ScriptX Language Guide

December 1995

Kaleida Labs

ScriptX Language Summary – Version 1.5

Comments and Line Continuation

```
--      comment
//      C++ style comment
/* inset comment */
\       continue expression on next line
```

Variable Declaration

```
local [ constant ] variable, variable, ...
local [ constant ] variable [ := value ]
global [ constant ] variable, variable, ...
global [ constant ] variable [ := value ]
```

Variable Assignment

```
variableDeclaration := value
ivAccess := value
elementAccess := value
```

Hexadecimal Numbers

```
0xN
```

String Constants

```
"string"
```

String Escapes

```
\<nnnn>hex escape for unicode characters
\r      return
\t      tab
\"      doublequote
\\      backslash
```

Array and KeyedLinkedList Literals

```
#(element, element, element ...)
#(key:value, key:value, ...)
```

Element Access

```
collection[key]
```

Range Literals

```
startval to endval [ by increment ]
startval [ exclusive|inclusive ] to
    endvalue [ exclusive|inclusive ] continuous
```

Name Literals

```
@name
```

Operators and Functional Equivalents

```
+   sum
-   sub
-   negate
*   mul
/   quo
=   equal
==  eq
!== ne
<>  nequal
!=  nequal
<   lt
>   gt
<=  le
>=  ge
```

Logical Expressions

```
expresion and expression
expression or expression
not expression
```

Compound Expressions

```
(expression; expression; ...)
(
    expression
    expression
    ...
)
```

Function Calls

```
function arg arg arg ...
function positionalArg positionalArg ... keyarg:value \
    keyarg:value ...
function(arg, arg, arg, ...)
function(regulararg, regulararg, ..., keyarg:value,
    keyarg:value, ...)
function()
apply functionName [ arguments ]collectionOfArguments
```

Instance and Class Variable Access

```
object.ivname
class.cvname
```

Object Coercion

```
originalobject as newclass
```

Conditionals

```
if test then trueexpression else falseexpression
if test then trueexpression
if test do trueexpression

case [ testexpression ] of
    tag: expression
    ...
    [ otherwise:expression ]
end
```

Repeat Loops

```
repeat while|until test do expression
repeat expression while|until test
```

For Loops

```
for sources [ while|until test ] do expression
for sources [ while|until test ] collectIt
for sources [ while|until test ] selectIt

sources:  number
          index := collection|rangeliteral
          index in collection|rangeliteral

collectIt: collect [into collection] [by method]
          [as collectionClass] expression

selectIt: select item [into collection] [by method]
          [as collectionClass] if test
```

Block and Loop Controls

```
continue -- to continue in e a loop
exit [ with expression ] -- to exit from a loop
return expression -- functions and methods only
```

Threads and Pipes

```
expression &
collection | function
collection | CollectionClassOrInstance
```

Object and Class Definition

```
[local] object [ objectname ] ( classlist )
    [ keywordarg:value, keywordarg:value
        keywordarg:value
    ...]
    [ inst[ance] var[iable]s
        varname [: value ]
        [qualifier] varname
    ...]
    [ inst[ance] methods
        methoddef
    ...]
    [ settings
        ivname:value, ivname:value
        ivname:value
    ...]
    [ contents
        item, item
        item ]
end

[local] class classname (classlist)
    [class var[iable]s
        varname [:value ]
        [ qualifiers ] varname
    ... ]
    [inst[ance] var[iable]s
        varname [: value ]
        [ qualifiers ] varname
    ... ]
    [class methods
        methoddef ]
    [inst[ance] methods
        methoddef ]
end

qualifiers:       [ readonly ] [ transientOption ] \
                    [ reference ]
transientOption:  transient [ initializer function ]
```

Function and Method Definition

```
[local] f[u]n[ction] functionName arguments -> body
( arguments -> body )
[class] method methodName self arguments -> body

arguments:    positionalArguments \
                  [#rest args ] [ #key keys ]
keys:         key:varName ( defaultValue )

nextMethod self arguments
apply nextMethod arguments -- init or other method with
                                  #rest args
```

Free Methods

```
method methodName self { object object } \
        arguments -> body
[ class ] method methodName self { class class } \
        arguments -> body
```

Setter and Getter Methods

```
[class] method setter self arg -> body
[class] method getter self -> body

setter:   varnameSetter | get varname
getter:   varnameGetter | get varname
```

Modules

```
module moduleName
    [ exports variables ]
    [ uses module module module ... ]
    [ uses module with
        [ imports everything ]
        [ imports variable, variable, variable, ... ]
        [ excludes variable, variable, variable, ... ]
        [ exports everything ]
        [ exports variable, variable, variable, ... ]
        [ prefix prefix ]
        [ renames oldName:newName oldName:newName
        ... ]
    end ]
end

inModule module
getModule @module
```

Exceptions

```
guard
    expression
    expression
    ...
[ catching
    catchList ]
[ on exit
    exit code ]
end

catchlist: exception [arg]: action
    exceptionClass [arg]: action

caught error
throw again
```

Other (Available Only in ScriptX Listener)

```
?         -- last value in listener
--<<<    -- execute input file up to >>>
--<<<-   -- execute input file, but don't print it
```

Contents

Figures

Tables

Preface

This document is part of the ScriptX Technical Reference Series. This series is for programmers using ScriptX to develop interactive multimedia tools and titles. This series includes the following documents:

- The *ScriptX Components Guide* provides an overview of ScriptX architecture, conceptual explanations about the organization of the ScriptX classes into components, and script examples showing how the classes work together. It covers ScriptX from the multimedia title, down to the operating system devices. This manual is essential to anyone designing and building multimedia titles in ScriptX. It is the companion volume to the *ScriptX Class Reference*.
- The *ScriptX Class Reference* is a detailed reference to the ScriptX class library that provides, in dictionary form, a complete specification of the classes, methods, variables, and functions available for building multimedia titles and tools in ScriptX. It is the companion volume to the *ScriptX Components Guide*.
- The *ScriptX Language Guide* (this manual) is a practical guide to using the ScriptX programming language. It provides complete functional descriptions of the language as well as concrete descriptions of tasks you might do when actually working with the ScriptX language. Anyone programming in ScriptX will want to use this book.
- The *ScriptX Tools Guide* provides information about the ScriptX development process that is not covered in the other manuals. The first part discusses how to use the browsers, the Listener and other tools that are supplied with ScriptX. All users will want to read this part. The second part explains how to extend ScriptX by loading classes written in C, and discusses platform-specific issues. Developers who wish to add classes written in C to ScriptX will want to read the second part. The third part of the *ScriptX Tools Guide* discusses how to build additional tools in ScriptX. Tool developers will want to read the third part.
- The *ScriptX Quick Reference* summarizes information about the ScriptX Language and Class Library. It includes the grammar of the language, listings of components and their classes, and an alphabetical reference to classes, including class variables, instance variables, and methods.

Who Should Read This Book

Anyone interested in learning about how to write ScriptX scripts should read this book. This book describes the ScriptX authoring language and provides many annotated examples that demonstrate how to perform basic tasks in ScriptX.

Summary of Contents

This book is a reference manual for the overall structure of the ScriptX scripting language, including syntax summaries and examples of its use. It contains nine chapters and two appendixes.

Summary of Chapters

Chapter 1, “Introducing the ScriptX Language,” is an overall introduction to the ScriptX language and to its use of object-oriented programming concepts.

Chapter 2, “ScriptX Building Blocks,” describes the basic parts of ScriptX that are used to build larger expressions: numbers, strings, arrays, and arithmetic.

Chapter 3, “Working with Objects,” includes information about creating and using ScriptX classes and objects, both the system-defined classes and any others that may be written in scripts.

Chapter 4, “Conditionals and Loops,” describes the constructs in the ScriptX language for compound expressions and control flow.

Chapter 5, “Functions, Threads and Pipes,” shows how to define functions in ScriptX and describes the thread and pipe operators.

Chapter 6, “Defining Classes and Objects,” describes how to define and specialize new and existing classes and objects in the ScriptX language.

Chapter 7, “Collections,” describes the classes that can contain other objects, and the searching and selecting protocol for those objects.

Chapter 8, “Exceptions,” outlines the ScriptX exception system and the constructs in the language for catching and handling exceptions.

Chapter 9, “Modules,” describes the module system in ScriptX, which allows name space packaging and management.

Appendixes

Appendix A, “ScriptX Reference,” contains an EBNF grammar for the ScriptX Language. It includes detailed discussion of tokens, operators, reserved words, and expression syntax.

Appendix B, “Unicode Escape Characters,” contains a table of Unicode values for special characters that are common to the Macintosh, OS/2, and Windows.

Manual Conventions

This manual is set primarily in Palatino and ITC Avant Garde, except that code examples and code words, including names of classes, functions, generic functions, and variables, are set in `Courier`.

If the return value of an expression is relevant to the discussion, that result is shown after an arrow on the line following the code (if it’s not relevant, nothing is shown):

```
2 + 2
⇨  4
```

Ellipses (. . .) replace parts of example scripts that are not relevant to the discussion:

```
class example (MyClass)
    . . .
end
```

tax:description

Definitions of syntax from the ScriptX language are shown with an initial graphic (▼) to allow you to find them easily in the text.

▼ `if` *expression* `do` *expression*

Syntax for function calls is shown with a different initial graphic (◆). Global functions are functions that are not associated with any particular class or obejct. Generic functions are used to call methods on individual objects. Generic and global functions are not part of the ScriptX language itself; they are defined by the core classes, and by the scripted classes and objects you create in your own programs.

◆ `append` *collection value*

In syntax definitions, the `Courier` typeface indicates reserved words. Variable parts of the expression are in italic and are explained in the body of text following the definition.

Parts of syntax definitions that are optional are shown with brackets surrounding the optional part. These brackets are not part of the construct itself. They should be omitted in scripts:

▼ `exit` [`with` *expression*]

Parts of scripts that are intended to be repeated multiple times contain an ellipsis (. . .) at the end of the series:

▼ *expression expression expression* . . .

C H A P T E R

Introducing the ScriptX Language

1

This volume provides a definitive description of the ScriptX authoring language, part of the ScriptX Language and Class Library. It contains the syntax for ScriptX and provides information on using the language effectively. The book is designed to offer several different paths that readers can follow, depending on their familiarity and experience with other programming languages, with object-oriented programming, and with formal grammars.

Appendix A, "ScriptX Reference," is intended as an alternative entry-point and reference for advanced users. Appendix A presents an annotated EBNF grammar of the ScriptX language.

This chapter, "Introducing the ScriptX Language," provides an overview of the ScriptX language. It includes a section that introduces concepts in object-oriented programming, beginning on page 12. Readers who are familiar with other object-oriented programming languages may want to skip this section, proceeding directly to the following chapter.

Whatever path you take through this book and the other documentation you receive with your ScriptX Language and Class Library, you will want to reinforce your reading with hands-on experience. The ScriptX Language and Class Library includes an interactive listener window that can be used for trying out the examples in this book. The Listener provides you with instant feedback, evaluating expressions as you enter them. If you prefer not to type, you will find complete listings of all scripts, organized by chapter, on your CD–ROM. For directions on using the ScriptX Listener, see the *ScriptX Developer's Guide*.

The description of the ScriptX language and the code examples used in this book are independent of any existing authoring tool or environment. Although you should be able to enter code examples in this book in any ScriptX environment that allows it, the examples in this book are intended to be more illustrative than tutorial.

ScriptX—A Technical Overview

The ScriptX language is part of an overall framework for multimedia title development. ScriptX is a cross-platform, dynamic, object-oriented programming language with influences that include CLOS (Common Lisp Object System), Smalltalk, and Dylan, as well as scripting languages such as Lingo and HyperTalk. The language was designed to be easy to use in an interactive session.

It is expected that most ScriptX titles will eventually be generated by authoring tools or development environments specialized for creating multimedia titles. Those tools will effectively hide much of the actual code of the ScriptX title.

However, many of those authoring environments may require some additional scripting with ScriptX. For example, an environment may require the author to fill in parts of a title with specific scripts (for example, to describe the action of a button), or to provide other capabilities that the tool itself does not provide. In addition, some environments may provide an interactive environment, such as the ScriptX Listener, which can be used to enter ScriptX expressions and see an immediate result.

The ScriptX Language

The following section describes the main features of the ScriptX language:

- Programs written in the ScriptX language are compiled into a fast, portable form called *bytecode*. The bytecode compiler is part of the ScriptX Language and Class Library. Complete ScriptX titles consist of scripts in this bytecode form, plus media content. Titles are compiled into platform-independent bytecode which runs on the Kaleida Media Player for each platform. The bytecode compiler allows ScriptX programs to have both the flexibility of an interpreted language and the speed of a compiled language. Compilation is immediate and interactive. ScriptX is characterized as a scripting language in part because classes, objects, functions, and methods can be created and compiled incrementally. New methods can be added to an existing class, or an instance of a class, without recompilation.
- ScriptX is a pure object system. ScriptX is uniformly object oriented. In ScriptX, in contrast with C++, every value is an object reference. Many object-oriented languages allow a mixture of object and non-object types. ScriptX does not. In ScriptX, every unit of information is an object. Every construct in the language is an expression, and every expression returns an object. Every operand in every operation is an object. Some core classes do store information internally that is not in the form of an object. For example, individual characters in a string are not objects. However, in the ScriptX language, such data can only be evaluated and manipulated as objects.
- The ScriptX language is rooted in the Kaleida object system and the core classes. This differs from C++, in which, at least in theory, a developer might begin a new application by developing the equivalent of `RootObject` for that application. (In ScriptX, `RootObject` is the root system class from which all other classes in the application inherit.)
- ScriptX implements *latent typing* of variables, also known as dynamic typing. This contrasts with *static typing* in both C and C++. Static typing means that type is associated with the location the value is assigned to, not with the value itself. By contrast, latent typing means that type is associated with and intrinsic to the object itself. Static typing requires an explicit declaration of type for any location that stores a value. With static typing, all types that can be used in a program must be declared at compile time. With latent typing in ScriptX, a location does not actually store a value, but rather a reference to an object, generally stored elsewhere. One could say that all ScriptX variables are of a single type—*object reference*. In practice, this means that type checking is associated not with the compiler but with the object itself. Type checking, which is optional, can be performed at runtime.

- ScriptX allows any variable to store a reference to any object. In untyped languages, expressiveness is limited by the ability of the language to parse and interpret data from the input stream. Lingo, ToolScript, and Hypertalk—untyped languages that are popular with existing multimedia authoring tools—accept only data that can be represented as a string. In these languages, data is stored and manipulated in other forms internally, but all data must be represented implicitly as a string in a program. ScriptX has a single data type, an object reference, which has the flexibility to accommodate every kind of data and program construct, including classes, objects, functions, methods, modules, directories, and files.
- ScriptX allow objects to be assigned to variables at runtime, a feature known as *dynamic binding*. This allows new objects to be imported into a program to interact with existing objects. If these objects understand a common set of protocols, they can communicate at runtime. Dynamic binding allows programmers to create a style of program that is open-ended. For example, a title developer could ship a new set of scenes and characters to complement an existing title.
- ScriptX offers a full range of control structures, including branching, case statements, for loops, and repeat loops that terminate at either the beginning or the end of the loop. The ScriptX for loop is notable in that it can iterate over the members of a range or collection of objects. ScriptX control structures are implemented as expressions that return values. This allows control structures to be nested within other control structures.
- ScriptX is modular. Using modules, a programmer can specify well-defined interfaces to programming libraries. Through the use of modules, a programmer can protect a program's namespace. Modules allow groups of programmers to collaborate on large programming tasks.

The Kaleida Object Model

The following section describes the main features of the Kaleida object model.

- Classes are themselves objects. Class variables are really instance variables of class objects, and class methods are instance methods of a class object. Class objects are real, denotable objects in ScriptX that can be queried, assigned to variables, and acted upon in the same ways as all other objects. Class objects exist in a hierarchy, with each class object inheriting information from the classes "above" it in the hierarchy. Subclasses add or refine information that makes them more specific than the classes higher up in the hierarchy. Class and instance variables are inherited through the class hierarchy. Each class is an instance of its metaclass. For a description of the Kaleida metaclass network, see the "Object System Kernel" chapter in the *ScriptX Components Guide*.
- Classes and objects defined using the ScriptX language become fully-featured, first-class representations in the ScriptX environment. They interact seamlessly with existing objects and with the language. Although there are internal differences, there is no distinction, in programming interface, between a scripted and a non-scripted class. This allows title and tool development to take place in a modular and incremental fashion, with the ability to re-use and specialize any user-developed classes and objects.

- Methods, like class and instance variables, are inherited through the class hierarchy. When a generic function is invoked, if the particular object that it is invoked on does not define a method for it, ScriptX searches for an appropriate method in each superclass up the inheritance tree until an implementation is found. (The system maintains an efficient dispatch cache to make this searching essentially negligible.) That method is then invoked, with the original arguments, and it operates on the original object.
- Methods are invoked indirectly through the use of *generic functions*, allowing for the separation of interface and implementation. Generic functions allow you to use a single generic name and argument list to refer to multiple implementations of the given generic function that may exist in different classes. In object-oriented programming terminology, this behavior is known as *polymorphism*. Generic functions are called identically to regular functions, with the restriction that a generic function always has a first positional argument, its target object.
- When classes or instances are multiply inherited, instance variable access and method dispatch become more complicated. (Method dispatch determines which method implementation is selected when a generic function is called, as just described.) In singly-inherited classes, the search for an appropriate implementation of a generic function is linear. If a method definition is not found in the current class, its superclass is searched, and so on up to `RootObject`. In multiply-inherited classes, the order in which each class is searched determines which method is invoked, especially in cases where different superclasses may define different implementations for that generic function. ScriptX searches the class hierarchy in a depth-first, topological fashion. (For more information on inheritance, see “Multiple Inheritance” on page 127 of Chapter 6, “Defining Classes and Objects.”) Each individual superclass is searched before any common superclasses are searched.
- Objects that persist between user sessions can be saved to the ScriptX object store. Program elements, including objects, classes, functions, methods, variables, and modules, can be saved to the object store, just as other objects that represent data are saved. The object store is designed for flexible load management—objects can be preloaded into memory for better performance. In the object store, objects are stored in a variety of containers, which are themselves objects. Once data or program elements are stored in containers, they can be moved from system to system, from platform to platform, to run on any system that supports the Kaleida Media Player.
- Memory management is integrated into the ScriptX environment, and is invisible to the ScriptX programmer. Memory allocation is automatic and occurs when objects are created. Memory deallocation is also automatic; memory for unused objects is reclaimed through the use of a real-time, incremental garbage collector when there are no longer any references to the object. The “Memory Management” chapter of the *ScriptX Components Guide* describes memory management with the Kaleida Media Player.
- Access to instance and class variables, the getting and setting of values, is implemented through accessor methods. Each class and instance variable is associated with a corresponding getter method that returns its value. If the value of a class or instance variable can be changed by a script, there is also

a corresponding setter method that sets its value. It is this implementation of variable access that enables ScriptX to support *encapsulation* of data in objects.

ScriptX and the Core Classes

Most object-oriented languages, including ScriptX, are coupled with an extensive class library. Unlike some, however, ScriptX is a *rooted* language, meaning that many ScriptX values are interpreted directly as objects from this library. The ScriptX library provides a rich body of core classes containing powerful abstractions for multimedia programming. For example, there is a hierarchy of classes inheriting from the class `Clock`, which handles timing and synchronization. The programmer can simply use these core classes and specialize them to fit particular needs. The ScriptX core classes are described in the *ScriptX Components Guide* and the *ScriptX Class Reference*.

Being a rooted language, ScriptX was designed with the core classes library in mind. It therefore has many language constructs that know about and incorporate the methods that all objects are expected to have. The result is that ScriptX has multiple surface syntax; in other words, it often has more than one way to express the same thing. These alternate ways of expressing something, which are in many cases shortcuts for longer operations that use methods, are especially important in the areas of object creation and operator syntax. For example, `StringConstant` and `Array` are two core classes where shortcut language constructs can be used to create an instance of the class. By just entering the data which comprises a `StringConstant` object, you create a `StringConstant` object. Similarly, just entering the contents of an array in array form creates an `Array` object. The following are some examples showing both the shortcut language syntax and the corresponding longer version using method syntax:

Language Syntax	Equivalent Method Syntax
`"I am a string"`	`new StringConstant \` `string:"I am a string"`

`global ar := #(1, 2)`	`global ar := new Array`
	`append ar 1`
	`append ar 2`

`global kl := #(1:"one", 2:"two")`	`global kl := new KeyedLinkedList`
	`append kl 1 "one"`
	`append kl 2 "two"`

The examples below demonstrate using operator syntax and array access, which are shorter and more natural than using the equivalent method syntax:

Operator Syntax	Equivalent Method Syntax
`2 + 3`	`sum 2 3`
`5 * 4`	`mul 5 4`
`ar[1]`	`getOne ar 1`
`ar[1] := 2`	`setOne ar 1 2`

What is Object-Oriented Programming?

You can look at object-oriented programming as a conceptually simple extension of what good programmers have always done or as a radically new approach to software creation. In truth, object-oriented programming is both. As such, you can approach learning it from either point of view. The rest of this chapter explores object-oriented programming as an extension to conventional, procedural programming.

Procedural Programming

Procedural programming is based on two entities: data structures and functions. Data structures contain information. Functions act on that information. Traditional programming tools typically create and maintain these two entities separately.

Smart programmers have always found ways to reuse data structures and functions. Reusing them is a good idea—they are already tested, they behave in understood ways, and they often represent data or actions that many programs need.

However, once you move a data structure from one program to another, you discover that you need some functions to manage it. Or, when you move a function, you discover the need for a data structure for it to manipulate. While traditional programming languages don't provide much support for this sort of reuse, good programmers have still done it.

Combining Structure and Function

Object-oriented programming combines data structures and the functions that act on them into a single unit. This combination of a data structure plus the functions that act on that data structure is called an *object*.

You can change the contents of the object's data structure using the functions that come with the object. These functions don't need a separate data structure. You can think of the object as a component.

Combining the two basic entities—data structures and functions—into objects makes it easier to move them to other programs and reuse them. Data structures and functions that represent the information and tasks of a particular enterprise can be saved and shared, making programming tasks quicker to complete and easier to maintain.

Protecting Data

Objects provide the advantage of data protection. Many programming problems are the result of changing a data structure in a way that the programmer didn't expect or didn't intend.

An object controls access to its data by making data inaccessible from outside—you can only get and set its values by using functions for getting and setting that are provided by the object. The correct object-oriented programming term is *encapsulation*. Encapsulation allows an object to provide quality control for its own data.

Thinking in Types

You can think of each type of object as representing a template. This template is similar to the defined types that many other languages support, like a record or a structure. As a template, the object can be used to create new objects that share similar features. Each object created from the template gets a new, initialized copy of the data structure defined for that type of object. Each object created from the template also gets access to the functions that act on its data structure. You can think of the template as a factory for building objects of a particular type.

In object-oriented programming, such a template is called a *class*. Every object is built from instructions that are associated with its class. Objects of the same type are *instances* of the same class. Every object is an instance of any classes that it belongs to or inherits from.

Properties and Methods

Suppose you want to represent dogs in a computer program. You start by listing things you want to keep track of about a dog. In procedural programming the things you want to keep track of are stored in your data structure. Your data structure might contain items or fields such as name, owner, breed, and temperament. Each object will store data for one dog instance (See Figure 1-1).

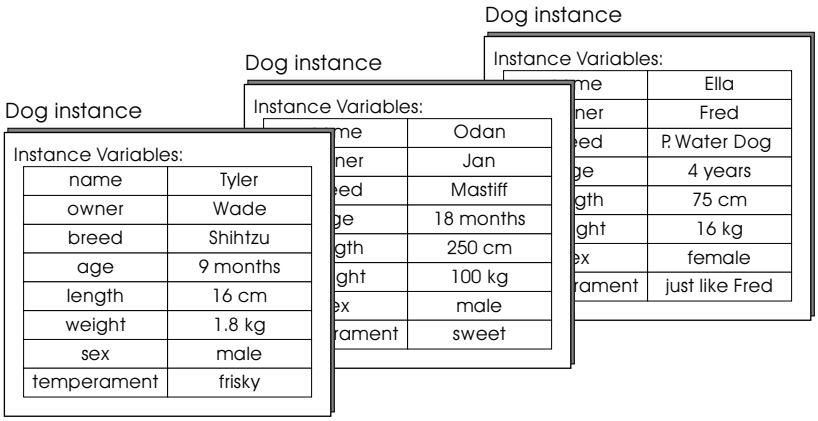


Figure 1-1: Storing data for dog instances

In object-oriented programming, these items or fields are sometimes called *properties*. Most properties are associated with a particular dog, that is, with a given instance of the class `Dog`. Properties that are associated with a particular dog are called *instance variables*. You might also want to define properties for your program that are associated with all dogs—for example, you might want to keep track of the population of dogs. Properties that are associated with the `Dog` class in general are called *class variables*.

Now that you have a data structure for a dog, the next step in designing a program is to figure out what a dog does. A dog performs certain activities, which most programmers would refer to as procedures or functions. In your dog program, these functions might include eat, sleep, run, fetch, bark, bite, and sniff. In object-oriented programming, the actions that an object carries out are sometimes called *behaviors*.

If you wrote a procedural program, most of your program would consist of functions. In ScriptX, functions that are associated with a particular object or class of objects are called methods, in part to differentiate them from functions that are not associated with any object. To summarize, functions that are associated with objects are called *methods*; those that are not associated with objects are called simply *functions*.

Creating Objects and Classes

Although the formal discussion of the syntax for defining objects and classes doesn't occur until Chapter 6, ScriptX is a very natural and readable language. This is a good place to jump in and examine ScriptX code.

This code sample creates a class called `Dog` and an instance of `Dog` called `nikki`. Enter the first eight lines of code below, beginning with the word `class`. This creates the `Dog` class. The `Dog` class can then be used as a template for creating `Dog` objects. Each `Dog` object will have all the properties, such as `owner` and `breed`, that are defined in the template. The `Dog` class also defines three instance methods—`bark`, `fetch`, and `sniff`—just enough to give you the general idea.

```
--
```

```
-- this code example, and others throughout this volume, are on
-- the CD-ROM with the ScriptX Language and Class Library
--
class Dog()
    instance variables
        name, owner, breed, age, length, weight, sex, temperament
    instance methods
        method bark self -> print "makes a lot of noise"
        method fetch self -> print "fetches a stick"
        method sniff self -> print "sticks nose into things"
end
⇨  Dog
```

The return line (⇨ `Dog`) indicates that the ScriptX bytecode compiler has compiled the `Dog` class. That means you can now create `Dog` objects.

The next four lines of code create the first `Dog` object, which is assigned to the variable `nikki`. In this example, the value of four of the instance variables for `nikki` are set at initialization. Since the other four are ignored, their values will be `undefined` until they are explicitly set.

```
object nikki (Dog)
    settings name:"Nikki", owner:#("Jocelyn","Ken"), sex:@female,
        breed:"English Springer Spaniel", temperament:@nervous
end
⇨  Dog@0xef8908
```

Instance variables are like slots or buckets. You can put any ScriptX object into the "slots" that are defined for the dog. This example uses several of the most common classes of object in the system. The dog's `name` instance variable contains a `StringConstant` object `"Nikki"`. The dog's `owner` instance variable contains an `Array` object. Each item in that array is also a `StringConstant` object. The `sex` and `temperament` instance variables contain `NameClass` objects, also known as name literals.

The return value (⇨ `Dog@0xef8908`) tells you the `Dog` object exists, and it gives the memory address. Of course, this memory address will differ with each computing session, and on each platform. (Unlike the real Nikki, this dog stays put at `0xef8908` until you no longer want her around!)

Ignoring for now the ScriptX language syntax, notice how easy it is get back information about `nikki`. Here's what you type into the Listener to get access to the `owner` instance variables defined by the object `nikki`.

```
nikki.owner
⇨  #("Jocelyn", "Ken")
```

Now consider the three instance methods defined for the class `Dog`: `bark`, `fetch`, and `sniff`. In object-oriented programming you can think of *calling* a method as sending a message to an object, telling that object to perform some operation.

```
fetch nikki
```

```
⇨  "fetches a stick"
   OK
```

The third line (`OK`) is the return value of the `fetch` method. The `print` function in the definition of the `fetch` method prints `"fetches a stick"` to the Listener window. This string is not itself a return value—printing a string is just an operation that `fetch` carries out. Throughout this manual, an arrow (⇨) indicates output from ScriptX to your Listener or debugger window, including both printed messages and return values. Output from ScriptX is also indented, to indicate that you do not type it in.

This example shows how you can create a description of a dog's properties and behavior as part of a dog template. This template is the `Dog` class, from which usable `Dog` objects can be created. Each `Dog` from this template has the same properties and the same general behavior, as described by its methods.

Note that each dog could behave differently, even though all `Dog` objects share the same behavior. This is because a dog's methods have access to the object's own data. You could write a `fetch` method such that a `Dog` object with good temperament fetches faster than a `Dog` object with bad temperament.

Beyond Traditional Programming

So far, object-oriented programming simply represents a codified system for doing what good programmers have always done: protect their data and reuse trusted, tested code. This section focuses on things you can do with object-oriented programming that couldn't be simulated in procedural programming languages.

Defining by Difference

Many times, programmers need to create new code that is similar to existing code. In particular, there is often a need to modify the functionality of something that already works, without totally rewriting it. Object-oriented programming provides just this type of extended reuse, allowing you to create a new class using an existing class as a base. The new class is a *specialization* of the existing class. You only need to define how a new class is different from its parent class—what properties and behavior it adds to or modifies in the base class. In other respects, the new class is the same as the base from which it was created.

Object-oriented programming is characterized by *inheritance*. This means that a specialized class inherits the properties and behavior of its parent, which leads to a hierarchy of classes, grouped together based on similarities in their data structures and methods. This grouping of classes provides a family tree of classes that range from general to specialized.

Going back to the dog example, say you want a world that is populated by many kinds of dogs. The example created one class, `Dog`. For some programs, that might be adequate. But what if you want to create a program that really models all the different ways that dogs behave?

Object-oriented programmers often design a hierarchy of classes that reflect the specialization of behavior and properties they need to model. The `Dog` class has general properties and behavior for all dogs. Using inheritance and specialization, you can specialize that `Dog` class to create more specialized classes that have particular characteristics and behavior. For example, you could divide dogs into various categories like hunting dogs and lap dogs as shown in Figure 1-2. There is no right way to categorize dogs; how you set up the dog hierarchy should depend on how you want to use them. Think of all the ways to divide up the world of dogs!

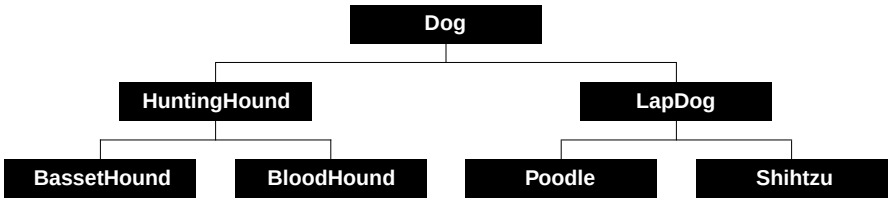


Figure 1-2: Creating inheritance hierarchies of dogs

One of the advantages to creating a hierarchy of classes is that different classes can share the same behavior, or methods. This sharing of behavior is called *polymorphism*. Each class or object that inherits a method can define its own version of that method.

Technically, when your ScriptX program calls a method, it does not call the method directly. For each method name, ScriptX creates a *generic function*. Unlike regular functions, generic functions are always associated with an object. Generic functions look just like regular functions, except that the first argument is always an object. You always call a generic function on a particular object. ScriptX determines which method to call, based on the value of the first argument to the function, which is the object you are calling the function on. Methods are really versions of generic functions, defined for a particular class or object.

Generic functions reduce the complexity of the system of objects. A programmer can get work done using the small vocabulary of generic functions that is shared by a family of objects. Rather than a single function with a single behavior, a generic function stands for a whole range of general behavior, shared by a whole set of objects. Each class of object supplies an appropriate implementation of the generic function . A programmer can easily learn to use all dog classes, since every dog implements `bark`, `fetch`, and `sniff`—the advantage is that each dog can implement them in its own way. For example, each specification of the `Dog` class can have its own implementation of `bark`. The following example shows how to specialize the `Dog` class by creating two new subclasses: `HuntingHound` and `LapDog`. In object-oriented programming terminology, `Dog` is a *superclass* of `HuntingHound` and `LapDog`. `HuntingHound` and `LapDog` are *subclasses* of `Dog`. Notice the specialization of the `bark` generic function on the `HuntingHound` and `LapDog` classes.

```
class HuntingHound (Dog)
    instance methods
        method bark self -> print "wooof, wooof"
```

```
end
⇨  HuntingHound
class LapDog (Dog)
    instance methods
        method bark self -> print "yip, yip, yip, yip, yip"
end
⇨  LapDog
```

Hunting hounds and lap dogs continue to share the same implementation of the `fetch` and `sniff` methods, since they are not specialized. The `bark` method, although it was already defined by the `Dog` class, has been redefined. In object-oriented programming terminology, to specialize behavior in this manner is to *override* a method.

To override a method is not necessarily to replace it with a completely new version. A subclass can invoke the version of a method that is provided by a superclass in its own implementation of that method. Here is an example.

```
class Shihtzu (LapDog)
    instance methods
        method bark self -> (
            nextMethod self
            print "jumps up and down"
        )
end
⇨  Shihtzu
object tyler (Shihtzu) settings name:"Tyler" end
⇨  Shihtzu@0xefcf88
bark tyler
⇨  "yip, yip, yip, yip, yip"
    "jumps up and down"
    OK
```

Note – Since you are used to getting a return value in the Listener window, from now on, return values are shown only where required by the discussion.

In the example above, the expression `nextMethod self` causes any instances of `Shihtzu` to call the `bark` method defined by the next superclass that implements it. In this case, the next implementing superclass is `LapDog`. Each instance of `Shihtzu` first calls the more general `bark` method for all instances of `LapDog`, and then continues with the specialized version for a `Shihtzu` object.

Polymorphism, the sharing of behavior, allows you to be more general because you are specific in indicating which object should fulfill the request. The programmer uses the generic function to describe what should be done and leaves it to the object to determine how exactly to do it.

By comparison, functions in a procedural program have only a single version. You must create a separate function or add special case code and additional arguments for each new thing you want to do. Modifying a procedural program increases the complexity of the system, and makes it harder for you to maintain and scale your code.

Protocols

A class that defines a common set of methods that every subclass either inherits a definition of, or creates its own definition for, is said to define a *protocol*. For example, the `Dog` class defines a protocol in the `bark`, `fetch`, and `sniff` methods. These three methods are implemented for every instance of `Dog`, making them the `Dog` protocol. A protocol is a set of generic functions that is defined for every class in some branch of a class hierarchy.

In some object-oriented programming environments, the concept of protocol is much more formal. In ScriptX, it is an informal term that identifies a set of shared behaviors. As a programmer, it makes your life easy to know that every dog, whether a golden retriever or a cocker spaniel, knows how to bark. Each subcategory of dog, each individual dog breed, or even each individual dog, can define its own bark, but you are guaranteed that if you have a dog, it knows how to bark.

Defining New Behavior

You can create completely new behavior in a subclass. This example creates a `BassetHound` class that implements the `drool` method, a method that other dogs, with less active salivary glands and more active facial muscles, might not share. In the dog world example, drool behavior is peculiar to basset hounds.

```
class BassetHound (HuntingHound)
    instance methods
        method drool self -> print "slobbers all over everything"
end
-- now create an instance of BassetHound
object vaps (BassetHound)
    settings name:"Vaps", owner:"The Crosbys"
end
-- call the drool generic function on vaps
drool vaps
⇨  "slobbers all over everything"
```

Note the following distinction, which is important in object-oriented programming. In the previous example, it isn't the `BassetHound` class that drools. The `BassetHound` class is a template used to define the properties and behavior of basset hounds. To be more precise, the `BassetHound` class is a template that defines the ways in which basset hounds differ from hunting dogs, and from dogs in general. But you have to create an actual instance of `BassetHound`, a `BassetHound` object, in order to see a dog drool.

Multiple Inheritance

Up to this point, the model for the world of dogs contained only instances of some subclass of `Dog`. That might do if you were only interested in one aspect of a dog's behavior. But in the real world, systems are far more complex. Some unlucky dogs are wild or stray, and do not have owners. Instead, they have territories and live in packs. Other dogs are tame, and have both owners and

veterinarians. Naturally, wild dogs behave very differently from pets. But so far, the dog world example is set up as if all dogs have a property called `owner`, an instance variable that stores the owner's name.

Fortunately, ScriptX allows any object or class to inherit from more than one parent class. This *multiple inheritance* allows you to factor the behavior of dogs into several different parent classes. To *factor* behavior is to divide or separate aspects of behavior in logical ways. The ability to factor information or behavior is one of the benefits of multiple inheritance. For example, you can keep the `Dog` class, renamed `Canine`, and create two new classes, `Pet` and `WildAnimal`, that contain aspects of canine life that are peculiar to domestic and feral dogs (see Figure 1-3). Mixing these classes together will produce dogs with the desired characteristics.

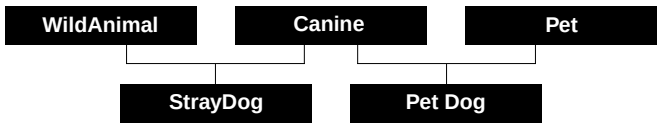


Figure 1-3: Multiple inheritance and dogs

The following example creates a `PetDog` class that uses multiple inheritance to define properties and behavior. The new class is a template for creating objects that share characteristics of all of its parents. `PetDog` inherits from both the `Canine` and `Pet` classes, which incorporate the general properties and behavior of the previously defined `Dog` class, and add new ones.

Note – If you have been using the ScriptX Language and Class Library in one continuous session since the beginning of this chapter, you will not be able to redefine the `Dog` class until you eliminate all instances of `Dog` from the system. Instead of reusing `Dog`, this example uses the name `Canine` as a root class for dogs.

```
class Canine ()
    instance variables
        age, length, weight, sex, temperament
    instance methods
        method bark self -> print "makes a lot of noise"
        method sniff self -> print "sticks nose into things"
        method sleep self -> print "lazy dog sleeps all day"
end
class Pet ()
    instance variables
        name, owner, breed, veterinarian, spayed
    instance methods
        method fetch self -> print "fetches a stick"
end
class PetDog (Pet, Canine)
end
```

The `PetDog` class does not actually define any new behavior or properties for `PetDog` objects. An instance of `PetDog` will have all the instance variables and instance methods that its superclasses define. Now create a `PetDog` object.

```
object tammy (PetDog)
    settings name:"Tammy", owner:"the Metzenbergs", sex:@female,
        spayed:@true, breed:"Siberian Husky", veterinarian:"Dr. Donovan"
end
fetch tammy
⇨  "fetches a stick"
sniff tammy
⇨  "sticks nose into things"
```

`PetDog` object `tammy` defines properties of both a `Canine` object and a `Pet` object. She barks, sniffs, and sleeps like a canine, and she fetches like a pet.

Although you would probably want to override the `fetch` method, you could easily create a `Cat` class and combine it with `Pet` to create the new class `PetCat`. In object-oriented terminology, `Pet` is being used to *mix in* characteristics of `Pet` with another class to create a new subclass that offers additional features.

As you work with ScriptX, you will find many examples of multiple inheritance in use. With multiple inheritance, it is easy to add new features to classes and objects you create. ScriptX ships with a library of media classes called the core classes. You can add functionality to your own classes by mixing them in with these predefined media classes. For example, if you wanted to have your `PetDog` object appear in a ScriptX window, you could create a new class that mixes `PetDog` with `TwoDShape`, one of the core classes. As an instance of `TwoDPresenter`, a `TwoDShape` object can present a graphic object, such as a bitmap. Mixing in the `TwoDShape` class would give your objects the ability to display a target object (such as a bitmap) in a window with a two-dimensional coordinate system.

Ease of Modification

Suppose you have already built a system that works. You have a trusted, tested sequence of logic. Do you want to add new functionality, when any change might break your entire system?

In an object-oriented environment, you can isolate changes so that they have an effect on only a particular class, a group of classes, or even a particular object. Suppose you have a pet dog that sleeps on top of his doghouse. You don't have to create a new class, or modify any existing classes. You can add this new behavior to a particular `PetDog` object when you create the object.

```
object snoopy (PetDog)
    instance methods
        method sleep self -> (
            print "sleeps on top of his doghouse"
            nextMethod self
        )
    settings name:"Snoopy", owner:"Charlie Brown", breed:"Beagle"
end
sleep snoopy
⇨  "sleeps on top of his doghouse"
    "lazy dog sleeps all day"
    OK
```

As this example shows, specialization in ScriptX is not limited to defining new classes. You can add new instance variables and define or override instance methods at any level, even for a single object.

Connections Between Objects

In ScriptX, as in all object-oriented languages, you often build complex data structures by connecting objects. ScriptX stores values by reference. When you assign or set an instance variable, you store a reference to another object. In this respect, instance variables create a network of connections between objects. These object connections allow you to connect information in ways that are natural and logical, avoiding duplication of information and functionality.

Suppose you want to know where pet dogs live. As defined in the previous section, the `PetDog` class inherits from both `Pet` and `Canine`. From the `Pet` class, a pet dog inherits the instance variable `owner`. If an owner has an `address` instance variable, you can get access to that information through the dog. The following example defines the `Owner` class and creates an instance of `Owner` and an instance of `PetDog`.

```
class Owner ()
    instance variables
        name, address
end
-- create an owner
object jan (Owner)
    settings name:"Jan", address:"Fairfax, California, USA"
end
-- create a dog for the owner to own
object odan (PetDog)
    settings breed:"Mastiff", sex:@male, name:"Odan"
end
```

If the dog's `owner` instance variable is set, you can figure out where the dog lives by examining the owner's `address` instance variable.

```
odan.owner := jan
odan.owner.address
⇨  "Fairfax, California, USA"
```

This connection makes additional information available about a pet dog, without modifying the `Canine` or `PetDog` classes. (In an actual program, you could make this connection at initialization, and introduce error and type checking.)

Building and Maintaining Libraries

Traditional libraries of code are collections of pretested functions that extend a language. These functions typically address narrow areas of specific behavior, such as database design or special mathematics. Rarely are all the necessary data structures provided. Most libraries can't attempt to solve complete, application-level problems. The library code is intended to be connective. As

the programmer, you write the majority of the code in an application. You construct the problem-solving framework and use library code where appropriate to fill in the gaps.

Object-oriented libraries are collections of classes arranged in hierarchies that resemble family trees. The library classes and the objects created from those classes are complete and ready to use. They can address larger scale problems than function libraries can. The programmer can use them to accelerate development, making it possible to prototype rapidly by using library objects as a base for exploration.

More importantly, it is possible for an object-oriented system to provide a set of objects that have built-in relationships. By creating collections of cooperating objects, it's possible for an object-oriented system's libraries to act as pretested problem-solving frameworks. The framework provides the majority of the code. You plug your objects into the slots left in the framework.

It's easy to modify object-oriented systems without breaking them, because the changes are localized in individual objects, not spread out across many data structures and functions. Adding new behavior to an object-oriented system can often be as simple as creating a new type of object and placing an object of that type in an existing system of code. The system is modified by changing the type of the object that receives a method. The trusted, tested sequence of logic doesn't have to change.

Special Kinds of Classes

The ScriptX core classes library defines a few classes that cannot be specialized. This means you cannot create new subclasses of those classes. You cannot add methods or instance variables to those classes, or to instances of those classes. These classes are said to be *sealed*. Sealed classes are often internally optimized and as such do not provide behavior that would be available to subclasses. The sealed classes include classes that represent numbers, Boolean values, names, gates, and threads. Most of the classes in the core classes are not sealed.

Another special kind of class, common throughout the core classes, is an *abstract* class. When a class hierarchy is organized, information is factored into superclasses so that common subclasses can share and reuse it. Often, when information is factored in this way, classes can result that provide partial information for their subclasses, but do not contain enough information to create fully-featured instances of themselves. These classes are called abstract classes. An abstract class can be subclassed or mixed in with another class, but it cannot be instantiated. Abstract classes exist as a basis for defining other classes. ScriptX prevents you from instantiating an abstract class. Any class that is not abstract is *concrete*.

A final distinction is between *scripted* classes and the core classes themselves. A scripted class is one that is created using the ScriptX language. (The core classes are created with Objects-In-C, an extension of the C programming language developed by Kaleida Labs, Inc.) All scripted classes are unsealed. Although there are some internal differences between scripted classes and the core classes, their external interface is the same.

