine if multiplication required typing **`12.multipliedBy(3)`** instead of **`12 * 3`**). Descriptive lists of commonly used operators appear in Tables 1.2 and 1.3. A more complete list can be found in a SC overview file titled "Operators."

TABLE 1.2 List of common unary operators.

Method Usage	Description
`abs(x);`	Absolute value. Non-negative value of *x*, i.e., distance from zero.
`ceil(x);`	Round up to the nearest whole number.
`floor(x);`	Round down to the nearest whole number.
`neg(x);`	Negation. Positive numbers become negative and vice-versa.
`reciprocal(x);`	Return 1 divided by *x*.
`sqrt(x);`	Return the square root of *x*.
`squared(x);`	Return *x* raised to the power of 2.

TABLE 1.3 List of common binary operators.

Name	Method Usage	Symbolic Usage	Description
Addition	N/A	**`x + y;`**	
Subtraction	N/A	**`x - y;`**	

TABLE 1.3 Continued.

Name	Method Usage	Symbolic Usage	Description
Multiplication	N/A	`x * y;`	
Division	N/A	`x / y;`	
Exponentiation	`x.pow(y);`	`x ** y;`	Return *x* raised to the power of *y*.
Modulo	`x.mod(y);`	`x % y;`	Return the remainder of *x* divided by *y*.
Rounding	`x.round(y);`	N/A	Return *x* rounded to the nearest multiple of *y*.

"Order of operations" refers to rules governing the sequence in which operations should be performed. In grade school, for example, we typically learn that in the expression 4 + 2 * 3, multiplication should happen first, and then addition. SC has its own rules for order of operations, which may surprise you. If an expression includes multiple symbolic operators, SC will apply them from left-to-right, without any precedence given to any operation. If an expression includes a combination of method calls and symbolic operators, SC will apply the method calls first, and then the symbolic operators from left-to-right. Parenthetical enclosures have precedence over method calls and symbolic operators (see Code Example 1.8). In some cases, it can be good practice to use parentheses, even when unnecessary, to provide clarity for readers.

CODE EXAMPLE 1.8. EXAMPLES OF HOW PRECEDENCE IS APPLIED TO COMBINATIONS OF METHODS, SYMBOLIC BINARY OPERATORS, AND PARENTHETICAL ENCLOSURES.

```
// symbolic operators have equal precedence, applied left-to-right:
4 + 2 * 3; // -> 18

// parentheses have precedence over symbolic operators:
4 + (2 * 3); // -> 10

// methods have precedence over symbolic operators:
4 + 2.pow(3); // -> 12

// parentheses have precedence over methods:
(4 + 2).pow(3); // -> 216

// parentheses first, then methods, then binary operators:
1 + (4 + 2).pow(3); // -> 217
```

1.6.2 STRINGS

A string is an ordered sequence of characters, delineated by an enclosure of double quotes. A string can contain any text, including letters, numbers, symbols, and even non-printing characters like tabs and new lines. Strings are commonly used to represent text and may be used to provide names or labels for certain types of objects. Several examples appear in Code Example 1.9.

Certain characters need special treatment when they appear inside a string. For example, what if we want to put a double quotation mark inside of a string? If not done correctly, the internal quotation mark will prematurely terminate the string, likely causing the interpreter to report a syntax error. Special characters are entered by preceding them with a backslash, which in this context is called the *escape character*. Its name refers to the effect it has on the interpreter, causing it to "escape" the character's normal meaning.

CODE EXAMPLE 1.9: EXAMPLES OF STRINGS AND USAGE OF THE ESCAPE CHARACTER.

```
// a typical string
"Click the green button to start.";

// using the escape character to include quotation marks
"The phrase \"practice makes perfect\" is one I try to remember.";

// using the escape character to include new lines
"This string\nwill print on\nmultiple lines.";
```

We know the plus symbol denotes addition for integers and floats, but it also has meaning for strings (the usage of one symbol for multiple purposes is called *overloading* and is another example of polymorphism). A single plus will return a single string from two strings, inserting a space character between them. A double plus does nearly the same thing but omits the space. These and a handful of other string methods appear in Code Example 1.10.