Wrapping Up Objects

In summary, objects are data structures that are associated with both data and the functions, which are called methods, that act on that data. Objects are created from classes, which act as templates for building objects. Each object is an *instance* of its parent class.

Classes are organized in a hierarchy that resembles a family tree, based on the characteristics they share. Classes *inherit* the properties and behavior of their parent classes. When it is created, or *instantiated*, each object gets a copy of the instance variables and access to the instance methods that are defined for its class.

With each new class of objects, a programmer is free to define new instance variables and methods, or to override existing methods to behave in a new, appropriate way for the particular class. This redefinition can even incorporate the parent's version of the same method.

This class inheritance tree ranges from very general types at its root to more specialized types at the branches. The types in a particular branch of the tree share a common way of doing things. New classes can be added at any point in the tree. Almost any existing type can be the parent for a new type.

Much of the real power of an object-oriented environment is found in the class libraries that can be included with an object-oriented language. They allow a programmer to concentrate on the problem at hand, building solutions from reliable, tested components.

The primary benefits of object-oriented programming are reduced complexity, increased reusability, and extensibility. Objects are a way of bundling data and functions together to make a reusable software component. Objects protect their internal workings, hiding their data and generalizing their functions so that there are fewer details for a programmer to deal with. Classes are factories for making usable objects. Classes provide the basis for making new types simply by describing how the new type is different from an existing type.

Object-oriented programming gives you the ability to create a general framework which, because of polymorphism, continues to work with new and unexpected specializations of the system. All the basic frameworks in ScriptX, such as the animation compositor, depend on polymorphism. Because of polymorphism, you can rely on existing frameworks to work with new objects that you define in your title.

C H A P T E R

ScriptX Building Blocks

2

This chapter describes many of the basic building blocks of the ScriptX language, including expression syntax, comments, lexical constructs such as end-of-line, literals such as numbers and strings, variable scope, declaration, assignment, and simple operators.

Expressions

ScriptX is an expression-based language. An *expression* is a language construct that can be evaluated to yield a value. In ScriptX, every complete construct in the language yields a value, and that value is always an object.

Valid expressions in ScriptX include things that you might not consider to be expressions in other languages, such as loops and conditionals. Since every construct in ScriptX yields a value, you can write more flexible code by nesting expressions within other expressions and passing around their results.

When a ScriptX expression is evaluated, the resulting value is always an object, and that object is an instance of a class (either one of the ScriptX core classes or a class you have defined). Where the distinction between the value and the class of an object is important, this book refers to each separately. For example, the expression `2 + 2` yields an `ImmediateInteger` object whose value is `4`.

End-of-Line

ScriptX is a flexible, command-based language which, unlike some other programming languages, does not require statement punctuators such as semicolons. In ScriptX, an end-of-line most commonly signifies the end of an expression. You can break an expression over multiple lines, but only if the break occurs at a point where the expression is incomplete.

For example, you could write the expression `a < b` as follows:

```
a <
b
```

In this example, the end of the expression has not yet occurred at the end of the first line, so ScriptX looks to the next line for the expression to be completed. That same expression could not be written this way:

```
a
< b
```

Because `a` is a complete expression in ScriptX, the end-of-line after `a` is considered the end of the expression. The second line would then result in an error, since the ScriptX bytecode compiler expects to find a new expression beginning on that line.

A sentence is a complete syntactic construct that can be evaluated by the ScriptX bytecode interpreter. The bytecode interpreter does not evaluate an expression until you enter a “complete sentence” in the Listener window. An incomplete sentence can be continued on the following line. The following generalizations apply to completion of ScriptX sentences.

1. When the interpreter is expecting a closing delimiter, such as a parenthesis, square bracket, or brace, the sentence is incomplete.
2. A complete sentence cannot end with a separator, such as a comma, colon, or function body separator (`->`).
3. A complete sentence cannot end with a binary operator, such as an arithmetic or comparison operator (+, –, *, /, =, ==, <>, !=, !==, <, >, <=, >=), the collection access operator ([ ]), the function operator (( )), or the assignment operator (:=).

Line breaks can make a ScriptX program less readable. Some programmers use the backslash (\ ) to indicate all line breaks, even in cases where it is not required by ScriptX syntax. Most programmers break lines only in obvious places, such as after separators and binary operators.

A script cannot be broken in the middle of a literal (such as a number or string) without causing a syntax error, or changing the meaning of the expression.

```
-- this example generates a syntax error
global myArray := #(33, 44, 55, 6
6, 77, 88, 99)
-- ** Syntax error: ("syntax error") at "6" (SyntaxError)
```

In the next example, the line break is interpreted as a newline character:

```
global myString := "This is my
string"
⇨  "This is my
   string"
```

To force evaluation of an incomplete sentence in the scripter, type `!!` and press **enter**. Two exclamation marks act as a terminator, allowing you to begin entering a new expression. If the expression that the scripter was currently evaluating is incomplete, termination reports a syntax error.

```
(a + b) < !!
-- ** Syntax error: ("syntax error") at "(a + b) < !" (SyntaxError)
```

For more information on ScriptX syntax and expressions, see Appendix , “ScriptX Reference.”

Continuation Over Multiple Lines

Most ScriptX expressions can be written on a single line. Some expressions have recognizable boundaries that allow them to be broken over multiple lines. If an expression is too long to fit on a single line, and cannot be divided, you can continue an expression on the next line by placing a backslash (\ ) at the end of the line.

A backslash indicates that an expression continues on the next line. The backslash itself is treated as white space. If you prefer, you can use a backslash wherever there is a line break, for readability. When an example in this book uses the backslash, the second line is indented to show continuation from the previous line.

```
1 + 2 + 3 \
   + 4 + 5
```

Since tabs are treated as blank space in ScriptX, you can use tabs freely to make your ScriptX code more readable. One place you might not want to use a tab is in a string—you wouldn't want tab characters embedded in your strings.

```
print "This is an example of a very long expression, one that \
is too long to fit on one line."
```

The example above shows a call to the `print` function. Function calls are the most common case of an expression that is too long to fit on one line but cannot be split over multiple lines unless you use a backslash. Function calls are described in further detail in Chapter 3, "Working with Objects."

Finally, multiple expressions can be placed on one line, separated by semicolons:

```
t := b * b; if t < j then t else j; print b; print t
```

Comments

A comment, which is a text annotation added for readability, is ignored by the compiler. The ScriptX language implements two styles of comments: end-of-line comments and inset comments. Since they can extend over many lines, inset comments are often called multiple-line comments.

End-of-line comments begin with two dashes (`--`) or two slashes (`//`). Everything that follows the second dash or slash, up to the end of the line, is considered a comment and is ignored by the compiler. Note that a backslash (\ ) within a comment is ignored. To continue a comment onto the next line, you must precede that line with dashes or slashes as well.

```
-- This is a comment that is quite verbose so I have
-- no choice but to continue it on the following line.

// This comment style is for lost souls who suffer C++ envy
```

End-of-line comments are useful for appending to a line of code:

```
2 + 5 - (15 / 4) -- some random arithmetic
true == false // a silly comparison
```

Inset or multiple-line comments (`/* this is a comment */`) use the same notation as the C programming language. Unlike C, ScriptX allows inset comments to be nested, as in the following example.

```
/*  This is part of a comment which
    will continue for many lines,
/*  and so is this,
    and this too!
*/  But the comment is not finished yet!
    This is still part of the same comment!
    It ain't over till the woman of size sings!
*/
```

The ability to nest multiple-line comments provides a versatile mechanism for "commenting out" sections of source code. Although nested comments are more flexible than the unnested C-style comments, several other kinds of errors are possible. For example, by deleting a comment terminator that is nested within another comment, a developer might inadvertently comment out other parts of a program.

Numbers

One of the simplest constructs in ScriptX is the number literal. A number literal evaluates to an instance of the appropriate `Number` class. Figure 2-1 shows the available number classes. The "Numerics" chapter of the *ScriptX Components Guide* provides additional information on the range and precision of each of these classes.

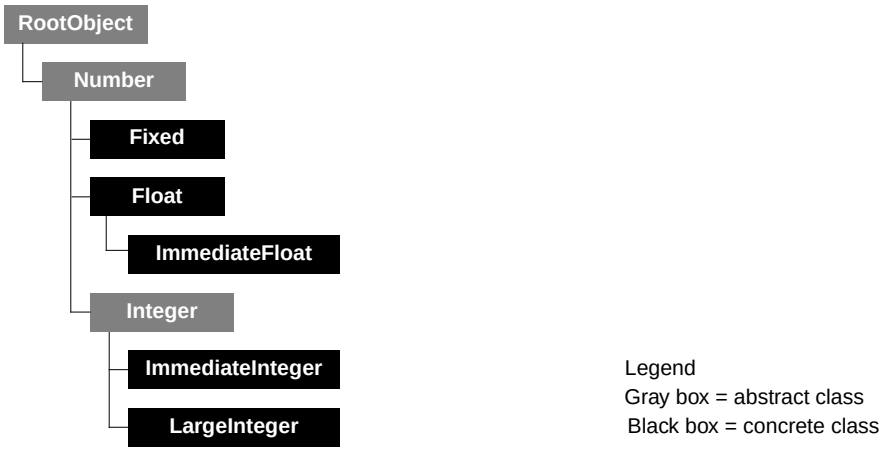


Figure 2-1: ScriptX number classes

Floating-point numbers must begin with an integer—that is, you must use `0.1234` rather than simply `.1234`. Floating-point numbers with no fractional value must include a `0` after the decimal, that is, `4.0` or `12345.0`, to differentiate them from integers. For example, `4.0` is treated as a floating point number, while `4` is treated as an integer.

Floating point numbers can also be represented using exponential notation.

```
6.023e23 -- Avogadro's number
```

Negative numbers are preceded by a minus sign.

```
-123.456
6.626e-27 -- Planck's constant
-0.99999999
```

Keep in mind that floating point values, unlike integers, are not stored with exact precision on computer systems.

```
4.2 -- it will be stored as an ImmediateFloat object
⇨  4.19999885559082
```

Hexadecimal Numbers

You can use numbers in scripts in hexadecimal form using the `0x` notation.

```
0xFFF
0x04
```

When a hexadecimal number literal is evaluated, ScriptX creates an instance of `ImmediateInteger` or `LargeInteger`, just as it would for its decimal equivalent. Only integer hexadecimal numbers are supported by ScriptX.

Strings

String literals in ScriptX consist of one or more characters delimited by double quotes (").

```
"This is a string."
```

The value of a string literal is an instance of the class `StringConstant`, which stores the string as a sequence of UTF values, an optimized representation of 31-bit Unicode/ISO 10646 values. Unicode allows ScriptX to represent most international characters.

Strings can contain any characters, including single quotes, operators, and other characters that would be interpreted differently by ScriptX outside the string. The following strings contain character sequences that make them look like other ScriptX constructs:

```
"Is it a string? Yes, it's a string."
"-- A string with a comment in it"
"a string with an assignment := in it"
"a string with arithmetic in it: 4 + 4"
```

Strings can also include end-of-line characters; that is, strings can be spread over several lines. End-of-line characters are preserved in a string as newline characters:

```
"Jack and Jill
went up the hill
to fetch a pail of water"
```

Table 2-1 lists special characters that can be included in strings as escape sequences.

Table 2-1: Special characters in strings

Characters	Meaning
\r	return
\n	new line
\t	tab
\"	double-quote
\\	backslash
\<nnnn>	hexadecimal values for Unicode characters

Note that ScriptX handles the problem of different implementations of the newline character (\n) on different platforms by forcing the newline character from all platforms to the value 13.

When the text string is compiled, ASCII special characters are converted into their internal equivalents, and are not displayed again as escape characters.

```
"a string with a \n return in it"
⇨  "a string with a
return in it"
```

Non-ASCII Unicode characters are represented as Unicode escape characters when they are printed to a stream, such as the Listener window.

```
-- \<2122> represents the trademark symbol ( ™ )
"ScriptX\<2122> is a really awesome language"
⇨  "ScriptX\<2122> is a really awesome language"
```

At this time, the operating systems on which the Kaleida Media Player runs do not support the Unicode standard. Full support is expected in the future. In the interim, ScriptX maps common Unicode characters to platform-specific character sets. Characters are displayed correctly as the target of a text presenter, provided that the character is supported by the underlying system, and that an appropriate glyph is available in the display font.

ScriptX does not map platform specific (non-ASCII) characters to their corresponding unicode characters. Representation of non-ASCII characters requires the use of escape sequences, as in the example above. Appendix , “Unicode Escape Characters,” contains a listing of many commonly used

special characters and their hexadecimal Unicode values. For an explanation of Unicode and UTF encoding, see the "Text and Fonts" chapter of the *ScriptX Components Guide*.

A Note on Strings and String Constants

String literals, described in this section, are instances of the class `StringConstant`. Instances of `String` and `StringConstant` appear the same in most operations, but functions or operators (such as the comparison functions described on page 40) that are concerned with the classes of those objects may differentiate between them.

When you attempt to modify a string constant, some operations convert it automatically to a `String` object, its editable counterpart. These operations include string addition and subtraction, described on page 43. Other operations require that you explicitly coerce the string constant to a string. Coercion of objects from one class to another is described in Chapter 3, "Working with Objects."

System Objects

ScriptX provides several objects, created at start-up, that serve as system constants. Table 2-2 describes those objects.

Table 2-2: System objects

Object	Use
`true, false`	The `true` and `false` objects are the two instances of the `Boolean` class. If `true` and `false` are coerced to a `Number` class, `true = 1` and `false = 0`, so that `true > false`.
`undefined`	Equivalent to `NULL` in other languages; `undefined` should be used to mean "has no value."
`unsupplied`	The value assigned to any keyword arguments that were not given explicit values when the function or method was called. Keyword arguments to functions are described in Chapter 5.
`empty`	Used by functions that operate on collections such as arrays to indicate "element asked for was not found." Do not use `empty` to mean `undefined`; use `undefined` instead. See page 163 for more information about the `empty` object.
`OK`	Used by functions that have specific side effects and no relevant return value (similar to functions in C that return *void*). Functions that return `OK` should always return `OK` regardless of the outcome of the function (unless the function reports an exception and the thread it is running in dies). See Chapter 5 for more information about functions and threads. See Chapter 8 for more information about exceptions.

You may be accustomed to languages where certain functions, like void functions in C and C++, return no value. ScriptX is a language of expressions, a language where every expression returns a value. That return value is always an object. Think of a system object as a pointer to some fixed location in

memory. A system object is simply a value that is the outcome of so many ScriptX expressions that it has been predefined by the system. All such values are defined as objects, providing completeness to the ScriptX object system.

You can use system objects in your own scripts. Be careful to use them exactly as they are defined in Table 2-2, so that the classes and objects you create can work with other classes and objects.

Names

There are two kinds of names in ScriptX, lexical names and name literals. Lexical names are used to name variables and other denotable things, including functions and local function arguments. Name literals can be used as labels for function arguments, as keys in keyed collections, or anywhere else that a simple label might be required.

ScriptX names are case insensitive. That is, upper and lower case letters are treated the same. However, ScriptX remembers the case used the first time a new name appears and uses that case whenever it displays that name.

Throughout the Kaleida Technical Reference Series, the following case conventions are used:

- Names of functions, methods, and variables are often compounded from several words. In cases like this, the first letter in every word except the first is capitalized:

```
appendNew
beginDecompressSeq
```

- Names of modules and classes have all words capitalized, including the first word:

```
SequencePlayer
KeyedLinkedList
```

ScriptX defines about a hundred reserved words that cannot be used as lexical names. (They can be used to name keyword arguments for functions.) For a complete list of reserved words, see Table A-3 on page 236.

The ScriptX core classes define the names of many classes, variables, methods, keywords, global functions, global variables, and global constants. These names are imported into any module that uses the ScriptX module. A script can override these names, both globally and locally. It is quite common to override them locally. For example, a class may use the name of another class internally as the name of an instance variable, often an instance variable that it uses to store an embedded object. Although it is permitted, it is not good programming practice to override any of these names globally.

Lexical Names

Lexical names are used to name things in a program that the compiler must interpret during compilation, such as variables, local functions, and function arguments.

Names must begin with an alphabetic character or an underscore (_). They can contain only uppercase and lowercase letters (A–Z, a–z), numbers, or the underscore character. Names cannot contain spaces, tabs, special characters, or non-alphanumeric characters such as symbols and punctuation. They can be up to 256 characters in length.

Here are some examples of valid names:

```
foo
mylist
_x
HugePulsingCreatureFromTheCratersOfVenus
bignum4500
oswald_acted_alone
```

Here are some examples of names that are invalid:

```
23skidoo -- invalid, because it starts with a number
curses@(*&#!*@& -- invalid, because it contains symbols
spread out -- interpreted as two names, because it contains a space
aïda -- contains an accented or inflected character
```

Name Literals

Unlike lexical names, which are used to name things at compile time, name literals denote `NameClass` instances which exist at run time. Name literals evaluate to `NameClass` objects.

- To create a name literal from a simple name, use an at sign (`@`) before the name itself:

  ```
  @foo
  ```

- To create a name literal from an expression, call `intern NameClass` on the expression:

  ```
  intern NameClass ("foo" + "bar")
  ```

Name literals follow the same upper and lower case conventions as lexical names. Here are some more examples and uses of name literals:

```
@utterly_undefined_thing
prin variableThing @normal debug -- used as a label for an argument
animals := #(@snake:"cobra", @bird:"sparrow", @fish:"guppy")
```

Name literals form a proper set. Multiple occurrences of a name literal with the same sequence of characters always evaluate to the same object. For example, the names `@lefthanded`, `@LeftHanded`, and `@LEFTHANDED` are not merely

equal—they represent the same object. Name literals can be stored and compared more efficiently than strings, since multiple instances of strings may contain the same sequence of characters but are still different objects.

Name literals are useful as labels to make your scripts more readable. For example, if a function has an argument that is always one of three values, you can use name literals for those values instead of using numbers or strings. Additionally, name literals can be used as keys in key-value pairs, or anywhere else you need a valueless label in a script. Name literals are used extensively in this way by the ScriptX core classes.

Variables

This section describes variables and scope, including variable declaration and assignment. It also provides a brief introduction to modules. Modules are covered in greater detail in Chapter 9, “Modules.”

Variable Scope and Extent

The *scope* of a variable definition determines where that variable is defined and where it can be referred to by name. The *extent* determines how long that variable remains defined.

ScriptX provides two different kinds of scope for its variables: global and local. Variables that are declared `global` have a global scope. This means they continue to exist for the duration of the program’s execution, and they are visible in modules that use the current module. Variables that are declared `local`, however, can be declared and used only within the context of a limited variable scope. Local declarations cease to exist once that variable scope has ended.

ScriptX does not directly define a static scope, as in C and C++, but equivalent functionality is available through closures. A closure variable is a variable that persists in memory like a global variable, but is visible only within a local scope. Closures and closure variables are defined on page 107.

The ScriptX language has several constructs that introduce a new variable scope different from that of the scope surrounding it, including function and method definitions and some loops. Compound expressions are the most commonly used construct that defines a new variable scope. See “Compound Expressions” on page 53 for more information about local variables and examples of how variables act within different variable scopes.

By default, ScriptX variables are global, including the variables that are used to name functions, classes, and some objects. However, you should explicitly declare each variable you use to be either `local` or `global` using the expressions described in the section “Declaring Variables” on page 37.

A Note on Modules

In addition to the concepts of local and global variables, ScriptX also provides support for `modules`. Modules in ScriptX are a packaging system that allows control over multiple global variable namespaces. Global variable names are only visible within the module where they are declared, and in modules that use that module. Modules allow for integration of large titles from components made by many different programmers without naming conflicts.

Every ScriptX expression is compiled and executed within the context of a given module. Global variables defined and used within a single module are only visible to other expressions that are visible within that module. Different modules can have different global variables with the same name. Using modules, you can import, export, and rename variables across modules to combine parts of ScriptX programs without worrying about whether different parts of a program use the same names.

You do not need to define modules, or even understand modules, in order to use ScriptX. Most ScriptX environments provide a default module in which you can define variables, enter expressions, execute scripts, and experiment with ScriptX. ScriptX modules are useful when you are creating larger programs or parts of programs that you expect to distribute and be used by other ScriptX programs. Chapter 9, “Modules,” describes how to define and use multiple modules to manage variables in your ScriptX program.

Declaring Variables

Variables are declared using the `local` or `global` construct.

▼ `local` *varname, varname, varname* `. . .`
`global` *varname, varname, varname* `. . .`

The *varname* is the name of the variable to be declared. Multiple variable names can be specified in the same expression, separated by commas.

Local variables can only be declared and referred to by name inside a construct that introduces a new variable scope (such as a compound expression, described on page 53). You cannot declare them at the outermost level.

Global variables can only be declared at the outermost scoping level. It is considered good programming practice, in ScriptX as in all languages, to declare a global variable before it is used. Many programmers declare global variables at the beginning of a program or module.

```
global density, mass, velocity
```

You do not have to declare variables before they are used. Be forewarned, however, that the first time you reference a variable that has not yet been declared (either by assigning an object to it or by simply referring to it by

name), that such a variable is declared `global` (regardless of the scope in which it is defined) and a warning is printed to your environment's debugging window or stream:

```
mishegoss := @normal
-- ** Warning: Undeclared global mishegoss
⇨  @normal
```

For the sake of brevity, many examples in this book do not explicitly declare global variables. Readers should assume that any global variables used in examples have already been declared.

If you try to reference a variable that has been declared, but has not yet had anything assigned to it, ScriptX reports an exception.

```
global pickup
print pickup
-- ** While calling from the listener at:
   0:  PUSH-EXTERNAL-VARIABLE Scratch:pickup
-- ** Scratch:pickup does not have a variable value
   (UninitializedVariable)
```

Variables can be assigned initial values using the assignment operator (`:=`), which is described in the section "Assignment" on page 38. A colon can also be used to assign an initial value to a variable.

```
global pickup := 52
global defenseBudget:3.18e11, moonsOfMars:2
```

Constants

You can make both local and global variables read-only by following either the `local` or `global` reserved word with the reserved word `constant`. Constants have values that cannot be changed by assignment, and as such must have values assigned to them at time of declaration:

```
global constant halfTon := 1000
global constant lightSpeed := 2.998e8 -- speed of light, meters/sec
```

Assignment

To assign a value to a variable, use the assignment operator (`:=`).

▼ *location* `:=` *expression*

On the lefthand side, *location* is a variable name or declaration. Most expressions in the ScriptX language can be used on the righthand side. For more information on the assignment expression, see the discussion on page 245 in Appendix A, "ScriptX Reference."

```
local var := 15
```

```
x := "string that goes in x"
```

Variables can be assigned to a value without being declared first, as in the previous example. A warning is printed if you do so. Variables that are assigned a value before they are declared are automatically declared as `global` variables, regardless of the scope in which they are used.

Variable assignment expressions evaluate to the value assigned; this allows you to nest or cascade assignments within other expressions:

```
z := 14 + (y := 3 - (x := 6))
⇨  11
```

In this example, the expression `x := 6` evaluates to `6`, and `y` evaluates to `–3`, allowing the value `11` to be assigned to `z`.

Variable assignment in ScriptX is reference assignment. Assigning one variable to another does not copy the object that variable contains. Both variables contain a reference to the same object. Consider the following assignment expressions:

```
myRect := new Rect x2:100 y2:100
⇨  [0, 0, 100, 100] as Rect
myOtherRect := myRect
⇨  [0, 0, 100, 100] as Rect
```

In the first assignment, a new instance of the `Rect` class is assigned to `myRect` (the `new` method is described in the next chapter). In the second assignment, `myRect` is then assigned to `myOtherRect`, which has the effect of setting `myOtherRect` to reference the same `Rect` object that `myRect` references.

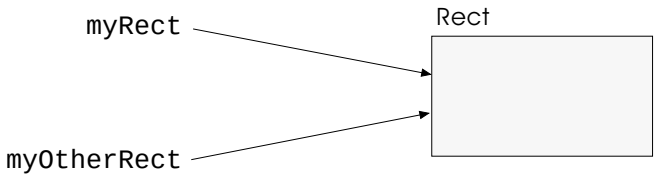


Figure 2-2: Pointer assignment

If you change any properties of `myRect`, you also change `myOtherRect`, since they both refer to the same object in memory.

```
myRect.y2 := 60
⇨  60
myOtherRect
⇨  [0, 0, 100, 60] as Rect
```

Variables and the Garbage Collector

As a ScriptX program runs, many objects are created. When there are no longer any references to those objects, the garbage collector automatically removes them from memory and reclaims the space they were using. This happens when all variables that reference an object have other values assigned to them, and there are no other references, such as by membership in an array or through another object's instance variables.

Each variable declaration uses only one word of memory, four bytes on a 32-bit processor, until an object is assigned to it. Objects are whatever size they are when instantiated. Once an object is assigned to a variable, ScriptX keeps that object in memory until there is no longer an active reference to it, including any other variables that reference that object. Of course, an object may itself contain references to other objects. Some objects, such as arrays, contain references to objects that are their members or elements.

Memory assigned to a local variable can be reclaimed at the end of the scope in which that variable is declared. Memory assigned to a global variable cannot be recovered unless the value of the variable is set to `undefined`, or the variable is unbound.

To allow the garbage collector to reclaim memory from unused objects, use local variables whenever possible. For global variables, set the value of a variable that references the object to `undefined` (or some other value) when the object is no longer needed. This allows memory to be reclaimed without undoing the declaration itself.

```
global garbage := "taking up space"
⇨  "taking up space"
garbage := undefined
⇨  undefined
garbage
⇨  undefined
```

In this example, `garbage` still exists as an empty slot in memory. This slot points to the `undefined` object, a system object.

ScriptX Operators and Simple Expressions

This section describes the basic operators ScriptX has for forming simple expressions. Table 2-3, "ScriptX operator precedence, highest to lowest," shows the order in which ScriptX expressions are evaluated.

ScriptX includes operators for the following:

- arithmetic
- equality
- magnitude
- logical expressions

Most ScriptX operators give the programmer the option of using white space both before and after the operator. Many programmers prefer to add white space, for readability.

```
z:=x*y -- less readable, but the compiler still understands it
z := x * y -- more readable
```

ScriptX is permissive about space around operators, with one exception. The subtraction operator (-) requires a trailing space. The minus sign without a trailing space is interpreted as the unary operator for negation.

```
3 - 5 -- with trailing space
⇨  -2
3-5 -- without trailing space, the compiler doesn't understand
⇨  -- ** "Ill-formed expression" (generalError)
3 +- 5 -- not very readable, but the compiler understands it
⇨  -2
3- 5 -- ugly, but the compiler understands it
⇨  -2
```

For a formal treatment of operators, see Appendix , "ScriptX Reference."

Operator Precedence

Table 2-3 summarizes the precedence of ScriptX operators and expressions. Operators higher up the table are evaluated first if they occur side-by-side in an expression. Operators in the same table cell have equal precedence and are evaluated from left to right, unless otherwise stated.

Table 2-3: ScriptX operator precedence, highest to lowest

Operator (s)	Notes
collection[*key*]	element access in collections, described on page 50
object.variable	class and instance variable access, described in Chapter 3
function arg arg . . . *function* (*arg, arg, . . .*)	function calls, described in Chapter 3
object `as` *class*	object coercion, described in Chapter 3
`-`	negation, described on page 42
`* /`	multiplication, division, described on page 42
`+ -`	addition, subtraction, described on page 42

`= == <> != !==`
`> < >= <=`
`contains`
	equality and magnitude, described on page 44.
	`contains` is used for collection membership, described on page 52

`not`	logical not, described on page 48.
`and`	logical and, described on page 48
`or`	logical or, described on page 48.
`in` `by`	for expressions association is from right to left.
`|`	pipe operator, described in Chapter 5

Table 2-3: ScriptX operator precedence, highest to lowest

Operator (s)	Notes

`repeat`
`guard`
`exit`
`for`
`return`
	loops and block control expressions are described in Chapter 4, while the `guard` expression is described in Chapter 8.

`&`	thread operator, described in Chapter 5
location `:=` *value*	assignment operator, described on page 38

You can change the order of evaluation by surrounding expressions with parentheses. ScriptX evaluates the innermost expressions first:

```
(((3 + 15) * 2) == 36)
(3 < 4) and (5 >= 2)
```

For additional information on operators and expression syntax, see Appendix , "ScriptX Reference."

Arithmetic

The ScriptX language includes the following infix operators for simple arithmetic:

- `+` addition
- `-` subtraction
- `*` multiplication
- `/` division

Each of these operators requires two operands. When two operands for an arithmetic expression are of different number classes, one operand may be promoted internally to accommodate the range of the other class.

The following examples demonstrate the promotion of `x`, which is an `ImmediateInteger` object. The `getClass` method is defined by `RootObject`, the root class from which all ScriptX classes inherit, so it is available to all objects.

```
global x:2, y:3.5, z:123.456789, w:9876543210, a, b, c
getClass x
⇨  ImmediateInteger
getClass y
⇨  ImmediateFloat
getClass z
⇨  Float
getClass w
⇨  LargeInteger
a := x + y
⇨  5.5
```

```
getClass a
⇨  ImmediateFloat
b := x + z
⇨  125.456789
getClass b
⇨  Float
c := x + w
⇨  9876543212
getClass c
⇨  LargeInteger
```

The result may be demoted for efficiency, if there is no loss of range.

```
global p:3.23, q:4.23, resultObject
getClass p
⇨  ImmediateFloat
getClass q
⇨  ImmediateFloat
resultObject := p - q
⇨  1
getClass resultObject
⇨  ImmediateInteger
```

String Arithmetic

You can use the addition and subtraction operators on strings.

String addition concatenates its operands into a single string:

```
"this" + "that"
⇨  "thisthat"
"Once upon " + "a time"
⇨  "Once upon a time"
```

String subtraction removes the first instance of the second string operand from the first:

```
"one two three" - "one"
⇨  " two three"
"banana" - "an"
⇨  "bana"
```

The following code illustrates an easy way to print out the value of a variable. Note that the parentheses are necessary and that the integer *x* must be coerced to a `String` object in order to be added to another string.

```
x := 10
print ("x: "+ x as String)
⇨  "x: 10"
```

String subtraction can make stripping the extension away from a file name easy:

```
myFilename := “Answers.doc”
myPrefix := myFilename - “.doc”
⇨  “Answers”
```

Note – In these examples, string addition and subtraction convert the original string, an instance of the class `StringConstant`, into an instance of the class `String` (its editable counterpart). `StringConstant` is a subclass of `String`. Operations that specifically test for an object’s class may not consider `String` and `StringConstant` to be the same, even though they are both strings.

Tests of Equality and Magnitude

ScriptX provides several operators for testing equality (whether two objects are equal in value), identity (whether two objects are exactly the same object), and magnitude (whether one object is greater or less than another). This section describes those operators.

In addition to the equality and comparison operators, ScriptX also defines a full set of object comparison functions that can be used in a more general way than many of the operators. These functions are based on four generic functions, which can be redefined in new classes and objects. ScriptX equality and comparison functions are described in Chapter 3, “Working with Objects,” and in the “Global Functions” chapter of the *ScriptX Class Reference*.The Comparison protocol is described in the “Object System Kernel” chapter of the *ScriptX Components Guide*.

Equality and Identity

ScriptX has two infix operators for testing object identity (`==` and `!==`), and three for testing equality (`=`, `!=`, and `<>`). All five operators return either `true` or `false`.

Identity tests determine whether two references refer to the same object. In an object-based system, two objects are the same if they are the same object in memory. ScriptX defines two identity operators:

`==`	tests whether two references are identical
`!==`	tests whether the references are not identical (the negation of `==`)

The following examples illustrate the use of the identity operators:

```
num1 := 3.14159265 -- a Float object
num2 := num1
num2 == num1
⇨  true
num1 !== num2
⇨  false
```

```
rect1 := new Rect x2:50 y2:50
rect2 := new Rect x2:50 y2:50
rect1 == rect2
⇨  false
```

In the first two identity tests, `num1` and `num2` are the same object in memory. In the third test, although the `Rect` objects in `rect1` and `rect2` were defined in the same way, and appear to be the same, they are different objects in memory, so the identity test returns `false`. (Chapter 3, "Working with Objects," describes how to define objects using the `new` method.)

Note – Two `ImmediateFloat` or `ImmediateInteger` objects that have the same value always appear to be the same object in identity tests. For more information on immediate objects, see the "Numerics" chapter in the *ScriptX Components Guide*.

Equality tests determine whether two objects have the same value. The objects may or may not be the same object in memory. (If they are the same object in memory, then they are equal.) Equality tests are only concerned with the data the two objects contain.

`=`	tests whether two objects are equal
`!=, <>`	tests whether the two objects are not equal (the negation of =). The not-equal operators are equivalent and can be used interchangeably

The following examples illustrate the use of equality operators:

```
num1 := 3.14159265 -- a Float object
num2 := num1
num2 = num1
⇨  true

rect1 := new Rect x2:50 y2:50
rect2 := new Rect x2:50 y2:50
rect1 = rect2
⇨  true
```

In the second example, the objects are different objects in memory (the identity test evaluated to `false`), but they have been defined the same way with the same parameters. Their values are the same, so the equality test returns `true`.

If two objects are not comparable, they are not equal.

```
num1 = rect1
⇨  false
```

Objects sometimes contain deeply embedded references to other objects. For example, each of the `Rect` objects defined above might be the boundary for another ScriptX object, such as a `TextPresenter` object. These `TextPresenter` objects, in turn, might be embedded in `PushButton` objects.

ScriptX allows each class to determine with which other classes it is comparable, and what the criteria are for equality between comparable objects. For example, a class can determine whether the comparison of two objects should be shallow or deep. (A deep comparison compares deeply embedded structures.) For more detail on comparison of objects, see Chapter 3, "Working with Objects."

Ordering

ScriptX has four infix magnitude operators that test whether one object is of greater "value" than another:

<	less than
>	greater than
<=	less than or equal to
>=	greater than or equal to

Magnitude operators only work if the operands they are given are comparable—that is, from the same or similar classes. If you try to compare objects from classes that are not comparable, ScriptX reports an error. Every object responds to the `isComparable` method, a method defined by `RootObject` and inherited by all classes in the system. The following test shows that `num1` and `rect1`, defined in the previous section, are not comparable.

```
isComparable num1 rect1
⇨  false
```

Each class determines which classes it is comparable with, by specializing the `isComparable` method. If a class is comparable with another, then it must determine how the comparison is made. For information on the Comparison protocol, see the "Object System Kernel" chapter of *ScriptX Components Guide*.

Magnitude expressions on comparable objects return `true` or `false`, as shown in the following examples:

```
3 < 4
⇨  true
3 <= 3
⇨  true
"apple" < "zucchini"
⇨  true
```

Equivalent Functions

Most of the ScriptX operators for arithmetic, equality, and magnitude described in the previous sections are shorthand for generic or global function calls. During compilation, those operators are converted into the appropriate function call with the appropriate arguments. For example, ScriptX translates

the expression `2 + 2` into the generic function call `sum 2 2`. The `sum` method is defined by the `Number` classes, allowing this expression to be evaluated correctly by any numbers.

Since ScriptX translates operators to generic function calls before evaluating an expression, you can extend the use of many of those operators to scripted classes that you create yourself. If you define each of the comparison methods that are part of the Comparison protocol in your class, ScriptX can then use instances of your class as arguments to the appropriate operator.

You can also call these functions yourself in a ScriptX expression, instead of using their operator counterparts, as shown in these examples:

```
sum 2 2
⇨  4
equal "asparagus" "celery"
⇨  false
```

Table 2-4 contains a list of ScriptX operators and their equivalent functions. These functions are all generic functions except for `eq` (the equivalent function for `==`) and `ne` (the equivalent function for `!==`), which are global rather than generic. Note that this means that `eq` and `ne` cannot be specialized, whereas the generic functions can. For information on defining functions, see Chapter 5, “Functions, Threads and Pipes.” For information on creating new classes and objects that use these functions, see .”

Table 2-4: Operators and their equivalent functions

Operator	Function	Meaning
+	sum	addition
-	sub	subtraction
*	mul	multiplication
/	quo	division
=	equal	equality
==	eq	identity
<>, !=	nequal	inequality
!==	ne	not identity
<	lt	less than
>	gt	greater than
<=	le	less than or equal to
>=	ge	greater than or equal to

ScriptX defines other arithmetic functions and tests of equality and magnitude that are not generic and cannot be specialized. See the “Global Functions” chapter of *ScriptX Class Reference*, and the class definitions of `Number` and `Integer` in the same volume.

Logical Operators

Expressions that return `true` or `false` can be grouped into logical expressions using `and`, `or`, and `not`.

▼ *expression* `and` *expression*

expression `or` *expression*

`not` *expression*

The `and` logical expression only evaluates to `true` if all of the sub-expressions (*expression* in the syntax) on either side of the `and` operator also evaluate to `true`. If any of the sub-expressions evaluates to `false`, the entire `and` expression evaluates to `false`. The `and` expression is short-circuiting; that is, the expression stops evaluating its sub-expressions after the first `false` expression is found.

The `or` logical expression evaluates to `true` if any of the sub-expressions on either side of the `or` evaluate to `true`. The `or` expression stops evaluating its sub-expressions once the first `true` expression is found. The `or` expression only evaluates to `false` if all of its sub-expressions also evaluate to `false`.

The `not` expression negates the value of *expression*. If *expression* evaluates to `true`, the `not` expression evaluates to `false`, and vice versa.

```
((3 + 2) < (4 + 5)) and 1 >= 1
⇨ true
"this" < "that" or "penny" > "pound"
⇨ false
not ("feet" = "feet")
⇨ false
```

Arrays and Keyed Linked Lists

The ScriptX language provides two literal constructs that directly represent collections of objects in scripts: an array literal and a keyed linked list literal. In addition, ScriptX provides two simple language constructs for accessing members and testing for membership in collections of all kinds.

`Array` and `KeyedLinkedList` are two of the many subclasses of `Collection` that can be used in a ScriptX program. Collections provide comprehensive behavior for groupings of objects. `Array` and `KeyedLinkedList` are prime representations of two categories of collection: those with explicit keys and those without. Collections are described in detail in Chapter 7, “Collections,” and in the “Collections” chapter of *ScriptX Components Guide*.

Array Literal

The array literal construct contains any number of expressions, separated by commas, surrounded by parentheses, and beginning with a hash sign (#):

▼ `#(`*expression*`,` *expression*`,` *expression*`,` *expression*`, . . . )`

When the array construct is evaluated, each individual expression is also evaluated, in order, so that the resulting array contains the result of each of those expressions. Arrays themselves are instances of the class `Array`. Here are some examples of array literals:

```
#(2, 4.2 + 5, "Fresno", 3 < 4, #(@doug, @jocelyn, @jim))
⇨  #(2, 9.19999694824219, "Fresno", true, #(@doug, @jocelyn, @jim))
#() -- empty array
⇨  #()
```

In some languages, all items in an array must be of the same type. The first example demonstrates that in ScriptX, members of an array can belong to different classes. The five members of this array belong to the classes `ImmediateInteger`, `ImmediateFloat`, `StringConstant`, `Boolean`, and `Array`, respectively.

Keyed Linked List Literal

The keyed linked list literal is similar to the array literal, except that the expressions within the parentheses are unordered key-value pairs. The keyed linked list construct evaluates to an instance of `KeyedLinkedList`.

▼ `#(`*key*`:`*value*`,` *key*`:`*value*`,` *key*`:`*value*`, . . . )`

In the keyed linked list literal, *key-value* pairs are separated by commas, with colons separating the *key* from the *value*. Like the array construct, all the expressions within the keyed list literal are evaluated before the instance of the `KeyedLinkedList` is created.

The *value* part of a key-value pair can be any expression. To specify most complex expressions as the *key* part of the pair, you must specify that expression inside parentheses (and the result of that expression becomes the key itself). The following expressions can be used without parentheses as the *key* part of the key-value pair:

- a number, string, or name (`45`, `"this or that"`, `@nameMe`)
- a variable name
- an array access expression (`myArray[1]`, described in “Access to Members of Collections”)
- a reference to a class or instance variable (`myRect.x1`, described in Chapter 3, “Working with Objects”)

Here are several examples of the use of keyed linked list literals:

```
#(@fruit: "apple", @vegetable: "carrot", @legume: "kidney bean")
#(:) -- empty keyed list
#(1:"money", 2:"show", 3:"ready", 4:"go")
```

```
#( #(1,2,3):@oneToThree, #(4,5,6):@fourToSix, \
   #(7,8,9,10):@sevenToTen) )

x := 1; y := 4
#( (x < y):@smallX, (x > y):@smallY )
⇨  #(true:@smallX, false:@smallY)
```

Literals and Other Collections

You can use the array and keyed list literals, together with coercion, to represent any kind of collection "literally" in a script.

```
#("Bornstein","Agostino","Reiman","Jacobi") as SortedArray
⇨  #("Agostino", "Bornstein", "Jacobi", "Reiman") as SortedArray
```

Coercion is discussed in the section "Coercing Objects to Other Classes" on page 67, and again in the section "Coercing Between Collection Classes" on page 160.

Access to Members of Collections

`Array` and `KeyedLinkedList` are two of the `Collection` classes from the ScriptX core classes. The `Collection` classes implement a common set of methods for operations on the items they store.

The ScriptX language provides access to members of collections with the element access expression. The element access expression provides a reference to an element by its position or key.

▼ *collection*`[`*position*`]`
collection`[`*key*`]`

In the element access expression, *collection* must evaluate to an instance of `Collection` such as `Array` or `KeyedLinkedList`. *position* is an ordinal position within the array, and *key* is a key in a key-value pair.

Like the keys in the keyed linked list literal, you can specify complex expressions as the *collection* part of the element access expression by surrounding them with parentheses. You can also specify that the following expressions are the *array* or *keyedLinkedList*:

- a variable name
- an actual array or keyed linked list (`#(1, 2, 3)`)
- another array access expression (`myArray[1]`)
- a reference to a class or instance variable (`myRect.x1`, described in Chapter 3, "Working with Objects")
- an anonymous function with no arguments (described in Chapter 5, "Functions, Threads and Pipes")

- `nextMethod` (described in Chapter 6, “Defining Classes and Objects)

If the given *key* is not found, or the *position* is out of range, the element access expression returns the `empty` object. `empty` is used specifically in the collection classes to mean “element not found.”

```
myMenu := #(@breakfast:"coffee", @lunch:"burrito",@dinner:"pizza")
⇨  #(@breakfast:"coffee", @lunch:"burrito", @dinner:"pizza")
myMenu[@midnightSnack]
⇨  empty
```

The element access expression is translated by the compiler into a call to the generic function `getOne` . The `getOne` generic function is implemented by all of the collection classes, and as such, the element access expression is also available to any collection. See Chapter 7, “Collections,” for more information about the collection classes.

These examples show how to use the element access expression:

```
array1 := #( "un", "deux", "trois" )
array1[1]
⇨  "un"
array1[3]
⇨  "trois"
array1[0]
⇨  empty
#( "once", "twice", "thrice", "four times")[4]
⇨  "four times"
keyedList1 := #("snake":"cobra","bird":"sparrow","fish":"guppy")
keyedList1["bird"]
⇨  "sparrow"
array2 := #(array1)
array2[1]
⇨  #( "un", "deux", "trois" )
array2[1][2]
⇨  "deux"
(if array1.size > 1 then array1 else #(undefined))[1]
⇨  "un"
```

Setting Items

You can also use the element access expression, as described in the previous section, to change the value of the item at that key or position.

▼ *collection*[*position*] := *value*
collection[*key*] := *value*

As in the element access expression, *collection* is a factor or expression within parentheses that results in a collection such as an array or keyed linked list. The *position* represents an ordinal position within the collection, while the *key* represents a key associated with one item. The *value* is either the value for the given key or the value to be placed at that position in the array.

The element access expression, when used with an assignment operator, is compiled into a call to the generic function `setOne`. The `setOne` generic is implemented by all collection classes, and as such, the element access construct is available to any collection.

These examples show how to change or add an element to a collection such as an array using the element access and assignment expressions:

```
global houseplant := #(@palm) -- create an array
⇨  #(@palm)
houseplant[2] := @africanViolet -- add a second element
⇨  @africanViolet
housePlant
⇨  #(@palm,@africanViolet)
houseplant[3] := @begonia -- add a third element
⇨  @begonia
housePlant
⇨  #(@palm,@africanViolet,@begonia)
houseplant[1] := @fern -- set a new value for first element
⇨  @fern
housePlant
⇨  #(@fern,@africanViolet,@begonia)
```

Element Access and Other Collections

Since the element access expression is a shorthand for `getOne` or `setOne`, it can be used with any class that implements these generic functions. In the ScriptX core classes, many of the media classes are implemented as collections, including windows, menus, documents, animation lists, and even title containers. The element access expression is used with all of these classes, and with scripted collection classes as well.