CODE EXAMPLE 1.10: COMMON STRING METHODS AND OPERATIONS.

```
"Hello" + "there!"; // -> "Hello there!"

"Some" ++ "times"; // -> "Sometimes"

"I'm a string.".size; // return the number of characters in the string

"I'm a string.".reverse; // reverse the order of the characters

"I'm a string.".scramble; // randomize the order of the characters

"I'm a string.".drop(2); // remove the first two characters

"I'm a string.".drop(-2); // remove the last two characters
```

1.6.3 SYMBOLS

A symbol is like a string, in that it's composed of a sequence of characters and commonly used to name or label things. Where a string is used, a symbol can often be substituted, and vice-versa. A symbol is written in one of two ways: by preceding the sequence of characters with a backslash (e.g., **`\freq`**), or by enclosing it in single quotes (e.g., **`'freq'`**). These styles are largely interchangeable, but the quote enclosure is the safer option of the two. Symbols that begin with or include certain characters will trigger syntax errors if using the backslash style. Unlike a string, a symbol is an irreducible unit; it is not possible to access or manipulate the individual characters in a symbol, and all symbols return zero in response to the **`size`** method. It is, however, possible to convert back and forth between symbols and strings using the methods **`asSymbol`** and **`asString`** (see Code Example 1.11). Symbols, being slightly more optimized than strings, are the preferable choice when used as names or labels for objects.

CODE EXAMPLE 1.11: CONVERSION FROM STRING TO SYMBOL AND VICE-VERSA.

```
"hello".asSymbol.class; // -> Symbol

\hello.asString.class; // -> String
```

1.6.4 BOOLEANS

There are exactly two instances of the Boolean class: **`true`** and **`false`**. These values are special keywords, meaning we can't use either as a variable name. If you type one of these keywords into the code editor, you'll notice its text will change color to indicate its significance. Booleans play a significant role in conditional logic, explored later in this chapter.

Throughout SC, there are methods and binary operators that represent true-or-false questions, and which return a Boolean value that represents the answer. These include less-than/greater-than operations, and various methods that begin with the word "is" (e.g., **`isInteger`**, **`isEmpty`**), which check whether some condition is met. Some examples appear in Code Example 1.12, and a list of common binary operators that return Booleans appears in Table 1.4.

CODE EXAMPLE 1.12: EXAMPLES OF CODE EXPRESSIONS THAT RETURN BOOLEAN VALUES.

```
1 < 2; // -> true

1 > 2; // -> false

1.isInteger; // -> true

1.0.isInteger; // -> false

"hello".isEmpty; // -> false

"".isEmpty; // -> true
```

TABLE 1.4 Common binary operators that return Boolean values.

Symbolic Usage	Description
`x == y`	Return **`true`** if *x* and *y* are equal, **`false`** otherwise.
`x != y`	Return **`true`** if *x* and *y* are not equal, **`false`** otherwise.
`x > y`	Return **`true`** if *x* is greater than *y*, **`false`** otherwise.
`x < y`	Return **`true`** if *x* is less than *y*, **`false`** otherwise.
`x >= y`	Return **`true`** is *x* is greater than or equal to *y*, **`false`** otherwise.
`x <= y`	Return **`true`** is *x* is less than or equal to *y*, **`false`** otherwise.

1.6.5 NIL

Like true and false, **`nil`** is a reserved keyword in the SC language, and is the singular instance of the **`Nil`** class. Most commonly, it represents the value of a variable that hasn't been given a value assignment, or something that doesn't exist. We rarely use nil explicitly, but it shows up frequently, so it's helpful to be familiar. The **`isNil`** method, demonstrated in Code Example 1.13, can be useful for confirming whether a variable has a value (attempting to call methods on an uninitialized variable is a common source of error messages).

CODE EXAMPLE 1.13: USAGE OF THE `isNil` METHOD.

```
(
var num;
num.isNil.postln; // check the variable – initially, it's nil
num = 2; // make an assignment
num.isNil.postln; // check again – it's no longer nil
)
```

1.6.6 ARRAYS

An array is an ordered collection of objects. Syntactically, objects stored in an array are separated by commas and surrounded by an enclosure of square brackets. Arrays are like strings in that both are ordered lists, but while strings can only contain text characters, an array can contain anything. In fact, arrays can (and often do) contain other arrays. Arrays are among the most frequently used objects, because they allow us to express an arbitrarily large collection as a singular unit. Arrays have lots of musical applications; we might use one to contain pitch information for a musical scale, a sequence of rhythmic values, and so on.

As shown in Code Example 1.14, we can access an item stored in an array by using the **`at`** method and providing the numerical index. Indices begin at zero. As an alternative, we can follow an array with a square bracket enclosure containing the desired index.

CODE EXAMPLE 1.14: ACCESSING ITEMS STORED IN AN ARRAY.

```
x = [4, "freq", \note, 7.5, true];

x.at(3); // -> 7.5 (return the item stored at index 3)

x[3]; // alternate syntax
```

Most unary and binary operators defined for numbers can also be applied to arrays, if they contain numbers. Several examples appear in Code Example 1.15. If we apply a binary operator to a number and an array, the operation is applied to the number and each item in the array, and the new array is returned. A binary operation between two arrays of the same size returns a new array of the same size in which the binary operation has been applied to each pair of items. If the arrays are different sizes, the operation is applied to corresponding pairs of items, but the smaller array will repeat itself as many times as needed to accommodate the larger array (this behavior is called “wrapping”).

CODE EXAMPLE 1.15: BEHAVIOR OF ARRAYS IN RESPONSE TO COMMON UNARY AND BINARY OPERATIONS.

```
[50, 60, 70].squared;         // -> [2500, 3600, 4900]

1 + [50, 60, 70];             // -> [51, 61, 71]

[1, 2, 3] + [50, 60, 70];     // -> [51, 62, 73]

[1, 2] + [50, 60, 70];        // -> [51, 62, 71]
```

The **dup** method, defined for all objects, returns an array of copies of its receiver. An integer, provided as an argument, determines the size of the array. The exclamation mark can also be used as a symbolic shortcut (see Code Example 1.16).

CODE EXAMPLE 1.16: USAGE OF dup AND ITS SYMBOLIC SHORTCUT, THE EXCLAMATION MARK.

```
7.dup; // -> [7, 7] (default size is 2)

7.dup(4); // -> [7, 7, 7, 7]

7 ! 4; // -> [7, 7, 7, 7] (alternate syntax)
```

Arrays are a must-learn feature, rich with many convenient methods and uses—too many to squeeze into this section. Companion Code 1.1 provides an array "cheat sheet," covering many of these uses.

1.6.7 FUNCTIONS

We'll inevitably write some small program that performs a useful task, like creating chords from pitches or converting metric values to durations. Whatever this code does, we certainly don't want to be burdened with copying and pasting it all over our file—or worse, typing it out multiple times—because this consumes time and space. Functions address this problem by encapsulating some code and allowing us to reuse it easily and concisely. Functions are especially valuable in that they provide an option for modularizing code, that is, expressing a large program as smaller, independent code units, rather than having to deal with one big messy file. A modular program is generally easier to read, understand, and debug.

A function is delineated by an enclosure of curly braces. If you type the function that appears in Code Example 1.17 from top to bottom, you'll notice the IDE will automatically indent the enclosed code to improve readability. If you write your code in an unusual order, the auto-indent feature may not work as expected, but you can always highlight your code and press the [tab] key to invoke the auto-indentation feature. Once a function is defined, we can evaluate it with **`value`**, or by following it with a period and a parenthetical enclosure. When evaluated, a function returns the value of the last expression it contains.