Testing for the Presence of an Element

To see if an object is in an array or keyed list, use `contains`:

▼ *collection* `contains` *thing*

n this syntax, *collection* is an expression that evaluates to a `Collection` object, and *thing* is the object in the collection for which you are testing. The `contains` expression returns either `true` or `false`, depending on whether or not the element is present in the collection.

The `contains` construct is compiled into a call to the generic function `isMember` before the expression is evaluated. The `isMember` generic function is implemented by all collection classes, and as such, the `contains` construct is available with any collection.

Here are some examples of how `contains` is used in scripts:

```
x := #("one", "three", "five")
x contains "one"
⇨  true
x contains "two"
⇨  false
#(1:"one",2:"two",3:"three") contains "two"
⇨  true
```

Compound Expressions

Compound expressions, sometimes called block expressions or just blocks, are simply a series of complete expressions surrounded by parentheses. Those expressions can be on separate lines, on the same line separated by semicolons, or in any combination, just like normal expressions.

▼ `(`

expression

expression

`. . .`

`)`

Compound expressions allow you to control the order in which an expression is evaluated. A compound expression can also be used to group several expressions into what appears, from the point of view of surrounding expressions, as a single expression. Thus, you can use a compound expression anywhere a single expression would work. The result of evaluating a compound expression is the value of the *last* expression in the block, unless a specific exit or return value has been specified using a block control expression. See “Conditionals and Loops” on page 81, for information on using compound expressions with block control expressions such as `exit` and `continue`.

A compound expression yields an object, just like a regular expression. A compound expression can be used in place of most ScriptX expressions.

```
-- globals cannot be explicitly declared in a block
global m:0, n:1
```

```
-- a compound expression, because it is enclosed in parentheses
t := (if m < n then m else n)
⇨  0

(
x := "string1"
y := "string2"
print (x + y)
)
"string1string2"
⇨  OK
```

If the final statement in a compound expression is a local declaration, it returns `undefined` rather that the value of the local declaration. Note the difference between the first and second expressions.

```
global y := (local x := 10)
⇨  undefined
global y := (x := 10)
⇨  -- ** Warning: Undeclared global x
   10
```

Compound Expressions and Variable Scope

Compound expressions define a new local variable scope for the expressions within them. Local variables declared in the block are only available to the expressions within that block that occur after the declaration.

```
global z, x := 15
(
   local q:= 10
   z := x + q
)
z -- evaluate z
⇨  25
q-- evaluate q
-- ** Warning: Undeclared global q
```

Note that `z` continued to exist outside the block because it was declared globally. The variable `q`, however, was declared locally, and so is only available within the block expression in which it was defined. In the example that follows, note that the local definition within the block overrides the global definition outside the block.

```
global var1 := "string"
(
   local var1 := "a different string"
   var1   -- the block returns the value of this because it's
          -- the last expression in the block
)
⇨  "a different string"
var1
⇨  "string"
```

The first `var1` evaluates to the value of the local assignment, as declared in the block. The second `var1` , which occurs after the end of the block, returns the value of the global `var1` variable.

When ScriptX encounters a variable reference, it first checks the local scope for the value of that variable, and then each surrounding variable scope. This causes local variables to “hide” the values of global variables and local variables of the same name, including variables that may contain substrate classes, objects, and functions.

The following example shows that the global variable is visible inside the block until it is overriden by a local variable with the same name. After the block’s scope has ended, the local variable no longer exists.

```
global a, b:2, c:3
(
    a := b
    local b:5
    c := b
)
⇨  5
a -- evaluate a
⇨  2
b -- evaluate b
⇨  2
c -- evaluate c
⇨  5
```

C H A P T E R

Working with Objects

3

ScriptX is an object-oriented language that operates on a hierarchy of classes and objects. The language makes no distinction between the core set of ScriptX classes and the classes and objects that you define yourself. Creating new instances, accessing and changing instance variables, and calling methods are operations which are used with all classes. This chapter contains an overview of the basic operations on classes and objects in ScriptX.

Creating New Objects

In ScriptX, all information you work with in the process of writing a program is contained in objects. Objects can be created in many ways in ScriptX. Objects are created as literals, as the result of a function call to other objects, as a side effect of those functions, or explicitly by you in a script. This latter case is the focus of this section.

ScriptX provides three primary ways in which you can directly create new objects:

- using a literal construct
- using the `new` generic function
- using the `object` expression

A Note On Memory Allocation

Memory allocation in ScriptX is automatic. When you create a new object, the appropriate amount of memory is allocated for that object in the system.

Literals

Literals are special constructs in the language that allow you to represent many common objects directly in a script. Numbers, strings, and array constructs, as described in the previous chapter, are all literals. Here are some examples:

- The literal `4` yields an `ImmediateInteger` object with the value 4.
- The literal `"cat"` yields a `StringConstant` object with the value `"cat"`.
- The literal `#(1,2,3,4)` yields an `Array` object that contains four `ImmediateInteger` objects with the values 1, 2, 3, and 4.

new

Only a few select classes in the ScriptX core classes have literal representations in the ScriptX language. The most common way to create an instance of any class in a script is by using the `new` generic function:

- `new` *class* `[` *key*`:`*value* *key*`:`*value* `. . . ]`

Function calls are discussed later in this chapter, beginning on page 65. The calling sequence for the generic function `new` has three components:

- `new`, the name of a generic function
- a name, the name of the class (or an expression that returns the name)
- a set of key-value pairs, separated by spaces, called *keyword arguments.*

Keyword arguments specify initial parameters of the new object, for example, its initial state or size. Often, these parameters set initial values for instance variables. The `new` method creates the instance and then passes those keyword arguments internally to the object's `init` generic function. The keyword arguments that a class allows are defined by that class's `init` method, and by all of its superclasses which implement an `init` method.

Each class definition in the *ScriptX Class Reference* specifies the keyword arguments that are used by that class. Scripted classes also define keyword arguments, and they can specialize any behavior they inherit. For example, a scripted class can supply different default values for a keyword argument it inherits. For more information on specializing the `init` method, see page 136.

Keyword arguments can be specified in any order, and many of them are optional. Any optional keywords you do not specify in calling `new` are given default values by the `init` method when the object is initialized.

The `init` method determines which keyword arguments are optional and what the default values are. Each `init` method applies the `init` method defined by its superclasses. Superclasses may define other keyword arguments. The `init` method may specialize or override any behavior it inherits from its superclasses. In this way, a class's `init` method, together with that of its superclasses, determines which keyword arguments are defined, which are required or optional, and what the default values are.

Many classes define no keyword arguments:

```
myStencil := new Path
new LinkedList
```

If some keywords are optional, you can specify values for only the ones you are concerned with, and allow the others to be initialized to defaults:

```
myRect2 := new Rect x2:500 y2:500 -- x1 and y1 are set by default
bigArray := new Array initialSize:150 -- growable is set by default
```

Because `new` evaluates to a new instance of the given class, you can nest expressions using new within other expressions:

```
rectangular := new TwoDShape stencil:(new Rect x2:500 y2:500)
```

The `new Rect` expression evaluates to an instance of the `Rect` class, which is then used as the value for the `stencil` keyword argument to the `TwoDShape` class. By nesting `new`, you write more concise code and avoid extra variables.

Note that the `new` generic is not a part of the ScriptX syntax. It is actually a generic function, a generic you call on class objects. Recall that in the ScriptX object system, classes are themselves objects. You can call the `new` generic function on any concrete class to create an instance of that class. (Concrete and abstract classes are defined on page 25.)

In most cases, ScriptX reports an exception if you attempt to call `new` on an abstract class. However, in some special cases the Kaleida Media Player creates a new instance of a concrete subclass of that abstract class automatically. (Examples include `KeyboardDevice` and `MouseDevice`.)

For more information on `new`, see the "Overview" chapter of the *ScriptX Class Reference*. For information on the object system, see the "Object System Kernel" chapter of *ScriptX Components Guide*.

Using the object Expression

An alternative to the `new` generic for creating new instances of a class is to use the `object` definition construct. The `object` construct allows you to give initial values to any instance variables, including those that are not initialized by keyword arguments.

Note – The complete form of the `object` definition expression allows you to fully specialize a new object. You can add new instance variables, define new methods, and even create an object that is an instance of a mixture of classes. This chapter describes only some aspects of this expression. For more information, see Chapter 6, "Defining Classes and Objects."

The simplest form the `object` expression creates a new instance of a class. Just as with `new`, the `object` expression allows you to specify any available or required keyword arguments:

▼ `object [` *variableName* `] (` *classes* `)`
`[` *keyword*`:`*value*`, ` *keyword*`:`*value*`, . . . ]`
`end`

In the `object` expression, *classes* is the class or list of classes this object is an instance of. It can be a list of several classes, separated by commas. *variableName* is an optional variable to which you can assign this object. The *keyword*`:`*value* pairs are any keyword arguments. As with the generic function `new`, you can specify keyword arguments in any order. You can specify keyword arguments on the same line, separated by commas, on separate lines, or in any combination:

```
object (Rect) x1:35, y1:35, x2:50, y2:50 end

object (Rect)
    x1:35, y1:35
    x2:50, y2:50
end
```

```
object (Array)
    initialSize:100
    growSize:10
end
```

The `object` expression evaluates to an instance of the new class, which allows it to be nested within other expressions. You can also assign the new object to a variable, using one of two forms: you can specify a *variableName* in the first line of the `object` expression, or you can put the variable name on the left side of an assignment statement, with the `object` expression on the right.

```
object myNewRect (Rect) x2:100, y2:100 end -- constant variable
myOtherNewRect := object (Rect) x2:100, y2:100 end -- not constant
```

The difference between these two forms is that the first form, in which you include the variable name inside the `object` expression, causes the variable `myNewRect` to be declared `constant`. This prevents you from assigning anything else to that variable once the object has been created. You can, however, use the `object` expression multiple times to redefine the object assigned to a variable.

Initializing Instance Variables

You can use the `object` expression to initialize the values of any instance variables defined by that object. In many cases, the object may use keyword arguments to give many of its instance variables initial values. However, if there are other instance variables defined by the object that are not set directly by keyword arguments, you can specify initial values for those instance variables as well.

▼ `object [` *variableName* `](`*classes*`)`

```
    [ keyword:value, keyword:value, . . . ]
    settings
        variable:value
        . . .
end
```

The `settings` reserved word, which should always occur just after any keyword arguments in an `object` expression, indicates that the following instance variables should be initially set. You can then specify any number of *variable*`:`*value* pairs. Just as with the keyword arguments, these *variable*`:`*value* pairs can be specified on a single line separated by commas, on separate lines, or in any combination.

For example, the class `TwoDShape` defines instance variables for its `fill` and `stroke` values as well as variables for the `x` and `y` coordinates of the shape. The `init` method for `TwoDShape` defines keyword arguments for `fill` and `stroke`, but not for the coordinates. This example uses keyword arguments to set the fill and stroke for the shape, and then `settings` to initialize the values of `x` and `y`.

```
rect1 := object (TwoDShape)
    fill:(new Brush color:redColor) -- keyword argument
    stroke:(new Brush color:greenColor) -- keyword argument
    settings
        x:50, y:50
end
```

Instance Variable Access

Instance variables define the attributes or properties of an object. For example, instance variables may be used to hold information such as coordinate position (for graphic objects), tempo and number of channels (for audio objects), or form of compression (for video objects).

Many instance variables are assigned initial values through keyword arguments to the `new` generic or the `object` definition construct. Once the object has been created, you can use the language constructs described in these sections to get and set the values of those instance variables.

Variable Access Expression

You query the value of an instance variable by using the dot syntax:

▼ *object.instanceVariable*

object is an object (or an expression that yields an object), and *instanceVariable* is the name of the instance variable.

In the following example, a new instance of the class `Line` is created. Instances of `Line` have four instance variables for the coordinates of their starting and ending points: `x1`, `y1` and `x2`, `y2`. This example uses keyword arguments to set those variables, and then it accesses those values:

```
myLine := new Line x1:10 y1:10 x2:65 y2:65
myLine.x1
⇨  10
-- test: is the endpoint higher than the startpoint?
myLine.y2 < myLine.y1
⇨  false
```

If an object's instance variable holds another object that in turn has its own instance variables, you can "chain" references to those variables:

```
object myRectangularShape (TwoDShape)
    stroke:blackBrush, boundary:(new Rect x2:100 y2:100)
end
myRectangularShape.boundary.x2
⇨  100
```

Note that in order to specify most complex expressions as the *object* part of the instance variable construct, you must specify that expression inside parentheses so that it is evaluated first. The only expressions that can be used without parentheses as the *object* part of an instance variable expression are:

- a variable name
- An array or keyed linked list literal (`#(1, 2, 3)`)
- an array access expression (`myArray[1]`)
- a reference to another class or instance variable (`myRect.x1`)
- an anonymous function with no arguments, resulting in an object (described in Chapter , "Functions, Threads and Pipes")
- `nextMethod` (described in Chapter , "Defining Classes and Objects)

Instance Variable Access—An Efficiency Note

Since all access to instance variables is through method calls (see "Setters, Getters, and Real and Virtual Variables" on page 145) it is best to avoid repeated calls to an object nested several levels deep.

Using an example from Chapter 1, we got the address of Odan's owner using the construct `odan.owner.address`. This involves two function calls, one for each "`.`". Let's say you want to send out a mailing to every dog owner living in the same city as Odan's owner, so you need to compare each address in your list to Odan's address. You could greatly improve efficiency by assigning Odan's address to a variable and then using that variable for comparison, thus accessing the address just once instead of every time you make a comparison. The following code fragment, though not complete, illustrates the point:

```
address := odan.owner.address -- assign to a variable
for i := 1 to addressList.size do -- create a for loop to iterate over
                                  -- an array of addresses
   if addressList[i] = address do -- compare each item to variable
      ... -- your function for sending mail
```

A further efficiency could be achieved by using the `Collection` method `forEach` in place of the `for` loop, but the `for` loop is easier to understand in an example at this point.

Changing the Values of Instance Variables

Use the assignment operator (`:=`) to set the value of an instance variable:

▼ *object.instanceVariable := value*

As before, `object` is an object or an expression that yields an object, *instanceVariable* is the name of the instance variable, and *value* is an expression that sets the new value.

```
myLine := new Line x1:10 y1:10 x2:65 y2:65
-- Change the line's starting point to be 0,0
myLine.x1 := 0
myLine.y1 := 0
-- Change the line's endpoint to be 10,10
myLine.x2 := 10
myLine.y2 := 10
```

Most instance variables are read-write. That is, you can both query their values and change them. A few are read-only. If you try to change them, an error is reported. Whether an instance variable is read-write or read-only is defined by the class.

Class Variables

Class variables are similar to instance variables, except that they are variables defined by an entire class, rather than by a single instance of the class. Think of class variables as instance variables for the class itself. To query or change the value of a class variable, use the same ScriptX construct as for instance variables, specifying the class (or an expression that yields a class) instead of an object:

class`.`*varname*

For example, `interests` is a class variable defined by the `Event` class. The result of entering the following code is a collection of the interests that have been posted in `myEvent`.

```
myEvent.interests
```

Calling Functions

There are two kinds of functions in ScriptX: regular functions and generic functions.

Regular functions are just like functions in other languages; they have a single interface (its name and arguments), and a single implementation (the function definition), as shown in Figure 3-1.

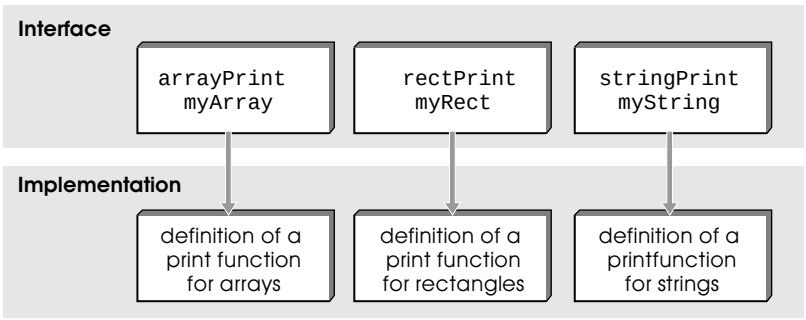


Figure 3-1: Regular functions

Generic functions are used to invoke methods that are defined by classes and objects. Unlike regular functions, generic functions have a single interface, but they can select from many different implementations of that function

(methods) based on the object specified as the first argument. When you call a generic function, it selects an appropriate method to invoke and transfers control to that method, passing along any arguments you specified (Figure 3-2). For a definition of generic functions, see the discussion on page 19 and page 12.

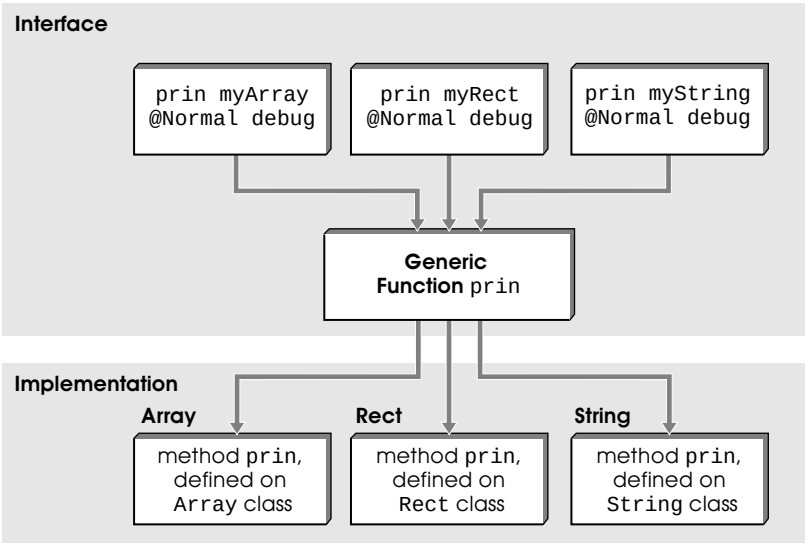


Figure 3-2: Generic functions

In Figure 3-2, `prin` is a generic function that takes three arguments, an object to print, an argument that determines how to print it, and a stream object (`prin` is described in detail on page 75). Since each class has a different printable representation, there are separate methods in each class that implement `prin`. The generic function `prin` uses the class of the object (`myArray`, `myRect`, and `myString`, instances of `Array`, `Rect`, and `String`) to choose the appropriate implementation.

Both regular and generic functions are called identically in scripts. There are three forms of function calls:

▼ *functionName positionalArgs keywordArgs*
functionName`(`*positionalArgs*`,` *keywordArgs*`)`
functionName`()`

The first two forms are used to call regular or generic functions with arguments. Both forms are equivalent and are described below. The third form is used to call regular functions without any arguments This third form cannot be used with generic functions because they require at least one argument—the object the generic function is to operate on. The empty argument list () is used to indicate to ScriptX that this is a function call and not a variable reference.

In the first two forms, *positionalArgs* are any required positional arguments for that function, and *keywordArgs* are any keyword arguments that the function or generic function may have defined. Keyword arguments are *key*`:`*value* pairs,

where the key and the value are separated by a colon. Positional arguments, if any, must appear before any keyword arguments and must be specified in the correct order. Keyword arguments, if any are defined, can appear in any order.

The second form for calling functions is one that may seem more familiar to those who are accustomed to other programming languages. In this form, all the arguments, positional or keyword, are separated by commas and surrounded by parentheses.

The following examples illustrate calling both regular and generic functions. These examples use the following functions:

- the `new` generic function, as described on page 59
- the `append` generic function, which appends an element to the end of a collection such as an array
- the `getNth` generic function, which retrieves the element at the given position in a collection such as an array
- the `getClassName` generic function, which returns a string containing the name of the class of which this object is an instance
- the regular functions `helloWorld` (which takes no arguments, but prints a message to the debugging stream), and `HamCheeseOnRye`, which, when called with the appropriate arguments, returns a congratulatory message. These functions do not exist as part of the the core ScriptX system and are shown for syntactic purposes only

```
cheez := #("swiss", "cheddar", "roquefort")
append cheez "havarti" -- returns the key of the item added to cheez
⇨  4
getNth cheez 3
⇨  "roquefort"
getNth(cheez, 4)
⇨  "havarti"
getClassName cheez
⇨  "Array"
helloWorld() -- function with no args
⇨  "Hi mom!"
hamCheeseOnRye meat:@liverwurst cheese:@swiss bread:@rye
⇨  --** "@liverwurst is not an appropriate sandwich ingredient"
hamCheeseOnRye(meat:@ham, cheese:@swiss, bread:@wholewheat)
⇨  --** "@wholewheat is not an appropriate sandwich ingredient"
hamCheeseOnRye meat:@ham cheese:@swiss bread:@rye
⇨  "a most excellent sandwich."
```

Coercing Objects to Other Classes

Object coercion refers to the ability to convert an instance of one class into an instance of another. For example, you might coerce an integer (an instance of `ImmediateInteger`) into its string representation (an instance of `String`, `StringConstant`, or `Text`). In ScriptX, however, object coercion does not

change the original object, as the term coerce might imply. Instead, coercion in ScriptX creates a new instance of the target class, and uses the original object for deriving the contents of that new instance.

The `as` expression coerces an object to another class:

▼ *object* `as` *class*

In this expression, *object* is the object you want to coerce, and *class* is the class you want the object to be coerced to. The types of classes a given object can be coerced to is determined by the destination class, or if the destination class can't coerce, the object's own class. If neither attempt at coercion succeeds, an exception is reported.

```
123 as Float
⇨  123.0
123 as String
⇨  "123"
123 -- the original object is still 123
⇨  123

myList := new LinkedList
myList as Fixed -- this attempt to coerce reports an exception
-- ** Cannot coerce #() as LinkedList into a Fixed. (CantCoerce)
```

When an object is coerced to another class, some of the information the original object contained can be lost. For example, when you convert an instance of a number class with more precision to one with less precision, you may lose the extra information that the additional precision provided:

```
43.5 as ImmediateInteger
⇨  43
```

In this example, the original object (an instance of `ImmediateFloat`) was coerced to the class `ImmediateInteger`, which truncated the fractional part.

Table 3-1 provides some simple examples of how objects can be coerced.

Table 3-1: Class coercion examples

Original Class	Sample Expression	Result
`ImmediateInteger`	`123 as Float`	`123.0`
`ImmediateInteger`	`123 as LargeInteger`	`123`
`ImmediateInteger`	`123 as String`	`"123"`
`Float`	`123.5 as ImmediateInteger`	`123`
`String`	`"345" as ImmediateInteger`	`345`
`String`	`"345" as Fixed`	`345.0`

Note that in order to specify most complex expressions on either side of the `as` construct you must specify those expressions inside parentheses so that they are evaluated first. The only expressions that can be used without parentheses in an `as` expression are, where appropriate:

- a number, string, or name literal (`45`, `"this or that"`, `@nameMe`)
- A variable name
- an array or keyed linked list literal (`#(1, 2, 3)`)
- an array access expression (`myArray[1]`)
- a reference to a class or instance variable (`myRect.x1`)
- an anonymous function with no arguments (described in Chapter , “Functions, Threads and Pipes”)
- `nextMethod` (described in Chapter , “Defining Classes and Objects”)

A coercion expression is actually compiled as a call to the global function `coerce`. A script can call `coerce` as an alternative to the `as` operator. For example, the following expressions coerce a `StringConstant` object, that is a string literal, to a `Text` object, a kind of string that can be edited. These two coercion expressions are equivalent:

```
"When in the course of human events" as Text
coerce "When in the course of human events" Text
```

The `coerce` function, in turn, calls one of two underlying generic functions. The Coercion protocol comprises a set of generic functions that allow scripted classes to be coerced to and from other classes, including substrate classes. Each class can implement its own version of `morph` or `newFrom` to define which classes of objects it can be coerced to, and to perform the coercion operation. By specializing these generics, scripted classes can be coerced freely using the `as` construct in the ScriptX language. If an object cannot be converted, it reports the `cantCoerce` exception. For more information on coercion, see the “Object System Kernel” chapter of the *ScriptX Components Guide*.

Comparing Objects

Chapter , “ScriptX Building Blocks,” describes a set of operators that are used to compare objects for equality, identity, and magnitude. That chapter also notes that those operators are shorthand for a set of equivalent global functions, which are summarized on page 48. This section summarizes several additional ScriptX comparison functions.

These comparison functions are part of a suite of object comparison functions that ScriptX provides. Because they are based on generic functions, you can override the generics in the classes and objects you define to extend the object Comparison protocol.

Chapter , “Defining Classes and Objects,” describes how to specialize your classes to allow for comparison of objects.

For more information on the Comparison protocol, see the "Object System Kernel" chapter of the *ScriptX Components Guide*. For complete definitions of ScriptX comparison functions, see the "Global Functions" chapter of the *ScriptX Class Reference*.

isComparable and cmp

Two functions that are useful for comparing objects are the generic function `isComparable` and the `cmp` global function. The `isComparable` generic function, defined by `RootObject`, is one of the four generics that are the basis of the Comparison protocol. `isComparable` returns `true` or `false`, depending on whether its arguments are comparable. If `isComparable` returns `true`, then comparison operators and comparison functions (described in Chapter , "ScriptX Building Blocks") can be used on the tested objects without causing an exception.

Each class can override `isComparable` to indicate which classes it can be compared with. The default implementation of `isComparable` simply states that two objects are comparable if they are of the same class.

```
isComparable 1 2
⇨  true
isComparable 1 "banana"
⇨  false
```

The `cmp` global function is a general purpose comparison function. When it is called with two arguments (which must be comparable), it uses the `equal` (`=`) and `lt` (`<`) generic functions to return one of three values:

- `@same` if the arguments have the same values (`equal` evaluates to `true`)
- `@before` if the first argument is less than the second (`lt` returns `true`)
- `@after` if the first argument is greater than the second (`lt` returns `false`)

The following examples demonstrate the `cmp` function:

```
cmp 1 1
⇨  @same
cmp 1 4
⇨  @before
cmp 4 1
⇨  @after
```

Universal Comparison

In addition to simple comparison tests, ScriptX also defines a set of global functions for determining universal equality and magnitude. The universal comparison functions provide an ordering for all objects, even those that may not be comparable. These functions can be used to sort any objects.

Universal comparison functions can compare two objects of different classes. If two objects are of the same class, and that class does not implement a method for `localLT`, then `ult` can still report an exception. Generally, all objects for which comparison is meaningful, such as numbers and strings, implement a method for `localLT`. To insure that the universal comparison functions work for a given object, use `canObjectDo` to query the object to see whether it implements `localLT`. For information on `canObjectDo`, see page 74.

The seven universal comparison functions are `ueq`, `une`, `ult`, `ugt`, `uge`, `ule`, and `ucmp`.

The functions `ueq` (universal equal) and `une` (universal not equal) test for object equality. The `ueq` function returns `true` only if:

- The classes of its arguments are the same, *and*
- the values of its arguments are equal

Conversely, `une` returns `true` if either the classes of its arguments are not the same *or* the values of its arguments are not the same.

```
arg1 := 1.9 -- an instance of ImmediateFloat
arg2 := 1.9 as Float
arg1 = arg2 -- equivalent to the equal global function
⇨  true
ueq arg1 arg2 -- test for universal equality
⇨  false
```

In this example, `arg1` is assigned the number 1.9 (an instance of class `ImmediateFloat`), and `arg2` is assigned the same value, coerced to the `Float` class. Because the *values* of these numbers are the same, the equality test returns `true`. However, because the *classes* of these objects are different, `ueq` returns `false`.

The global functions `ugt`, `ult`, `uge`, and `ule` are used for universal comparisons of magnitude. (They are the universal counterparts of the `>` (`gt`), `<` (`lt`), `>=` (`ge`), and `<=` (`le`) operations, respectively). Each of these functions returns `true` if one of the following rules is true:

- If the classes of the arguments are the same, then the methods return the same result that the non-universal comparison would have returned.

  For example, for the expression `ult 1 2`, the classes are the same (`ImmediateInteger`), and the value `1` is indeed less than `2`, so the function returns `true`. Since `lt 1 2` returns `true`, `ult 1 2` returns `true`.

- If the classes are different, then the names of the classes are compared alphabetically.

  For example, for the expression `ult 1 "this"`, the two classes (`ImmediateInteger` and `String`) are different, so the names of the classes are compared. Because `ImmediateInteger` is alphabetically "less than" `String` (that is, "I" comes before "S"), the expression returns `true`.

Universal comparison functions exist to provide a total ordering, which is not necessarily a meaningful ordering, like that provided by their non-universal counterparts. Sorting objects alphabetically by their class name may not be very meaningful, but it does allow objects of any class to be sorted together.

Finally, the `ucmp` function is the universal equivalent of the `cmp` function. `ucmp` returns one of the following values, based on the following rules:

- `@same` if the `ueq` function returns `true` for the arguments.
- `@before` if the `ult` function returns `true` for the arguments.
- `@after` if the `ueq` and `ult` functions both return `false` for the arguments.

Finding Information About Classes and Objects

This section describes generic functions that are useful for finding information about a class or object, such as what its superclasses are, whether it defines a particular generic function, or the class of a given object. These methods are defined by the classes `RootObject`, `RootClass`, and `Behavior`, and as such, they are available to all classes and objects. For more information, see the "Object System Kernel" chapter of the *ScriptX Components Guide*, and the definitions of `RootObject` and `Behavior` in the *ScriptX Class Reference*.

Note – `RootObject`, `Behavior`, and `RootClass` define several generic functions that expose API that is private, and not considered part of the ScriptX Language and Class Library. Any classes, objects, instance variables, or methods not documented in the Kaleida Technical Reference Series, or in associated release notes, are not supported by Kaleida. Since such API is likely to change with future versions of ScriptX, using it in a title or tool may result in future incompatibilities with Kaleida products.

Some of the generic functions discussed here return a very large collection of objects, generally in an array. By default, only the first ten elements of a collection are printed to a stream. You might want to read ahead to section "Output" which begins on page 74 to explore options for printing a complete listing.

◆ `getSubs` *class*
`getSupers` *class*

The `getSubs` and `getSupers` generic functions return an array of all the subclasses or superclasses of *class*, respectively. For `getSupers`, this includes all the classes up to `RootObject`. The generic function `getSupers` returns an array in exact inheritance precedence order.

- `getDirectSubs` *class*
  `getDirectSupers` *class*

  In contrast with `getSubs` and `getSupers`, the generics `getDirectSubs` and `getDirectSupers` return an array of immediate (direct) subclasses or superclasses of *class*, respectively.

- `isDirectSub` *class otherClass*
  `isSub` *class otherClass*
  `isMemberOf` *class object*

  The generic functions `isDirectSub`, `isSub`, and `isMemberOf` test the relationship between two classes, or between a class and an object. The `isDirectSub` generic returns `true` if *class* is a direct subclass of the class *otherClass*; the `isSub` generic returns `true` if *class* inherits from the class *otherClass*. The `isMemberOf` generic returns `true` if the given *object* is an instance of *class*.

- `allInstances` *class*

  The generic function `allInstances` returns an `Array` object containing a list of all instances of *class* that currently exist in the system. Note that instances of a subclass are considered instances of the given class.

- `getClassName` *object*
  `getClass` *object*

  The `getClassName` function returns a string containing the name of the class, of which *object* is an instance. The `getClass` function returns the class object of which the *object* is an instance. Comparisons using `getClass` are more efficient than they would be with `getClassName`, since `getClassName` requires the system to create and compare strings. The `getClass` generic function references the object's own class object directly.

- `isAKindOf` *object class*

  The `isAKindOf` generic function tests the relationship between *object* and *class*. If *object* is an instance of the given class, or if *object* is an instance of a class that inherits from this class, `isAKindOf` returns `true`.

- `getDirectGenerics` *object*
  `getAllGenerics` *object*

  The `getDirectGenerics` function returns an array containing all the generic functions that are implemented by methods that are defined directly by the class that *object* is an instance of. The `getAllGenerics` function returns all the generic functions available to *object*, whether they are implemented in the class itself or in any of its superclasses.

- `canObjectDo` *object generic*

  The `canObjectDo` function tests whether *object* has an implementation (a method) for the generic function *generic.* This generic functions is often called on class objects.

- `ivNames` *classOrObject*
  `allIvNames` *classOrObject*

  The `ivNames` function returns a sorted array of instance variables defined only on the class or object. The `allIvNames` function returns a sorted array of all instance variables available to the class or object, including instance variables inherited from all the superclasses of this class. Both these functions return only those instance variables that occupy real places in memory, which may not be all the available instance variables. For a complete list, use `getDirectGenerics` or `getAllGenerics`, and locate those generic functions that end with `-Getter`.

Output

Every ScriptX object can be printed using a set of global functions that print a text-only representation of that object to an output stream. How an object's printable representation appears depends on how that class has defined it. In many cases, the printable representation may be nothing more than the name of the object's class and its memory address. In other cases, the class may define its printable representation to contain significant information about the value or contents of that object.

Printable representations are useful for debugging purposes. Be careful when using an object's printable representation to refer to that object—the object's printable representation may not be the same thing you would type into a script to create or query that object.

ScriptX defines a default stream named `debug` for printable representations. In some ScriptX environments, the debug stream may be attached to a listener or debugging window. You can also create additional streams and use them as the stream argument to each of the functions described in this section.

The Short Answer

The simplest way to get the printable representation of an object is to call the function `print` on it. The `print` global function writes a representation of the object to the `debug` stream. This form of printing is used extensively in examples throughout this book.

```
t := new Rect
print t
⇨  [0, 0, 0, 0] as Rect
r := new LinkedList
print r
⇨  #() as LinkedList
```

Note – The function call `print "t"` appears to have the same output in the Listener window as the simply typing the string literal `"t"`, followed by a newline character. The difference is that the `print` function actually prints to a stream, the `debug` stream, which can be directed to the Listener window, or to another window, such as a debugger.

You can do simple formatted output by coercing the printable representation of any object into a string, adding it to another string as a label, and then printing the result:

```
print ("r contains: " + (r as String))
⇨  "r contains #() as LinkedList"
```

The prin Generic Function

The basic generic function for printing objects is `prin`. Many classes in the ScriptX core classes implement a `prin` method; `prin` is commonly specialized by scripted classes.

◆ `prin` *objectToPrint option stream*

where:

- *objectToPrint* is the object you want printed
- *option* tells how to print the information (see below)
- *stream* is a writeable instance of `ByteStream`, or one of its subclasses.

The `prin` method prints *objectToPrint* to *stream* using the specified *option* without including a newline character at the end of the line.

See the "Streams" chapter of *ScriptX Components Guide* and the *ScriptX Class Reference* for more information about streams.

Unlike the `print` function, which is a global function that is not associated with any class, `prin` is a generic function. It is defined as a method by the root system class, `RootObject`, and can be specialized by any class in the system that requires a special printed representation. The `prin` generic function is used as a primitive by all other printing functions, most of which, like `print`, are global functions. By specializing `prin` for a class, you can assure that all ScriptX printing functions can print that class. (In certain classes that contain recursive structures, it is necessary to specialize the `recurPrin` generic function as well.)

By default, ScriptX provides only one appropriate stream for printable information—the debugging stream, referenced by the system global variable `debug`. Streams can also be instances of the class `String` or a stream that you have created yourself.

The *option* argument to `prin` may be one of the following names. Some classes, notably the `Exception` classes, define other options available to `prin`, but at least these four must be available:

- `@normal` prints a "regular" representation of the object.
- `@complete` prints a complete representation which may include more information than `@normal`, depending on the class. In collections such as `Array`, for example, the `@normal` representation only prints the first ten elements. The `@complete` representation prints all the elements.
- `@debug` prints even more information suitable for debugging purposes, including the class of the object, or in the case of collections such as arrays, the class of the collection and the classes of each of its items.
- `@unadorned` prints a plain representation of the object, which may vary depending on the class. For example, it prints an instance of `String` without quote characters as delimiters. It prints a name literal, a `NameClass` object, without the identifying "@" sign.

In the following examples, the result (specified by ⇨) is the output printed to the stream, not the result of the `prin` expression itself:

```
a := "test string"
prin a @normal debug
⇨  "test string"
prin a @debug debug
⇨  StringConstant: "test string"
prin a @unadorned debug
⇨  test string

t := #(1,2,3,4)
prin t @normal debug
⇨  #(1, 2, 3, 4)
prin t @unadorned debug
⇨  1234
prin t @debug debug
⇨  Array@0x3904e8: #(ImmediateInteger: 1, ImmediateInteger: 2,
   ImmediateInteger: 3, ImmediateInteger: 4)
```

```
t := #(1,2,3,4,5,6,7,8,9,10,11,12,13,14,15)
prin t @normal debug -- default is only 10 elements
⇨  #(1, 2, 3, 4, 5, 6, 7, 8, 9, 10, ...)
prin t @complete debug -- @complete prints all the elements
⇨  #(1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15)
```

Other Printing Functions

The generic function `prin` is used by several global functions, summarized here, which can be used as shortcuts to the `prin` generic. Printing functions are described in detail in the "Global Functions" chapter of the *ScriptX Class Reference*.

- `prinln` *self arg* [*stream*]

  `prinln` operates in the same manner as `prin`, except it also prints a newline character after the output.

- `prin1` *self* [*stream*]

  `prin1` is equivalent to `prin` with the `@normal` option. The *stream* argument is optional and defaults to `debug`.

- `print` *object* [*stream*]

  A combination of `prinln` and `prin1`, the `print` function prints the object representation to *stream* with the `@normal` option and a newline character. The *stream* argument is optional and defaults to `debug`. See the discussion and examples that begin on page 75.

- `prinString` *object arg*

  `prinString` yields a new `String` instance containing a printed representation of the given object.

- `printString` *object*

  `printString` is equivalent to `prinString`, except that it forces the `@normal` option.

- `shortPrin` *object arg stream length*

  `shortPrin` is equivalent to `prin`, except that `shortPrin` prints only the number of characters specified by *length*.

Formatted Output

Generalized formatted output is available with the `format` function:

- `format` *stream formatString object option*

The `format` global function prints a formatted string to a stream, where *stream* is the stream to print to, (for example, `debug`), *formatString* is the string to print (which can include substitution characters), *object* is the object you want a printable representation of, and *option* is one of the four printing options, as described on page 76. The final argument is optional—if it is not supplied, it is passed as `undefined` and defaults to `@normal`.

Note – The *stream* and *object* arguments to `format` are in a different order than in `prin` and its associated functions.

The *formatString* argument is a string that may contain substitution characters. Those substitution characters represent the object being printed, and are replaced in the resulting output by the printable representation of that object. For single objects, use "`%*`" as a substitution character.

```
i := 15
format debug "I is %*\n" i @normal
⇨  I is 15
```

Note that the format string can contain special characters such as newlines.

To print out a number of objects in one format string, you can put those objects in an array and use the array element substitution character. For multiple objects, the substitution character is "`%n`" where *n* is the position of the element in the collection. The "`%`" can be used as a terminator after `n`. If you use this form for multiple elements in a collection, the *option* argument of the format expression must be a collection of options (such as an array), with a separate option for every element being printed.

```
format debug "%1 and %2% are dead\n" \
     #("rosencrantz", "guildenstern") #(@unadorned,@unadorned)
⇨  rosencrantz and guildenstern are dead
```

As a shorthand for printing an array of options, the letters `n`, `c`, `u`, and `d` can be inserted after individual items. This allows items to be printed out with different formats. When individual formats are supplied, they override formats that are supplied in the fourth argument.

```
-- in this example, format codes %n1 and %u2 make the final
-- argument unnecessary
format debug "%n1 and %u2 are dead\n" #("rosencrantz", "guildenstern")
⇨  "rosencrantz" and guildenstern are dead
-- in this example, @unadorned overrides @debug for both items
format debug "%u1 and %u2 are dead\n" \
     #("rosencrantz", "guildenstern") #(@debug,@debug)
```

```
⇨  rosencrantz and guildenstern are dead
```

If you use the "%*" substitution in a format string on a collection object, the resulting output string contains the entire printable representation of that object:

```
format debug "The array is: %*" w @normal
⇨  The array is: #("rosencrantz", "guildenstern")
```

The "%n" substitution only works on collections that implement the `getNth` method. See Chapter , "Collections," or the "Collections" chapter of the *ScriptX Components Guide* for more information about collections. If you use the "%n" substitution on objects that are not collections, ScriptX reports an exception.

Recursive Output

Objects that contain other objects, for example collections such as arrays or keyed lists, may contain multiple copies of the same object, and may even contain references to themselves. In such a case, the printable representation of that object has a special format to manage recursion:

```
keepOnGoing := #(1,2,3)
append keepOnGoing keepOnGoing
```

In this example, `append` added the array `keepOnGoing` to itself as its own fourth element. Without a special format to represent this condition, the printable representation of `keepOnGoing` would go into an infinite loop trying to print its fourth element:

```
#(1, 2, 3, #(1, 2, 3, #(1, 2, 3, ...
```

Instead, `keepOnGoing` prints using a special format that looks like this:

```
print keepOnGoing
⇨  #1=#(1, 2, 3, #1#)
```

In this format, the object that contains a reference to itself is specified by a number (`#1` in this case), an equal sign, and the printable representation of itself. Then, in the list of elements, any elements that refer to that same object are indicated by that number, surrounded by hash signs (`#1#` in this case).

Collections that contain multiple copies of the same object are printed using a recursive format:

```
x := "elbow"
b := #(x, x)
print b
⇨  #(#1="elbow", #1#)
```

In this example, the `#1="elbow"` part refers to the first occurrence of x, and `#1#` refers to all other occurrences.

In objects that contain multiple recursive or repeated objects, those collections are numbered consecutively:

```
a := "elbow"
b := "knee"
c := #(a, b, b, a)
print c
⇨  #(#2="elbow", #1="knee", #1#, #2#)
```

C H A P T E R

Conditionals and Loops

4

This chapter outlines the structures in the ScriptX language for handling conditionals such as `if` and `case`, and loops such as `for` and `repeat`, as well as several expressions for control of these loops.

Conditionals: if and case

Conditionals are used to perform an action or combination of actions based on the outcome of a given test. ScriptX has three forms of conditionals:

- `if` . . . `do` . . .
- `if` . . . `then` . . . `else` . . .
- `case` . . . `end`

If Conditional

ScriptX has two `if` expressions: `if` . . . `do` and `if` . . . `then` . . . `else`.

▼ `if` *conditional* `do` *trueExpression*

`if` *conditional* `then` *trueExpression* `else` *falseExpression*

Use `if` . . . `do` if you are only interested in the `true` condition, `and` `if` . . . `then` . . . `else` if both the `true` and `false` conditions must be handled. You can use `if` . . . `then` without the `else` statement, but the compiler will be waiting for the `else` statement, so you will not immediately see the result of an `if` . . . `then` statement in the Listener window. Once another expression has been entered, however, you will see the results of both expressions. If you type an `if` . . . `then` without the `else` and want to see the result right away, you can use the operator `!!`, which is legal only at the top level when an expression is complete.

The *conditional* part of the `if` expression is an expression that evaluates to either `false` or something other than `false`. In both forms of the `if` expression, *trueExpression* is evaluated if the value of *conditional* is not `false`. That is, the test expression does not have to evaluate specifically to the `true` object for *trueExpression* to be evaluated—it just has to evaluate to something other than `false`. Both *trueExpression* and *falseExpression* are often compound expressions.

Both forms of the `if` expression return a value—that value is the value of either *trueExpression* or *falseExpression*, depending on which one was evaluated. Because `if` expressions return a value, you can nest `if` expressions inside other expressions, such as an assignment.

If the test in the `if . . . do` expression returns `false`, the whole `if` expression returns `undefined`.

Here are some examples using `if` expressions:

```
global x := 1, y := 5, p
if x > 0 do x := 0

p := if x > 0 then x else 0

if x < y then x
else (
    print "x is greater than y, setting x to 0"
    x := 0
)
```

case Conditional

Case expressions are used to switch between possible operations based on the result of evaluating an initial expression.

▼ `case` [*test*] `of`

tag `:` *expression*

`. . .`

[`otherwise :` *expression*]

`end`

The value of *test* is compared to the value of each of the tags in the tagged expressions (the *tag*`:`*expression* clause) in succession using the `equal` global function as a test of object equality. When this comparison first evaluates to `true`, then the *expression* (often a compound expression) is evaluated. Once a match is found and the *expression* is evaluated, the `case` expression terminates and yields the value of that *expression*. No other comparisons are made. Note that the `case` expression, like the `if` expression, always yields a value. If there is no match, the value is `undefined`.