CODE EXAMPLE 1.17: DEFINING AND EVALUATING A FUNCTION.

```
(
f = {
   var num = 4;
   num = num.squared;
   num = num.reciprocal;
};
)

f.value; // -> 0.0625

f.(); // alternate syntax for evaluating
```

The function in Code Example 1.17 is not particularly useful, because it produces the same result every time. Usually, we want a function whose output depends on some user-provided input. For instance, suppose we want our function to square and take the reciprocal of an arbitrary value.

We've already seen similar behavior with **`pow`** (which is, itself, a sort of function). We provide the exponent as an argument, which influences the returned value. So, in our function, we can declare an argument of our own using an **`arg`** statement. Like a variable, an argument is a named container that holds a value, but which also serves as an input to a function, allowing a user-specified value to be "passed in" at the moment of execution. If no value is provided at execution time, the default value of the argument (defined in the declaration) will be used. Providing default values for arguments is not required, but often a good idea.

CODE EXAMPLE 1.18: DEFINING AND EVALUATING A FUNCTION WITH AN ARGUMENT.

```
(
f = {
   arg input = 4;
   var num;
   num = input.squared;
   num = num.reciprocal;
};
)

f.(5); // -> 0.04 (evaluate, passing in a different value as the input)

f.(); // -> 0.0625 (evaluate using the default value)
```

Code Example 1.19 shows a syntax alternative that replaces the **arg** keyword with an enclosure of vertical bar characters (sometimes called "pipes") and declares multiple arguments, converting the code from Code Example 1.4 into a function. When executing a function with multiple arguments, the argument values must be separated by commas, and will be interpreted in the same order as they appear in the declaration.

CODE EXAMPLE 1.19: DEFINING AND EVALUATING A FUNCTION WITH MULTIPLE ARGUMENTS, DECLARED USING THE "PIPE" SYNTAX.

```
(
g = { |thingA = 7, thingB = 5|
   var result;
   thingA = thingA.squared;
   thingB = thingB.reciprocal;
   result = thingA + thingB;
};
)

g.(3, 2); // -> 9.5 (thingA = 3, thingB = 2);
```

TIP.RAND(); ARGUMENTS AND VARIABLES

New users may wonder about precise differences between arguments and variables, and what "rules" might apply. In some respects, they're similar: each is a named container that holds a value. Variables are ordinary, named containers that provide the convenience of storing and referencing data. An argument, on the other hand, can only be declared at the very beginning of a function, and serves the specific purpose of allowing some input that can be passed or routed into the function during execution. Variable declarations can only occur at the beginning of a parenthetically enclosed multi-line code block, or at the beginning of a function. If a function declares arguments and variables, the argument declaration must come first. It's not possible to spontaneously declare additional variables or arguments somewhere in the middle of your code.

1.6.8 GETTING AND SETTING ATTRIBUTES

Objects have attributes. As a simplified real-world example, a car has a color, a number of doors, a transmission that may be manual or automatic, etc. In SC, we interact with an object's attributes by applying methods to that object. Retrieving an attribute is called *getting*, and changing an attribute is called *setting*. To get an attribute, we simply call the method that returns the value of that attribute. For setting an attribute, there are two options: we can follow the getter method with an equals symbol to assign a new value to it, or we can follow the getter method with an underscore and the new value enclosed in parentheses. The following pseudo-code demonstrates essential syntax styles for getting and setting, which reappear throughout this book. Note that an advantage of the underscore syntax is that it allows us to chain multiple setter calls into a single expression.

```
x = Car.new; // make a new car

x.color = "red"; // set the color

x.numDoors_(4).transmission_("manual"); // set two more attributes

x.numDoors; // get the number of doors (returns 4)
```