You can have any number of *tag*`:`*expression* clauses in a `case` expression. Whereas the *expression* can be any expression (including a compound expression), the *tag* is a limited form of expression, evaluated before the return value of *test* is matched against it. The *tag* can be one of the following:

- a number, string, or name literal (`45`, `"this or that"`, `@nameMe`)
- a variable name
- an array or keyed linked list literal (`#(1, 2, 3)`)
- an array access expression (`myarray[1]`)
- a reference to a class or instance variable (`myrect.x1`)
- an anonymous function with no arguments (described in Chapter , “Functions, Threads and Pipes”)
- `nextMethod` (described in Chapter , “Defining Classes and Objects.”

- any other inner-level expression enclosed in parentheses

The optional `otherwise` clause identifies the default case, if none of the tags result in `true`:

```
case pet of
    @cat: print "meow"
    @dog: print "woof"
    @bird: print "chirp"
    @snake: print "hiss"
    otherwise: print "mysterious animal noise"
end

bool := case i of
    0:false
    1:true
    otherwise: (print "out of range"; undefined)
end
```

If the *test* is omitted, the tags are expected to be expressions that evaluate to `true` or `false`. When the first tag expression to yield `true` is evaluated, the `case` expression terminates and yields the value of its corresponding expression, which it evaluates.

```
case of
    (isComparable a b): (
        case of
            (a > b): print "a is greater than b"
            (a = b): print "a is equal to b"
            otherwise: print "a is less than b"
        end
    )
    otherwise: print "a and b are not comparable"
end
```

If the `otherwise` tag is omitted and none of the other tags match the value of the test expression, the value of the `case` expression is `undefined`.

Repeat Loops

ScriptX has two forms of `repeat` loops: one that repeats as long as a conditional test is not false (`repeat . . . while`), and one that repeats until the conditional test is not false (`repeat . . . until`). Each form has two variants, providing a total of four possible forms:

▼ `repeat while` *conditional* `do` *expression*
`repeat until` *conditional* `do` *expression*
`repeat` *expression* `while` *conditional*
`repeat` *expression* `until` *conditional*

In all four forms, *conditional* is the loop control expression, which must be an expression that evaluates to either `false` or `not false`, and *expression* is the body of the loop containing the expression (often a compound expression) that is executed at each iteration of the loop.

In the `repeat` forms that have the *expression* before the *conditional*, the *expression* is evaluated at least once before the *conditional* is evaluated. In the other forms (the `do` forms), if the `while` test evaluates to `false`, or the `until` test to `not false`, then *expression* is never evaluated.

All `repeat` expressions return the `OK` object, a special system object, regardless of the outcome of their conditional tests or actions.

```
global x := 10
repeat while x > 0 do (
    print (sin x)
    x := x - 1
)

x := 10
repeat (
    print x
    x := x - 1
) until x == 0
```

Using the `exit` construct defined on page 94, you can exit from within a repeat loop, returning a different value.

for Loops

In ScriptX, the `for` loop is very sophisticated and includes

- concurrent looping of multiple iteration sources
- collecting and selecting values based on tests
- compounding the results of several loops

ScriptX has three forms of `for` loops:

- `for` *sources* [*conditional*] `do` *expression*
  `for` *sources* [*conditional*] `collect` [`into` *collection*]
  [`by` *function*] [`as` *collectionClass*] *expression*
  `for` *sources* [*conditional*] `select` *expression* [`into` *collection*]
  [`by` *function*] [`as` *collectionClass*] `if` *conditional*

In each form of the `for` expression, *sources* is the source of iteration and *conditional* is an additional, optional test that can be used to control the loop.

Simple Iteration

The simplest use of the ScriptX `for` loop is to execute an expression a specified number of times. This iteration can take three forms:

- `for` *numericExpr* `do` *expression*
  `for` *item* `:=` *rangeOrCollection* `do` *expression*
  `for` *item* `in` *rangeOrCollection* `do` *expression*

In the first form, *numericExpr* is the number of times to iterate, or an expression that results in a number; and *expression* is the expression, often a compound expression, to evaluate at every iteration. The expression is evaluated the number of times specified.

```
for 9 do print "hello"
```

```
allSpaces := #()
for 52 do append allSpaces " "
```

The last two forms are for iterating over ranges or collections. The *item* part of the `for` loop names the local iteration variable, which holds successive values from the range or collection as the loop iterates. This local iteration variable is available within the body of the loop. The *rangeOrCollection* part is either a range, as described below, or a collection such as an array. Finally, the *expression* part of the loop, as in the previous form, is an expression, often a compound expression, that is evaluated at each iteration. Note that `item :=` and `item in` are equivalent forms, and both ranges and collections can be used with either form.

In all of these forms, the `for` loop returns the last value of the loop body expression when the loop ends.

Here are two examples of using this form of `for` loop. The sections that follow supply more examples.

```
global i, myArray := #(1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12)
for i := 1 to 10 do print myArray[i]
global theMin := 25
for i in #(1, 435, 234, 23, 8) do
    if i < theMin do theMin := i
```

```
⇨ undefined
```

The `for` loop evaluates to `undefined` because in the last iteration, the `if` expression evaluates to `undefined`; *theMin*, however, has a value of 1, as you would expect.

Note – A ScriptX `for` expression actually works by creating an object called an iterator. Within a for loop, a script can perform many operations that would be impossible without access to this iterator. Iterators, an advanced topic, are discussed in the "Collections" chapter of the *ScriptX Components Guide*. Also note that when you want to minimize memory usage and garbage collection, you can avoid creating an iterator by using the `forEach` method on a collection instead of using a `for` expression.

---

for Loops and Ranges

Explicit ranges are specified using the range literal construct. You can also use any expression that yields a `Range` object.

▼ *startValue* `to` *endValue* `[ by` *increment* `]`

The *startValue* side of the range is the number from which to start iterating, the *endValue* side is the number with which to end, and the optional *increment* is the amount by which to increment at each iteration. If `by` *increment* is omitted, the increment is assumed to be 1.

```
for i := 1 to 23 do print i
for i in 0 to 100 by 10 do print (i * i)
```

To decrease the index at each iteration, simply reverse the order of the *startvalue* and *endvalue* of the range. For decreasing ranges, you must always state the amount by which to decrement:

```
for i := 10 to 1 by -1 do print (i + i)
```

The range syntax described here is actually a range literal that creates an instance of one of the subclasses of `Range`. That literal can be used anywhere in a script. Ranges are described in greater detail in the section on "Ranges" beginning on page 157, and in the "Collections" chapter of the *ScriptX Components Guide.*

for Loops and Collections

You can use `for` loops to iterate over collections such as arrays and linked lists. At each iteration, each item in the collection in turn is assigned to the iteration variable (`i` in these examples). The loop body is then evaluated with the local iteration variable set to that value.

```
for i in #(1,2,3,4) do print i
for i := #("Francois","Inez","Louis","Margot") do
    format debug "%* did it!\n" i @unadorned
```

Note that ranges, described in the previous section, are also collections, which allows you to form `for` expressions like this:

```
global countdown := 10 to 1 by -1
for i in countdown do
    (format debug "%* seconds to blast off!\n" i @normal)
```

Multiple Sources of Iteration

All `for` loops can have multiple sources of iteration, separated by commas. Each source is iterated at each loop. The loop terminates when any one of the sources of iteration ends.

```
-- in this example, the first iterator terminates first
for 14, i := 1 to 1000 by 23 do
       (prin i @normal debug; prin " " @unadorned debug)
⇨  1 24 47 70 93 116 139 162 185 208 231 254 277 300 OK

-- in the following example, the second iterator terminates first
for 14, i := 1 to 1000 by 230 do
       (prin i @normal debug; prin " " @unadorned debug)
⇨  1 231 461 691 921 OK

for i := 1 to 10, j := 1 to 10 by 2 do (
    format debug "I is %1. J is %2\n" #(i, j) #(@normal,@normal)
    print (i * j)
)

-- print only the first 8 elements in the array listOfPeople
for 8, i in listofPeople do print i
```

Additional Tests

In `for` expressions you can also specify an additional, optional test to stop each form of the loop from further iterations. Using tests in addition to standard iterations allows both the simplicity of `for` loops and the flexibility of a `repeat` loop.

▼ `for` *source* `while` *conditional* `do` *expression*
`for` *source* `until` *conditional* `do` *expression*

In this form of the `for` expression, the *source* part of the loop is any of the forms described in the previous sections, and *conditional* is an expression that returns `false` or an object whose value is not `false`.

```
x := 1
for 23 while x <= 5 do (
    print x
    x := x + 1
```

```
)
⇨ 1
   2
   3
   4
   5
   6
```

You see `1` through `6` in the Listener window because the `for` expression above prints `1` through `5` and then returns `6`, which is the value of the `for` expression (that is, the value of `x`, the last expression inside the `for` expression) at the end of the loop.

Here's the same loop performed using `until` rather than `while`:

```
x := 1
for 23 until x > 5 do (
   print x
   x := x + 1
)
⇨ 1
   2
   3
   4
   5
   6
```

Here's another example using `while`:

```
for m := 1 to 100, n := 1 to 100 by 5 while m * n * n < 10000 do (
   prin m @normal debug; prin ", " @unadorned debug
   prinLn n @normal debug
)
```

Collecting or Selecting Results

The examples of `for` loops in the previous sections all used the `do` reserved word to begin an expression that is evaluated each time the loop iterates. The `for` loop also has forms that allow you to collect the results of an iterated expression into a collection such as an array, or to collect a particular value of an iterated expression based on a given test.

▼ `for` *sources* [`(while|until)` *conditional*] `collect`
[`into` *collection*] [`by` *function*] [`as` *collectionClass*] *expression*

`for` *sources* [`(while|until)`*conditional*] `select` *expression*
[`into` *collection*] [`by` *function*] [`as` *collectionClass*] `if` *conditional*

In both forms, *source* is the source (or sources) of iteration, as described in "Simple Iteration" on page 87 and *conditional* is the optional `while` or `until` test described in "Additional Tests."

The `collect` form builds a collection (an array by default) of the values of the loop body at each iteration. At each iteration the *expression* is evaluated, and the values of that expression are collected.

```
for i := 1 to 10 collect i
⇨  #(1, 2, 3, 4, 5, 6, 7, 8, 9, 10)

sinvals := for i := 1 to 3 collect sin i
⇨  #(0.841470984807897, 0.909297426825682, 0.141120008059867)
```

The `select` form allows you to select and assemble a collection of the values of the loop at each iteration based on a given test. If the test returns `true`, the value of *expression* at each iteration is put into a collection (an array by default).

Note that to use a complex expression in the *expression* part of the `select` form, you must specify that expression inside parentheses. The only expressions that can be used without parentheses in the `select` form of the `for` loop are:

- a variable name (for example, the name of the index variable)
- an array access expression (`myArray[i]`)
- a reference to a class or instance variable (`myRect.x1`)
- an anonymous function with no arguments (see “Anonymous Functions” on page 105 of Chapter , “Functions, Threads and Pipes”)
- `nextMethod` (see “Defining Class and Instance Variables” on page 145 of Chapter 6, “Defining Classes and Objects”)

Here are some examples:

```
negativesins := for i := 1 to 10 select (sin i) if (sin i) < 0
⇨  #(-0.756802, -0.958924, -0.279415, -0.544021)

global status := #(@doorClosed:false, @keyInIgnition:true,
       @propellerMoving:true)
print status
⇨  #(@doorClosed:false, @keyInIgnition:true, @propellerMoving:true)
for i in #(@doorClosed, @keyInIgnition, @propellerMoving)
    select i if status[i] = true
⇨  #(@keyInIgnition, @propellerMoving)
```

into

The optional `into` clause allows you to append collected or selected items to an existing collection such as an array:

```
squares := #()
for s in 1 to 10 collect into squares (s * s)
```

The `into` clause is most useful for several disconnected (either sequential or nested) `for` loops that modify the same collection:

```
nums := for i in 1 to 5 collect i
for i in 20 to 23 collect into nums i
for i in 40 to 45 collect into nums i
prin nums @complete debug
⇨  #(1, 2, 3, 4, 5, 20, 21, 22, 23, 40, 41, 42, 43, 44, 45)

global doors, openDoors := #()
doors := #(@front:false,@back:true,@side:true,@garage:false)
for i in #(@front,@back,@side,@garage)
select i into openDoors if doors[i] = true
⇨  #(@back, @side)
```

Note that the expression following the reserved word `into` must evaluate to an instance of a collection. That collection expression can be one of the following expressions:

- a variable name (for example, the name of the index variable)
- an array or keyed linked list literal (`#(1, 2, 3)`)
- an array access expression (`myArray[i]`)
- a reference to a class or instance variable (`myrect.x1`)
- an anonymous function with no arguments (see “Anonymous Functions” on page 105 of Chapter , “Functions, Threads and Pipes”)
- `nextMethod` (see the discussion of `nextMethod` on page 132 of Chapter 6, “Defining Classes and Objects”)
- any other expression, surrounded by parentheses.

by

The optional `by` clause in either the `collect` or `select` form of the `for` expression allows you to specify a function that is used to add each new item to the final collection. This function, which is called a collector, can be either a regular function or a generic function that has the required form. To be used as a collector, a function must return the collection into which the items are being collected.

The collector function is called with two arguments on each pass through the loop. The first argument is a collection (specified by the `into` clause, or created based on the `as` clause). The second argument is the current value of the loop body. Several generic functions in the Collection protocol have the required form, and others can easily be embodied in a scripted function. The following script joins a sequence of collections into one using `merge`, a generic function that every `Collection` object provides a method for.

```
global firstArray := #(1,2,3,4,5)
global secondArray := #(@yes,@no,@maybe)
global thirdArray := #("sow","ewe","cow","hen")
bigJumbledArray := #(firstArray,secondArray,thirdArray)
niceNeatArray := #()
for i in bigJumbledArray collect into niceNeatArray by merge i
⇨  #(1, 2, 3, 4, 5, @yes, @no, @maybe, "sow", "ewe", ...)
```

By default, a `for` expression calls the `appendReturningSelf` global function, which is defined for sequences. This global function is similar to `append`, except that it returns the collection itself. (By contrast, the generic function `append` returns the key of the item that was appended.) Another useful function for collecting items is the global function `addManyValues`.

In the following example, items are collected into a string using a function that is built around `prepend`, a generic function for which all sequences define a method. This has the effect of reversing the order of a string. (Keep in mind that `prepend` is an inefficient way to add items to some sequences, including arrays and strings. It is used here only for illustration purposes.) For more information on functions, see Chapter , “Functions, Threads and Pipes.”

```
global reagan := new String string:""
speech := "Facts are stupid things"
-- define a function that can act as a collector or selector
function prependReturningSelf sequence value -> (
    prepend sequence value
    return sequence
)
-- use the collector function to reverse the string
for i in speech collect into reagan by prependReturningSelf i
⇨  "sgniht diputs era stcaF"
```

For a variation on this script which uses an anonymous function, see page 104 of Chapter 5, “Functions, Threads and Pipes.”

The only types of expressions that can be specified after `by` without parentheses are the following, where appropriate:

- a variable name (for example, the name a generic function or a class)
- an array access expression (`myArray[i]`)
- a reference to a class or instance variable (`myRect.x1`)
- an anonymous function (see “Anonymous Functions” on page 105 of Chapter , “Functions, Threads and Pipes”)
- `nextMethod` (see “Class and Instance Methods” on page 120 of Chapter , “Defining Classes and Objects”

To specify complex expressions in a `by` clause, you must specify them within parentheses.

as

The optional `as` clause specifies a collection class into which elements are collected or selected. Collections are described in Chapter , “Collections.” Note especially the section on strings as collections.

```
for i in "jabberwocky" collect as Array i
⇨  #(106, 97, 98, 98, 101, 114, 119, 111, 99, 107, ...)

for i in "jabberwocky" select i as String if i > 105
⇨  "jrwoky"
```

The first example collects characters in a string literal (a `StringConstant` object) into an array. They are coerced to Unicode values, which are `ImmediateInteger` objects, before being added to the array. The second example selects only those elements of a string for which the Unicode value is greater than `105`.

The expression following the `as` reserved word must evaluate to a class. The types of expressions that can be specified after `as` without parentheses are the same as for the `by` clause, listed above. To specify a complex expression in an `as` clause, you must specify it within parentheses.

Loop Control Expressions

Loop control expressions are used to affect the flow of control in a compound expression such as those used with `for` and `repeat` loops.

▼ `continue`

`exit [ with` *expression* `]`

The `continue` expression causes the closest enclosing `for` or `repeat` loop to immediately begin its next iteration. If there are no further iterations, the loop ends.

```
for i in #(1, 3, 6, 9) do (
     if (i * i) = 36 do continue
     print i
)
⇨  1
   3
   9
```

If `continue` is used in the `collect` or `select` forms of the `for` loop, the value of the loop expression is not added to the collection if the loop is interrupted. For example:

```
for i := 1 to 5 collect (
   if i = 3 then continue else i
)
⇨  #(1, 2, 4, 5)
```

The `exit` expression is used to immediately stop execution of the innermost `for` or `repeat` loop, continuing the execution of the surrounding loop or block. Loops that are exited with `exit` alone return `undefined`. To specify a return value for the loop, use the `exit with` form. The return value of the entire block is then the value of *expression.*

```
val := for i in #(1, 3, 6, 9) do (
     if i = 6 do exit with "Six"
     print i
)
```

```
⇨  1
   3
   “Six”
print val
⇨  "Six"
```

C H A P T E R

Functions, Threads and Pipes

5

This chapter describes how to define ScriptX functions and also discusses the ScriptX thread and pipe operators.

Defining Functions

Functions provide a way to group a series of common operations under a single name. As described in Chapter 3, "Working with Objects," there are two kinds of functions in ScriptX: regular functions and generic functions. This section describes how to define regular functions. Generic functions are created as a side effect of defining a method on a class or object. Method definition is described in Chapter 6, "Defining Classes and Objects."

Functions, like all things in ScriptX, are objects (literally, instances of a subclass of `AbstractFunction`, typically `ByteCodeMethod`). There are two ways to define functions. A named function definition creates a new variable and immediately assigns the function object to it. An anonymous function is not assigned to a variable, although it can be assigned in a subsequent expression. Anonymous functions are described on page 105.

If you are defining a recursive function, you must use a named function. This is true because a named function is the only kind of implicit or explicit variable declaration in which you can use a name within the initializing expression and have it refer to the variable being defined. In other cases, it refers to an outer scope variable of that name.

Function Syntax Summary

Regular functions with names are defined using the following syntax:

▼ `function` *fnName* [*positionalArgs*] [`#rest` *restArg*] [`#key` *keywordArgs*] \
`->` *body*

The `function` reserved word, which can be shortened to simply `fn`, signals that this is the start of a function definition, and *fnName* is the name of the variable this function object is assigned to. After the function name are the parameters to the function: *positionalArgs* specify arguments that are required by the function and must be specified in a specific order (position), *restArg* holds an array of optional arguments given when the function is called, and *keywordArgs* is a list of keyword arguments. All three types of arguments are optional, although they must be specified in this order.

Finally, *body* is the expression, often a compound expression, that makes up the body of the function. The expressions within the function body are executed when the function is actually called.

Function Names and Variables

The *fnName* part of the function definition is the name of the variable that this function object is assigned to once it has been created. Like all variables in ScriptX, if *fnName* has not been previously declared, it is automatically declared global. You can specify that *fnName* is to be declared locally by preceding the function keyword with the word `local`. Note that just as with local variables, you can only define a local function inside a local scope (that is, not at the top level).

```
local function sortIt theList ->  . . .
```

Additionally, the variable that this function is assigned to is automatically declared `constant` in the function definition, which means that you cannot assign anything else to it once the function has been created (except by redefining another function). This is to prevent accidentally overwriting the definition of a function with a simple assignment. If you expect to reuse the global variable name to which you've assigned a function elsewhere in your script, consider using the anonymous function construct instead (or a local variable, if appropriate).

Functions with Positional Arguments

Functions with positional arguments are the simplest form of function, and they look and operate just like functions in other languages. When the function is called, the positional arguments must all be specified and must be in the right order for the function to work.

You can specify any number of positional argument parameters (including none), but they always appear first in the function definition, before any "rest" arguments or keyword arguments.

```
function square n -> n * n
square 10
⇨  100
function beepMe -> print "beep!"
beepMe()
⇨  "beep!"
```

Note – If a function has no arguments, empty parentheses are required.

Functions with Keyword Arguments

Functions with keyword arguments provide the most flexibility for the user. You can call such a function with different numbers of arguments, in any order, and some or all of those arguments can be optional. However, calling functions that have been defined to use keyword arguments involves a performance hit because all the keyword arguments must be processed by ScriptX before the function is executed.

To specify that your function takes keyword arguments, use the `#key` keyword in the function definition, followed by the definitions of the keywords themselves:

▼ `function` *fnName* [*positionalArgs*] [`#rest` *restArg*] [`#key` *keywordArgs*] \
`->` *body*

Keyword arguments are optional in the function definition, but if they are included, they must be specified after both the positional arguments and the optional "rest" argument (described in the next section).

You can specify any number of keyword arguments after the `#key` reserved word. Keyword arguments look like this:

▼ *keyName*`:[` *argName* `] [ (` *initialValue* `) ] . . .`

where:

- *keyName* is the name of the keyword, as specified when the function is called. The optional *argName* is a keyword (specified when the function is called) that is assigned to in the body of the function. If you do not specify *argName* in the function definition, the keyword value is assigned to a local variable of the same name as the keyword.
- *argName* is most often used when the local variables used for the keywords may conflict with other system-defined variables. Because functions start a new variable scope, the variable names of your keys as used in the body of the function may hide the variable names of classes or objects or other functions that you may want to make use of in the body of that function. Specifying a separate *argName* allows you to continue to access the original values of those variable names while still being able to use meaningful keyword names.
- *initialValue* is the optional value this keyword argument is to initially be set to if a value is not given for that argument when the function is called. This is often called the default value. The initial value is always specified within parentheses. If you do not specify a initial value for a keyword, and no value for that keyword is given when the function is called, the value of that keyword in the body of that function is the `unsupplied` object. Note that `unsupplied` is different from `undefined`; `undefined` must be specifically indicated by the user for it to appear in the body of a definition (and may have a specific meaning), but `unsupplied` is used if the user did not specify that key at all.

If a function with keyword arguments is called with duplicate keywords, the value of the *first* one in the series is used.

Here are some examples:

```
-- reportArgs simply prints out its a and b values
function reportArgs #key a: b: -> (print a; print b)
reportArgs a:10 b:20
```

```
⇨  10
   20
reportArgs a:100
⇨  100
   unsupplied
reportArgs a:10 b:30 a:50
⇨  10
   30

-- reportArgs2 is the same as reportArgs, but the values of the
-- keys are assigned to the local variables moo and quack
-- instead of a and b.
function reportArgs2 #key a:moo b:quack -> (
   print moo; print quack
)
reportArgs2 a:10
⇨  10
   unsupplied

-- reportArgs3 defines default values for a and b so
-- neither will appear as unsupplied
function reportArgs3 #key a:(1) b:(2) -> (
   print a; print b
)
reportArgs3() -- use all defaults
⇨  1
   2
reportArgs3 a:20
⇨  20
   2

-- reportArgs4 defines default values
-- and assigns them to local variables
function reportArgs4 #key a:moo(1) b:quack(2) -> (
   print moo; print quack
)
reportArgs4()
⇨  1
   2
reportArgs4 a:6
⇨  6
   2
```

Functions with Rest Arguments

ScriptX allows you to define a function that you can call with any number of arguments. For example, you could create a function that adds up all of the numbers passed into it—two numbers, ten numbers, or whatever—the important thing is that the number of arguments is not fixed.

The `#rest` argument allows you to define a function with a variable number of arguments. In this sense, "rest" means "the rest of the arguments after the positional arguments, except the denoted keyword arguments." The function collects those "rest" values into an array, and in the body of the function you can access elements in that array.

Unlike positional arguments, rest arguments are not required—when you call the function you can supply no rest arguments, or as many as you need to.

Notice in the following function definition that `#rest` takes a single argument *restArg*. The combination `#rest` *restArg* comes after any positional arguments and before any keyword arguments. By convention, the argument `args` is typically used for *restArg* (though you could name it whatever you want), so what you often see in examples is the combination `#rest args`.

▼ `function` *fnName* [*positionalArgs*] [`#rest` *restArg*] [`#key` *keywordArgs*] \
`->` *body*

This is an example of a function that takes no arguments other than its rest arguments:

```
-- addEmUp takes a variable number of arguments, sums them all
-- and returns the sum
function addEmUp #rest args -> (
    local sumArgs := 0
    for i in args do sumArgs := sumArgs + i
)
addEmUp 1 2 3 4 5
⇨ 15
addEmUp 3 4
⇨ 7
```

In place of using `args` in the previous example, we could have used `numList`, to denote the list of numbers.

This example defines a function that takes a single required argument and a variable number of rest arguments (each argument happens to itself be an array):

```
-- joinArray takes a single required array and a variable number
-- of other arrays. It builds a single array of all the
-- elements in all its arguments
function joinArray array1 #rest otherArrays -> (
    for i in otherArrays do addMany array1 i
    return array1
)
joinArray #(1,2,3) #(4,5,6)
⇨ #(1, 2, 3, 4, 5, 6)
joinArray #() #(@angora,@persian) #(@egyptianmau) #(@housecat)
⇨ #(@angora, @persian, @egyptianmau, @housecat)
```

Combining Rest and Keyword Arguments

If your function definition uses both the `#rest` and `#key` reserved words to define both rest and keyword arguments, the non-positional arguments can only be in the form of keyword-value pairs (*keyword*:*value*). You cannot mix arbitrary numbers of `#rest` arguments and keyword arguments.

When a function with both rest and keyword arguments is called, the keywords and values supplied to the function are stored into the rest argument in sequence (key, value, key, value, and so on) with the keywords appearing as `NameClass` objects.

```
function showRest #rest args #key a: b: c:(100) ->
    print args debug

showRest a:10 b:20 c:30
#(@a, 10, @b, 20, @c, 30)

showRest a:10
#(@a, 10)

showRest()
#()
```

Note that the compiler translates a keyword argument into a pair of conventional arguments. The following function calls are equivalent:

```
grok foo:10 moof:20
grok @foo 10 @moof 20
```

Function Return Values

You can specify a function's return value using the `return` block control expression:

▼ `return` *expression*

When a `return` expression is encountered in the body of a function definition, the function is immediately exited with the value specified by *expression*.

Functions that do not specify a specific return value return the value of the last expression evaluated while executing the function.

```
function factorial n -> (
    if n <= 0 then return 1
    else return (n * (factorial (n - 1)))
)
-- now, try a few test values
factorial 0
⇨  1
factorial 4
⇨  24
function sumAndPrint #rest args -> (
    local mySum := 0
```

```
    for i in args do mySum := mySum + i
    format debug "The sum is: %*\n" mySum @normal
    return mySum
)

sumAndPrint 10 20 34 45
⇨  The sum is: 109
    109
```

Anonymous Functions

The anonymous function expression evaluates to a function object (actually, an instance of the class `ByteCodeMethod`). Unlike function definitions using the `function` construct, anonymous function definitions don't cause a variable to be declared implicitly, and so are not accessible by naming the variable to which they are bound (hence the term *anonymous* function). Anonymous functions are useful for nesting function definitions in other expressions, or as arguments to other functions.

▼ `(` [*positionalArgs*] [*RestArg*] [*KeywordArgs*] `->` *body* `)`

Anonymous functions can be defined with the same three types of arguments as regular functions: positional arguments, rest arguments, and keyword arguments. Just as with regular functions, all three types of arguments are optional, but must be specified in the order shown.

```
(n -> n * n)
(a b -> a + b)
(#rest nums -> for i in nums do print (i * 10))
```

Anonymous functions can be used wherever normal function calls are used:

```
function addTwo a b -> a + b
addTwo 4 5
⇨  9
(a b -> a + b) 4 5
⇨  9

for i in #(1,2,3,4) collect (n -> n as String) i
⇨  #("1", "2", "3", "4")
```

In the ScriptX core classes, many generic functions take a function as an argument. An anonymous function is an ideal way to pass a function object as an argument, since it doesn't require assigning a name. The following example demonstrates the use of an anonymous function in a callback script. The second argument in the third expression, `addTimeCallback`, registers a callback that will print the clock's time in 15 seconds. This printing action is called as an anonymous function.

```
global myClock := new Clock
myClock.rate := 1
addTimeCallback myClock (c -> print c.time) \
    myClock #() (myClock.time + 15) true
```

Because an anonymous function expression evaluates to a function object, you can assign an anonymous function expression to a variable, and then use that variable name to call the function, just as if you had created that function with the `function` expression. The difference is that the variable to which you assign the anonymous function is not a constant variable, so it can be reassigned to any other object.

```
cubeMe := (a -> a * a * a)
cubeMe 2
⇨  8
```

As mentioned earlier, if your function requires recursion (a recursive function calls itself to accomplish an operation), you cannot use an anonymous function. Recursion depends on a special binding of a variable name to a function definition before that function object has been created. The `function` expression provides a local binding that allows recursion. An anonymous function does not.

```
-- this is OK
function sumTheNum arg -> (
    if arg <= 0 then return 0
    else return (arg + (sumTheNum (arg - 1)))
)

sumTheNum 2
⇨  3
sumTheNum 10
⇨  55
```

In the following example, an anonymous function is bound to a variable. It works as intended at runtime. However, it is less efficient than the same function would be with a `function` binding, since the compiler cannot optimize for recursion.

```
-- an inefficient way to do recursion
sumTheNum2 := (arg -> (
    if arg = 0 then return 0
    else return (arg + (sumTheNum2 (arg - 1)))
))
```

The body of an anonymous function can be a series of expressions. The following script repeats an example that first appeared on page 92, where it was defined using a named function.

```
global backwards := new String string:""
global palindrome := "Satan, oscillate my metallic sonatas"
for i in palindrome collect into backwards by \
    (sequence value -> prepend sequence value; sequence) i
⇨  "satanos cillatem ym etallicso ,nataS"
```

In this example, the body of the anonymous function contains two expressions. At each iteration through *palindrome*, the anonymous function prepends *i* to *backwards* and then returns the new version of *backwards*. The value of the final expression (the final version of *backwards* in this example) is the return value of the entire function. A ScriptX `for` expression can act as a collector, accumulating items into a collection. Since the `by` clause in a `for` expression must evaluate to a function that returns the collection itself, this anonymous function combines `prepend` and an expression that returns the collection. In this way, the anonymous function makes it possible to use `prepend` as a collector function. For information on the `for` expression, see the discussion which begins on page 86.

Anonymous functions can be used to provide a "wrapper" around a function, perhaps to change its signature or return value. In the previous example, an anonymous function acts as a wrapper around the generic function `prepend`, creating a version of `prepend` that returns the modified sequence. This can be especially useful if you are storing a function in a variable or table. For a code example that demonstrates the use of anonymous functions in a function lookup table, see the section "Using the Collections Component" in the "Collections" chapter of the *ScriptX Components Guide*.

Closures

A closure is a function that refers to a closure variable, also called a free variable. This variable is declared as a local variable in some higher scope that contains the definition of the closure. The closure itself can be defined as either a local function or an anonymous function. A closure is created any time a local or anonymous function with free variables "escapes" from the scope in which those variables are defined.

The following example creates a closure around a local function called `elapsedTime`. This closure can then be used to keep track of the time that has passed since the last time it was called. In a ScriptX title, it could be used to do simple timekeeping without creating a clock, and without defining any new global variables. ScriptX defines a global instance of `CalendarClock` for simple timekeeping, stored in a global variable called `theCalendarClock`. (Since it is accurate only to one second, the clock stored in `theCalendarClock` is not intended for system timekeeping.) The closure function that is defined here refers to this clock internally.

```
function closureMaker -> (
    -- x is the free variable or closure variable
    local x := theCalendarClock.time
    -- elapsedTime is the function that will be returned as a closure
    local fn elapsedTime -> (
        local result := theCalendarClock.time - x
        x := theCalendarClock.time
        result
    )
```

```
    -- the next expression is the return value of this function
    -- it returns the local function elapsedTime
    -- that turns it into a closure
    elapsedTime
)
⇨  #<ByteCodeMethod Scratch:closureMaker of 0 arguments>
```

In this example, a closure is created when the local function is returned by a global function. This means that `elapsedTime` "escapes" to the top level, even though it was defined as a local function.

Each time `closureMaker` is called, it returns another closure. The closures it creates share code with the other closures on the `elapsedTime` function, but each closure maintains its own value for the closure variable `x`. The value of this closure variable is accessible only through the closure. In the following example, a call to `timeKeeper` answers the question, "How much time has passed since I called you last?"

```
timeKeeper := closureMaker() -- create a closure
⇨  #<ByteCodeMethod anonymous of 0 arguments>
-- now, wait seven seconds and try timeKeeper
timeKeeper()
⇨  0:0:7:0 as Time
```

The ScriptX closure construct is more versatile and powerful than a static variable in C or C++. A closure can be used to encapsulate a value, such as a counter, so that it is only accessible through a function. The following example creates a closure from an anonymous function.

```
-- defines a closure on an anonymous function
function counter -> (local x:0; (-> x := x + 1))
```

Each time `counter` is called, it returns a new closure that behaves isomorphically with repect to other counters, while retaining its own closure variables, as the following example demonstrates:

```
myCounter := counter()
myCounter()
⇨  1
myCounter()
⇨  2
-- now create another counter
global anotherCounter := counter()
anotherCounter()
⇨  1
myCounter() -- myCounter still retains its own closure variables
⇨  3
anotherCounter() -- so does anotherCounter
⇨  2
```

Using apply to Call Functions

The `apply` function is a special way of calling functions where some or all of the arguments that you want to send to that function are available in a collection such as an array. It takes the arguments out of the collection and "applies" the function to them as if you had written them down explicitly in a normal function call.

▼ `apply` *function otherArgs collectionOfArgs*

In using `apply`, *otherArgs* can be any number of interim arguments. The final argument to `apply`, however, must be a collection such as an array (or an expression that results in a linear collection). That array contains any remaining arguments to the *function*, and can contain any number of elements, including none `#()`.

The `apply` function is often useful when you are calling a function that defines rest arguments from within another function that also defines rest arguments. It allows you to "spread out" an array of arguments into a function call, regardless of how many elements the original array had in it.

```
function addEmUp #rest args -> (
    local sum := 0
    for i in args do sum := sum + i
    return sum
)
global a := #(1,4,9,16)
global b := 1 to (a.size)
myArgs := merge a b
-- use apply to call a function with rest arguments
apply addEmUp myArgs
⇨  40
```

A typical use of `apply` is in initialization (`init`) methods for classes and objects. All `init` methods define rest arguments, and you are required to pass them along to a superclass's `init` methods. For more information on initialization methods and `apply`, see Chapter 6, "Defining Classes and Objects."

Threads

Threads are independent processes that run in parallel with one another, sharing processor time. For example, you could process a long search or compilation in the background in one thread, and run a continuous animation in another thread, while still interacting with the main application. A ScriptX title starts running in the main thread, which can be referenced through the global variable `theMainThread`. The title can create and run other threads at any time.

The ScriptX thread system is described in detail in the *ScriptX Components Guide*. The ScriptX language provides a shorthand construct for spawning threads, similar to that used in the UNIX shell languages. The ampersand character (`&`), when used at the end of an expression as a postfix operator, starts that expression running in a separate thread.

```
(p := importBitmap "monalisa") &
computePrimesTo 10000 &
```

The example above starts threads running functions called `importBitmap` and `computePrimesTo` in parallel with the main program. Any number of separate threads can run at the same time, subject to the limitations of memory and processor time, allowing parallel processing of different tasks.

The thread operator (`&`) creates a thread object, with the expression that precedes it as the code that runs in the thread. The thread is immediately active. You can test the status of the thread or control the thread by using any of the global functions or generic functions described in the *ScriptX Class Reference*.

```
t := computePrimesTo 100 &
repeat until (t.status = @done) do
    print "tick"
print "computation completed"
```

This example executes a `computePrimesTo` function as a thread and then prints “tick” to the debug stream until the status of the thread changes from `@running` to `@done`.

The & operator is a shorthand form for the `callInThread` global function, which creates a new `RegularThread` object with an operating priority of `@normal`. The value of `priority`, either `@normal` or `@high`, determines how much processing time the thread receives. See the “Threads” chapter of the *ScriptX Components Guide* for more details on thread priority.

Pipes

Pipes are used to iterate over a source collection, range, or other sequenceable class. Each element is fed through the pipe to a function, or a collection class or instance. Pipes are often a useful shorthand for many nested `for` loops.

▼ *from* | *to*

The pipe operator is an infix shorthand for calling the `pipe` generic function, which you can specialize in a scripted class. Tables 5-1 and 4-2 describe how `pipe` is implemented by certain classes in the core classes.

Table 5-1: Pipe forms

Piping From	Results in
instance of `Collection`	each item in the collection is piped to the class or function on the *to* side of the pipe operator.
instance of `Range`	each number in the range is piped to the class or function on the *to* side of the pipe operator.

Table 5-2: Pipe forms

Piping To	Results in
`Collection` class	a new instance of that class is created and the source items are collected into that instance.
instance of `Collection`	source items are appended to the end of the existing `Collection` object.
a function	source items are used as the arguments to the function and an instance of `LinkedList` is used to collect the results. The function being piped to must be defined to be a function with only one argument.

The following examples demonstrate the use of pipes:

```
1 to 5 | LinkedList
⇨  #(1, 2, 3, 4, 5) as LinkedList

#(3,4,5,6) | #(1,2)
⇨  #(1, 2, 3, 4, 5, 6)

function double n -> 2 * n
1 to 10 | double
⇨  #(2, 4, 6, 8, 10, 12, 14, 16, 18, 20)

#(1, 2, 3, 4) | (n -> 2 * n)
⇨  #(2, 4, 6, 8)
```

You can also “cascade” pipe expressions, that is, string two or more pipe expressions together.

```
function squares n -> n * n
1 to 5 | double | squares
⇨  #(4, 16, 36, 64, 100)
```

A useful trick is to pipe a long collection to the `print` function. The collection would otherwise be displayed in truncated form. The following example collects a list of generic functions that the `Rect` class implements. Only the first few values are reproduced here.

```
getAllGenerics Rect | print
⇨  init()
   localEqual()
   prin()
```

```
    morph()
    copy()
    . . .
```

As an intermediate step, the collection could be piped to `SortedArray`, so that it is alphabetized.

```
getAllGenerics Rect | SortedArray | print
⇨   bboxGetter()
    copy()
    drawSelf()
    finalize()
    heightGetter()
    heightSetter()
    . . .
```

This script creates a list of classes of objects that implement the generic function `pipe`. Note the use of an anonymous function. The anonymous function is especially useful here as a "package" around `canObjectDo`, a generic function that requires two arguments. Since `pipe` works only with functions requiring one argument, the anonymous function solves the problem. It takes the one argument supplied by `pipe`, passes it to `canObjectDo`, and supplies `pipe` as the second argument to `canObjectDo`. The result of the anonymous function is then passed on to the next pipe.

```
getSubs RootObject | (y -> if canObjectDo y pipe then y else empty) \
       | SortedArray | print
⇨   AccessoryContainer
    ActionList
    ActuatorController
    ApplyTree
    Array
    ArrayList
    Bounce
    Btree
    ByteString
    . . .
```

C H A P T E R

Defining Classes and Objects

6

Up to this point, discussion has centered on simple examples using existing classes and methods in the ScriptX core classes. ScriptX also allows you to define your own classes, and to specialize individual instances of classes.

Classes and objects created in the ScriptX language using the syntax in this chapter are fully-featured, first-class objects. This means that the ScriptX language, the parser, and the compiler treat user-defined classes and objects in the same way as built-in system classes and objects. This allows you to easily build powerful specialized extensions to ScriptX on top of the core ScriptX implementation.

This chapter describes the syntax for developing new singly- and multiply-inherited classes and objects, and for defining methods and variables within them. It also describes some methods that are commonly overridden in subclasses so that those classes can interact effectively with the language and with other classes in the ScriptX core classes library.

Defining Classes

The most common way to extend the power of ScriptX is to define new subclasses that specialize the existing core classes. You can even create your own hierarchy of classes for whatever application your ScriptX program requires. Defining your own classes allows you to create comprehensive class libraries that can be re-used in any modules and scripts.

Syntax Summary

To define a new class, use the `class` expression.

▼ `class` *variableName* `(` *classList* `)`

```
[ class variables
. . . ]
[ instance variables
. . . ]
[ class methods
. . . ]
```

```
   [ instance methods
   . . . ]
end
```

The `class` expression has an introductory clause followed by four main sections for defining the features of the new class: `class variables`, `class methods`, `instance variables`, and `instance methods.` Each part of the expression is described in the following sections.

You must specify the sections of the `class` expression in the order in which they appear above, although you do not need to include all of them. Use only the sections you need in order to create your specialized subclass. Each section can be on a separate line, as shown in the syntax, or all can be on the same line, or the two styles can be used in any combination.

Classes and Variable Names

The first clause of the `class` expression specifies the name of the new class and the inheritance of this class.

▼ `class` *variableName* `(` *classList* `)`

```
   . . .
end
```

lThe *variableName* part of this line specifies the name of the variable to which this class object will be assigned once it has been created. Like all variables in ScriptX, if the variable named by *variableName* has not been previously declared, it is automatically declared global. You can specify that the variable name is to be declared locally by putting the reserved word `local` in front of the reserved word `class`. Note that just as with local variables, you can only define a local class inside a local scope (that is, not at the top level).

```
(
   local class SortedLinkedList (LinkedList)
   end
)
```

This example creates a subclass of the class `LinkedList` (with no further specialization) and assigns it to the local variable `SortedLinkedList`.

In addition, the variable to which a class is assigned is declared `constant`, which means that you cannot assign anything else to it once the class has been created. This is to prevent accidentally overwriting the definition of a class with a simple assignment. You can, however, redefine a class using the name of the original class. (Some restrictions apply; see “A Note on Redefining Classes” on page 117 for more information.)

Classes and Inheritance

The first clause of the class expression also specifies the inheritance of this class.

▼
```
class variableName ( classList )
    . . .
end
```

The *classList*, within parentheses, specifies a list of classes from which this class inherits (that this class is a subclass of). The *classList* can be a single class, or a list of classes separated by commas. If you specify multiple classes in *classList*, your new class multiply inherits from all those classes. See "Multiple Inheritance" on page 127 for more information on classes with multiple superclasses.

If *classList* is not specified, (the parentheses are empty), then the class is a direct subclass of `RootObject`.

A Note on Redefining Classes

You can redefine a class by using the `class` expression with the same name as a previously-defined class. There are several things to note about redefining classes:

- You cannot redefine any of the ScriptX core classes. Although you cannot redefine core classes, you can add additional methods to a class, and you can redefine methods. Generally, you should redefine methods in a scripted subclass, since other classes and objects may depend on the class's default behavior.
- Each time you use the `class` expression to define a class, all previous definitions of that class are replaced. You cannot incrementally add to the definition of a class using multiple `class` definitions.
- You cannot redefine a class that has existing instances, since those instances will not work with the new class. To remove instances of a class, you must remove all references to those instances. Removing those references may include setting global variables that hold those instances to some other value (for example, `undefined`), or removing instances from other structures such as collections. Once all references to an instance are gone, the ScriptX garbage collector removes that instance.

Class and Instance Variables

Class and instance variables specify the properties or state of a given class or object. You can think of class variables as instance variables defined directly on the class object itself. (Technically, that is exactly what they are.)

Instance variables are variables whose values can vary from instance to instance. For example, the class `Car` may define an instance variable called `color`. Each instance of `Car` can then have its own value for `color`.

Class variables are variables that relate to the class itself, and are accessible to and shared by all the instances of a class, including the class itself. For example, the class `LuxuryCar` might have a `features` class variable that holds an array of features common to all instances of luxury cars, such as `@antiLockBrakes` or `@woodPaneling`. Those features are shared by all instances of `LuxuryCar`, and if the value of that variable is ever changed, it affects all those instances equally.

Both class and instance variables are accessed through the use of special generic functions called setters and getters. Setters change (or set) the value of a variable; getters query for (or get) the current value of a variable. ScriptX language constructs for querying and changing class and instance variables are described beginning on page 63. Use these generic functions to indirectly gain access to the variable itself.

ScriptX provides several mechanisms for defining class and instance variables with many different features, including read-only variables, variables that automatically calculate their values when they are queried by some other method in the class or object, or variables that discard their values when stored on disk. Each of these more complex forms for defining variables, as well as details on setter and getter generic functions, is described in detail in "Defining Class and Instance Variables" on page 145.