1.6.9 LITERALS

In Section 1.3.3, we compared a house to a blueprint of a house to better understand the conceptual difference between classes and instances. If this example were translated into SC code, it might look something like this:

```
x = House.new(30, 40); // create a house with specific dimensions

x.color_("blue"); // set the color

x.hasGarage_(true); // set whether it has a garage
```

In this imaginary example, we make a new house, paint it blue, and add a garage. This general workflow—creating a new thing, storing it in a variable, and interacting with it through instance methods—is quite common. In fact, most of the objects we'll introduce from this point forward will involve a workflow that follows this basic pattern. We won't necessarily use **`new`** in every case (some classes expect different creation methods), but the general idea is the same.

Interestingly, we haven't been using this workflow for the classes introduced in this section. For example, we don't have to type **`Float.new(5.2)`** or **`Symbol.new(\freq)`**. Instead, we just type the object as is. Classes like these, which have a direct, syntactical representation through code, are called *literals*. When we type the number seven, it is literally the number seven. But an object like a house can't be directly represented with code; there is no "house" symbol in our standard character set. So, we must use the more abstract approach of typing **`x = House.new`**, while its literal representation remains in our imagination.

Integers, floats, strings, symbols, Booleans, and functions are all examples of literals. Arrays, for the record, exist in more of a grey area; they have a direct representation via square brackets, but we may sometimes create one with **`Array.new`** or a related method. There is also a distinction between literal arrays and non-literal arrays, but which is not relevant to our current discussion. The point is that many of the objects we'll encounter in this book are not literals and require creation via some method call to their class.

`TIP.RAND();` OMITTING THE "NEW" METHOD

The **`new`** method is so commonly used for creating new instances of classes that we can usually omit it, and the interpreter will make the right assumption. For example, using our imaginary "house" class, we could write **`x = House(30, 40)`** instead of **`x = House.new(30, 40)`**. However, if creating a new instance without providing any arguments, we can omit **`new`** but cannot omit the parentheses, even if they are empty. For example, **`x = House.new()`** and **`x = House()`** are both valid, but **`x = House`** will be problematic. In this third case, the interpreter will store the house class, instead of a new house instance.

1.7 Randomness

Programming languages are quite good at generating randomness. Well, not exactly—programming languages are good at producing behavior that humans find convincingly random. Most random algorithms are pseudo-random; they begin with a "seed" value, which fuels a deterministic supply of numbers that feel acceptably random. Regardless of how they're generated, random values can be useful in musical applications. We can, for example, spawn random melodies from a scale, shuffle metric values to create rhythmic variations, or randomize the stereophonic position of a sound.

1.7.1 PICKING FROM A RANGE

The **`rrand`** method (short for "ranged random") returns a random value between a minimum and maximum, with a uniform distribution, meaning every value in the range is equally likely to appear. If the minimum and maximum are integers, the result will be an integer. If either boundary is a float, the result will be a float. The **`exprand`** method also returns a value within a range, but approximates an exponential distribution, which means that on average, the output values will tend toward the minimum. This method always returns a float, and the minimum and maximum value must have the same sign (both positive or both negative, and neither can be zero). These methods are demonstrated in Code Example 1.20. This book favors the "method(receiver)" syntax for these methods because it highlights their ranged nature more clearly.

CODE EXAMPLE 1.20: RETURNING A RANDOM NUMBER FROM A RANGE.

```
rrand(1, 9);         // random integer between 1 - 9,
                     // uniform distribution

rrand(40.0, 90.0);   // random float between 40 - 90,
                     // uniform distribution

exprand(1, 100);     // random float between 1 - 100,
                     // exponential distribution
```

Which of these two methods should you use? It depends on what the output will be used for. A dice roll or coin flip, in which all outcomes are equally probable, can be accurately simulated with **`rrand`**. However, our ears perceive certain musical parameters nonlinearly, such as frequency and amplitude. Therefore, using a uniform distribution to generate values for these parameters may produce a result that sounds unbalanced. Consider, for example, generating a random frequency between 20 and 20,000 Hz. If using **`rrand`**, then the output will fall in the upper half of this range about half the time, (between approximately 10,000 and 20,000 Hz). This may seem like a wide range, but from a musical perspective, it's only one octave, and a very high-pitched octave at that! The other half of this range spans approximately 9 octaves, so the sonic result would appear saturated with high frequencies, while truly low frequencies would be rare. By contrast, **`exprand`** would provide a more natural-sounding distribution. So, don't just pick one of these two methods at random (no pun intended)! Instead, think about how the values will be used, and make an informed decision.