The simplest way to create a class or instance variable in a class is to use either the `class variables` or `instance variables` sections of the `class` expression.

▼ `class` *variableName* `(` *classList* `)`

```
   class variables
      varName
      varName:initialValue
      . . .
   instance variables
      varName
      varName:initialValue
      . . .
end
```

The reserved words `class variables` can be shortened to `class vars`. Similarly, the reserved words `instance variables` can the shortened to `inst variables` or simply `inst vars`. Each short form is equivalent to its longer counterpart. After these reserved words, you can specify any number of variable names (here, *varName*) on separate lines, or all on one line separated by commas, or in any combination.

You can also specify initial values for those variables (*initialValue*) by supplying a colon and the value after the variable. The initial value of a class variable is assigned when the class is created; the initial value for an instance variable is assigned each time a new instance of that class is created. If you do not specify initial values for a class or instance variable, its value is `undefined`.

Each of these forms creates "real" instance variables, that is, variables that exist as slots in memory. They also construct pairs of setter and getter functions automatically for each of the variables listed when the class is created. For more details on the setter and getter functions, see "Defining Class and Instance Variables" on page 145.

The following example shows the definition and use of a class variable:

```
class PurpleDragon ()
    class vars numDragons : 0
    instance methods
        method moreDragons self -> (
            (getClass self).numDragons := (getClass self).numDragons + 1
        )
end
```

This example defines a class `PurpleDragon`. Since no class is specified in the class inheritance list, the class inherits directly from `RootObject`, the root system class. The `PurpleDragon` class defines a class variable, `numDragons`, which is initially set to `0`. It also defines an instance method to increase the value of the class variable `numDragons` by one. Instance methods are described in the section "Class and Instance Methods" on page 120.

Now, you can instantiate `PurpleDragon` and call `moreDragons` on any of those instances. The `numDragons` class variable is increased:

```
dragon1 := new PurpleDragon
dragon2 := new PurpleDragon
PurpleDragon.numDragons -- the initial value
⇨  0
moreDragons dragon1
⇨  1
moreDragons dragon2
⇨  2
PurpleDragon.numDragons -- find out new value
⇨  2 i
```

A Note on Class Variables and Inheritance

Class variables, like instance variables, are inherited from class to subclass. However, each class that inherits a class variable from a superclass gets a different version of that class variable; all it inherits is the name. It provides its own "slot" for that variable. This differs from C++, in which class variables are shared by a class, all its instances, and all its subclasses and their instances.

For example, suppose you have a class `LuxuryCar`, which defines a class variable `paneling`, whose default value is `@oak`.

```
class LuxuryCar ()
    class variables paneling:@oak
end
```

Now, suppose you created a subclass of `LuxuryCar` called `BargainLuxuryCar`. The `BargainLuxuryCar` class inherits the `paneling` class variable (and its value) from `LuxuryCar`:

```
class BargainLuxuryCar (LuxuryCar) end
BargainLuxuryCar.paneling
⇨ @oak
```

However, because each class owns its own slot for class variables, even inherited ones, if you change the value of `LuxuryCar`'s class variable `paneling`, `BargainLuxuryCar`'s version of the class variable `paneling` is not affected.

```
LuxuryCar.paneling := @teak
⇨ @teak
BargainLuxuryCar.paneling
⇨ @oak
```

Class and Instance Methods

Methods are implementations of generic functions that are defined directly by an object or class, or that are accessible to an object or class through inheritance. Class methods operate on the class object itself, and so are related to the class as a whole (much like class variables). Instance methods operate on an instance of a class. To define class or instance methods, use the `class methods` or `instance methods` sections of the class expression:

▼
```
class variableName ( classList )
    class methods
        methodDefinition
        . . .
    instance methods
        methodDefinition
        . . .
end
```

The reserved words `instance methods` can be shortened to `inst methods`; both forms are equivalent.

Both the `class methods` and `instance methods` reserved words are followed by any number of individual method definitions. Method definitions are very similar to function definitions—see "Defining Methods" on page 129 for more information.

One of the most commonly defined instance methods is the `init` method, which determines how an instance of a class is to be initialized when the `new` method is called on the class. For detailed information on defining initialization methods, see the discussion of the `new`, `init`, and `afterInit` methods that begins on page 136.

Defining Objects

The best way to extend ScriptX is through the use of specialized subclasses, which can then be reused in many different situations.

However, there may be circumstances where you do not want to create a new subclass, for example, if you want to change only one instance of a class to have some specific behavior. For this reason, ScriptX also allows specialization of individual instances through the `object` expression.

Chapter 3, "Working with Objects," described simple uses of the `object` expression to create an object and to set its instance variables. This section describes the full form of the expression, which allows you to define instance variables and methods, set initial instance variables, and provide a list of initial elements for container objects such as arrays.

Syntax Summary

The syntax for the `object` expression has an initial definition clause, a set of optional keyword arguments for initializing the object, and four main sections: `instance variables`, `instance methods`, `settings`, and `contents`.

▼
```
object [ variableName ] ( classList )
       [ keyword:value, keyword:value . . . ]
   [ instance variables
   . . . ]
   [ instance methods
   . . . ]
   [ settings
   . . . ]
   [ contents
   . . . ]
end
```

You must specify the sections of the expression in this order, although all of them are optional. You can use any subset of these sections to create an instance of a class with the specializations you need. Each section can be on a separate line, as shown in the syntax, or all on the same line, or in any combination.

Objects and Variable Names

The first clause of the `object` expression includes an optional *variableName*, which specifies the variable that the object is assigned to when it is created.

▼ `object [` *variableName* `] (` *classList* `)`

```
   . . .
end
```

Like all variables, if the *variableName* has not been previously declared, it is automatically declared `global`. You can specify that *variableName* is to be declared locally by preceding the `object` expression with the `local` reserved word. Note that just as with local variables, you can only define a local object inside a local scope (that is, not at the top level).

```
(
local object myList (LinkedList)
   . . .
end
)
```

The expression above creates a new instance of the class `LinkedList` (by calling `new` on `LinkedList`) and assigns it to the local variable `myList`.

The variable to which this object is assigned is automatically declared `constant`, which means that you cannot assign anything else to it once the object has been created. You can, however, redefine this variable to hold another object by using the `object` expression again with that same variable name.

If you leave off *variableName*, the `object` expression simply yields an object of the appropriate class. This allows you to nest the `object` expression within other expressions:

```
myShape := new TwoDShape \
    boundary:(object (Rect) x2:30, y2:20 end) \
    stroke:(object (Brush) color:blackColor, pattern:grayPattern \
           settings linewidth:3 end)
```

Objects and Inheritance

The first line of the `object` expression also specifies the class or classes this object is an instance of.

▼
```
object [ variableName ]( classList )
   . . .
end
```

The *classList*, which must be within parentheses, specifies a list of classes from which the new object inherits. This list can be a single class or a list of classes separated by commas. If you specify multiple classes in *classList*, all of those classes contribute to the newly created object. See "Multiple Inheritance" on page 127 for more information on objects with multiple superclasses.

If *classList* is not specified (that is, the parentheses are empty), then the object is a special instance of `RootObject`.

Redefining Objects

Each successive use of the `object` expression, using the same variable name, makes the old definition unavailable, effectively overwriting any specializations you may have made to that object.

You can add method definitions to any existing object using free method definitions. Free methods are described in the section "Free Method Syntax" on page 133.

Initialization Keywords

▼
```
object [ variableName ] ( classList )
   [ initKey:argument, initKey:argument . . . ]
   . . .
end
```

Keyword arguments, as described in Chapter 3, "Working with Objects," are used to define the initial parameters of a new object, which may include setting the initial values of some of its instance variables. The keyword arguments that an object requires are defined by that class's `init` method and by the `init` methods defined by all its superclasses, for objects with multiple superclasses. You can look up a particular class in the *ScriptX Class Reference* for a list of keyword arguments that it uses.

Keyword arguments can be specified in any order, and many of them are optional. If you do not specify an optional keyword argument in the `object` expression, those keywords are given default values when the object is initialized (as defined by that class's `init` method). Again, see the *ScriptX Class Reference* for a list of keyword arguments for each class, including default values.

Instance Variables

The `instance variables` section of the `object` expression is used to add new instance variables to an object, just as it was used to add new instance variables to a class in the `class` expression.

▼ `object` [*variableName*] (*classList*)

```
   instance variables
      varName
      varName:initialValue
      . . .
end
```

The `instance variables` reserved words can be shortened to `instance vars`, `inst variables`, or simply `inst vars`. All forms are equivalent.

Instance variable names can be specified either as names only (in which case their initial values are undefined), or they can be initialized by specifying a *varName*:*initialValue* pair separated by a colon. The *initialValue* of the variable can be any expression, including nested `object` definitions:

```
object lotsOfVars (Player)
   instance variables
      firstIV
      secondIV:@cowabunga
      thirdIV:object (PushButton) inst vars pb1 end
end
```

You can specify variable definitions on separate lines, on the same line separated by commas, or in any combination.

```
object foo (Array)
   instance vars
      a, b
      c:"purple"
      d:3, e:4
end
```

Instance Methods

▼ `object` [*variableName*] (*classList*)

```
   instance methods
      methodDefinition
      . . .
end
```

The `instance methods` section of the `object` expression, followed by any number of method definitions, adds new methods or redefines existing methods for the given object. See "Defining Methods" on page 129 for more information on defining methods.

The reserved words `instance methods` can also be specified as `inst methods`. Both forms are equivalent.

Settings

▼ `object` [*variableName*] (*classList*)

```
   settings
      varName: value
      . . .
end
```

The reserved word `settings` allows you to set initial values for instance variables defined by the class or classes this object is an instance of. You can also use the `settings` section to specify values for new instance variables you defined in an `instance variables` section. (Adding your own instance variables or instance methods to an object definition is called specialization.)

You can specify variable assignments on separate lines, on the same line separated by commas, or in any combination.

```
-- first create the Novel class as a template for this example
class Novel () inst vars author, title, characters end

-- now create an instance of book
object myNovel (Novel)
   inst vars
      authorsSister
   settings
      author:"Emily Bront\<00e9>"
      title:"Wuthering Heights"
      characters:#("Heathcliff","Catherine")
      authorsSister:"Charlotte Bront\<00e9>"
end
```

Using the `settings` clause is equivalent to first creating the object and then setting its instance variables with separate assignment statements. The following code is equivalent to the example above except that it does not include the specialization of adding the instance variable *authorsSister*.

```
myNovel := new Novel
myNovel.author := "Emily Bront\<00e9>"
myNovel.title := "Wuthering Heights"
myNovel.characters := #("Heathcliff","Catherine")
```

Using `settings` has the advantage of using fewer expressions to accomplish the same thing, and of having the new object be initialized with its instance variables already set to the values you require.

Contents

▼ `object` [*variableName*] (*classList*)

```
    contents
        element, element, element, . . .
end
```

The `contents` section of the `object` expression is used to specify the initial contents of objects that can hold other objects, such as collections. Chapter 7, "Collections", describes this set of utility classes and how to use them.

Each of the elements in the `contents` section is added to the object using the `addToContents` method. Objects that inherit from `Collection` have a default version of this method. If you are using a class or object that does not inherit from `Collection`, you must implement the `addToContents` method for the `contents` section to work. You can also override `addToContents` in this object, or in a class that this object inherits from, to change the default behavior (for example, to print a debugging message as each element is added).

A Note on Special Classes Created by object

In ScriptX, every object is by definition an instance of some class and receives its behavior and state information from that class. If an object is created by the `object` expression and has specializations (additional instance variables or methods) or is an instance of multiple classes, then a class for that special object does not truly exist.

To get around this paradox, when you use the `object` expression, ScriptX creates a special "hidden" class for the object created. In most cases you do not need to refer to or use these special classes. Be aware, however, that using the `getClass` function on a specialized object (particularly one that is an instance of multiple classes) may result in unusual results:

```
object specialObject (TwoDShape, Dragger) end
getClass specialObject
```

⇨ `TwoDShapeDragger`

Multiple Inheritance

Multiple inheritance allows a subclass or instance to be created that inherits behavior from multiple superclasses. The new subclass or object inherits both class and instance variables and methods from each superclass, allowing you to combine the behavior of each superclass into a single subclass or object.

To create a new class or object that inherits from several superclasses, list the classes in the first line of the class or object definition, separated by commas:

▼ `class` *variableName* `(`*class*`,` *class*`,` *class*`, . . . )`

▼ `object [` *variableName* `] (`*class*`,` *class*`,` *class*`, . . . )`

The newly created class inherits behavior and properties from all of its superclasses. This has the following effects:

- The keyword arguments of the multiply-inherited class or object are the set of all the keyword arguments that were available in each original class. All required keyword arguments are still required by the new class or object.
- All class and instance variables inherited from each of the superclasses are available to the new class or object.
- All methods available from each superclass are available to the new class or object. However, if more than one of the superclasses defines a method with the same name, the superclass that receives the generic function call for that method is determined by the order in which the classes are specified in the original class list.

Inheritance in multiply inherited ScriptX classes operates in a *depth-first topological* order. This means that all the superclasses of the first class in the list are searched before the superclasses of the next class in the inheritance list, and so on up the inheritance chain. To view this order , call `getSupers` on the new class.

You can control the inheritance of your subclass or object by controlling the order of its superclasses in the `class` or `object` definition. For example, if your class inherits from two classes, you can often simply reverse the order of those classes if the wrong method is being invoked first. Figure 6-1 shows a simple hierarchy of classes that illustrates the meaning of depth-first topological ordering.

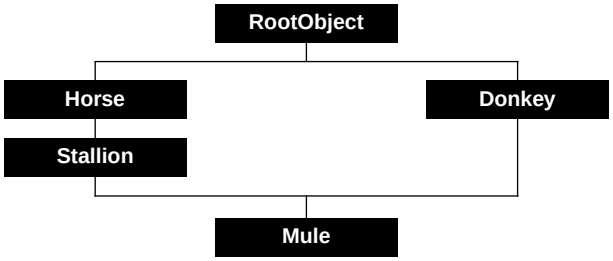


Figure 6-1: An illustration of depth-first topological order.

Here are class definitions for `Horse`, `Stallion`, and `Donkey`, the parent classes of `Mule`:

```
class Horse () end
class Donkey () end
class Stallion (Horse) end
class Mule (Stallion, Donkey) end
```

If `Mule` is defined so that it inherits first from `Stallion`, then methods defined by both `Stallion` and `Horse` take precedence over methods defined by `Donkey`. To check the specific order in which ScriptX searches classes for a given generic function, you can use the `getSupers` generic function. `getSupers` returns an array of the superclasses of a given class in depth-first topological order, which is the order in which they are searched.

```
getSupers Mule
⇨ #(Stallion, Horse, Donkey, RootObject)
```

If `Mule` inherits first from `Donkey`, then methods defined by `Donkey` take precedence over methods defined by either `Stallion` or `Horse`. Notice how the ordering of classes reported by `getSupers` changes when we redefine the `Mule` class.

```
class Mule (Donkey, Stallion) end
getSupers Mule
⇨ #(Donkey, Stallion, Horse, RootObject)
```

We can use `getSupers` to determine the inheritance of any ScriptX class, no matter how complicated. Here are several examples drawn from the core classes.

```
class BouncyArray (Projectile, Array) end
getSupers BouncyArray
⇨ #(Projectile, Array, Sequence, LinearCollection,
   ImplicitlyKeyedCollection, Collection, RootObject)

class WeirdInheritance (QueuedEvent, TwoDShape, LinkedList) end
prin ((getSupers WeirdInheritance) as Array) @complete debug
⇨ #(QueuedEvent, Event, TwoDShape, TwoDPresenter, Presenter,
   LinkedList, Sequence, LinearCollection, ImplicitlyKeyedCollection,
   Collection, RootObject
```

The following example demonstrates the importance of inheritance order in classes and objects. The `BassetHound` and `PitBull` classes, subclasses of `Dog`, have their own implementations of the generic function `getTemper`.

```
class Dog () inst vars name end

-- create subclasses of Dog for the breeds BassetHound and PitBull
class BassetHound (Dog)
    instance methods
    method getTemper self -> (
        format debug "%*'s temper is good.\n" (self.name) @unadorned
    )
end

class PitBull (Dog)
    instance methods
    method getTemper self -> (
        format debug "%*'s temper can be bad.\n" (self.name) @unadorned
    )
end
```

`Frookie` and `Noodle` are both mutts. `Noodle` inherits first from `BassetHound`, so his temperament is usually good. `Frookie` inherits first from `PitBull`, so his temperament can be very bad.

```
object Frookie (PitBull, BassetHound) settings name:"Frookie" end

object Noodle (BassetHound, PitBull) settings name: "Noodle" end

-- test the getTemper method on both dogs
getTemper Frookie
⇨  Frookie's temper can be bad.
getTemper Noodle
⇨  Noodle's temper is good.
```

Defining Methods

This section describes the ScriptX syntax for defining methods. It also describes common ways of using method definition and redefinition to make your class effectively interact with other classes.

- "Method Definition Syntax" on page 130 describes the general syntax for defining methods.
- "Free Method Syntax" on page 133 describes the free method syntax, which allows you to add methods to existing classes and objects from outside the `class` and `object` expressions.
- "Overriding Methods" on page 134 describes how to override or augment methods defined elsewhere in the inheritance hierarchy. It also explains why you might want to do that.

Methods and Generic Functions

You call a method by calling a generic function on an object; the generic function redirects this function call to the appropriate method. Chapter 3, "Working with Objects," describes the relationship between methods, defined by classes and objects, and generic functions, which are used to access those methods. For a definition of generic functions, see page 19.

When you define methods using the expressions described in this section, you do not have to do anything extra to create a new generic function or to make sure the existing generic function knows about your new definition. ScriptX automatically updates the appropriate generic function for your method definition or creates a new generic function for new methods.

Note – The generic function you name, and thus, the one you specialize, is the one that is visible in the current module. You can have many different generic functions of the same name in different modules. For information on names and modules, see "Modules" on page 187.

Method Definition Syntax

Method definitions are almost identical to function definitions. Unlike functions, method definitions require at least one argument—the object or class on which the method is invoked. That argument is typically called *self*, and it is always the first argument to that method.

Because method and function definitions are so similar, this section provides only a general summary of the syntax itself. For more details, see the section "Defining Functions" on page 99 of Chapter 5, "Functions, Threads and Pipes." Methods are defined using the following general syntax.

▼ `method` *methodName* `self` *arguments* `->` *body*

where:

- *methodName* names the generic function which invokes this method.
- `self` holds the current object on which this method operates. (Any other argument name can be used here besides `self`; the word `self` is used by convention.)
- *arguments* are required positional arguments, rest arguments, or keyword arguments. Examples of each form are shown on page 131.
- *body* is an expression, often a compound expression, that is executed when the method is invoked.

A method can also have keyword arguments, as described in "Defining init Methods" on page 138. However, since they complicate the syntax and are often not needed, they are described later.

As elsewhere, you must refer to class and instance variables in the body of a method definition using the standard class and instance variable access expression (`self.`*variable*). ScriptX does not provide a mechanism for automatically accessing class and instance variables by name within a method definition.

Methods, like functions, can use the `return` expression to specify the value the method returns. Without an explicit return, the method returns the value of the last expression evaluated. When in doubt on a return value, consider returning `self`, `undefined`, or `OK`.

```
object myObj (RootObject)
    inst vars a:500
    inst methods
    method incrementA self inc -> (
        self.a := self.a + inc
    )
end

class GenericClass (RootObject)
    instance methods
    method printClass self -> (
        print ("I'm an instance of " + (getClassName self))
    )
    method printMe self -> (
        format debug "This is me: %*.\n" self @normal
        printClass self
    )
end

global myGenericClass := new GenericClass
printMe myGenericClass
⇨  This is me: GenericClass@0x1161b7c
    “I’m an instance of GenericClass”

class MyLL (Array)
    instance methods
    method addItemToBeginning self item -> (
        addNth self 1 item
        format debug "%* added " item @normal
        return self
    )
end

global myLinkedList := new MyLL
addItemToBeginning myLinkedList 12345
⇨  12345 added #(12345) as MyLL
```

More Examples

The following class, `MyClass`, defines three methods:

- `addAllIVs`, which takes no required arguments (except `self`) and sums the values of the instance variables `a`, `b`, and `c`.
- `addEmUp` takes a `#rest` argument, and sums all the values of its arguments, then adds the value returned by `addAllIVs`.

- `changeIVs` takes three keyword arguments (`incA`, `incB`, `incC`), for which the default values are all 10, and increments the values of `a`, `b`, and `c` with the appropriate increment.

```
class MyClass ()
    instance vars a,b,c
    instance methods
    method addAllIVs self -> (
        self.a + self.b + self.c
    )
    method addEmUp self #rest allArgs -> (
        local s := 0
        for i in allArgs do (s := s + i)
        s := s + (addAllIVs self)
    )
    method changeIVs self #key incA:(10) incB:(10) incC:(10) -> (
        self.a := self.a + incA
        self.b := self.b + incB
        self.c := self.c + incC
        print self.a; print self.b; print self.c
        return self
    )
end
```

Now that the class and its methods have been defined, here are some examples of its use:

```
object exmpl (MyClass)
    settings a:1, b:5, c:12
end
exmpl.a
⇨ 1
exmpl.b
⇨ 5
exmpl.c
⇨ 12
addAllIVs exmpl
⇨ 18
addEmUp exmpl 3 8 4 6
⇨ 39
changeIVs exmpl -- no keywords, defaults are all 10
⇨ 11
    15
    22
changeIVs exmpl incA:4 incB:-7
⇨ 15
    8
    32
```

Free Method Syntax

Free methods are methods that can be defined outside the boundaries of a class or object definition. Free methods are useful for adding or redefining method definitions in existing classes or objects without having to redefine the entire class or object.

Free method definitions look similar to regular method definitions, with the addition of a special clause that specifies the class or object to which this method belongs. There are two forms of method definition: one for adding instance methods to an object, and one for adding either class or instance methods to a class.

▼ `method` *methodName* `self { object` *object* `}` *args* `->` *body*

▼ `[ class ] method` *methodName* `self { class` *class* `}` *args* `->` *body*

In both forms, *methodName* is the name of the generic function that invokes this method. If the `class` reserved word is included before the `method` reserved word, that method is a class method.

The *args* part of each free method definition supplies the arguments this method takes (besides `self`). These can be positional arguments, rest arguments, or keyword arguments. See Chapter 5, "Functions, Threads and Pipes," for a description of each of these types of arguments.

The expression within braces is called a restriction, and is used to point to the class or object that the method is associated with. Note that the restriction must come after the `self` argument, but before any other arguments.

Finally, the *body* part of the method definition is the expression, often a compound expression, that the method evaluates when invoked.

The first form adds an instance method to the object specified by *object*. The expression in braces holds an object, and can be one of the following expressions, where appropriate:

- a variable name
- an anonymous function definition with no arguments
- an array access expression (`myArray[1]`)
- a reference to a class or instance variable (`self.x`)
- any other expression, contained within parentheses

Here is an example:

```
tryThisOut := #(1,2)
method appendSum self {object tryThisOut} n m ->
   (append self (n + m); return self)
appendSum tryThisOut 3 4
⇨  #(1,2,7)
```

The second form adds either an instance method or a class method to the class specified by *class*, which can contain one of the expressions from the list above for *object*.

```
class MyClass () end
method jellydonut self {class MyClass} name ->
    format debug "%* is not a jellydonut!\n" name @unadorned

i := new MyClass
jellydonut i "cruller"
⇨  "cruller is not a jellydonut!"
```

Adding Methods to the ScriptX Core Classes

Using free method definitions, it is possible to add new methods to those ScriptX core classes that are not sealed. This allows you to extend the behavior of those classes, and of classes that inherit from those classes.

When adding methods to the core classes, be careful not to override existing methods. That is, avoid defining a method of the same name as one that already exists in that class, particularly initialization (`init` and `afterInit`) methods. A class may depend on internal behavior defined by those methods; overriding those methods may cause errors in the operation of that class.

As an alternative to overriding existing methods in the ScriptX Core Classes, consider creating a subclass of that class with your own definitions instead.

Overriding Methods

The previous sections described how to create entirely new methods for your class or object. However, the other important use of method definitions is to override an existing method. Overriding a method means providing a different implementation for an inherited method. This can happen either in class specialization or instance specialization. For an example of class specialization, `TwoDShape` defines the `draw` method; a subclass of `TwoDShape` that re-implements the `draw` method is said to override the `draw` defined in its superclass. For an example of instance specialization, an instance of `TwoDShape` that re-implements the `draw` method is said to override the `draw` defined in its class.

When you override an existing method, your method must have the same name and positional arguments as the original method, but can have new keyword arguments. In addition, the method can use the functionality provided by superclasses by calling `nextMethod`. As such, there are two ways to override an existing method:

- If you want *add to* the existing functionality provided in superclasses, call `nextMethod`.
- If you want to *replace* (rather than add to) the functionality provided by the superclasses, don't call `nextMethod`. Use this approach when the behavior you want is not an addition to the existing behavior. However, some methods (`init`, `afterInit`) require that you call `nextMethod`.

The first option reuses the existing implementation, while the latter does not. In either case, you can provide your own specialization in the body of the method. The latter option was described in the previous section "Method Definition Syntax" on page 130.

You use `nextMethod` to call the overriding method from the body of a method, like this:

- `method` *methodName* `self` *arguments* `-> (`
  `. . .` *optionally do something here* `. . .`
  `nextMethod self` *arguments*
  `. . .` *optionally do something else here* `. . .`
  `)`

The `nextMethod` expression passes the call upward through the inheritance hierarchy, calling methods with the same name, so that each superclass can invoke its own implementation of that method on the instance. Another way to look at this is that invoking `nextMethod` simply allows you to call the original methods that would have been invoked had you not overridden. Each `nextMethod` encountered calls the next method up the chain, hence the term `nextMethod`.

Since `nextMethod` calls methods in superclasses, you should supply to `nextMethod` any arguments that you want those superclasses to handle. For example, since the `draw` method in `TwoDShape` takes three arguments (`self`, `surface`, `clip`), when you create a subclass of `TwoDShape` where you override `draw`, you would call `nextMethod` with those same arguments:

```
method draw surface clip -> (
    nextMethod self surface clip
)
```

In another example, the following method definition overrides the `append` method for a given object such that the original `append` is called only if the item to be appended to this object is an instance of the `ImmediateInteger` class. (The `append` method is defined in the `Sequence` class.) Otherwise, an error message is printed to the `debug` stream and the method returns `undefined`:

```
global myArrayOfIntegers := #()

-- override the append method on myArrayOfIntegers
method append self {object myArrayOfIntegers} item -> (
    if (getClass item = ImmediateInteger) then (
        nextMethod self item
    )
    else (
        format debug "Not an ImmediateInteger: %*\n" item @normal
        return undefined
    )
)
```

Multiple Inheritance with nextMethod

Technically, because ScriptX supports multiple inheritance, a method can have more than two chains of inheritance up to `RootObject`. In this case, `nextMethod` calls the next method in depth-first order, as described in "Multiple Inheritance" on page 127. Use `getSupers` to see the order in which the methods are called. For example, given a class that inherits from both `TwoDShape` and `Bounce`, `nextMethod` calls the classes in the order shown here:

```
class BouncingShape (TwoDShape, Bounce)
end

getSupers BouncingShape | print
⇨  #<Class Substrate:Bounce>
   #<Class Substrate:TwoDController>
   #<Class Substrate:Controller>
   #<Class Substrate:IndirectCollection>
   #<Class Substrate:Collection>
   #<Class Substrate:TwoDShape>
   #<Class Substrate:TwoDPresenter>
   #<Class Substrate:Presenter>
   #<Class Substrate:RootObject>
```

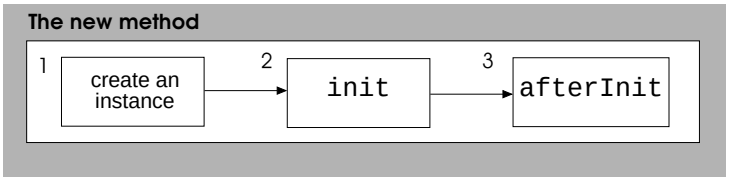
Note – The ScriptX core classes provide mechanisms to define additional behavior for certain methods and protocols in the core classes. For example, if you are interested in overriding any of the methods that add or delete items in collections, as the example above does, consider the advantages of using the `IndirectCollection` class. For more information, see the "Collections" chapter of the *ScriptX Components Guide*.

Defining the new, init, and afterInit Methods

When you create a new instance of a class using the `new` method or `object` construct, several things happen that create and initialize the instance of that class. (The following sections on initialization apply equally to the `new` method and the `object` construct.) As shown in the figure, the `new` method (1) allocates memory for the new object and sets up other internal values; it then (2) calls `init` on the newly created instance to initialize it, passing along any arguments that had been specified to `new`; it finally (3) calls `afterInit` for post-initialization, again passing along any arguments that had been specified to `new`. Notice that `new` is a class method (operates on a class), while `init` and `afterInit` are instance methods.

Note – While its is routine to call the `new` method, you should never directly call `init` or `afterInit` (which are automatically called by `new`); doing so will not re-initialize the object, and might put the instance into an unknown state or cause unexpected behavior. Calling `init`, in particular, might cause your program to crash.

This section describes `init` and `afterInit`, and how to override them in your own classes to specialize the initialization.



When you create a subclass of an existing class, you may often want to override how it is initialized. To do this, you override the `init` and `afterInit` methods in your subclass. You are not required to define either an `init` or `afterInit` method in your subclass; you only need to define them if your subclass's initialization behavior is different from that of its superclasses.

The `new` method should almost never be overridden. (You could override `new` if you want to limit the number of instances that can be created.)

The `init` method initializes the new instance of its class. Every instantiable class has an `init` method, which contains implementation to set up some of the initial values of a new instance.

Be aware that the instance is in an incomplete state until `init` is done. This is very important to keep in mind. It is not safe to do anything that may have the side effect of calling a method on this instance, which is still in an incomplete state. Calling a method on an incomplete instance can cause your program to crash.

However, the `afterInit` method is a safe place to perform all post-initialization work. Since the object is fully initialized at this point, it's safe to call a method on it. The default `afterInit` method for most core classes is empty. The `Collection` classes have an `afterInit` method that allows you to add initial keys and values to that new collection. Where `afterInit` comes in handy, again, is in subclasses you define.

If you create a subclass of `Window`, you might define `afterInit` to create whatever objects the window is supposed to initially requite or contain, such as pushbuttons, bitmaps, players, and other objects. Then every time `new` is called on that class, the new window will automatically get the objects it needs.

Both `init` and `afterInit` are defined in similar ways and are described in the following sections.

Keyword Arguments

Keyword arguments are arguments in a method or function that have explicit keywords. You can recognize a keyword argument by the colon (:) used to separate the keyword from the value:

keyword`:`*value*

The `new` method for many classes uses keyword arguments. For example, the following code creates a new instance of a `Point` class; it uses the keywords `x` and `y` to intialize the point to coordinates 10,20:

```
new Point x:10 y:20
```

Keyword arguments are described in the following section "Defining init Methods."

Defining init Methods

The definition of the `init` method has the syntax shown below. It is defined to take only one positional argument, `self`, one rest argument *restArg* (which is actually a collection of arguments), and any number of keyword arguments *keywordArgs*. Keyword arguments are not restricted to only the `init` method; any method definition can contain keyword arguments.

◆ `method init self #rest` *restArg* `[ #key` *keywordArgs* `] -> (`

```
    . . .
    apply nextMethod self keywordArgs restArg
    . . .
)
```

where:

- *restArg* is an argument (typically `args`) that holds the collection of keyword arguments passed into `init`.
- *keywordArgs* is an optional series of newly defined keyword arguments, each of which can be renamed with *argName* and can have an initial value which must be inside parentheses. Keyword arguments have this syntax:

  keyName`:[` *argName* `] [ (` *initialValue* `) ] . . .`

- `nextMethod` passes the original method call (including arguments) up the inheritance hierarchy, so that each superclass has a chance to perform its particular initialization on the new instance.
- `apply` calls `nextMethod` with all members of *restArg* as arguments. For more information, see "Using apply to Call Functions" on page 109 in Chapter , "Functions, Threads and Pipes."

As you recall, the `new` method automatically calls `init`. When you call any function or method with a series of keyword arguments, these arguments are wrapped in a collection. In this case, the collection of keyword arguments are passed to *restArg* in `init`.

Use `#key` to define any new keyword arguments. An example of a keyword argument is `scale:(10)`. Notice that the parentheses are required around the default value, even if the value is simply a number and not a complex expression. For a complete description of keyword arguments, see the discussion that begins on page 100 of Chapter 5, "Functions, Threads and Pipes."

In general, the `new` method that calls `init` can contain a variable number of keyword arguments, and should therefore include `#rest` *restArg*, as shown, which collects those keyword arguments together into the collection *restArg*. (It's interesting to note that `#rest` can collect any kind of arguments. It is used to collect keyword arguments in the `init` method, whereas it can be used to collect non-keyword arguments in functions.)

If the `new` method that calls `init` does not contain a variable number of keyword arguments, then you do not need to include `#rest` *restArg*. However, about the only time you can be sure of this constraint is with a non-subclassable (sealed) class that inherits directly from `RootObject`.

Unless you really know what you're doing, there are only two things you can do inside the body of `init`:

- Call `nextMethod`, optionally passing in keywords with initial values along to superclasses
- Assign values to the instance variables that the subclass itself defines

---

Note – When you are defining an `init` or `afterInit` method, you should set only those instance variables that are defined within that class. However, if you find you must set an instance variable that is "owned" by a superclass, you should call `nextMethod` *before* you set its value. Setting an instance variable that is "owned" by a superclass and then calling `nextMethod` after that may result in unexpected behavior, sometimes even a system crash.

It's important to call `nextMethod`, because `init` is implemented in `RootObject`; `nextMethod` ensures that that implementation is called. Notice that it does not matter what `init` returns; its return value is ignored.

Here is a "minimal" implementation of the `init` method which does no specialization:

```
method init self #rest args -> (
    apply nextMethod self args
)
```

In this example, any keywords supplied to the `new` method are passed into `init` as the collection `args`; this collection is then passed up to the `init` method of its superclasses using `apply nextMethod`.

Initializing Keywords and Instance Variables

Notice you have three places you can set initial values in the `init` definition:

- In the keyword arguments that are defined in the `#key` line of `init`. If the same keyword appears in *restArg* and *keywordArgs*, the method accepts only the first one. Therefore, since *restArg* appears before *keywordArgs* at this point, *restArg* takes precedence over *keywordArgs*.
- As keyword arguments to `nextMethod` in the body. Here the precedence order is opposite that of the previous case—since *keywordArgs* appears before *restArg*, *keywordArgs* takes precedence over *restArg*.

- As explicit assignment statements in the body, such as `self.scale := 10`. If this occurs after `nextMethod`, it takes precedence over the two previous places.

Important – It may not be safe in the body of `init` to set instance variables for a superclass or for any other class, or to call methods on any objects, because doing so may have the side effect of calling a method on this instance, which is still in an incomplete state. Calling a method on an incomplete instance can cause your program to crash.

Examples of Overriding init

Here are some examples of class definitions that override `init`:

The `BlahShape` class specializes `Rect` to add an extra instance variable, which is initialized to a linked list.

```
class BlahShape (Rect)
    inst vars
        blahList
    inst methods
        method init self #rest args -> (
            apply nextMethod self args
            self.blahList := new LinkedList
            return self
        )
end
```

The `TextTable` class, suitable for creating a table, specializes `ArrayList` by adding three instance variables. It specializes the `init` method to set values for those instance variables, supplying a default of 10 columns and a name of "Laura's Table". It also supplies a default value for `initialSize` (one of the keyword arguments that `ArrayList` defines) to set a TextTable instance to 30 elements.

```
class TextTable (ArrayList)
    inst vars
        recClass, columns, name
    inst methods
    method init self #rest args \
        #key columns:(10) tabClass: name:("Laura's Table") -> (
            apply nextMethod self initialSize:(30) args
            self.recClass := tabClass
            self.columns := columns
            self.name := name
            return self
    )
end
```

`ArrowPresenter` is a subclass of `TwoDPresenter` that adds two instance variables and overrides by setting values for those instance variables. Its `init` method also supplies a different default value for one of the keyword arguments that the `TwoDPresenter` class uses.

```
class ArrowPresenter (TwoDPresenter)
    instance variables strokePaint, direction
    instance methods
        method init self #rest args ->(
            apply nextMethod self boundary:(new Rect x2:15 y2:15) args
            self.strokePaint := blackBrush
            self.direction := @up
            return self
        )
end
```

Passing Initial Values in init to Superclasses

If you calculate or otherwise determine values for keyword arguments that need to be supplied to the `init` methods defined by superclasses, you can pass those values in keyword-value pairs using `nextMethod`.

For example, if your superclass's `init` method defines a `scale` keyword argument, but you want instances of your class to always have a scale of `15`, you could supply the value `15` to the `scale` keyword in the `nextMethod` call.

```
-- Initial value of scale is always 15, even if passed in another value
method init self #rest args -> (
    apply nextMethod self scale:15 args
    return self
)
```

Now, even if an alternate value for `scale` is supplied when this method is called, a scale of `15` is always used. Both values appear in the `nextMethod` arguments (one explicitly as `scale`, the other is in `args`), but `nextMethod` always chooses the *first* value of the key it finds.

In addition, a subtle point with the `init` method (or with any function or method that defines both rest and keyword arguments) is that the `arg` argument collects only keyword arguments that have been supplied directly in the call to `init`; it does not collect keyword arguments for which you've simply specified default values in the `init` definition.

For example, say that your superclass takes an optional `scale` keyword. Therefore, if you want to be able to override the default value, you might assume that simply specifying a default value to the keyword argumenta after `#key` will work:

```
-- Incorrect implementation
-- This method does not pass the default scale value to its superclasses
method init self #rest args #key scale:(15) -> (
    apply nextMethod self args
)
```

The problem in this example is that unless you specifically give a value for `scale` when you create an instance of your class, the value of `scale` does not get appended to `args`, and so does not automatically get passed to your superclass's `init` method.

To pass the default value of any keyword argument you define in your own `init` method to your superclass's `init`, you must explicitly pass that keyword along in the `nextMethod` call, and supply its current value:

```
-- Correct implementation
-- This method passes the default value of scale to its superclasses
method init self #rest args #key scale:(15) -> (
    apply nextMethod self scale:scale args
)
```

This allows you to specify the default value of `15` to the `scale` keyword argument. It also allows the value of `scale` to be overridden by a value supplied with `new` or `object`.

Defining afterInit Methods

The `afterInit` method is identical in structure to the `init` method. Just as with `init`, the `afterInit` method requires a rest argument, and you must call `apply nextMethod` at some point in the body of `afterInit` so that all superclasses have a chance to add their post-initialization behavior.

However, it is much less restrictive than `init` in what you can implement inside the body, since *self* at this point is fully initialized.

◆ `method afterInit self #rest` *restArg* `[ #key` *keywordArgs* `] -> (`

```
    . . .
    apply nextMethod self keywordArgs restArg
    . . .
)
```

See the previous init method definition for descriptions of *restArg*, *keywordArgs*, and the other elements of this definition.

The following is a simple example using `init` and `afterInit`. The class `SameKey` inherits from `SortedKeyedArray`, but for every key-value pair in the array, each key is exactly the same object. Only the values vary. The `SameKey` class has an instance variable called, appropriately, `key`, which holds the key that is repeated for every key-value pair. When you call `new` on this class, you have two available keyword arguments: `key`, which specifies the value for the `key` instance variable, and `vals`, which holds an array of potential values for the new collection. During initialization of the new object, the value of `key` is assigned to the `key` instance variable, and the contents of `vals` are added to the collection.

```
class SameKey (SortedKeyedArray)
    instance vars
        key
    instance methods
        method init self #rest args #key key: -> (
            apply nextMethod self args
            self.key := key
```

```
        )
        method afterInit self #rest args #key vals: -> (
            apply nextMethod self args
            for i in vals do add self self.key i
        )
end
```

The `init` method defines a keyword argument, `key`, to hold that key, and assigns that value to the instance variable `self.key`. Because adding a new element to a collection requires a method call on that object, an `afterInit` method is required. The `afterInit` method defines an additional keyword argument, `vals`, which holds an array of possible values for the new class. The `afterInit` method then adds the new key-value pair (using the key specified in the `key` instance variable) to the new collection.

When a new instance of `SameKey` is created, it looks like this:

```
newSameKey := new SameKey key:@flavor \
    vals:#(@cherry,@watermelon,@peach)
⇨  #(@flavor:@peach, @flavor:@watermelon, @flavor:@cherry) as SameKey
```

Comparison Functions

ScriptX provides several default functions for comparing objects, as described in Chapter , “Working with Objects.” When you create new subclasses or objects, your new classes may need to specialize those functions so that they behave properly.

For example, instances of `String` can be compared. The ordering used by the `localLt` method defined by the `String` class is based on a lexicographic ordering of Unicode values. (The ASCII character set is a subset of the Unicode character set.) Suppose you want to create a specialized class of strings (`DeutschString`) that handles all those lugubrious vowel sounds in the German language. You want the ScriptX comparison functions to correctly sort German strings like `"Frühling"`, `"Österreich"`, and `"Mädchen"` that contain inflected vowels.

You can specialize the three generic comparison functions listed below by specifying method definitions in your class. These generic functions take two arguments, *self* and another object, and return `true` or `false`.

- `isComparable`
- `localLt`
- `localEqual`

You do not need to redefine all of these to affect the behavior of many comparison functions; you only need to define those that are different from your class’s inherited definition. You can use `nextMethod` to call the original definitions of these functions from within your definition.

isComparable

The `isComparable` generic function returns either `true` or `false`, depending on whether the objects (and they must be objects, not classes) can be compared. By default, `isComparable` returns `true` if each argument is of the same class.

```
(getClass self) = (getClass anotherObject)
```

However, some objects may be comparable even though they are not instances of the same class. For example, numbers are generally comparable. Instances of `Number` are comparable because each number class defines its own versions of `isComparable`, `localLt`, and `localEqual`. If you define a new class, you might want to specialize these three methods in your class so that instances of other classes can be compared with instances of your class.

```
method isComparable self other -> (
    if ((nextMethod self other) == true) or \
        (isAKindOf self MySuperClass)
    then return true
    else return false
)

method isComparable self other -> (
    if ((allIvNames other) contains @target)
    then return true
    else return false
)
```

localLt and localEqual

The `localLt` and `localEqual` generic functions are used by their functional equivalents, `lt` and `equal` (`<` and `=`) to specify whether one object is less than or equal to another in value. They are also used as the basis for many of the other comparison functions, such as `cmp`, `le`, `ge`, and `gt`.

Both `localLt` and `localEqual` generally assume that their arguments are comparable, that is, that `isComparable` returns `true`.

When redefining `localLt` and `localEqual`, be careful not to actually call any comparison functions or operators to compare those same arguments. Because those functions use the methods you are now defining, you can create a circular method call by doing this. For example, do not do this:

```
method localEqual self other -> (
    if (self = other)
        then return true
        else return false
)
```

Note that the test, `self = other`, uses the `equal` function, which then uses `localEqual` to test its arguments. However, you've just redefined `localEqual` to test the arguments using `equal`, and so on.

You can avoid this circularity by using `nextMethod` to compare the objects, or to compare parts of those objects. For example, you might compare the values of instance variables:

```
method localLt self other -> (
    if (isComparable self other)
    then if ((nextMethod self other) == true) and \
           (((size self) < (size other)) == true)
        then return true
        else return false
    )
    else (report Unordered #(self, other))
```

For more information about comparison functions and the ScriptX Comparison protocol, see the chapter "Object System Kernel" in the *ScriptX Components Guide*.

Defining Class and Instance Variables

Previous sections in this chapter described a simple form for defining class and instance variables. This section contains further details on how to define class and instance variable in specific ways, including

- A description of setter and getter functions and their relationship to the variables they are changing and querying
- The difference between real and virtual variables
- Defining real instance variables, including adding special behavior for storing those variables when this class is stored to the ScriptX object store
- Defining and overriding the setter and getter generic functions to create virtual variables or to specialize the behavior of real variables

Setters, Getters, and Real and Virtual Variables

The ScriptX language provides the following forms for accessing or changing class or instance variables:

classOrObject.variableName
classOrObject.variableName := *value*

In contrast with many other object-oriented programming languages, these forms for querying and changing instance variables do not read or write the variable directly. Instead, they call special generic functions called *setters* (to change the variable) and *getters* (to query a variable's value), which then operate on the class or object to retrieve or change the value of that variable.