1.7.2 RANDOMLY PICKING FROM A COLLECTION

We sometimes want to select a random value from a collection, instead of a range. If the possible outcomes are stored in an array, we can use **`choose`** to return one of them (see Code Example 1.21). This method doesn't remove the item from the collection; it simply reports a selection while leaving the collection unaltered.

CODE EXAMPLE 1.21: SELECTING A RANDOM ITEM FROM A COLLECTION.

```
(
var scale, note;
scale = [0, 2, 4, 5, 7, 9, 11, 12];
note = scale.choose;
)
```

Suppose we have a bag of 1,000 marbles. 750 are red, 220 are green, and 30 are blue. How would we simulate picking a random marble? We could create an array of 1,000 symbols and use **`choose`**, but this is clumsy. Instead, we can use **`wchoose`** to simulate a weighted choice, shown in Code Example 1.22. This method requires an array of weight values, which must sum to one and be the same size as the collection we're choosing from. To avoid doing the math in our heads, we can use **`normalizeSum`**, which scales the weights so that they sum to one, while keeping their relative proportions intact.

CODE EXAMPLE 1.22: WEIGHTED RANDOMNESS.

```
(
var bag, pick;
bag = [\red, \green, \blue];
pick = bag.wchoose([750, 220, 30].normalizeSum);
)
```

1.7.3 GENERATING AN ARRAY OF RANDOM VALUES

As we've seen, **`dup`** returns an array of copies of its receiver and can be used to generate an array of random values. However, there is a catch: if we simply call **`dup`** on a method that generates a random choice, it will duplicate the result of that random choice. The workaround involves enclosing the random generator in curly braces—thus creating a function—and then duplicating that function. This works because the behavior of **`dup`** is slightly different when applied to a function. Instead of merely duplicating the function, **`dup`** will duplicate and evaluate each function copy it creates. Therefore, the random process will be performed once for each item that populates the returned array.

CODE EXAMPLE 1.23: USAGE OF dup WITH METHODS THAT GENERATE RANDOMNESS.

```
rrand(40, 90).dup(8); // 8 copies of 1 random value

{rrand(40, 90)}.dup(8); // 8 uniquely chosen random values
```

1.8 Conditional Logic

Life is full of choices based on certain conditions. If it's a nice day, I'll ride my bike to work. If ice cream is on sale, I'll buy two. In computer programming, conditional logic refers to mechanisms used for similar decision-making, with many possible musical applications. For example, if a section of a piece of music has been playing for at least three minutes, transition to the next section. If the amplitude of a sound is too high, reduce it. This section focuses on expressions of the form “if-then-else,” which tend to be relatively common. Other conditional mechanisms are documented in a reference file called “Control Structures.”

1.8.1 IF

One of the most common conditional methods is **`if`**, which includes three components: (1) an expression that represents the test condition, which must return a Boolean value, (2) a function to be evaluated if the condition is true, and (3) an optional function to be evaluated if false. Code Example 1.24 demonstrates the use of conditional logic to model a coin flip (a value of 1 represents “heads”), in three styles that vary in syntax and whitespace. The second style tends to be preferable to the first, because it places the “if” at the beginning of the expression, mirroring how the sentence it represents would be spoken in English. Because the entire expression is somewhat long, the multi-line approach can improve readability. Note that in the first expression, the parentheses around the test condition are required to give precedence to the binary operator **`==`** over the **`if`** method. Without parentheses, the **`if`** method is applied to the number 1 instead of the full Boolean expression, which produces an error.