The use of setters and getters to access a class or object's variables allows a great deal of flexibility in querying or changing that variable.

- You can override the setter and getter functions for your class or object to provide additional behavior when a variable is accessed or changed. For example, you could update other variables that are affected when this

variable changes. You can print debugging messages, or you can make sure that the value to which a variable is being assigned is of a specific class or within a set range.

- Because of the existence of setters and getters, the class and instance variables themselves do not have to actually exist in memory. The setter and getter functions can instead calculate the values of those variables from some other features of the class or object, and return that value.

This last feature, using setters and getters to mimic the existence of a real variable, is known as creating *virtual* class or instance variables. Virtual variables are different from *real* variables, which occupy real locations in memory (and take up space in the final object). Both real and virtual variables appear the same from the outside, and both are accessed in the same way. The only difference lies in their definition and storage within your class or object.

Defining Real Variables

To define real class or instance variables you use the `class variables` or `instance variables` sections of the `class` or `object` definitions, as described earlier in this chapter on page 117 and again on page 124.

▼
```
class variableName ( classList )
   class variables
      varName
      varName:initialValue
      . . .
   instance variables
      varName
      varName:initialValue
      . . .
end
```

In addition to these simple forms where you specify either the name of the variable or the name and its initial value, each variable definition can also specify a set of optional keywords to specify, primarily, how this variable is to be handled when this class or object is stored in the ScriptX object store. Those keywords appear in a specific order before the variable name and its initial value:

▼
```
[ readOnly ] [ transient [ initializer function ] ] \
   [ reference ] varName [ :initialValue ]
```

The qualifiers to each variable are all optional, but they must be specified in the order shown for them to be processed correctly.

The first optional qualifier, `readOnly`, indicates that this variable can be retrieved but cannot be changed. When this variable is created, only a getter method is constructed for it.

The remaining qualifiers are used for classes and objects that are stored into ScriptX storage containers, part of the ScriptX object store. By default, when a class or object is stored, all of its class and instance variables are also stored with their current values. Additionally, when the class or object is loaded from the storage container, all the objects that its instance variables hold are also retrieved.

The remaining three optional qualifiers allow you to specialize how each variable is to behave when stored:

The `transient` qualifier specifies that when the class or object is stored, the value of this variable is discarded. Transient variables have initial values of `undefined` when they are restored, unless an initializer function is specified.

The `initializer` qualifier specifies that when a transient object is restored, the function specified by *function* is executed to set the initial value of this instance variable. (Only transient variables can take an initializer.) The *function* expression can hold any of the following expressions, where appropriate:

- a variable name
- an anonymous function definition
- an array access expression (`myArray[1]`)
- a reference to a class or instance variable (`self.x`)
- any other expression, contained within parentheses

The `reference` qualifier specifies that when the class or object this variable belongs to is restored, the object this variable holds is not restored along with it. Instead, that object is only restored when the variable is queried. The `reference` qualifier is useful for variables that hold very large objects that do not immediately need to be loaded into memory when your ScriptX program starts running.

For further details on the ScriptX object store and how objects are stored, see the *ScriptX Components Guide*.

Defining Virtual Class or Instance Variables

Virtual class or instance variables, as mentioned in the introduction to this section, are variables that occupy no slots in memory and take up no space in the object. Virtual variables do not actually exist; instead, their values are calculated by setter and getter functions when the variable is accessed or queried, and their values returned as if the variable did actually occupy a real location in memory. In this way, virtual variables mimic the appearance of real class or instance variables.

Virtual class or instance variables are particularly useful for variables that may frequently change as a consequence of some other operation on that object. Take, for example, an object that holds other objects such as a collection. That

collection object might have a `size` instance variable that returns the number of elements in that collection. Any operation that adds or removes elements from that array would have to change the value of that instance variable, requiring an extra step (and more time) for each add or removal operation.

If `size` is a virtual instance variable, the size would never be stored anywhere in memory. Instead, when the `size` instance variable is queried, the getter method counts the items in the array and returns that value. No memory is used for that number, and each method that adds or removes elements from the array has one less operation it has to keep track of.

When you define a real class instance variable, ScriptX automatically creates a setter and getter method for that variable when the class or object is created. To define virtual instance variables, you must define setter and getter methods for those variables yourself.

ScriptX has two forms for defining setter and getter methods (two for creating setter methods, and two for creating getter methods):

▼ `method` *variable*`Setter self value ->` *body*
`method` *variable*`Getter self ->` *body*
`method set` *variable* `self value ->` *body*
`method get` *variable* `self ->` *body*

In each definition, *variable* is the name of the class or instance variable. In the first two definitions, the variable name appears just before the words `Setter` or `Getter` (with no space in between). For example, the getter method for the virtual instance variable `x` would be the `xGetter` method. In the second two definitions, the variable name appears after the reserved words `get` or `set`.

Each of the setter and getter forms are equivalent; that is, the `method set` form is equivalent to the *variable*`Setter` form and the `method get` form is equivalent to *variable*`Getter`.

Getter methods always have only one argument, `self`, and setters have two, `self` and the `value` the variable is to be "set" to. The *body* part of each method should always return the value of the virtual instance variable as its last argument.

If the virtual instance variable is to be considered read-only, define only a getter method for that variable and not a setter method.

The names of virtual instance variables should not be defined in a `class variables` or `instance variables` definition; remember, as virtual instance variables, simply the existence of the setter and getter methods gives those variables the appearance of reality.

The following examples show how to define setter and getter methods within a class or object definition. The first example is trivial. It simply says that the instance variable `ten` has a value of `10`.

```
method tenGetter self -> return 10
method tenSetter self value -> (
```

```
    print "Cannot change ten. ten is 10."
    10
)
```

The second example creates a virtual instance variable called `position`, which could be used to query or change the object's position in some two-dimensional space. Here `position` is built from the existing `x` and `y` instance variables. Setting `position`, an array of two elements, is accomplished by setting the values of `x` and `y`.

```
method get position self -> (
    return #(self.x, self.y)
)
method set position self value -> (
    self.x := value[1]
    self.y := value[2]
    return value
)
```

The third example defines a virtual instance variable `lineWidth` that provides direct access to an instance variable that is defined by a member object.

```
method lineWidthGetter self -> (
    return self.stroke.lineWidth
)
```

The `lineWidthGetter` method could be defined as a free method for subclasses of certain core classes, such as `TextPresenter`. (Note that you cannot change a core method in a core class, but you can change a method in a subclass of a core class.)

```
-- create a subclass of TextPresenter and defind lineWidthGetter as a
-- free method on it
class MyTextPresenter (TextPresenter) end
method lineWidthGetter self {class MyTextPresenter} -> (
    return self.stroke.lineWidth
)
-- now create a test of lineWidthGetter
global myTP := new MyTextPresenter \
    stroke:blackBrush fill:whiteBrush \
    boundary:(new Rect x2:200 y2:200) target:"inconceivable"
myTP.lineWidth
⇨ 1
```

Specializing Setters and Getters for Real Variables

As mentioned previously, real variables are also queried and changed through the use of setter and getter methods. By default, a pair of setter and getter generic functions are constructed by ScriptX for each class or instance variable when you create a class or object with the `class` or `object` expressions. You can, however, augment or change the behavior of those setters and getters for real variables, just as you did for virtual variables (for example, to print debugging messages or to change the value of some other part of the class or object).

You can either specialize the setter and getter for a variable that has been defined in this same class or object, or you can specialize the setter and getter for a variable that has been defined in a superclass and inherited by this class or object. To specialize setters and getters, you use the same method definition syntax that you used to define virtual instance variables, as described on page 147.

Specializing Setters and Getters for Inherited Variables

You can override or augment the setter and getter behavior in your class for a variable defined in a superclass in much the same way that you override other methods in other classes. You define a method of the same name in your class, and you call `nextMethod` to pass the method call up the class hierarchy to the appropriate implementation of that method.

For example, suppose you have defined a class called `VerbosePresenter` that inherits from the class `TwoDPresenter`. The `TwoDPresenter` class defines `x` and `y` instance variables to indicate the position of your presenter in a coordinate space. In `VerbosePresenter`, you want to augment the behavior of the setter methods for `x` and `y` to print a message to the debugging stream each time `x` or `y` is changed. You also want to accept floating point values and store them for precision, but round them to the nearest integer value for actual display.

To do this, store the values with higher precision in another set of instance variables which you define. Define setter methods for `x` and `y` to transform the value from floating point to integer, print a message, and then call `nextMethod`, which allows the superclass to actually set the variable:

```
class VerbosePresenter (TwoDPresenter)
    instance variables
        _x, _y
    instance methods
    method xSetter self value -> (
        self._x := value
        local val
        format debug "x is stored internally as %* " value @normal
        format debug "but displayed as %*\n" \
            (val := round value) @normal
        nextMethod self val
    )
    method ySetter self value -> (
        self._y := value
        local val
        format debug "y is stored internally as %* " value @normal
        format debug "but displayed as %*\n" \
            (val := round value) @normal
        nextMethod self val
    )
end
```

As previously noted, you should avoid setting any instance variables which are owned by superclasses before calling `nextMethod`. In the example above, the instance variables which are set (`_x` and `_y`) are not owned by any

superclasses because they are defined in the new class `VerbosePresenter`. Therefore, in this particular case, there is no problem with calling `nextMethod` last.

If you do not specify `nextMethod` as the last expression in the setter or getter method body, you need to explicitly return the value of the variable as the last expression.

```
method xGetter self {class VerbosePresenter} -> (
    local value := nextMethod self
    format debug "x is displayed as %* " value @normal
    format debug "but stored internally as %*\n" self._x @normal
    return value
)
method yGetter self {class VerbosePresenter} -> (
    local value := nextMethod self
    format debug "y is displayed as %* " value @normal
    format debug "but stored internally as %*\n" self._y @normal
    return value
)
```

Specializing Setters and Getters for Variables Defined In the Same Class

In the current version of ScriptX, you cannot truly override the default setter and getter methods for a real class or instance variable defined in that same class or object. The default setter and getter methods, as defined when you create a real variable in a `class` or `object` expression, provide a way of bypassing the normal setter and getter mechanism and directly gaining access to the memory location that stores the value of that variable. When you override the default setter or getter behavior for a variable, you lose the ability to directly access the variable. For example, to change the default behavior of a getter method for the variable `x`, you would have to use the expression `self.x` in the body of that getter method, which calls the getter method for `x` that you just defined, which queries `self.x`, and so on.

You can mimic the effect of augmenting the default setter and getter behavior by implementing your variable as a virtual variable, and then using a "placeholder" variable to hold the actual value of the variable whose behavior you are augmenting.

For example, suppose your class `Person` has an instance variable called `name` which can only have values of the class `String` or of one of its subclasses. Because of this restriction on the possible values of `name`, you need to define a `nameSetter` method that checks the class of the value to which the variable is set to make sure that it is a string, and reports an error otherwise.

Because you cannot change the default behavior of a real variable defined on this class, you create a placeholder for that real variable, for example, a real variable called `_name`. Then implement your `nameSetter` method, and in the body of the method definition, use the placeholder variable `_name` instead of the variable `name` to hold the actual name value:

```
class Person ()
    instance variables _name
    instance methods
        method nameSetter self value -> (
            if ((isaKindOf value String) == false) then
                print "bad value for IV name."
            else self._name := value
        )
end
```

Now, if you instantiate the class `Person`, you can use `name` just as if it were a real variable, and `nameSetter` checks to make sure the object you are assigning to `name` is of the right class.

```
foo := new Person
foo.name := 43
⇨  "bad value for IV name"
foo.name := "Elsa"
⇨  "Elsa"
```

This next example uses a fake placeholder variable (`_radius`) for the virtual variable `radius` in the class `MyRoundClass`:

```
class MyRoundClass ()
instance variables _radius
instance methods
method set radius self value -> (
    format debug "changing radius to %*" value @normal
    self._radius := value
)
method get radius self -> (
    format debug "The value of radius is %*" self._radius @normal
    return self._radius
)
end
```

C H A P T E R

Collections

7

Collections

Collections are classes that implement common aggregate data structures such as arrays, linked lists, and hash tables. The `Collection` class and its subclasses define behavior for collections and provide methods for adding and removing elements in collections, for querying and changing those elements, and for searching for elements in collections.

This section is not an exhaustive description of collection classes in ScriptX. It is intended as an introduction to the most useful classes and the most common ways of using those classes. Figure 7-1 depicts an abbreviated class-inheritance diagram for the Collections component. The sections that follow contain a short summary of many of the available collection classes, and of the most commonly used methods. For more information on collections, see the *ScriptX Components Guide* and the *ScriptX Class Reference*.

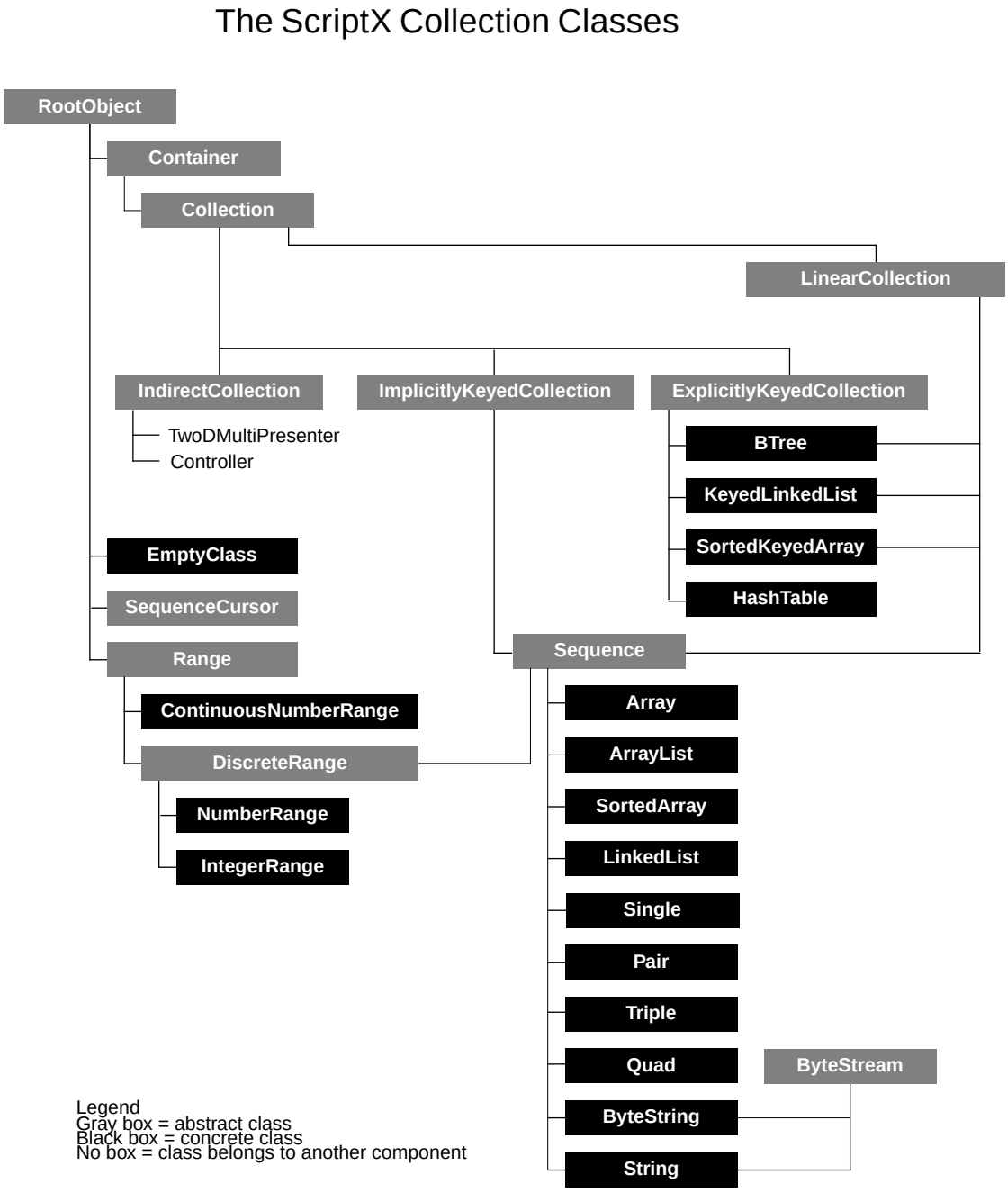


Figure 7-1: A few of the collection classes

Array, ArrayList, and LinkedList

The `Array` class provides the simplest data structure of the `Collection` classes, containing simply a linear array of elements. `LinkedList` is similar to `Array`, except it is implemented as a linked list of elements. Arrays are generally allocated with some number of "slots," (typically 20), and grow by an incremental number of elements when they run out of slots. Linked lists are variable in size; elements are added and removed dynamically as needed. The `ArrayList` class provides a hybrid of `Array` and `LinkedList`; its internal

structure is like a linked list of individual arrays. Arrays are typically faster for querying and changing elements, but linked lists are faster for inserting elements.

`Array`, `ArrayList`, and `LinkedList` are implicitly keyed sequences (they inherit from the abstract classes `Sequence` and `ImplicitlyKeyedCollection`), which means that their elements are accessed through their position in the list or array and not by some explicit key. Sequences begin with the ordinal position 1 (that is, the first element in a sequence is ordinal position 1). The following code illustrates using ordinal position to access an element of an array:

```
-- using the element access construct to get the first element
myArray[1]
-- using the method getOne to access the eighth element
getOne myArray 8
```

KeyedLinkedList

The class `KeyedLinkedList` is a list of explicit key-value pairs. Whereas sequences can be said to have implicit keys (their keys are their ordinal positions), instances of `KeyedLinkedList` have explicit keys. The values associated with those keys are usually accessed via the keys themselves.

SortedArray, SortedKeyedArray

The `SortedArray` and `SortedKeyedArray` classes provide implementations of an array and an array of key-value pairs, respectively, that are always sorted. If you add new elements to a sorted collection, they are inserted based on where they should appear in the sorting order.

You have the option of specifying the function that will be used to sort keys or items in the array. For example, if your sorted array contains items that are instances of some class, items can be sorted on the value of a particular instance variable. For a discussion of sorting with these classes, see the "Collections" chapter in the *ScriptX Components Guide*.

Note – Note that sorted arrays and sorted keyed arrays do not resort themselves if you modify one of their elements. If you modify a sorted array, you should do so by explicitly removing the item from the array, modifying it, and adding it back, so that the array will be sorted properly.

Ranges

Ranges are used to represent a range of numbers between two values. Ranges are divided into two broad categories, those which contain discrete numbers and those which do not. In the first category is the class `DiscreteRange`, which inherits from `Range` and `Sequence` and has subclasses `IntegerRange`

and NumberRange. These range classes containing discrete numbers are subclasses of Collection. Ranges containing numbers which are not discrete belong to ContinuousNumberRange and do not inherit from Collection.

- IntegerRange, a range of numbers in which the elements are integers
- NumberRange, a range of discrete numbers in which the elements can be of any subclass of Number except Integer
- ContinuousNumberRange, a range of non-discrete numbers

Single, Pair, Triple, and Quad

Single, Pair, Triple, and Quad are bounded sequences, that is, they can only contain a specific number of elements (one, two, three, and four, respectively). These classes take up less memory than a more general sequence (such as an array) with the same number of elements. Also, they provide slightly better performance because they are accessed directly, eliminating one level of indirection. Therefore, it is recommended that you use a Single, Pair, Triple, or Quad whenever it is appropriate.

```
global args := #(myString, 1, 0, 999) as Quad
```

Other Classes

ScriptX collections include two other classes which may be of interest:

- HashTable: an implementation of a hash table, useful for variable-size collections requiring very quick access
- BTree: an implementation of a B-tree structure, typically useful for very large indexed collections (such as databases)

Array and KeyedLinkedList Literals

Chapter 2, "ScriptX Building Blocks," described the literals in ScriptX for specifying instances of the Array and KeyedLinkedList classes. They are reproduced here for a summary:

▼ `#(` *element*`,` *element*`,` *element*`, . . . )`
`#(` *key*`:`*value*`,` *key*`:`*value*`, . . . )`

Range Literals

Ranges, such as those used in the for loop examples, are a special literal that creates an instance of the appropriate subclass of Range.

▼ *startValue* `to` *endValue*
startValue `to` *endValue* `by` *increment*

In both forms, *startValue* is the number to begin the range, and *endValue* is the number to end the range.

```
1 to 10
1 to 56
```

To iterate down from one number to another, put the larger number first in the range:

```
10 to 1 by -2
100 to 50 by -1
```

The `by` *increment* clause allows you to specify the amount by which each number in the range is incremented or decremented. If *startValue* is lower than *endValue*, the `by` *increment* clause is optional. In this case, there is a default value for *increment*, which is `1`. If *startValue* is larger than *endValue*, the increment must be explicitly stated.

```
1 to 10 by 2
1 to 100 by 7
0 to 1 by 0.1
10 to 1 by -1
88 to 0 by -8
```

Although the *startValue*, *endValue*, and *increment* are often numbers, they can also be an expression that results in a number. Keep in mind that most complex expressions must be specified inside parentheses for those expressions to be evaluated before the range expression. The following expressions can be used without parentheses, where appropriate:

- a variable name
- an array access expression (`myArray[1]`)
- a reference to a class or instance variable (`myRect.x1`)
- an anonymous function with no arguments, resulting in a number (described in the section “Anonymous Functions” on page 105 in Chapter , “Functions, Threads and Pipes”)
- a `nextMethod` call resulting in an object (The `nextMethod` call is described on page 132 in Chapter 6, “Defining Classes and Objects”).

Continuous Ranges

The range literal generally evaluates to an instance of the `IntegerRange` or `NumberRange` class. There is also a literal for `ContinuousNumberRange`, a range of all the numbers (floating-point and integer) between the end-points of the range. Although you can test the membership of a given number in a `ContinuousNumberRange`, you cannot count or list its elements.

To create an instance of `ContinuousNumberRange`, use the following literal syntax:

▼ *startNum* `to` *endNum* `continuous`

The `continuous` reserved word indicates that the range is to be continuous. You also have the option of specifying the exclusivity of the numbers that begin and end the continuous range. There are two reserved words for exclusivity:

```
exclusive
inclusive
```

Either the *startNum* or the *endNum*, or both, can be specified as `exclusive` or `inclusive`, where `exclusive` means that the range includes all numbers except that number itself, and `inclusive` means the range includes the number specified. The default is `inclusive.`

```
1 to 3 continuous
1 exclusive to 3 inclusive continuous
1 to 3 exclusive continuous
```

Note that in the last example, the `exclusive` reserved word only applies to the end number of the range. You must specify exclusivity separately for both ends of the range.

Creating New Instances of Collections

Just as with other objects you can use either the `new` method or the `object` expression to create a new instance of a subclass of `Collection`. You can specify the initial contents of a collection using the `contents` reserved word:

```
object myList (Array)
    contents "penny", "nickel", "dime", "quarter"
end
⇨  #("penny", "nickel", "dime", "quarter")
```

Probably the easiest way to create a new instance of a collection with an initial set of values is to use either the array or keyed list literal (as described on page 158), and then coerce the resulting `Array` or `KeyedLinkedList` object into an instance of the appropriate class. See the next section “Coercing Between Collection Classes” for examples.

Coercing Between Collection Classes

Collections can be coerced into other collection classes by using the `as` reserved word with the new collection class.

▼ *oldCollection* `as` *newCollection*

In this syntax, *oldCollection* is an expression yielding the original collection, and *newCollection* is the class of the collection to which it is coerced (or an expression resulting in a class).

```
col := #(4,2,1,3)
getClass col
⇨ Array
col as SortedArray
⇨ #(1,2,3,4) as SortedArray
col2 := col as KeyedLinkedList
⇨ #(4:4,2:2,1:1,3:3)
col2 as LinkedList
⇨ #(4,2,1,3) as LinkedList
```

The return values in the above examples are shown as if they had been printed using the `prin` method. The printable representations of most of the collection classes always refer to the class of the object as if that object had been coerced, even if it was originally created as an instance of that class. This is so that the printable representation of a collection class is the same as the syntax for creating that collection in a script. See "Printing Collections" on page 168 for more information.

Coercing between subclasses of `Collection` that are not of similar appearance may have unusual side effects. Table 7-1 describes the effect of many of these coercions.

Table 7-1: Coercions between collection classes

Original Collection	New Collection	Effect of Coercion

Sequences such as `Array` and `LinkedList`	Explicitly keyed collections such as `KeyedLinkedList`	The elements in the original collection become both the key and the value in the key-value pair (that is, the keys and the values in each are the same in the new collection).
		`#(1,2) as KeyedLinkedList`
		`⇨ #(1:1,2:2)`

Explicitly keyed collections such as `KeyedLinkedList`	Sequences such as `Array` and `LinkedList`	Only the values in the original key-value pairs are retained. The order of the elements in the new collection may not be the same as in the original collection.
		`#(1:"one",2:"two") as Pair`
		`⇨ #("one","two") as Pair`

Table 7-1: Coercions between collection classes

Original Collection	New Collection	Effect of Coercion
Unsorted collections such as `Array` and `KeyedLinkedList`	Sorted collections such as `SortedKeyedArray` and `SortedArray`	The elements in the original collection are sorted. `#(4,3,1,2) as SortedArray` ⇨ `#(1,2,3,4)`
Unbounded collections such as `Array` and `KeyedLinkedList`	Bounded collections such as `Single` and `Pair`	If the unbounded collection contains more elements than the bounded collection can hold, only the initial elements are included in the new collection. All others are discarded. If there are too few elements in the unbounded collection, the remaining elements are set to `undefined`. `#(1,2,3,4) as Pair` ⇨ `#(1,2) as Pair`

Accessing and Changing Elements in Collections

To get access to an element in a collection, use a method such as one of those described in the next section, or use the element access construct, which is actually syntactic shorthand for the `getOne` method:

▼ *collection* [*key*]

With this syntax, *collection* is an instance of a collection class, or an expression that returns a collection. If this collection is an implicitly keyed collection, then *key* represents the position of an element.

```
melons := #("honeydew","canteloupe","water","carnegie")
melons[1]
⇨ "honeydew"
melons[3]
⇨ "water"
```

If the collection is an explicitly keyed collection, then the key is an explicit key, or an expression that returns a key.

```
r := #(@one:"money",@two:"show",@three:"get ready",@four:"go!")
r[@two]
⇨ "show"
```

To change elements in a collection, use the same syntax as above, followed by an assignment. This form of element access is syntactic shorthand for the `setOne` method:

▼ *collection* [*key*] := *newValue*

For example:

```
melons := #("honeydew","canteloupe","water","carnegie")
melons[3] := "cassava"
print melons
⇨ #("honeydew","canteloupe","cassava","carnegie")
melons[5] := "water"
print melons
⇨ #("honeydew","canteloupe","cassava","carnegie","water")

r := #(@one:"money",@two:"show",@three:"get ready",@four:"go!")
r[@four] := "pause"
print r
⇨ #(@one:”money”, @two:”show”, @three:”get ready”, @four:"pause")
```

Summary of the Collection Protocol

This section contains a partial list of generic functions that a `Collection` object responds to. For a complete listing, see the definition of `Collection` in the *ScriptX Class Reference*.

The empty Object

Many of the methods described in the sections below return `empty` if the element or key you specify is not in the collection. The `empty` object is a special global constant used by the collection classes to mean “item not found.” It should not be confused with `undefined`. Both `empty` and `undefined` are system objects, objects that act as placeholders or constants.

Do not put the `empty` object into a collection as an element itself. If you assign the `empty` object to a collection, it becomes difficult to tell whether a method is returning the fact that the specified element was not found, or that it was found, but it was the `empty` object. Use `undefined` instead of `empty` to indicate that an item in a collection has no value.

Functions as Arguments

Some of the generic functions in the collections component take a function as one of their arguments. These generic functions provide a means of iterating through a collection to perform some action. In many cases, the function supplied as an argument is used to select which element or elements from the collection will be returned or operated on by the calling generic function. For instance, `removeAll` deletes from the collection all elements which cause the function passed to it to return `true`. A generic function such as `chooseAll` returns all items which cause the function to return `true`. In the case of `forEach`, however, the function is simply applied to each item in the collection, and there is no return value.

These generic functions all take three arguments. The generic function `forEach` is used to illustrate the form:

`forEach` *collection func arg*

collection is the collection to be operated on

func is the function to be applied to the items in the collection

arg is the argument to be passed to *func*

The function *func* is iteratively passed two arguments: an item from *collection* and the arguments contained in *arg*. If the collection has a natural order (inherits from `LinearCollection`), the items will be passed to *func* in that order. The content of the variable *arg* depends on how many arguments *func* requires. If *func* takes no arguments, *arg* needs to be `undefined`. If *func* takes two or more arguments, *arg* needs to be an array of the arguments.

```
--calling forEach to print each item to the debug stream
forEach #(2,4,6,8) print debug
```

The call to `forEach` in the example above is equivalent to the following code:

```
print 2 debug; print 4 debug; print 6 debug; print 8 debug
```

The generic functions `chooseAll`, `chooseOne`, `chooseOneBackwards`, `chooseOneBinding`, `forEach`, `forEachBackwards`, `forEachBinding`, `map`, `removeAll`, and `removeOne` all take a function as their second argument. See the *ScriptX Components Guide* and *ScriptX Class Reference* for more complete information.

Generic Functions for Accessing Elements in Collections

- `getOne` *self key*
  `getAll` *self key*
  `getNth` *self ordinal*

The `getOne` method gets the value given by the first instance of *key* in the collection *self;* `getAll` gets all the values associated with the given *key.* The `getNth` method, available only on collections that refer to an element's position as its key, gets the value at the ordinal position *ordinal*, and is the specialized form of `getOne` for collections with implicit keys. Keep in mind that ScriptX sequences begin at position 1, not position 0, as in some programming languages.

The `getOne` method is equivalent to the expression *self* `[`*key*`]`.

- `getOrdOne` *self value*

  The `getOrdOne` method is available only on collections that refer to an element's position as its key (that is, subclasses of the abstract collection `LinearCollection`). It returns the position at which the first instance of *value* is located in the collection *self*.

- `getKeyOne` *self value*
  `getKeyAll` *self value*

  The `getKeyOne` and `getKeyAll` methods act like `getOrdOne` on explicitly keyed collections such as `KeyedLinkedList`; given the *value,* they return the first key in the collection *self* whose value equals *value* (`getKeyOne`) or all keys whose values equal *value* (`getKeyAll`).

Generic Functions for Adding Elements to Collections

- `add` *self key value*

  The `add` method inserts the key-value pair specified by *key* and *value* into the collection *self*. For sequences such as arrays or lists, *key* is the position in the list.

- `addMany` *self another*

  The `addMany` method appends all the values (or key-value pairs) in the collection *another* to the collection specified by *self*. When the collections are both strings, this method can be used to concatenate them.

- `append` *self value*
  `prepend` *self value*

  The `append` method inserts the given *value* at the end of the collection *self*; `prepend` inserts it at the beginning.

Generic Functions for Deleting Elements from Collections

- `deleteOne` *self value*
  `deleteAll` *self value*
  `deleteNth` *self ordinal*

  The `deleteOne` method deletes the first instance of the given *value* in the collection *self*; `deleteAll` deletes all instances of *value* from *self*. With collections that have explicit key-value pairs, both the keys and the values are deleted. The `deleteNth` method, available only on collections that refer to an element's position as its key, deletes the instance of the element at the ordinal position *ordinal*.

- `deleteKeyOne` *self key*
  `deleteKeyAll` *self key*

  The `deleteKeyOne` method deletes the first instance of the key-value pair specified by *key* in the collection *self*. The `deleteKeyAll` method deletes all instances of that key and its values.

Generic Functions for Changing Elements in Collections

- `setOne` *collection key value*
  `setAll` *collection key value*

  The `setOne` generic function changes the binding of the first instance of *key* to *value* in the collection *collection*. If *key* does not exist, the key-value pair specified by *key* and *value* is added to the collection. The `setAll` method is equivalent to `setOne`, except that for all instances of *key*, the associated values are changed to *value*.

  In sequences such as arrays, *key* is the position of the element in the collection *collection*, and cannot be greater than the current length of the collection plus one. (If *key* does not exist in the collection, the key-value pair can be added to the end of the sequence but not to any position beyond that.)

  The `setOne` generic function is equivalent to the expression *collection* `[`*key*`]:=` *value*.

Generic Functions for Searching Collections

Collection access methods can be used to select a particular value, key, or key-value binding.

The generic functions that follow are the ones that are most useful for searching collections, and they are also examples of generic functions that take a function as an argument. Such functions have an iterator built in to their implementation. It is possible to create you own iterator and then apply your own operations, but using functions with a built-in iterator is more efficient.

- `chooseOne` *self func arg*

This function calls *func* once for each item in the collection *self,* supplying *func* with the argument *arg,* until *func* returns `true`. It returns the value of the first item in the collection for which *func* returns `true`, or `empty` if no call to this function returns `true`.

```
nameArray := #(“Hal”, “Sid”, “Su”, “Wanmo”, “Sid”)
chooseOne nameArray (x -> x = “Sid”) undefined
⇨ “Sid”
getOrdOne nameArray “Sid”
⇨ 2 -- the first occurrence of "Sid" is at ordinal position 2
```

- `chooseAll` *self func arg*

This function calls *func* once for each item in the collection *self,* supplying *func* with the argument *arg*. It returns a collection of all the elements in *self* for which *func* returns `true`. This function is probably the most useful one for searching collections.

```
-- select the numbers over 75
global numArray := #(4, 2.5, 109, 35, 3809, 1.0, 333.333)
chooseAll numArray (x -> x > 75) undefined
⇨ #(109, 3809, 333.333)

-- select the numbers over 75 which are integers
chooseAll numArray (x -> x > 75 and \
   (isAKindOf x Integer)) undefined
⇨ #(109, 3809)

-- select the numbers over 10000
chooseAll numArray (x -> x > 10000) undefined
⇨ #() -- there are no numbers over 10000
```

Other Useful Generic Functions and Properties

- *collection*`.size`

The `size` instance variable contains the number of elements in the *collection*. It is a virtual instance variable that is automatically calculated by its getter method. You cannot change it yourself.

- `isEmpty` *self*

The `isEmpty` generic function tests whether the collection *self* is empty. It returns either `true` or `false.`

- `isMember` *self value*

The `isMember` generic function tests whether the element specified by *value* is in the given collection, where *value* is the value of an explicitly keyed list, or an element in a sequence such as an array.

The following construct is another way to test membership in a collection:

```
myArray := #(1, 2, 3, 4)
myArray contains 3
⇨  true
```

Printing Collections

Collection objects, just like all objects, have a printable representation that can be printed to a stream using the standard printing functions described in Chapter , "Working with Objects" in the section "Output" on page 74. Instances of `Array` or `KeyedLinkedList` print in the same syntax as the array and keyed list literals:

```
print #("this and that")
⇨  #("this and that")
print #("this":"that")
⇨  #("this":"that")
```

Instances of classes other than `Array` or `KeyedLinkedList` use a similar format, with the name of the class after the array or list itself. This representation is also the same as the syntax for coercing between collections.

```
nums := #(2,3,1) as SortedArray
print nums
⇨  #(1,2,3) as SortedArray

gourmet := #(@capon:"meat",@brie:"cheese") as SortedKeyedArray
print gourmet
⇨  #(@brie:"cheese", @capon:"meat") as SortedKeyedArray
```

The `@normal` printed representation of a collection (that is, the one used by the `prin` generic function) is limited to printing only ten items, with an ellipsis at the end to show that there are more items in the collection. For collections with

more than ten items, you may want to print using either the `@complete` representation, or use a `for` loop to print all the elements of the collection individually.

```
series := (1 to 12) as Array
print series
⇨  #(1, 2, 3, 4, 5, 6, 7, 8, 9, 10, . . . )
prin series @complete debug
⇨  #(1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12)
for i in series do print i
⇨  1
   2
   3
   4
   . . .
```

A useful trick for printing collections is to "pipe" them to a printing function. As an intermediate step, you can even pipe them through a sort function or a sorted collection. For more information on the pipe operator, see "Pipes" on page 110. Here is an example of piping:

```
ScriptXQATeam := #("Bob Cotterill","Rajiv Joshi","Kathleen Peirce",
       "Su Quek","Felicia Santelli","Kim Swix","Maggie Womack",
       "Yosh Kashima","Bill Hogan")
ScriptXQATeam | SortedArray | print
⇨  "Bill Hogan"
   "Bob Cotterill"
   "Felicia Santelli"
   "Kathleen Pierce"
   . . .
```

Strings as Collections

A string (a `String`, `StringConstant`, or `Text` object) is actually a collection of integers which represent Unicode values corresponding to characters. Therefore, when you use collection methods on strings, you need to use the integer which corresponds to the Unicode value of a character rather than the character itself. For example, methods such as `add`, `append`, `prepend`, `deleteOne`, and `deleteAll` must be given an integer as *value* when they are applied to strings. Conversely, if the return value is an object from a collection, and the collection is a string, that return value will be an integer.

The collection access construct is the shortest way to find the Unicode value for a character. You simply create a string of one character and then access the first (and only) character:

```
"e"[1]
⇨  101
```

To get the character that an integer represents, you can create an empty string, add the integer to it, and then print it as a string:

```
function intToString intVal ->
```

```
    (
    local str := new String
    append str intVal
    print str
    )
intToString 99
⇨ “c”
```

Searching Strings

Methods which can be used to search collections can also be used to search strings. The most useful is `findRange`, which, when applied to a string, will search for the first occurrence of a match to the supplied string.

- `findRange` *self toMatch*

This function finds the first range of contiguous values inside the string *self* that matches the characters in the string *toMatch*. It returns the ordinal position of the first character in the matching range or 0 if there is no match.

```
global myStr := “Let’s see how this works.”
findRange myStr “see how this works”
⇨ 7
findRange myStr “see if this works”
⇨ 0
```

The global function `findNthContext`, which provide searching capabilities specific to strings, is useful for parsing. It allows you to search for the nth word, sentence, or paragraph. You can also supply a function designating a delimiter and search for the range of characters bounded by that delimiter.

- `findNthContext` *args context*

The argument *args* is a `Quad` which supplies the following :

the string to be searched

n, designating which occurrence to find

the cursor position to begin searching

the cursor position to end searching

The context is either @word, @sentence, @paragraph, or an anonymous function giving a delimiter.

The following example demonstrates using `findNthContext` but is not a complete explanation. More information is available in the global functions section of *ScriptX Class Reference*.

```
-- create a string
```

```
global str := "This is a sample string."

-- search for the fourth word in str, starting at the cursor position
-- 0 and continuing until the end (str.size)
global args := #(str, 4, 0, str.size) as Quad
findNthContext args @word

-- print out the word returned
copyFromTo args[1] args[3] args[4]
⇨ "sample"

-- search for the second set of characters bounded by ":"
global s := "first name:last name:street address:city:state"
global args := #(s, 2, 0, s.size) as Quad
findNthContext args (r -> r == ":"[1])

-- print out the characters returned
copyFromTo args[1] args[3] args[4]
"last name:"
```

A second global function for searching strings is `searchIndex`, which finds the first occurrence of the string you want to match. (Note that the string you want to match must be at least three characters long and cannot contain any spaces.) Searching is fast because `searchIndex` is actually searaching a signature index, which is built automatically when you create a `StringIndex` object and supply a string for its `string` instance variable.

◆ `searchIndex` *strIndex match searchContext wholeWord*

This function searches the `StringIndex` object *strIndex* for the first occurrence of *match* using *searchContext* to tell it where to search. The last argument, *wholeWord,* is either `true` or `false`, indicating whether the match must be a whole word (as opposed to only part of a word).

The following code example demonstrates using `searchIndex`. It will be easier to follow if you have read the entries for `SearchContext` and `StringIndex` in *ScriptX Class Reference*.

```
-- create a string
global myString := "Newton devised a new system."

-- create a StringIndex object
global strIndex := new StringIndex string:myString

-- create a SearchContext object suitable for beginning the search at
-- the beginning of the string
global sc := initialSearchContext strIndex 0

-- search for the first occurrence of "new" as a whole word
searchIndex strIndex "new" sc true

-- print out the result
copyFromTo sc.string sc.startOffset sc.endOffset
⇨ "new"
```

```
-- search for the first occurrence of "new" as a whole word or as part
-- of a larger word
searchIndex strIndex "new" sc false

-- print out the result (Note that searchIndex is not case sensitive.)
copyFromTo sc.string sc.startOffset sc.endOffset
⇨  "Newton"
```

C H A P T E R

Exceptions

8

An *exception* is an unusual or abnormal situation in a script or in the ScriptX system that might cause your ScriptX program to fail. For example, you might attempt to divide a number by zero. The ScriptX exception system allows you "catch" those exceptions and handle them gracefully. So, if a "divide by zero" exception occurs, you can catch the exception, print a warning, restore your variables to their original values, and continue executing the program.

The ScriptX exception system is made up of three parts:

- There is a set of exception classes, included with the ScriptX core classes, that provide exceptions for common situations. You can also create your own exceptions by defining new subclasses of these classes.
- The `guard` construct in the ScriptX language allows you to catch exceptions and handle them within your script.
- The `report` generic function allows you to report your own exceptions within your scripts.

This chapter describes how to catch and handle single exceptions, classes of exceptions, or all exceptions in your scripts, as well as how to report (sometimes called "throw") an exception.

The core classes define a standard set of exception classes that represent common errors, warnings, or other unwanted situations that may occur during the execution of a script. In addition, you can create your own classes and instances to represent kinds of exceptions and specific exceptions that your code might report during its execution.

For more details on the core set of exception classes, see the *ScriptX Class Reference*. For more information on defining your own exceptions, see "Creating Exception Subclasses and Instances" on page 185.

Catching and Handling Exceptions

ScriptX provides the following `guard` construct, which allows you to catch and handle exceptions within your scripts:

▼ `guard`

guardedCode

`[ catching`

exceptionName `[` *lexicalName* `]:[` *action* `]`

`... ]`

```
    [ on exit
        exitCode ]
end
```

The *guardedCode* is an expression (often a compound expression) containing the code to be guarded. Code that may report an exception must be guarded if that exception is to be caught. Any ScriptX expression can be guarded, with the following exceptions:

- A `global`, `unglobal`, or `local` expression cannot be guarded. (A `local` expression can be used inside a compound expression that is guarded. The `global` and `unglobal` expressions are permitted only at the top level.)
- Since they are permitted only at the top level, a module definition expression or an `in module` expression cannot be guarded. (These expressions are discussed in Chapter 9, "Modules.")

The optional `catching` section contains a tagged list of exceptions to check for, each one matched with an expression to execute if an exception occurs. This list is sometimes referred to as the *catch list*. Use *exceptionName* to specify one of three types of values: an instance of an exception class (to catch a specific exception), an exception class (to catch any exception within a class of exceptions), or the keyword `all` (to catch all exceptions).

Each item in the catch list can have a lexical name associated with its action. The optional *lexicalName*, if one is given, is bound to the argument that is reported with the exception, allowing the expression that catches and handles the exception to process this argument as well. When ScriptX reports an exception, the exception is bound to the global variable `throwTag`, and its reporting argument, if defined, to the global variable `throwArg`. These two global variables are covered in greater detail in the section "Retrieving Information about the Exception" on page 184. Think of each item in the tagged list of exceptions as being equivalent to `throwTag throwArg`, where `throwArg` is optional.

Expressions in the catch list are often compound expressions. These expressions could be used to print a message or handle some other consequence of catching the exception.

The optional `on exit` section contains an expression (often a compound expression) that is always executed whether or not the guarded code reports an exception. The expression in `on exit` is guaranteed to be executed.

Further details on each of these `guard` clauses are given in the remainder of this section.

caught and throw again

ScriptX also provides the `caught` and the `throw again` expressions, which, when used within the `catching` clause of the `guard` construct, prevent the remaining statements in the catch list from being executed after an exception has been caught.

The `caught` expression specifies that the exception has been caught. Once the exception has been caught, ScriptX immediately breaks out of the catching loop and does not check the remaining clauses in the `catching` clause or any other catch lists. The `caught` expression requires one argument, an expression containing the return value of the `guard` clause if that `guard` clause resulted in an exception. For example, if the guarded code is the generic function `addMany`, and the script attempts to add too many items to a collection, the argument to `caught`, and therefore the return value of `guard`, might be an error message:

```
global myArray := new Array initialSize:10 growable:false
global otherArray := #(1,2,3,4,5,6,7,8,9,10,11,12)
-- now try to add too many items to an array
guard
    addMany myArray otherArray
catching
    boundedError: caught (print "Tried to add too many items" debug)
end
⇨  "Tried to add too many items"
```

An examination of `myArray` shows that we succeeded in adding the first ten items before the exception was reported.

```
myArray
⇨  #(1, 2, 3, 4, 5, 6, 7, 8, 9, 10)
```

Like `caught`, the `throw again` expression causes ScriptX to stop checking the remaining statements in this `catching` clause. Unlike `caught`, the exception is then reported again, which causes ScriptX to test each of the exceptions in any surrounding `guard` constructs, or to drop the exception if there are no surrounding guard constructs. For example, if `x` is some number, and `y` is 0, the `catching` construct prints a message and then reports the `divideByZero` exception again, so any surrounding `guard` constructs can catch it if needed.

```
guard (z := x/y)
    catching
        divideByZero: (print "Divide By Zero"; throw again)
end
```

To `throw again` is to leave the exception uncaught. It is up to surrounding `guard` expressions to catch the exception. If there is no surrounding `guard` expression, or if the surrounding `guard` expression fails to catch the exception, the thread dies, and no value is returned.

What Happens In a Guard Construct

If no exceptions are reported during execution of the guarded code, the point of execution leaves the guarded code, executes the `on exit` code, and leaves the `guard` construct.

If an exception is reported during execution of the guarded code, the system stops executing the guarded code, executes the `on exit` code, then looks for all clauses in the catch list that catch the reported exception. For each clause in the catch list that catches the reported exception, the specified consequence occurs.

If any expression in the `catching` part of the `guard` construct uses the `caught` keyword, ScriptX immediately exits from the `guard` construct. None of the remaining statements in the current catch list or any surrounding catch lists are executed.

After ScriptX has checked the clauses in the current catch list, it reports (throws) the exception again, which means it checks for catch lists in any surrounding `guard` construct.

If any clause that catches the exception uses the `throw again` construct, the system throws the exception again immediately. In this case, it does not check the remaining statements in the catch list of the current `guard` construct, but immediately moves on to the surrounding one.

The system continues throwing the exception until the exception is explicitly caught by the `caught` construct. If the exception is not caught, if the `caught` construct is not specified for that exception, then the thread that was running when the exception occurred dies. If the exception occurs in the main thread, the title quits. Imagine the exception bouncing around from catcher to catcher until either something finally catches it for good, or it falls to the ground uncaught.

Specifying Exceptions to Catch

The `guard` construct, through the `catching` list, can catch individual exceptions, instances of a class of exceptions, or all exceptions.

To catch a particular exception, specify the `Exception` instance in the catch list. For example, this line catches only the `divideByZero` exception:

```
catching
    divideByZero : print "Divide by zero error. \n"
```

To catch all instances of a class of exception, specify the exception class in the catching list. For example, this line catches all reported math exceptions, including `divideByZero`, `lossOfRange`, `invalidNumber`, and any other exceptions that are instances of the class `MathException`.

```
catching
    MathException : print "Some kind of math error."
```

To catch all exceptions, specify the `all` keyword in the catch list. For example, this line catches all reported exceptions, including both system-defined exceptions and user-defined exceptions.

```
catching
    all : print "Look out, there's an error about."
```

An Example: Printing Messages

The best way to demonstrate the use of exceptions is by example. This example prints a message about each kind of exception that was caught. This is an overly simple example, and you would normally want to print out more information about the precise conditions under which the exception occurred. Details on how to retrieve and print information about an exception are given in “Retrieving Information about the Exception” on page 184.

In the example, note the use of `all: caught undefined` as the last clause in the catch list. This clause explicitly catches every exception to prevent it from being thrown again, thus ensuring that the point of execution leaves the `guard` construct and continues executing the remainder of the code. By putting this line at the end of the catch list, you ensure that all other consequences of an exception occur before the exception is explicitly caught. The `caught` construct must return a value, which in this case is `undefined`.

```
guard (
    width := box1.width
    length := box1.length
    ratio := width / length
)
catching
    noMethod : print "There's a method missing."
    divideByZero : print "Divide by zero error."
    MathException : print "Some kind of math error."
    all: caught undefined
on exit
    print "On exit code being executed"
end
```

guarded code

catch list

During the execution of the guarded expressions, several kinds of exceptions could occur. First, the `width` or `length` might not be defined on `box1`. Secondly, the values of those instance variables might be `undefined` or some other non-numeric value. Finally, the length of the box might actually be `0`.

- If either the `width` or `length` instance variable is not defined for `box1`, that variable does not have a setter or getter method, and so a `noMethod` exception is reported (for example “no `lengthGetter` method for `Box`”). The remaining expressions in the guarded code are not executed. First, the `on exit` code is executed. The exception is then intercepted and handled by the first clause in the catch list, which prints the message `"There’s a method missing."` to the `debug` stream.
- If `box1` is an object that implements both the `length` and `width` instance variables, but `length` contains `0`, the second expression in the guarded code reports a `divideByZero` exception, since the `ratio` expression attempted to divide the value of `width` by zero. The `on exit` code is executed. The exception is intercepted and handled by the second clause in the catch list, and the message `"Divide by zero error."` is printed. The exception is also intercepted and processed by the third clause, since the `divideByZero` exception is an instance of the class `MathException`. The message `"Some kind of math error."` is also printed.

- If `box1` is an object that implements both the `length` and `width` instance variables, but either one contains a non-number value, the third expression reports the `invalidNumber` exception. The `on exit` code is executed. The exception is intercepted and handled by the third clause in the `catch` list, since the `invalidNumber` exception is an instance of the class `MathException`. The message `"Some kind of math error."` is printed.

Note that none of the catchers in the catch list has caught the exception yet. If any kind of exception occurs, the exception is finally caught by the fourth clause in the catch list (`all`). Since the exception has been caught, no more catchers in the catch list, or in any catch lists of surrounding `guard` expressions, are executed. The value of the `guard` expression is `undefined`. Since the expression was finally caught, execution continues after the outermost `guard` expression.

Nesting Guard Constructs

Expressions to be guarded within a `guard` expression can consist of or include other `guard` expressions. The use of multiple `guard` levels enables you to tune your exception-catching code. For example, you can write code that checks all expressions in a block for exceptions of type *e1*, additionally checks a subset of the expressions for exceptions of type *e2*, and checks a further subset for exceptions of type *e3*.

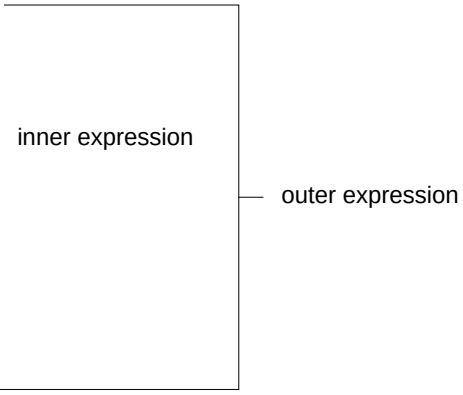
Usually, when an exception is reported by a guarded expressions in a `guard` construct, ScriptX checks each of the clauses in the catch list to find which ones catch the exception. Each exception in the catch list is checked, even if one has already caught the exception (unless one of the expressions uses the `caught` or `throw again` construct as discussed in "How to Control Throwing and Catching" on page 182.)

After checking all the clauses in the current catch list, ScriptX checks the catch list in the surrounding `guard` construct. This act of moving up to the surrounding `guard` construct to check for statements that catch an exception is known as throwing the exception. The system continues stepping up through the `guard` levels until one of the following occurs:

- The exception is explicitly caught by the `caught` construct, in which case the system exits from the outermost `guard` expression and continues executing the program.
- There are no more catch lists to check, in which case the thread in which the exception originated dies and execution of the process terminates.

The following example demonstrates how nested `guard` expressions work.

```
guard (
    S1
    guard
        S2
    catching
        e1: repair A
        e2: repair B
        e3: repair C
    end
)
catching
    e4: repair D
    e5: repair E
    all: caught undefined
end
S3
```



`S1` and `S2` represent ScriptX expressions that can be guarded. The symbols `e1`, `e2`, `e3`, `e4`, and `e5` represent lexical names, either instances of `Exception` or exception classes. In this example, expression `S1` is guarded against exceptions `e4` and `e5`, while expression `S2` is guarded against exceptions `e1`, `e2`, `e3`, `e4`, and `e5`.

Consider what happens if `S1` reports an exception:

1. The point of execution jumps to the catch list in the outer `guard` expression and checks if the reported exception is an instance of `e4`. If so, it executes the expression `repair D` and continues with the next catcher in the list.
2. The system checks if the exception is an instance of `e5`. If so, it executes the expression `repair E` and continues with the next catcher in the list, which is an `all` clause.
3. The clause `all: caught undefined` explicitly catches the exception and prevents it from being thrown again.
4. The system exits the outermost `guard` expression and executes the expression `S3`.

Consider what happens if `S2` reports an exception:

1. The system checks if the exception is an instance of `e1`. If so, it executes the expression `repair A` and continues with the next catcher in the list.
2. The system checks if the exception is an instance of `e2`. If so, it executes the expression `repair B` and continues with the next catcher in the list.
3. The system checks if the exception is an instance of `e3`. If so, it executes the expression `repair C` and continues with the next catcher in the list.
4. On reaching the final catcher in the catch list, the system throws the exception again, which means it checks the catch list in the surrounding `guard` expression if there is one, which there is.
5. The point of execution jumps to the catch list in the outer `guard` expression and checks if the reported exception is an instance of `e4`. If so, it executes the expression `repair D` and continues with the next catcher in the list.

6. The system checks if the exception is an instance of `e5`. If so, it executes the expression `repair E` and continues with the next catcher in the list, which is an `all` clause.

7. The catcher `all: caught undefined` explicitly catches the exception and prevents it from being thrown again.

8. The system exits the outermost `guard` expression and executes the expression `S3`.

In several examples outlined in this chapter, the exception is finally caught by the `all` clause. This clause is really just a macro that expands to `Exception`—it is a shorthand way of intercepting `all` instances of `Exception`. Note that there is no particular reason, although the examples in this chapter show it, that the `all` clause has to be last in the catch list. A catch list can have no `all` clause, or it can have several.

How to Control Throwing and Catching

The following sample code illustrates the use of `caught` and `throw again`:

```
guard (
    S1
    guard
        S2
    catching
        e1: (repair A; caught A)
        e2: (repair B; throw again)
        e3: repair C
    end
) catching
    e4: repair D
    e5: repair E
    all: caught undefined
end
S3
```

If `S2` reports an `e1` exception, the following happens:

1. The system checks if the reported exception is an instance of `e1`. If so, it executes the expression `repair A`. The `caught` statement terminates the search for matching exceptions. Execution exits from the `guard` expression and continues on to `S3`. The `guard` construct returns `A`.

If `S2` reports an `e2` exception, the following happens:

1. The system checks if the exception is an instance of `e1`, which it is not.

2. The system checks if the exception is an instance of `e2`. Since it is, it executes the expression `repair B`. The `throw again` expression causes the exception to be thrown again immediately, which means it stops checking catchers in the current catch list and starts checking catchers in the catch list of the surrounding `guard` expression.

3. The system checks if the exception is an instance of `e4`.

4. The system checks if the exception is an instance of `e5`.
5. The catcher `all: caught undefined` explicitly catches the exception and prevents it from being thrown again.
6. Execution moves out of the outer `guard` expression and continues on to expression `S3`

Reporting Exceptions

To report an exception, call the `report` function with two arguments:

- the exception to report
- an object containing information about the exception that occurred, which is used for the exception's format string (or default message) among other things

The actual object to be passed as the second argument varies from exception instance to exception instance.

For example, the format string for the `divideByZero` exception instance is `"Attempt to divide %* by zero"` where `%*` will be replaced by the number on which the attempt to divide by 0 was made. For instance, when an attempt is made to divide 25 by 0, a `divideByZero` error is reported by the following statement:

```
report divideByZero 25
```

The value 25 is passed to the exception's format string and the format string for the reported exception becomes:

```
"Attempt to divide 25 by zero."
```

The second argument to `report` can be a collection if you want to pass more than one object to the format string.

For example, the format string for the `noMethod` exception requires two input values: the method that was called and the class.

The following statement reports a `noMethod` exception when an attempt is made to call the nonexistent `turnPurple` method on an instance of the class `Window`:

```
report noMethod #("turnPurple", Window)
```

When this code is executed, an exception is reported. The formatted result for the exception is:

```
No "turnPurple" method for Window.
```

Retrieving Information about the Exception

Script uses two global variables to record information about the most recently reported exception:

- The `throwTag` global variable is bound to the most recently reported exception instance.
- The `throwArg` global variable is bound to the extra argument that was passed by the `report` method when the exception was reported.

To print the format string associated with the most recently reported exception, use the following call to the `prin` method:

```
prin throwTag throwArg debug
```

The exception classes specialize the `prin` generic function. (Recall that `print`, `prinln`, `format`, and the other global printing functions are derived from the generic `prin`, as discussed in the section on "Output" on page 74.) On most objects, the `prin` method takes three arguments: the object to print, a style argument, and an optional stream. In the case of exception objects, the second argument can be one of the following:

- a style argument (one of `@normal`, `@complete`, `@debug`, or `@unadorned`), in which case the name of the exception is printed in the indicated style
- any other argument, in which case the argument is passed to the exception's format string, and the format string is printed

When `prin` is called on an exception with a second argument of `throwArg`, it prints the format string for the exception. The global variable `throwTag` is always bound to the most recently reported exception.

To retrieve the information that the most recently reported exception received from `report`, get the value of the global variable `throwArg`.

The following example shows the use of `throwArg` and `throwTag`. Note that the `prinln` function does the same thing as `prin`, but adds a newline at the end of the printed output.

```
global a1:88, b1:0, c1, d1, e1
guard (
    c1 := a1 / b1
    d1 := a1 - b1
    e1 := a1 / c1
)
catching
    divideByZero returnArg: (
        format debug "The number divided by 0 is: %*\n" throwArg @normal
        prin "The default error message is:\n" @unadorned debug
        prinln throwTag throwArg
        caught undefined
    )
end
```

This script, upon attempting to divide `88` by `0`, catches the `divideByZero` error and prints the following messages to the debugging stream:

```
The number divided by 0 is: 88
The default error message is:
Attempt to divide 88 by zero. (DivideByZero)
```

Creating Exception Subclasses and Instances

To define subclasses of `Exception` or any of its subclasses, use the `class` construct as discussed in Chapter 6, "Defining Classes and Objects." For example, the following class declaration creates a class of exceptions that is particular to a class of objects called `People`:

```
class PeopleException (Exception) end
```

To create instances of exceptions, use `new` with appropriate exception class. The syntax for using `new` on exception classes is:

- `new` *ExceptionClass* `name: "`*name*`" format: "`*format string*`"`

The class *ExceptionClass* is the class to instantiate. The top-level exception class is `Exception`. The value in *name* is a string containing the name of the exception instance, for example `"tooOld".`The value in *format string* is the format string for the exception. The string is used as the default exception message, so it should contain useful information about the exception. The format string follows the substitution character rules as described in "Formatted Output" on page 78.

When an exception is reported, the second argument to the `report` method is passed to the format string for use by the substitution characters in the string. When calling `report` on an exception, make sure that the second argument matches the type of input required by the exception's format string. If the format string contains `%*`, the second argument to `report` should be an object. If the format string contains `%`*n*, the second argument to `report` should be a `LinearCollection` object containing at least *n* objects.

The following statement creates a `tooOld` exception as an instance of the exception class `PeopleException`.

```
tooOld := new PeopleException name: "tooOld" \
       format: "No one lives to be %* years old."
```

When this exception is reported, the second argument to `report` should be the unsuitable age. For example, the `People` class defines an `ageSetter` method that reports the `tooOld` exception if the age is greater than 115:

```
class People ()
    instance variables _age
    instance methods
        method ageGetter self ->
            return self._age
```

```
        method ageSetter self val -> (
            if val > 115 then report tooOld val
            else self._age := val
        )
end
```

If a script attempts to set a person's age to a value that is too large, `ageSetter` calls `report` on the `tooOld` instance of `PeopleException`, with an argument of `122`. This reports an exception:

```
abuela := new People
abuela.age := 122
-- ** No one lives to be 122 years old. (tooOld)
```

The default message is usually printed when the exception is reported but not caught. If the exception is caught by the catch list in a `guard` construct, the consequences of the exception depend entirely on the action defined by the clause in the catch list that catches the exception (which could include printing out the format string for the exception).

C H A P T E R

Modules

9

Modules provide a mechanism so that ScriptX programs can be packaged into components—components that can be compiled, managed, and integrated into other ScriptX programs without naming conflicts.

Many programming environments allow for only one set of global names. If you define a global variable, the variable's name is available in all parts of the program at all times. If the program that contains a variable's definition is incorporated and compiled into another program, that variable name is visible to all parts of that new program as well, and may conflict with variables of the same name that are defined elsewhere.

ScriptX provides a level of indirection between names and variables. A variable is an association between a name and a location in memory—a location which stores a value. In ScriptX, that value is always a reference to an object. This association between a name and a value cell is called a *name binding*. A module can be thought of as a collection of name bindings.

Using ScriptX modules, a given location in memory can have different names in different parts of a program. A module is a device for resolving name conflicts, so that groups of programmers can collaborate on a project. Using modules, you can encapsulate your program as a unit, and specify which classes, objects, functions, and global variables are visible outside that unit. ScriptX modules allow code that is compiled and tested for one project to be reused in other projects.

Modules have one other important function in ScriptX. Although the ScriptX class `ModuleClass` does not inherit from `Collection`, a module is best understood as a collection of name bindings. When a module is added to a title container, it stores its name bindings for classes, objects, functions, and variables to disk. This feature is useful as a mechanism for storing and retrieving compiled classes, functions, variables and objects.

Modules should not be confused with segmentation or dynamic loading of objects. When the ScriptX bytecode compiler is instructed to switch from one module to another, using the `in module` expression, all objects defined while executing in former module remain in memory. Any bindings that exist in the former module still exist, and can be used when that module becomes the current module again. Nor does loading a new module load any objects that the new module defines. A program must explicitly create or load any objects it defines.

Chapter Summary and Organization

This chapter is organized in five sections. ScriptX modules are designed to support teams of programmers who collaborate on large projects. Many readers, especially those who are using ScriptX for the first time, may not have

any immediate need for all the features of ScriptX modules. Those who are not using ScriptX for large projects may want to skip portions of the chapter until they actually need the features that are described.

"Module Basics" on page 190 describes essential features of modules in ScriptX. Programmers who are new to ScriptX may want to read just this section and return to the remainder of the chapter at a later date. If you do not intend to define modules or save compiled code between programming sessions, you should find everything you need to know about modules in this section.

"Module Concepts" on page 192 continues the discussion in the previous section, exploring underlying ideas that are necessary for defining your own ScriptX modules.

"Defining and Using Modules" on page 200 presents the syntax for creating a module in ScriptX, and for switching the compiler environment between modules. This section defines all the syntax for importing and exporting names and variables between modules.

"Organizing Modules" on page 219 presents a conventional technique for using modules called the interface/implementation model. This model can be used to create larger networks of modules. It allows modules to be managed without the need for complex use relationships. If your program contains only a single module, and you do not intend for the code to be usable by others, you can skip this section and go on to the next.

"Storing Modules" on page 227 discusses how to structure your ScriptX program so that it can be saved more easily. Without modules, you must individually store each class and object that your program uses, and you must also reconstruct any global variables that the program requires. By encapsulating the program within a module, classes, objects, functions, and global variables in the program are automatically saved for you.

Module Basics

All ScriptX expressions are compiled within a module. You can always tell what module you are currently compiling in by looking at the title bar of the active listener window. If no module has been defined, then ScriptX compiles expressions in the `Scratch` module. The title bar of the listener window reads "ScriptX Listener in module Scratch."

It is not necessary to define a new module to use ScriptX. If you do not define a module, your program compiles in the `Scratch` module, the default module. This module could just as easily have been called the "runtime" module in that it is created automatically, "from scratch," every time ScriptX runs, but cannot be saved. Any time you start running ScriptX, your program is compiled in the `Scratch` module until you explicitly declare that it is compiling in another module, by using the `in module` expression. If you are working in a user-defined module and then select **New Listener** from the ScriptX **File** menu, a new Listener starts in the `Scratch` module.

The Scratch module is one of three instances of ModuleClass that is created by the system. The global function currentModule indicates which module ScriptX is currently compiling in.

```
allInstances ModuleClass
⇨ #<#<Module Substrate>, #<Module ScriptX>, #<Module Scratch>
currentModule()
⇨ #<Module Scratch>
```

In addition to the Scratch module, ScriptX creates two system modules. The Substrate module is the module in which all global names from the core classes are defined and implemented. The ScriptX module is the interface module for the Substrate module. It is through the ScriptX module that names defined in the substrate are exported to other modules, including the Scratch module.

Note – The system does not allow you to redefine the ScriptX or Substrate modules. The ScriptX and Substrate modules implement the interface/implementation model, as described in the section "Organizing Modules" on page 219.

The Scratch module is suitable for testing brief scripts that are compiled from scratch each time they run. But if you want to save a project you have compiled, so that it can run in the Kaleida Media Player, it must be compiled and saved in another module.

An authoring environment can potentially define other "default" modules in addition to the ScriptX and Scratch modules. For example, an authoring tool could define new class libraries that are specific to some environment, or accessories that allow you access to platform-specific features. When you start up ScriptX in the Listener, the compiler is placed into the Scratch module. Other authoring environments may define a different "working" module.

The ScriptX module is the standard interface module which all other ScriptX modules and authoring environments include. This module exports all variable names in the ScriptX core classes, including names of classes, system objects, generic functions, and global variables. When you create a new module in which you plan to implement ScriptX code (an implementation module), you should import (use) the ScriptX module so that you have access to the core classes.

Conventions for Using Modules

The following suggestions make for good ScriptX programming practice, and good documentation too:

- Store each module definition by itself in a separate source file.
- Do not compile a single source file in more than one module.
- Do not store compiled code from a single source file in more than one container.

- Always store a module in the same container as code objects (classes, functions, etc.) that are compiled in that module.
- Use `in module` at the top of each source file that contains code for a given module. Everything in the file should belong to and be compiled in that one module. In this way, you can tell from looking at each source file which module it belongs to.
- Remember to delete old modules from containers as you redefine containers and restructure your title.

Module Concepts

Modules provide a mechanism for creating and managing multiple *namespaces*, similar to the packaging system in the Lisp language and the proposed "namespace" mechanism in C++. Each module has its own *global* scope, and variables that are declared global are only global within the boundaries of that module. The only real global names are the names of modules themselves.

This section describes the terms and concepts that underlie the use of modules in ScriptX:

1. To *export* a name is to make a name that is defined in a given module available for import into other modules. Only names that are exported are public. Each name that is to be public must be explicitly exported.
2. To *import* a name is to make a name that has been exported by another module available for use within a given module. Therefore, to access a name in another module requires two steps: that it be exported from that module and imported into this module. A module can import any name that is public.
3. A *name binding* is an association between a name and a value cell in memory. A ScriptX module can be thought of as a collection of name bindings for objects in a program.
4. A *variable* could be described as the set of name bindings (across modules) for a given value cell in memory. A variable is defined in only one module, the module in which it is initially assigned a value. Although a variable is visible and can be changed in all modules in which it has a name binding, it can be saved in a storage container only in the module which defines it.
5. A module *owns* a variable if it defines it by assigning a value to the name that is associated with the variable. A definition is a statement of ownership which combines the declaration of a name and the assignment of a value to that name.
6. Modules are not a security feature. Modules do not control access to objects in any way. The fact that a module owns a variable does not prevent a program passing a reference to the object that variable refers to on to a program that is compiling in another module. Modules only control access to objects *by name*.

As background reading for this section, you might want to review the ScriptX treatment of scope, lexical names, variables, and assignment from earlier sections of this volume.

Exporting Names

Every global name within a module is either exported (public) or unexported (private). By default, names are not exported. When you define a module, you specify which names are exported—that is, which names are available outside the module for import into other modules. Only names that are exported by some module can be imported by other modules.

Because a ScriptX variable can store a reference to any object, including classes and functions, exporting the name of a class or function makes those objects public, ready for import into other modules.

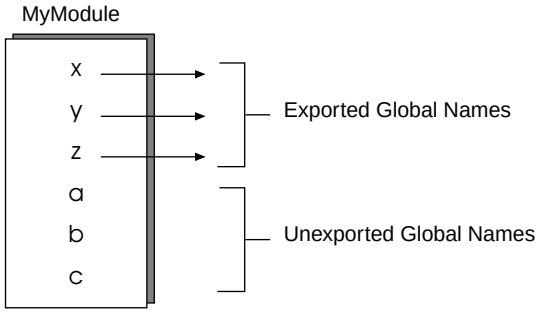


Figure 9-1: Exporting names from modules

A ScriptX module can export a name, even though it does not define the variable that is associated with that name. If a module exports names that are declared and defined by one of its client modules, it acts as an interface for that module. The section “Organizing Modules” on page 219 shows how this feature can be used to create a network of interface and implementation modules.

Importing Names

To access names in another module, you must export those names from that module and then import them into this module. Imported names appear in the new module just as if they had been declared there, and, if they have been defined as variables, their definitions come along as well. The importing module is considered to be using the first module, and the relationship between two modules is called a use relationship. Modules can use and be used by any number of other modules, as long as no circular use relationships are created.

In Figure 9-2, `MyOtherModule` uses `MyModule`. The names `x`, `y`, and `z` that are exported by `MyModule` are imported by `MyOtherModule`. The two variables named `x` in the two modules both point to the same value cell. Thus, if you set `x` to a new value in one module, its value is changed in both modules.

Although the name is exported from one module and imported into the other, access to its value cell is completely symmetric—it can be set or get from either module.

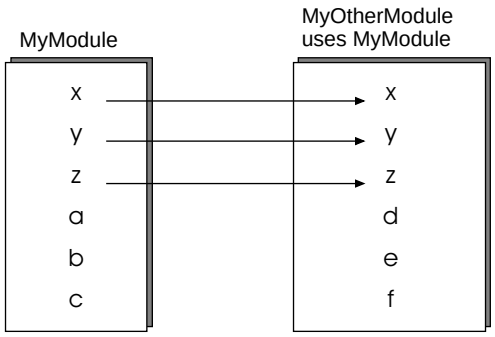


Figure 9-2: Importing names into modules

When you specify that one module uses another module, you can control which names from that module you want to import. You can also rename imported names in your own module, to prevent naming clashes, by prefixing the imported names with a set of characters or by simply giving the variables entirely different names. A renamed variable maintains its original definition. In effect, a ScriptX variable can have a different name in each module. Finally, imported names can also be re-exported, effectively "passing them along" to any modules that use the module that re-exported them.

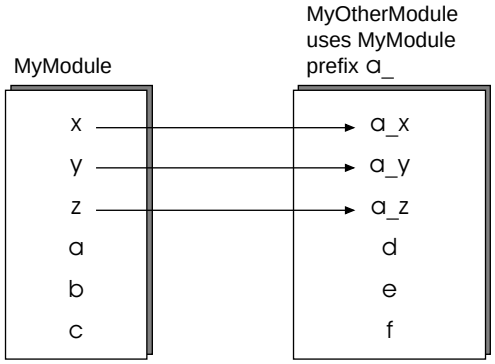


Figure 9-3: Importing names into a module, with a prefix

Name Bindings

A module is a collection of name bindings that the compiler uses at a particular point in the execution of a program. ScriptX name bindings allow for the separation of names and objects, so that objects can have different names in different modules. A different name can be "bound" to a given variable in each module in which it is visible.

A binding is an association between a name and a value cell in memory. A value cell stores a pointer to an object somewhere in memory. Bindings act as a level of indirection between names and value cells. This indirection is what

allows a ScriptX program to run with different namespaces. When the ScriptX compiler switches from one module to another, it is actually switching from one set of bindings to another.

Think of a binding as a way to hang on to an object. Objects do not have to be associated with bindings. References to objects are embedded within other objects, often many levels deep. Expressions that assign a new global name, such as an assignment, `function`, or `class` expression, create a binding.

The following example demonstrates what bindings are created when a simple ScriptX program runs. Although it is not a complete technical description, it shows in a heuristic way how objects are associated with names, through bindings, and how names are used in a ScriptX program to get access to objects. ScriptX manages a global namespace (there is one global namespace for each module), by keeping a table of name bindings.

```
class KittyCat ()
    inst vars
        favoriteFood
end
-- create an instances of Cat
object kiri (KittyCat)
    settings favoriteFood:"tuna"
end
⇨  KittyCat@0xe938c8
```

This program creates four name bindings. The first one is used to get access to the newly created `KittyCat` class, which is itself an object. The next two bindings are associated with instance variable access. Instance variable access is actually through generic getter and setter functions. These generic functions call the appropriate getter and setter methods. ScriptX automatically creates entries for generics in the module's name table. Finally, the script creates an object, an instance of `KittyKat`. This object is associated with a binding, bringing the total of new bindings in the system to four.

The following is a list of bindings that have been created so far in the execution of this program. Names in the first column are associated with objects, including classes and generic functions, in the second column. Lexical names are in lower case because they are interned in their downcase form. (In the scripter, you don't have to be concerned with case; you are free to use uppercase and lowercase to make your scripts more understandable.)

Lexical Name	Points To
`kittycat`	the `KittyCat` class
`favoritefoodgetter`	a generic function
`favoritefoodsetter`	a generic function
`kiri`	a `KittyCat` object at address `0xe938c8`.

Note that on initialization, the `favoriteFood` instance variable slot for the `kiri` object is filled with a string constant (that is, with a pointer to a `StringConstant` object). This string constant has no bindings of its own. A script can only get and set the contents of instance variables through some

binding or bindings. A simple instance variable access is actually a generic function call, which uses the bindings defined for the object and for the generic functions it provides a method for.

```
kiri.favoriteFood
⇨  "tuna"
```

The instance variable access expression `kiri.favoriteFood` is translated into a call to the getter generic function that is associated with this instance variable. Two bindings are required to get a value—one to the generic function and one to the object itself.

```
favoriteFoodGetter kiri
```

Each instance variable is really associated with two name bindings, one for its getter and one for its setter. If an instance variable of the same name exists elsewhere in the current module, it shares the same binding. A generic function handles method dispatch, routing calls to a getter or setter to the method defined by the appropriate object. This allows the same instance variable name to be used by many different objects within a module.

This chapter discusses syntax for importing and exporting variables in modules. What really happens in ScriptX is that a module imports and exports a set of name bindings. As a result, ScriptX modules do not just share values—they share value cells.

Defining Variables Within Modules

Each variable can have only one definition across modules, although that variable can be given a different name when it is imported into another module. The module that provides a definition for that module is considered to *own* that variable, and only the module that owns it can redefine it. When a module is stored into the ScriptX object store, all the variables that it owns and their most recent values are stored with it.

The only truly global names are module names. All variables are *declared* within some module. A variable can only be accessed (referenced) within some module. Since a given value cell can be bound to different names in different modules, and since the same name can be used to denote different variables in different modules, you do not know what variable something is until you know what module it was compiled in.

A variable is *owned* by the module that *defines* it. Only one module owns each variable. The ScriptX class definition, object definition, function definition, and global assignment expressions (a global assignment is a global declaration and assignment all in one expression) all declare and define a variable. Note that a variable defined when an object is assigned to a name. This name is usually declared as a name and assigned a value in the same expression, as in a class, object, or function declaration.

A module does not have to be the module that defines a variable to export a name for that variable. The ScriptX module system allows variables to be exported from one module, and then imported and defined in another module.

That second module, the one that defines the variable, does not have to re-export the variable for that definition to be visible, nor does a module that wants to use that variable need to use the module that defined it. By using the module that exported the variable, its definition comes along automatically. This separation between an exported variable and its definition allows you to organize and build networks of modules which can include circular use relationships between modules, multiple interfaces to the same module, and more comprehensible relationships amongst complex networks of modules that use and export many different variables. “Organizing Modules” on page 219 describes how to use modules in this way.

A single variable can be exported, imported, and renamed in different modules. Any module with access to that variable can change its value, if that variable has not been declared constant. However, only one module is considered to *own* that variable. When a module is stored in the ScriptX object store, it stores all variables it owns with their current values. If a variable is not owned by any modules, it cannot be stored with any of those modules.

The module that owns a variable is the module that provides its definition. A variable that is not defined in any module is not owned by any module.

The `class`, `object`, and `function` declaration expressions declare and assign the variable in one expression, allowing them to be owned by the module that defined them. You can separate declaration and assignment for both objects and functions. For classes, the name must be specified when the class is declared, and that name is automatically declared constant. Thus, a class name is always owned by the module in which it is declared.

A global variable that you create and assign by hand (using an assignment expression) requires two conditions to be owned by a module: you must both declare it and assign it in the same expression:

```
global myVariable := "variable's value"
```

Only declaration and assignment in the same expression confer ownership of a variable; declaring and assigning the variable in different expressions is not the same:

```
global anotherVariable
anotherVariable := "another value" -- anotherVariable is not owned
```

A variable that is declared but not assigned any value does not get saved with a module—the module does not yet *own* that variable. Note that when you define a variable in a ScriptX program you do not necessarily have to give it the same value it will hold when the module is stored. When modules are stored in the ScriptX object store, the module stores the most recent value for each variable that it owns. To make sure a variable is owned by a module without having to assign it its proper value right away, give it a value of `undefined` at declaration time:

```
global VarToBeChanged := undefined
```

Later on in the program, you can assign the value you want to be stored.

A module maintains its own table of name bindings. To export a variable is to allow other modules to have a binding for a particular value cell. To say that a particular module defines a variable means that its value cell was created in that module. The value cell is created when the variable is assigned a value. Other modules that have access to the variable have access only through the value cell in the module where it is defined. To import a variable into a module is to have a binding for that variable in the current module. Other options for using modules, such as `excludes`, `prefix`, and `renames`, also operate on this table of bindings.

If a module attempts to define a variable that is already shared, an error message results and the variable is not redefined. If a variable is exported (shared), then it is still only one variable (not a copy) and can have only one definition. For example, if a shared variable is defined in one module, assigned to in another module, and read in a third module, the third module reads the value that was assigned by the second module.

A shared variable does not have to have the same name in each module that uses it. You can use features like `prefix` and `renames` to give the variable multiple names, but it is still only one variable (not a copy) and has only one definition. Different modules may use different names, but they are still accessing the same location in memory.

If a variable is not exported, then you can use the same identifier to define a separate variable in another module. These variables have the same name within each module, but they refer to different locations in memory.

In this respect, a variable is really just a collection of name bindings, one for each module in which the variable is visible. A variable continues to exist and to occupy memory when the module it is defined in is no longer the module that the compiler is currently using. A variable is visible to scripts in the current module if the current module has a binding for it.

Examples—Ownership of Variables

ScriptX distinguishes between declaring a name and defining a variable. In practice the declaration of a name and the definition of a variable usually occur in the same expression, but the distinction is important.

The `global` expression declares a ScriptX lexical name. A name that is declared global is visible only within the module in which it is declared, unless that name is exported. In this way, the same lexical name can be declared in different modules.

In the following example, the module `Europe` exports four names: `finland`, `albania`, `portugal`, and `ireland`.

```
module Europe
    uses ScriptX
    exports finland, albania, portugal, ireland
end
in module Europe
global finland, albania:19, portugal := 27
```

When the global expression is combined with an assignment expression, the resulting expression both declares a lexical name and defines a variable. Since `albania` and `portugal` are assigned values, `albania` and `portugal` are both declared as names and defined as a variables.

At this point, a program can access the values of `albania` and `portugal`, since both names have been defined as variables. Although `finland` is declared as a name, it is not assigned a value, so `finland` is not defined as a variable. Likewise, `ireland` has not been assigned a value, so it cannot be accessed as a variable. Since `ireland` has not been declared either, assigning a value to `ireland` will cause the compiler to issue a warning. By contrast, `finland` has been declared, so the compiler does not issue a warning when a value is assigned to `finland`.

If compilation and execution switch to another module that uses the `Europe` module, the variables that are defined in `Europe` are visible there.

```
module World
    uses ScriptX, Europe
end
in module World
albania
⇨  19
portugal
⇨  27
```

At this point in the execution of the program, `finland` and `ireland` have still not been defined as variables. When they are assigned values in the `World` module, they are defined in the `World` module. Even though `finland` was declared in the `Europe` module, and `ireland` was exported by `Europe`, but not declared, both variables are now defined in `World`.

```
in module World
finland := 11
ireland := 94
```

If compilation reverts to the `Europe` module, `finland` and `ireland` retain their definitions as variables. They are visible in both `Europe` and `World`.

```
in module Europe
finland
⇨  11
ireland
⇨  94
```

Although all four of the variables in this example are exported by `Europe`, and three of them are declared there, it is where they are defined that determines how they are saved. Since `albania` and `portugal` are defined in the Europe module, they are saved to the object store with `Europe`. By contrast, `finland` and `ireland` are actually defined in the `World` module.

Only declaration and definition in the same module confers ownership. Since `finland` is not declared in the `World` module, it cannot be owned by any module, and is not saved. But `ireland`, even though it was exported by `Europe`, is declared and defined in the `World` module. Thus, `ireland` is saved with the `World` module.

The object, class, and function definition expressions implicitly declare a global name and define a variable as well, creating a binding for that variable in the current module.

```
module Universe
    uses ScriptX
end
in module Universe
class Star () end
object sun (Star) end
function orbit a b ->
    format debug "1%* orbits 2%*\n" #(a, b) @normal
```

In this example, `sun`, `star`, and `orbit` are declared as lexical names and defined as variables (a class definition is a constant). Each of these variables has a binding in the `Universe` module, meaning that a lexical name is associated with the variable in that module. Since `sun`, `star`, and `orbit` are defined as variables in `Universe`, they can be saved to the object store with `Universe`, and they can be exported to other classes.

Defining and Using Modules

Modules can be defined only at the top level in a ScriptX program. To define a module, use the module definition expression. A module definition expression returns a `ModuleClass` object. The complete module definition expression is summarized here. Each of its optional clauses is covered in greater detail later in this chapter.

▼ `module` *ModuleName*

`[ exports` *variable, variable, variable,* `. . . ]`

`[ exports [ readonly ] instance variables` *variable,* `. . . ]`

`[ uses` *module, module, module,* `. . . ]`

`[ uses` *module* `with`

`[ imports everything ]`

`[ imports` *variable, variable, variable,* `. . . ]`

`[ imports [ readOnly ] instance variables`

variable, variable, variable, `. . . ]`

`[ excludes` *variable, variable, variable,* `. . . ]`

`[ excludes [ readOnly ] instance variables`

variable, variable, variable, `. . . ]`

`[ exports everything ]`

`[ exports` *variable, variable, variable,* `. . . ]`

```
        [ exports [ readOnly ] instance variables
            variable, variable, variable, . . . ]
        [ prefix prefix ]
        [ renames oldName:newName, oldName:newName, . . . ]
        [ renames [ readOnly ] instance variables
            voldName:newName, oldName:newName, . . . ]
    end]
end
```

A `module` definition expression can include two main sections, which are both optional, and can be specified in any order. The `exports` options (described in the section "Exporting Variables to Other Modules" on page 206) specify which names are exported from this module. The `uses` options (described in the section "Importing Variables From Other Modules" on page 209) specify which names are imported from other modules.

In this syntax definition, and in many of the examples on the following pages, the individual sections of the module definition are shown on separate lines. You can specify a module definition in this way, all on a single line, or in any combination:

```
module foo exports x, y, z uses bar with prefix Z end end
```

```
module MyModule
    exports varOne, varTwo
    uses foo with excludes Z end
end
```

Defining a Module

The first line of the module definition specifies the name this module is to be known by:

▼ `module` *ModuleName*

```
...
end
```

Unlike ScriptX global variables, which are only global within the scope of a given module, module names are truly global. There is only one namespace for module names. Because of this, you should choose a module name that is unique and distinct:

```
module PaintInterfaceUser
module DeveloperModule_Rev15_345a
module LotsOfSillyClassesThatOnlyPartiallyWorkTogether
```

By convention, module names are capitalized according to the same rules as class names. Initial letters are capitalized, with every succeeding word also capitalized.

Redefinition of Modules

Modules can be redefined. When you redefine a module, the new definition completely replaces the existing definition. By redefining a module, you can export additional variables, or define new use relationships.

Redefinition of a module does not break existing code at runtime. For example, if you redefine a module to exclude some variable from another module, which the module was already using, the module continues to use that variable. It is possible to import or export additional variables, but not to remove a variable from the list of variables that are imported or exported. The following example demonstrates this:

```
module Blue exports ink, paper end
module Pink uses Blue end
in module Blue
global ink := "blue"
global paper := "white"
in module Pink
global stone := ink + paper
⇨  "bluewhite"

-- now redefine Pink and change the value of ink from within Blue
module Pink uses Blue with excludes ink end end
in module Blue
ink := "red"
in module Pink
global Stone := ink + paper
⇨  "redwhite"
```

The global variable `ink` is still accessible in module `Pink`, even though it has been redefined to exclude `ink`, and `ink` continues to reflect the value that was set for it in module `Blue`.

The following example shows that it is possible to end up with more than one binding for a variable in a given module:

```
module Green exports grass, leaves end
module Brown uses Green end
in module Green
global grass := "green"
global leaves := "green"
in module Brown
global compost := grass + leaves
⇨  "greengreen"

-- now redefine Brown so that it renames grass, and change value
module Brown uses Green with renames grass:herbs end end
in module Green
grass := "brown"
in module Brown
```

```
compost := grass + leaves
⇨  "browngreen"
global moreCompost := herbs + leaves
⇨  "browngreen"
```

In the module `Brown`, the variable `grass`, as defined by `Green`, is accessible through both the original name and its new name. It has two valid bindings.

Developers should be aware that the redefinition of modules is a convenience at runtime. Redefinition of a module cannot resolve all possible cases. If a program needs to redefine a module to exclude or rename variables that are already being used in existing code, the best option is to recompile.

Switching Modules

All ScriptX expressions are compiled within the context of a given module. To move the scope of the ScriptX compiler from one module to another, use the following expression:

▼ `in module` *ModuleName*

The `in module` expression places the ScriptX compiler (and therefore your Listener window or authoring environment) into the scope of the module specified by *ModuleName*, giving you access to all the variables defined in that module or imported from other modules. If the module you specify does not exist, ScriptX reports an exception. The ScriptX compiler remains in the specified module until `in module` is invoked again, placing the compiler in a different module. The `in module` expression is allowed only at the top level in your program.

Note – The ScriptX expression `in module` does not take a `ModuleClass` object as its target. It operates on a name literal, the scripter name of a module. Unlike other names in ScriptX, names of modules are not lexical names. With lexical names (names of constants and variables), the name represents the object itself. The name of a module is like a label for the module, a label which the `in module` expression recognizes. However, ScriptX functions that take a module as a parameter, such as the generic functions `load` and `store`, must be passed the `ModuleClass` object itself.

If you need access to the ScriptX core classes in your module, make sure the module that you specify in an `in module` expression has been defined to use the ScriptX module.

Generally, `in module` is used in a ScriptX program after its module definition, but before the remainder of the expressions (class definitions, variable declarations, and so on) that make up the program to be compiled within that module.

```
module DefinitionModule
    exports FirstClass, sumThem
    -- this exports getters and setters
    exports instance variables a, b, c
    uses ScriptX
end

module TestingModule
    uses ScriptX, DefinitionModule
end

in module DefinitionModule
class FirstClass ()
    instance variables a, b, c
    instance methods
        method init self #rest args #key a:(10) b:(10) c:(10) -> (
            apply nextMethod self args
            self.a := a
            self.b := b
            self.c := c
        )
        method sumThem self -> (
            print (self.a + self.b + self.c)
        )
end

in module TestingModule
global t := new FirstClass a:20
sumThem t
⇨  40
```

A new module can be defined within the scope of any other module, with the caveat that the module cannot use another module that is not yet defined. (For information on circular relationships, see page 211.) Defining a module does not switch the compiler into that module. Only the `in module` expression switches the compiler into another module..

Importing and Compiling Files with fileIn

The generic function `fileIn`, implemented by `DirRep` and `ByteStream`, is available only with the ScriptX Language and Class Library. (The Kaleida Media Player does not include the bytecode compiler, which compiles and executes scripts.) A large ScriptX project is typically compiled from a build file, and this build file usually includes a series of calls to `fileIn`. This design makes it easy to divide a large project into an number of smaller source files which are compiled in a set order.