CODE EXAMPLE 1.24: USAGE OF `if` TO MODEL A COIN FLIP.

```
// "receiver-dot-method" syntax:
([0, 1].choose == 1).if({\heads.postln}, {\tails.postln});

// "method(receiver)" syntax:
if([0, 1].choose == 1, {\heads.postln}, {\tails.postln});

// structured as a multi-line block:
(
if(
   [0, 1].choose == 1,
   {\heads.postln},
   {\tails.postln}
);
)
```

1.8.2 AND/OR

The methods **`and`** and **`or`** (representable using binary operators **`&&`** and **`||`**), allow us to check multiple conditions. For example, if ice cream is on sale, and they have chocolate, then I’ll buy two. Code Example 1.25 models a two-coin flip in which both must be “heads” for the result to be considered true. Again, parentheses around each conditional test are required to ensure correct order of operations.

CODE EXAMPLE 1.25: USAGE OF THE BINARY OPERATOR && TO MODEL A TWO-COIN FLIP.

```
(
if(
  ([0, 1].choose == 1) && ([0, 1].choose == 1),
  {"both heads".postln},
  {"at least one tails".postln}
);
)
```

1.8.3 CASE AND SWITCH

Say we roll a six-sided die and want to perform one of six unique actions depending on the outcome. A single **if** statement is insufficient because it envisions only two outcomes. If we insisted on using **if**, we'd need to "nest" several **if**'s inside of each other. Even with a small handful of possible outcomes, the code quickly spirals into an unreadable mess. Alternatively, a **case** statement (see Code Example 1.26) accepts an arbitrary number of function pairs. The first function in each pair must contain a Boolean expression, and the second function contains code to be evaluated if its partner function is true. If a test condition is false, the interpreter moves onto the next pair and tries again. As soon as a test returns true, the interpreter executes the partner function and exits the **case** block, abandoning any remaining conditional tests. If all tests are false, the interpreter returns **nil**.

CODE EXAMPLE 1.26: USAGE OF case TO SIMULATE A SIX-SIDED DIE ROLL.

```
(
var roll = rrand(1, 6);
case(
   {roll == 1}, {\red.postln},
   {roll == 2}, {\orange.postln},
   {roll == 3}, {\yellow.postln},
   {roll == 4}, {\green.postln},
   {roll == 5}, {\blue.postln},
   {roll == 6}, {\purple.postln}
);
)
```

A **switch** statement is similar to **case**, but with a slightly different syntax, shown in Code Example 1.27. We begin with some value—not necessarily a Boolean—and provide an arbitrary number of value-function pairs. The interpreter will check for equality between the starting value and each of the paired values. For the first comparison that returns true, the corresponding function is evaluated.

CODE EXAMPLE 1.27: USAGE OF `switch` TO SIMULATE A SIX-SIDED DIE ROLL.

```
(
var roll = rrand(1, 6);
switch(
    roll,
    1, {\red.postln},
    2, {\orange.postln},
    3, {\yellow.postln},
    4, {\green.postln},
    5, {\blue.postln},
    6, {\purple.postln}
);
)
```

1.9 Iteration

One of the most attractive aspects of computer programming is its ability to handle repetitive tasks. Iteration refers to techniques that allow a repetitive task to be expressed and executed concisely. Music is full of repetitive structures and benefits greatly from iteration. More generally, if you ever find yourself typing a nearly identical chunk of code many times over, or relying heavily on copy/paste, this could be a sign that you should be using iteration.

Two general-purpose iteration methods, **`do`** and **`collect`**, often make good choices for iterative tasks. Both are applied to some collection—usually an array—and both accept a function as their sole argument. The function is evaluated once for each item in the collection. A primary difference between these two methods is that **`do`** returns its receiver, while **`collect`** returns a modified collection of the same size, populated using values returned by the function. Thus, **`do`** is a good choice when we don't care about the values returned by the function, and instead simply want to "do" some action a certain number of times. On the other hand, **`collect`** is a good choice when we want to modify or interact with an existing collection and capture the result. At the beginning of an iteration function, we can optionally declare two arguments, which represent each item in the collection and its index as the function is repeatedly executed. By declaring these arguments, we give ourselves access to the collection items within the function.

In Code Example 1.28(a), we iterate over an array of four items, and for each item, we post a string. In this case, the items in the array are irrelevant; the result will be the same as long as the size of the array is four. Performing an action some number of times is so common, that **`do`** is also defined for integers. When **`do`** is applied to some integer **`n`**, the receiver will be interpreted as the array **`[0, 1, ... n-1]`**, thus providing a shorter alternative, depicted in Code Example 1.28(b). In Code Example 1.28(c), we declare two arguments and post them, to visualize the values of these arguments.

CODE EXAMPLE 1.28: USAGE OF do TO PERFORM SIMPLE ITERATIVE TASKS.

```
(a)      [30, 40, 50, 60].do({"this is a test".postln});

(b)      4.do({"this is a test".postln});

(c)      [30, 40, 50, 60].do({|item, index| [item, index].postln});
```

A simple usage of **collect** appears in Code Example 1.29. We iterate over the array, and for each item, return the item multiplied by its index. **collect** returns this new array.

CODE EXAMPLE 1.29: USAGE OF collect TO ITERATE OVER AN ARRAY AND RETURN A NEW ARRAY.

```
x = [30, 40, 50, 60].collect({|item, index| item * index});
// -> the array [0, 40, 100, 180] is now stored in x
```

Numerous other iteration methods exist, several of which are depicted in Code Example 1.30. The **isPrime** method is featured here, which returns true if its receiver is a prime number, and otherwise returns false.

CODE EXAMPLE 1.30: EXAMPLES OF ADDITIONAL ITERATION METHODS.

```
x = [101, 102, 103, 104, 105, 106, 107];

// return the subset of the array for which the function
returns true:
x.select({ |n| n.isPrime }); // -> [101, 103, 107]

// return the first item for which the function returns true:
x.detect({ |n| n.isPrime }); // -> 101

// return true if the function returns true for at least one item:
x.any({ |n| n.isPrime }); // -> true

// return true if the function returns true for every item:
x.every({ |n| n.isPrime }); // -> false

// return the number of items for which the function returns true:
x.count({ |n| n.isPrime }); // -> 3
```

1.10 Summary

The topics presented in this chapter provide foundational support throughout the rest of this book, and hopefully throughout your own explorations with SC. Due to the fundamental nature of these concepts, some of the code examples in this chapter may feel a bit dry, abstract, or difficult to fully digest. If you don't fully understand how and why these techniques might apply in a practical setting, that's reasonable. As you continue reading, these concepts will arise again and again, reflected through applications in synthesis, sampling, sequencing, and composition. For now, to provide some practical context, Companion Code 1.2 explores one possible application that ties several of these concepts together: creating a function that converts a pitch letter name and octave to the corresponding MIDI note number.

Notes

1 Matthew J. Wright and Adrian Freed, "Open SoundControl: A New Protocol for Communicating with Sound Synthesizers," in *Proceedings of the 1997 International Computer Music Conference*, Proceedings of the International Computer Music Association (San Francisco: International Computer Music Association, 1997). More information about OSC can be found at https://opensoundcontrol.stanford.edu/.

2 Technically, "method" and "message" are not synonymous, though they are closely related. A message is a more abstract concept of requesting an operation to be performed on an object. A method is a more concrete concept that encompasses the name and description of how a message should be implemented for an object. In a practical context, it is within reason to use these terms interchangeably.

3 At the time of writing, the documentation files are also available on the web at https://doc.sccode.org/.