Scripts that are imported using `fileIn` run in the `Scratch` module by default. The `fileIn` generic function defines a `module` keyword argument, which takes a `ModuleClass` object. This keyword can be used to specify which module the file is compiled in.

```
-- the file dogs.sx will be compiled in the AnimalInterface module
fileIn name:dogs.sx module:(getModule @AnimalInterface)
```

The `in module` expression, if it appears within a source file that is read in using `fileIn`, overrides `fileIn` in determining which module a given script is compiled in, but only within the scope of that file. Compilation reverts back to the previous module after `fileIn` finishes compiling and executing the script and returns a value.

The `fileIn` generic function It is possible to specify a module `fileIn` also specifies a module using the `module` keyword, the code within the file being imported is compiled within the named module. The `fileIn` generic function is described in the class definition of `DirRep` in the *ScriptX Class Reference*.

One disadvantage to using the `module` keyword with `fileIn` is that it can be hard to tell at a glance which module a given script is compiled in. For that reason, many programmers ignore the `module` keyword. They prefer to specify the current module explicitly at the top of each source file, using `in module` as the first expression in the file.

In contrast with `fileIn`, scripts that are read in and compiled using the **Open Title . . .** menu command in the current ScriptX Listener window run in the current environment, and place the ScriptX compiler directly into the modules specified by any `in module` expressions in the file.

Module Objects

The module definition, like all other ScriptX expressions, yields an object, in this case, an instance of `ModuleClass`. However, unlike class or function definition expressions, the name of the module is not a variable which is assigned to that module object, so you cannot use the module name to refer to the module object as you would any other variable.

In most cases you simply use the module as an environment in which you create and manipulate other objects. There may be cases, however, in which you need access to a `ModuleClass` object itself:

- When you want to dispose of that module, that is, allow the module itself and all the values of the variables it holds to be claimed by the ScriptX garbage collector.
- When you want to save a module to a ScriptX storage container.

Once a module has been defined, you can get access to the module object itself using either the `getModule` or `currentModule` functions:

◆ `getModule` *ModuleName*

The global function `getModule` returns the `ModuleClass` object that is specified by the given name. If you do not specify a valid module, `getModule` returns `false`. Unlike the `in module` expression, `getModule` cannot define a new module. The module referred to by *ModuleName* must already have been defined. Note that `getModule` does not affect which module ScriptX is currently compiling in. Only the `in module` expression can be used to switch from one module to another.

The `getModule` function returns `false` if it does not return a module. You can use this feature to test whether a module is defined or not:

```
if not (getModule @whatever) do
(
    -- This block executes only if the module @whatever is not defined
)
```

Once you have a module object to operate on, you can add that module to a container in the object store (as described in "Storing Modules" on page 227). For example, to append the `Definition` module to the `tc` title container:

```
append tc (getModule @Definition)
```

You can use any other functions that operate directly on module objects (such as `fileIn`, described above). Of course, the `ScriptX`, `Substrate`, and `Scratch` modules, which are defined by the system, cannot be saved.

- `currentModule()`

The global function `currentModule`, which is actually implemented as a macro, returns the `ModuleClass` object in which ScriptX is currently compiling. It reflects the state of the compiler during compilation, and is not meant to be used at runtime. ScriptX does not have a "currently active" module at runtime.

Do not use `currentModule` in scripts. It exists only for informational and debugging purposes.

- `deleteModule` *ModuleName*

A module can be removed from memory using `deleteModule`, but only if it is not being used by any other module.

The global function `deleteModule` can be called with either the name of the module, or a `ModuleClass` object as its argument. If the module is being used by another module, and cannot be deleted, `deleteModule` reports the `deletingUsedModule` exception. If the modules that are using it are then deleted, the module can be deleted.

Exporting Variables to Other Modules

To specify which global variables are exported outside this module, use the `exports` section of the module definition expression:

▼ `module` *ModuleName*

```
    exports variable, variable, variable, . . .
end
```

The `exports` reserved word is followed by a list of variable names that are visible outside this module. Those variables can be variables that are defined, or will be defined, within this module. They can also be variables that were imported from other modules, and are thus "passed along" by this module. Variable names can be specified on separate lines, on the same line separated by commas, or in any combination.

```
module Australia
    uses ScriptX
    exports beer, engineers
end
in module Australia
global beer := "Foster's"
global engineers := #("Wainwright", "Nicholson", "Williams")
```

You can use multiple `exports` sections in your module definition, which is useful for documentation purposes or to group sets of exported variables together into logical groups. The resulting module exports all of the variables in all the `exports` sections.

```
module California
    uses ScriptX
    exports chips, software
    exports almonds, avocados, cherries, figs, lettuce, wine
    exports entertainment, movies
end
in module California
global chips := #("PowerPC", "Intel")
global software := "ScriptX"
. . .
```

A module must explicitly export all the names that are to be visible outside the module. The point of modules is really to exclude names, to export only those names that are required by other modules. Consider the following analogies between ScriptX and C.

- The `exports` clause is a declaration of names that are visible outside the module, analogous to the `extern` statement in C.
- In C, you explicitly declare that a global variable is not visible elsewhere, by declaring that it is `static`. In ScriptX, a global variable is like a static variable by default. It must be explicitly exported to be visible in other modules. Thus, the default case is reversed.
- In C, the boundary between name spaces is the file. In ScriptX, the boundary is the module, which can extend across multiple files. Of course, a well-constructed program uses only one module in a given file.
- An interface module, defined later in this chapter (see "Organizing Modules" on page 219), is really analogous to a header file in C.

Exporting Classes

You can export a class simply by specifying the variable that contains that class in an `exports` section of a module definition. However, be aware that exporting the class name alone does not automatically provide access to that class's methods and variables. To have full access to an exported class from another module, you must not only export the variable that contains that class, but you must also explicitly export.

- The names of any methods (generic functions) defined by that class that do not already have a name binding, including any methods that the exported methods also use
- The setter and getter generic functions for the class and instance variables defined by that class

If you do not explicitly export these variables, those parts of that class are unavailable to any modules that use this module. For example, suppose you have the following class definition, where the `Person` class has two instance variables, `name` and `age`, and two methods, `printName` and `printAll`.

```
class Person ()
    instance variables
        name, age
    instance methods
        method printName self -> (
            prin ("My name is " + self.name + "\n") @Normal debug
        )
        method printAll self -> (
            printName self
            prin ("My age is " + self.age + "\n") @Normal debug
        )
end
```

In order for both instance variables and both methods to be available outside the module, you must explicitly export all of the name bindings that are defined by the class:

```
module PersonModule
    -- the class itself
    exports Person
    -- Person's instance variables (getter and setter generics)
    exports nameSetter, nameGetter, ageSetter, ageGetter
    -- Person's methods
    exports printName, printAll
end
```

Exporting Instance Variables

To make exporting instance variables easier, the ScriptX module definition includes special syntax for exporting instance variables:

▼ `module` *ModuleName*

```
   exports [ readonly ] instance variables variable, variable, . . .
end
```

The `instance variables` reserved words can be shortened to `instance vars` or simply `inst vars`. The list of variables can be supplied on one line separated by commas, on separate lines, or in any combination.

The optional `readonly` reserved word exports the list of instance variables specified by `variables` in a read-only form; that is, they can be queried but not changed.

The `instance variables` part of an `exports` clause is simply shorthand for exporting the setter and getter generic functions for those variables. If the `readonly` option is specified, only getter methods are exported. The following module definitions are equivalent:

```
module MyModule
   exports readonly instance variables x
   exports instance variables y
end

module MyModule
   exports xGetter, ySetter, yGetter
end
```

Importing Variables From Other Modules

To import variables from one module into another, use the `uses` clause of a module definition. There are two forms of `uses`: the short form (`uses`) that simply imports all the exported variables from the given module into this module, and the long form (`uses . . . with`), which allows control over which variables should be imported, and how they should be handled. For example, a `uses . . . with` clause can change their names or reexport them.

For the complete syntax for defining a module, see page 200. The syntax for both `uses` clauses in a module definition is as follows:

▼ `module` *ModuleName*

```
   [ uses module, module, module, . . . ]
   [ uses module with
      [ imports everything ]
      [ imports variable, variable, variable, . . . ]
      [ imports [ readOnly ] instance variables
         variable, variable, variable, . . . ]
      [ excludes variable, variable, variable, . . . ]
      [ excludes [ readOnly ] instance variables
         variable, variable, variable, . . . ]
      [ exports everything ]
```

```
        [ exports variable, variable, variable, . . . ]
        [ exports [ readOnly ] instance variables
            variable, variable, variable, . . . ]
        [ prefix prefix ]
        [ renames oldName:newName, oldName:newName, . . . ]
        [ renames [ readOnly ] instance variables
            voldName:newName, oldName:newName, . . . ]
    end]
end
```

The line containing the reserved word `uses` specifies the modules whose variables are to be imported. The first form is simply the reserved word `uses` followed by the other modules whose variables are to be imported. Note that with this form, *all* variables are imported from the specified modules. The modules can be specified on the same line separated by commas, on separate lines, or in any combination. Any module you specify in a `uses` definition must already have been defined. You can have a single `uses`, or you can have multiple `uses` clauses in the same module definition. Note that if you want the variables in the ScriptX core classes to be available to the expressions in this module, you have to explicitly use the ScriptX module.

The second form of `uses` clause, `uses with`, allows more control over how imported variables from individual modules are handled. The `uses with` clause specifies options for a single module. You must use an individual `uses with` clause for each module.

The `uses with` clause contains several sub-clauses, referred to in this chapter as *options*, all of which are optional and may be included in any order. Also, although they have been presented here on multiple lines and indented, they may also be specified on a single line, or in any combination.

The `uses` form is equivalent to the `uses with` form with `imports everything`. That is, the following two definitions are equivalent:

```
module Mocha
    uses ScriptX
end


module Mocha
    uses ScriptX with
        imports everything
end
```

All `uses with` option are described in the following sections.

uses

▼ `module` *ModuleName*

`uses` *module, module, module,* `. . .`

`end`

This is the simplest form of module definition that imports variables from other modules. All variables that are exported from the specified modules are imported into the module given by *ModuleName*.

A Note on Circular Use Relationships

A circular use relationship is defined as two modules that have `uses` clauses that import variables from each other. In this example, `Foo` uses `Bar` which uses `Foo`:

```
-- Don't define a circular relationship like this
module Foo
    exports x, y, z
    uses Bar
end

module Bar
    exports a, b, c
    uses Foo
end
```

Because a module's `uses` clause can only name modules that have already been defined, explicit circular use relationships cannot occur (the first definition returns a warning stating that module `bar` does not exist).

It is possible to define implicit circular use relationships between modules by exporting variable names from one module and defining those variables in another. This method of defining and using modules is described in "Organizing Modules" on page 219.

uses with imports

▼ `module` *ModuleName*

`uses` *module* `with`

`imports everything`

`imports` *variable, variable, variable,* `. . .`

`imports [ readOnly ] instance variables`

```
            variable, variable, variable, . . . ]
    end
end
```

The optional `imports` clause is used to specify which variables to import from the module specified by *module*.

The `imports` option, when used with the reserved word `everything`, simply imports all the variables that the given module exports. If you use `imports everything`, you cannot use any of the other forms of `imports` in the same `uses with` clause. Also, if you omit all `imports` options in a `uses with` clause, `imports everything` is assumed. The advantage to using `imports everything` rather than a simple `uses` clause is that other options, such as `prefix` and `renames`, are also available.

The `imports` option followed by a list of variables specifies exactly which variables to import into this module. The variable names can be specified on separate lines, on the same line separated by commas, or in any combination.

The `imports` option with the `instance variables` reserved words (or the `readOnly instance variables` reserved words) is syntactic shorthand for importing the setter and getter generic functions for the named instance variables (or, in the case of `readOnly`, only the getter method). The `instance variables` reserved words can be shortened to `instance vars` or simply `inst vars`. See “Exporting Classes” on page 208 for more information on importing and exporting classes and their methods and variables.

The following example creates a module and exports three global variables it defines.

```
module XYZ
    uses ScriptX
    exports x, y, z
end
in module XYZ
global x:10, y:"foo", z:#(56,567)
```

The following module uses `XYZ` and imports all variables from it.

```
module XYZimport1
    uses ScriptX
    uses XYZ with imports everything end
end
in module XYZimport1
print x
⇨  10
print y
⇨  "foo"
print z
⇨  #(56, 567)
```

The third module uses XYZ, but it imports only x and z from this module. Since y is undefined within this module, attempting to access y from this module reports an exception.

```
module XYZimport2
    uses ScriptX
    uses XYZ with imports x, z end
end
in module XYZimport2
print x
⇨  10
print z
⇨  #(56,567)
print y -- this reports an exception
⇨  -- ** XYZimport2:y does not have a variable value \
       (UninitializedVariable)
```

uses with renames

▼ `module` *ModuleName*

```
    uses module with
        renames oldName:newName, oldName:newName, . . .
        renames [ readOnly ] instance variables
            oldName:newName, oldName:newName, . . .
    end
end
```

The optional `renames` clause is used to import a variable from the given module and give it a new name. The definition of the variable, if any, is still valid in the new module under the new name. The `renames` keyword is followed by any number of *oldName* and *newName* pairs, on the same line separated by commas, on separate lines, or in any combination. The old and new variable names, *oldName* and *newName*, are separated by colons. The `renames` clause, used without an `imports` clause, assumes `imports everything`.

The `renames` clause with the `instance variables` reserved words (or the `readOnly instance variables` reserved words) is syntactic shorthand for renaming the setter and getter generic functions for the named instance variables (or, in the case of `readOnly`, only the getter method). See "Exporting Classes" on page 208 for more information on importing and exporting classes and their methods and variables.

The `renames` option overrides both `import` and `prefix`; the variables specified by `renames` are both imported and given the names specified by *newName*.

The following script sets up module XYZ, as in the previous example.

```
module XYZ
    uses ScriptX
    exports x, y, z
end
in module XYZ
global x:10, y:"foo", z:#(56,567)
```

This module uses module `XYZ`, imports only `x`, and renames it `externalX`.

```
module XYZrename1
    uses ScriptX
    uses XYZ with
        renames x:externalX
    end
end
in module XYZrename1
print externalX
⇨  10
print y
⇨  "foo"
```

The next module explicitly imports `x` and `y`, but renames `x`. Renaming `x` overrides `import x` so that `x` is imported, but with a new name.

```
module XYZrename2
    uses ScriptX
    uses XYZ with
        imports x, y
        renames x:otherX
    end
end
in module XYZrename2
-- this reports an exception
print x
⇨  -- ** XYZrename2:x does not have a variable value \
       (UninitializedVariable)
print otherX
⇨  10
print y
⇨  "foo"
```

uses with prefix

▼
```
module ModuleName
   uses module with
      prefix prefix
   end
end
```

The optional `prefix` clause is used to import variables and rename them by attaching a prefix (specified by *prefix*) to the variable name. Prefixing variable names is useful for resolving conflicts in variable names between modules or for simply indicating which module a variable came from. The `prefix` option, used without an `imports` option, assumes `imports everything`.

You can assign a prefix to all imported variables and then rename specific variables by using the `renames` clause for those variables.

The following script sets up module `XYZ`, as in the previous example.

```
module XYZ
   uses ScriptX
   exports x, y, z
end
in module XYZ
global x:10, y:"foo", z:#(56,567)
```

Module `XYZprefix1` imports all variables from module `XYZ` and prefixes variables from that module with `XYZ_`.

```
module XYZprefix1
   uses ScriptX
   uses XYZ with
      prefix XYZ_
   end
end
in module XYZprefix1
print x
⇨  -- ** XYZprefix1:x does not have a variable value
      (UninitializedVariable)
print XYZ_x
⇨  10
print XYZ_y
⇨  "foo"
```

Module `XYZprefix2` uses both a `prefix` clause and a `renames` clause. The `renames` clause overrides the prefix for specific variables.

```
module XYZprefix2
   uses ScriptX
   uses XYZ with
      prefix ext_
      renames x:fumbleWhizzy
```

```
    end
end
in module XYZprefix2
print ext_x
⇨  -- ** XYZprefix2:ext_x does not have a variable value
       (UninitializedVariable)
print ext_y
⇨  "foo"
print fumbleWhizzy
⇨  10
```

uses with excludes

▼ `module` *ModuleName*

```
    uses module with
        excludes variable, variable, variable, . . .
        excludes [ readOnly ] instance variables
            variable, variable, variable, . . .
    end
end
```

The optional `excludes` clause is used to explicitly exclude individual variables from a module. The list of variable names can be on a single line separated by commas, on individual lines, or in any combination. If you use `excludes` without an `imports` clause, `imports everything` is assumed.

The `excludes` option with the `instance variables` reserved words (or the `readOnly instance variables` reserved words) is syntactic shorthand for excluding the setter and getter generic functions for the named instance variables (or, in the case of `readOnly`, only the getter method). See "Exporting Classes" on page 208 for more information on importing and exporting classes and their methods and variables.

Like `renames` and `prefix`, `excludes` is used to prevent imported variable names from clashing with variable names in the current module. However, `excludes` is also useful when there are a lot of imported variables and most, but not all, are relevant; rather than specifying all the variables this module is interested in with an `import` clause, you can simply specify the variables that this module is not concerned with using `excludes`.

The example uses module `XYZ`, just as in the previous section.

```
module XYZ
    uses ScriptX
    exports x, y, z
end
in module XYZ
global x:10, y:"foo", z:#(56,567)
```

This module imports all the variables from module `XYZ`, excluding `y`.

```
module XYZexclude1
    uses ScriptX
    uses XYZ with
        imports everything
        excludes y
    end
end
in module XYZexclude1
print x
⇨  10
print y
⇨  -- ** XYZexclude1:y does not have a variable value \
       (UninitializedVariable)
```

uses with exports

▼ `module` *ModuleName*

```
    uses module with
        exports everything
        exports variable, variable, variable,  . . .
        exports [ readOnly ] instance variables
            variable, variable, variable,  . . .
    end
end
```

The optional `exports` clause is used to re-export any variables that have been imported from *module* using the `imports` or `renames` clauses (sometimes called "transitive exporting"). You also use it to export variables created within this module. If the variables have been renamed or prefixed upon import (using the `renames` or `prefix` clauses), those variables are exported using those new names.

The `exports` option, when used with the reserved word `everything`, simply re-exports all the variables that were imported from *module*. If you use `exports everything`, you cannot use any of the other forms of `exports` in the same `uses with` clause.

The `exports` option followed by a list of variables specifies exactly which of the variables to re-export. The variable names can be specified on separate lines, on the same line separated by commas, or in any combination.

The `exports` option with the `instance variables` reserved words (or the `readOnly instance variables` reserved words) is syntactic shorthand for re-exporting the setter and getter generic functions for the named instance variables (or, in the case of `readOnly`, only the getter method). See "Exporting Classes" on page 208 for more information on importing and exporting classes and their methods and variables. The `instance variables` reserved words can be shortened to `instance vars` or simply `inst vars`. All are equivalent.

The example uses module XYZ, just as in the previous section.

```
module XYZ
    uses ScriptX
    exports x, y, z
end
in module XYZ
global x:10, y:"foo", z:#(56,567)
```

Module XYZexport, which exports its own variables (a and b), imports everything from module XYZ, renames x to otherX, and transitively exports otherX and y.

```
module XYZexport
    exports a, b
    uses ScriptX
    uses XYZ with
        imports everything
        renames x:otherX
        exports otherX, y
    end
end

in module XYZexport
global a := "croissant"
global b := pi
print otherX
⇨  10
```

Module moreXYZexport uses XYZexport and imports all. The variables that get imported are y (from module XYZ), otherX (which is actually the variable x from module XYZ) and a and b (from module XYZexport).

```
module moreXYZexport
    uses ScriptX
    uses XYZexport with
        imports everything
    end
end

in module moreXYZexport
print a
⇨  "croissant"
print b
⇨  3.14159
print y
⇨  "foo"
print otherX
⇨  10
print x
⇨  -- ** moreXYZexport:x does not have a variable value \
       (UninitializedVariable)
```

Organizing Modules

This section describes a conventional approach to organizing ScriptX programs within modules called the interface/implementation model. This model can be used to create larger networks of modules. It allows modules to be managed without the need for complex use relationships that are difficult to understand.

For a simple ScriptX title that is to be distributed to others, you might define a single module that uses the ScriptX module. It can also use other modules—such as modules associated with library or accessory containers.

```
module MyOwnTitle uses ScriptX end
in module MyOwnTitle
-- build your title here . . .
```

For simple programs, encapsulating code within modules and importing and exporting variables is reasonably straightforward. Larger and more complex programs, with multiple modules that use and are used by each other, can create complex dependencies. In large networks of modules, it may become difficult to keep track of the relationships between modules.

The interface/implementation model is a model for designing networks of modules. Using the interface/implementation model, you can create modules with implicit circular use relationships. You can combine interfaces to construct customized, general interfaces that are based on the needs of the client module.

The interface/implementation model can also be used to define multiple interfaces to the same ScriptX program. For example, a code library could have an interface for end users and an interface for programmers—without duplicating any code

This section describes how the interface/implementation model can be used to create a better structure for large ScriptX programs.

Interfaces, Implementations, and Clients

The interface/implementation model for organizing modules separates modules into three types, which differ in the way they are defined and the way they are used.

- *Interfaces* are modules that export variables, but do not provide any definitions for those variables. Interface modules are never used in an `in module` expression and should have no code associated with them at all. Interface modules are used by other modules, but do not use any other modules.
- *Implementations* are modules that provide the definitions for the variables exported from interfaces, but do not export any variables themselves.
- *Clients* are simply modules that use an interface module. Implementation modules are clients of their interfaces. Other modules that in turn use interfaces are also clients.

Exporting Variables Defined Elsewhere

The ability to separate a module's interface and implementation depends on a feature of module interaction in ScriptX where the module that exports a variable is not the same module that owns its definition. Indeed, the module that owns the variable definition does not have to export that variable for its definition to be accessible. (You might want to review the discussion of variable definitions that begins on page 196.) This section contains a simple example which should clarify this relationship.

The module `BakeryInterface` exports the variable `rye`, but does not provide a definition for that variable. `BakeryInterface` is an interface module. Note that `BakeryInterface` does not use any other modules (including the ScriptX module).

```
module BakeryInterface
    exports rye
end
```

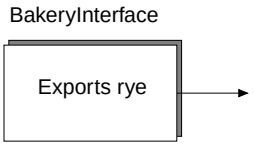


Figure 9-4: Interface module

A second module, `Bakery`, uses the `BakeryInterface` module and provides a definition for the imported variable `rye`. `Bakery` is the implementation module for `BakeryInterface`.

```
module Bakery
    uses BakeryInterface
end
in module Bakery
global rye := "rye bread"
```

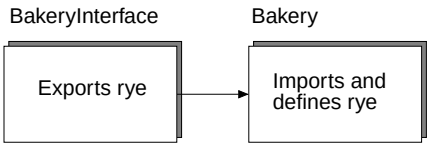


Figure 9-5: Implementation modules

The `Bakery` module does not, and should not, re-export the `rye` variable. The purpose of the `Bakery` module is simply to provide an implementation for variables exported elsewhere; it is not used by any other modules.

Now, any modules that use `BakeryInterface` (clients of the `BakeryInterface` module) automatically receive the definitions of those variables from `Bakery` even though they were not explicitly exported. For example, if the `Delicatessen` module uses `BakeryInterface`, `Delicatessen` has access to the definition of `rye` in `Bakery` even though `Bakery` didn't export `rye` and `Delicatessen` didn't use `Bakery`.

```
module Delicatessen
    uses ScriptX, BakeryInterface
end

in module Delicatessen
print rye
⇨  "rye bread"
```

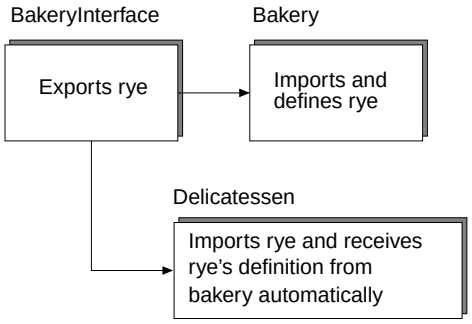


Figure 9-6: Client modules

Note that module `Delicatessen` needs to use the ScriptX module to have access to the `print` function. Like other global functions, `print` is a part of the object system, and not of the ScriptX language itself.

Although using modules in this way may seem like an unnecessary level of indirection, it allows you to organize and combine interface modules freely without worrying about where the contents of those interface modules are defined.

Figure 9-6 actually depicts the relationship between the three "system modules" that are defined by ScriptX: the `ScriptX`, `Substrate`, and `Scratch` modules. In Figure 9-7, the diagram in Figure 9-6 has been relabeled to demonstrate this relationship between system modules. The `ScriptX` modules acts as an interface module, and the `Substrate` module as an implementation module. The `Scratch` module, and any user-defined modules that use the `ScriptX` module, are clients of the `ScriptX` module.

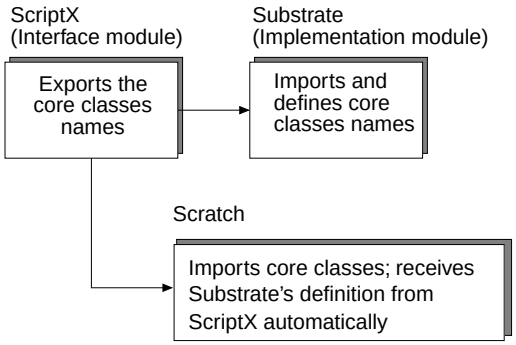


Figure 9-7: The ScriptX, Substrate and Scratch modules

Circular Module References

The separation of interface and implementation in modules allows you to create implicit circular relationships between modules that would not otherwise be possible. Direct circular use relationships in a module definition are not possible because of forward references to modules that have not yet been defined. However, by defining separate interface and implementation modules, you can create circular references to modules such that there are no forward references.

Here is an example of a small set of modules, for draw and paint operations. The implementation for each group of operations requires the use of the other, a circular reference. Using the interface/implementation model, the circularity is broken as all the interface modules are defined before any module tries to use any other module.

```
module DrawInterface
    exports line, rectangle, circle
end

module PaintInterface
    exports pencil, brush, fill
end

module DrawImplementor
    uses ScriptX, DrawInterface, PaintInterface
end

module PaintImplementor
    uses ScriptX, PaintInterface, DrawInterface
end
```

If you were to draw the relationships between these modules, they might look like this:

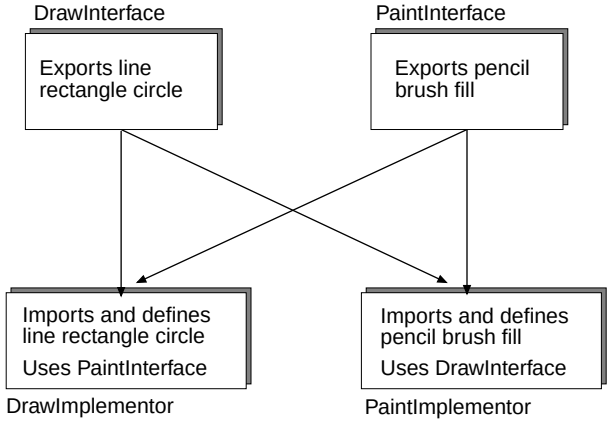


Figure 9-8: Circular module references

Here, the `DrawImplementor` module provides definitions for the `DrawInterface` module. It is also a client of the `PaintInterface` module, whose definitions are, in turn, provided in `PaintImplementor`. Finally, `PaintImplementor` is a client of `DrawInterface`, which completes the circularity.

Combining Module Interfaces

Another advantage of using the interface/implementation model to organize modules is that it allows you to create several interface and implementation modules, and then combine those interfaces in different ways. This is particularly useful for large projects where portions of the project may be produced by different programmers—each portion of the program would have its own interface which could then be combined into a single interface that provides access to the larger program as a whole.

For example, say you had two ScriptX programs, one for drawing functions (vector graphics) and one for painting (bitmap graphics). Using the interface-implementation module for organizing modules, you would create four modules: two interfaces and two implementors:

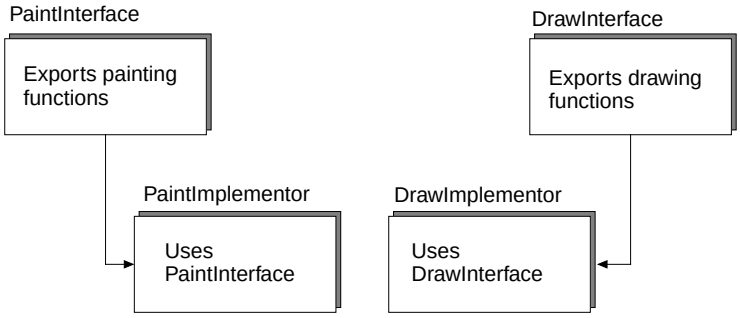


Figure 9-9: Paint and draw interfaces and implementations

Once the interfaces and implementations are defined, you can create a third interface (such as `GraphicsInterface`) that does nothing except use the other two interface modules, and re-export the variables it imported from those modules. Clients of `GraphicsInterface` have access to all the variables in both the `PaintInterface` and `DrawInterface` modules

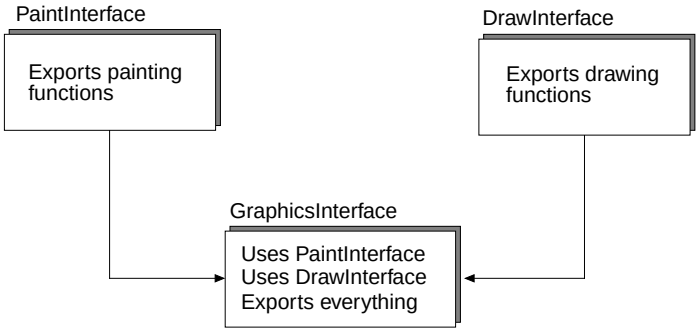


Figure 9-10: Paint and draw interfaces and implementations

Multiple Interfaces, One Implementation

Although the discussion up to this point has focussed on interfaces and implementations in pairs (one interface for each implementation), the most important part of the interface/implementation model is the separation of interfaces from implementations, not their organization. This section shows

examples of using multiple interface modules for the same implementation module, effectively providing many different "views" of a ScriptX program based on how much information you want to reveal to the clients of those interfaces.

Suppose you have created a large body of ScriptX code for producing various musical sounds. For that single large `MusicModule` implementation, you could have multiple interfaces, each of which provides access to only some of the variables within that module. For example, a `ClassicalMusic` interface module would provide access to `violin`, `flute`, and `harpsichord`; a `RockNRoll` module would include `electricGuitar` and `drumKit`; and a `Madrigals` module would include access to voices such as `altoVoice`, and `bassVoice`.

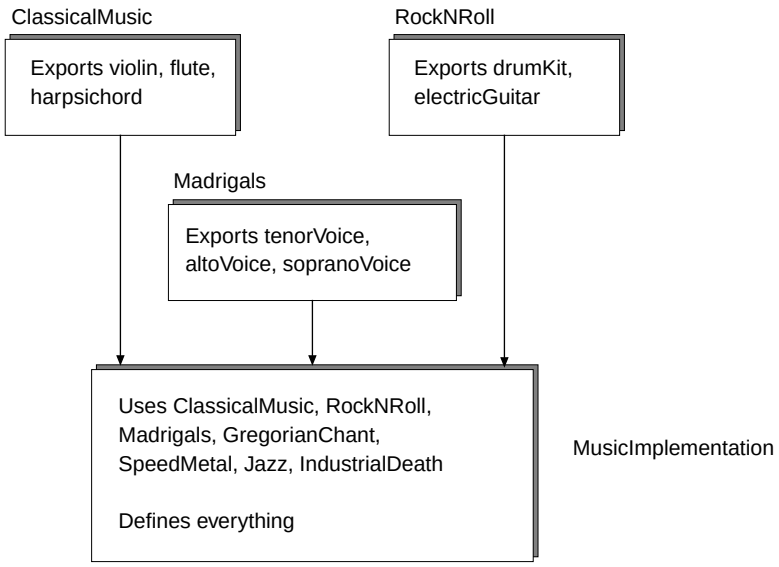


Figure 9-11: Multiple interfaces, one implementation

This same scheme could be applied to a complex ScriptX application where you want to limit some of the features of the application based on whether your user should have access to module `NewUser`, `IntermediateUser`, or `AdvancedUser`. By providing separate interfaces for each of those users, you could provide a subset of features to the new user, more features to the intermediate user, and a complete set to the advanced user.

The pass-through model is a variation on the standard interface/implementation model, useful for defining several discrete protocols and combining them into a single interface.

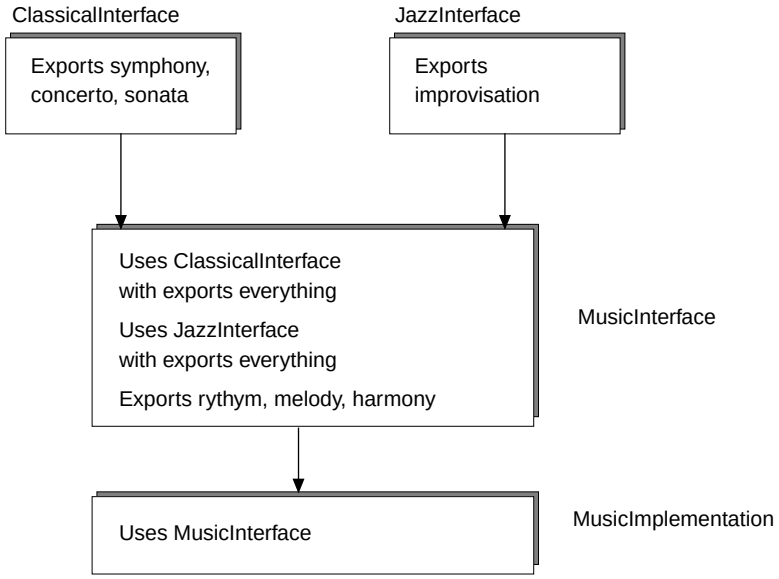


Figure 9-12: Pass-through interface/implementation model

Using a Build Module

A build module is a technique for creating a title without necessarily loading, or even saving, the code that is used in running the title. A complicated title, one with many implementation modules, and perhaps several interface modules, is a good candidate for using a build module.

The build module contains the code that builds the title. Often, it is a special module that corresponds with the build file for a project. The build file is compiled in the build module. By avoiding references from other modules to any variable that is defined in the build module, you can keep the code that is used to build the project from being loaded at runtime. A build module is often a transient object—an object that is not saved to the object store. The build module is the best place for variables that represent compilation status, since those variables should mean nothing at runtime.

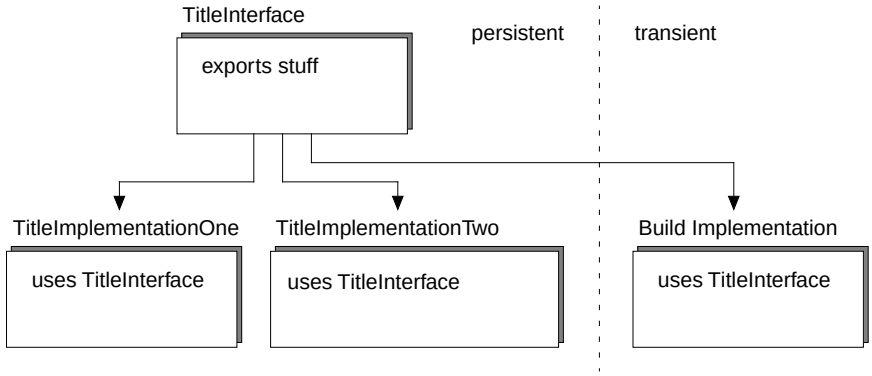


Figure 9-13: Using a build module to compile a project

Defining an Accessory Protocol

One possible application of the interface/implementation model is to create a protocol through which accessory programs can "bind" dynamically with a title at runtime. A ScriptX title does not have to be self-contained. Dynamic binding allows for the creation of titles that are shipped to end users with "hooks" for adding new scenes and characters.

A ScriptX title can be a constructive experience, one where users add new objects to create what is, in effect, their own title. In a sense, publishing a title that is a constructive experience means publishing a protocol.

Of course, two programs cannot interact without knowing something about each other. A title's accessory protocol is, in effect, a list of names that are recognizable to both the title and to accessories. These names represent classes, objects, functions, and generics that are public in any module that uses the accessory protocol. Although it is not necessary to determine all possible interactions between objects in advance, a set of names must be agreed upon.

```
module GidgetAccessoryInterface
    exports instance variables accessoryProtocols
    exports addToPresenter, GidgetClass
end
```

The `GidgetTitleImplementation` module, in turn, uses both the `ScriptX` and `GidgetAccessoryInterface` modules. Within this module, the `accessoryProtocols` instance variable, the generic `addToPresenter`, and the class `GidgetClass` are defined. Since they are defined there, they are also saved there.

```
module GidgetTitleImplementation
    uses ScriptX, GidgetAccessoryInterface
end
```

An accessory that uses the `GidgetAccessoryInterface` should define its own module.

```
module GidgetAccessoryImplementation
    uses ScriptX, GidgetAccessoryInterface
end
```

By now, it should be obvious that the accessory interface model is a familiar one. Think of the `ScriptX` module as the "accessory interface" for the ScriptX core classes. Every module that uses the `ScriptX` module is running as an accessory to ScriptX, which is implemented in the `Substrate` module. Just as the substrate makes a portion of the names it defines visible in the `ScriptX` module, any user-defined module can export some of the names it defines through its own interface module.

Storing Modules

One big advantage to structuring your ScriptX program using modules is that modules allow your program to be saved more easily into the ScriptX object store. Without modules, you must individually store each class and object that your program uses, and you must also reconstruct any global variables that the program requires. By encapsulating your program in a module or in multiple modules, and by saving only the module, all global variables and their values (classes, objects, and functions) that are defined within that module are automatically saved. When you save the module, every name that is defined in that module is saved as well. In effect, modules are a container for compiled code objects (classes, objects, and functions).

Storing Module Objects

You can save a module to a storage container such as a title, library, or accessory container. To store a module, you must first have access to the module object. Since module names are not lexical names, you can use the `getModule` function, described on page 205, to get access to the `ModuleClass` object itself, supplying the module's name as an interned `NameClass` object. Here, the `ModuleClass` object that represents module `MyModule` is assigned to the global variable `theModule`:

```
global theModule := getModule @mymodule
```

Once you have access to a module object, you can add it to a storage container just as you would any other object. This example creates a title container and adds a module to that container:

```
-- create a title container. theStartDir is the default for dir:
global tc := new TitleContainer path:"test.sx" dir:theStartDir
append tc (getModule @mymodule)
```

One weakness to this approach is that it builds the title container that is to store module `MyModule` outside of that module. What if the title container's startup action needs to refer to classes or objects that are defined in the module itself? The following code example demonstrates how to create a module and create the container that stores it within the same module. This example also saves that module to the title container without saving any variables that refer to the creation of the container.

First, create a module, `TopModule`. In this module, you create any classes and global variables that are to be stored with the module in your title or library container.

```
module TopModule
    uses ScriptX
end
in module TopModule
global topVar1 := "the first topVar"
class TopModuleTitle (TitleContainer)
    instance variables
        accessoryProtocols:#("string1", "string2", "stringEtc")
end
```

Suppose you want to avoid having the container itself store variables that refer to the creation of the container—that's wasted space. To make the code that builds a title or library container non-persistent, enclose it in parentheses and make all the "throwaway" variables local.

```
(
    local tc := new TopModuleTitle \
        dir:theScriptDir \
        path:"modacct.sxt" \
        name:"top"
    append tc (getModule @TopModule)
    tc.startUpAction := ( tc ->
        print "here we go"
        load tc[1]
        print topVar1
    )
    close tc
)
```

For more information on adding objects to title containers, library containers, and accessory containers, see the "Title Management" chapter of the *ScriptX Components Guide*.

Retrieving Objects from the Object Store

When your program runs from a ScriptX library container, the first thing it should do, presumably in its start-up script, is retrieve any saved modules from the object store. Recall that modules really have two functions in ScriptX. They exist at compile time to provide distinct compilation units, independent name spaces. But at runtime, modules serve as containers in which compiled class and function objects can be retrieved from the object store.

Of course, lexical names do not exist at runtime. A ScriptX program is compiled into bytecode. But the objects that those names represent during compilation must still be loaded into memory. Modules provide a convenient mechanism for saving and retrieving code.

Typically, a ScriptX title starts up by loading all modules in which its classes and functions are defined. During compilation, the order in which modules are defined and the order in which source files are compiled generally matters. It is quite possible for a class to be defined in one module, while its methods are defined in another module that uses the first module.

At runtime, the order in which modules are loaded generally does not matter. It is usually easiest to load all class and function objects at start-up. Modules are not useful as a program segmentation technique, since classes and functions, once loaded, cannot be unloaded.

Saving Interfaces and Implementations

When you add a module to any ScriptX library container, the ScriptX object store adds all the variables that it "owns." In addition, it also adds all the other modules it uses and any modules that also use it. By storing a module, you are effectively also storing everything that touches or is touched by that module.

In some cases, saving a module may cause more information to be stored than you want. Consider the following network of modules:

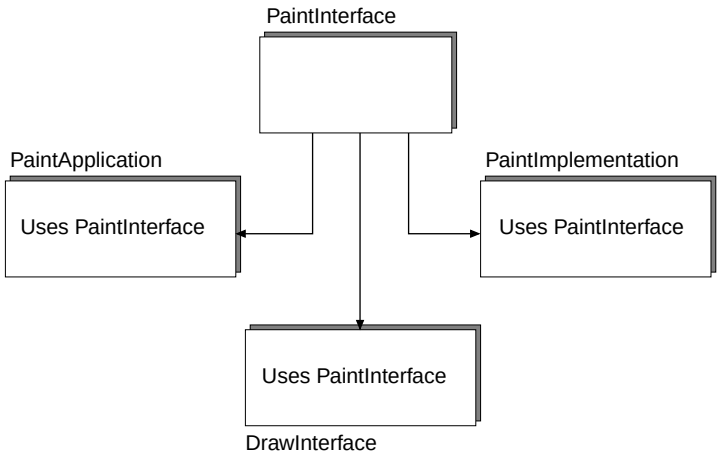


Figure 9-14: A network of modules to be saved

This example shows interface and implementation modules for painting functions, `PaintInterface` and `PaintImplementation`. In addition, it includes two clients of the `PaintInterface` module that use its variables: `PaintApplication` and `DrawInterface`.

Assume, for this example, that you wanted to store only the painting functions (`PaintInterface` and `PaintImplementation`) into a painting library. If you stored `PaintInterface` first, the ScriptX object store would include all the other modules that use `PaintInterface`—all the modules in the network.

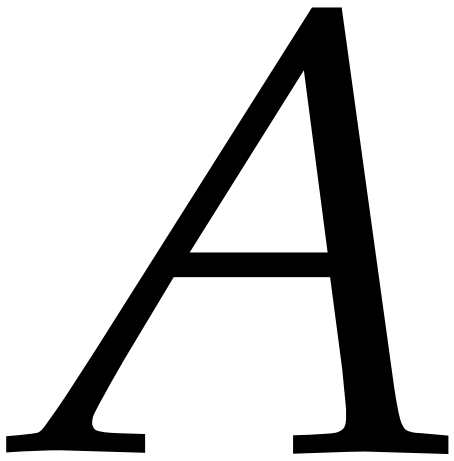
Storing `PaintImplementation` first, rather than `PaintInterface`, prevents this sort of runaway storage of modules. Because implementation modules, by definition, are never used by any other modules, only the modules that the implementation module itself uses are stored.

Here are some hints for storing only the modules you need:

- Add modules that use other modules to storage containers before adding modules that are used by other modules.
- Add only the implementation part of an implementation/interface pair.

APPENDIX

ScriptX Reference



This appendix provides a formal specification of the ScriptX language. It is intended as an entry point and reference for advanced users. It presents an annotated EBNF (Extended Backus-Naur Form) grammar of ScriptX. The complete EBNF grammar is listed separately at the end of the appendix.

The ScriptX grammar is organized into the following sections:

ScriptX Language Syntax

In an Extended Backus-Naur Form grammar, each unit of syntax is defined using tokens and other constructs and that are part of the language. For brevity and readability, this grammar does not depict all rules that govern the use of punctuation and white space with operators and other tokens. Some information about precedence of expressions and operators is also unspecified.

Reserved words and tokens, such as operators, that appear directly as shown in the input stream, are set in Courier boldface in lines of syntax, in lightface in body type. Non-terminals, units of syntax that are defined elsewhere in the grammar, are set in Palatino italics. Descriptive comments are used occasionally to indicate non-printing characters. Since ScriptX is not case sensitive, uppercase characters have been omitted.

`Courier boldface`	reserved words, operators, printing characters
`Courier lightface`	reserved words, operators, printing characters in body type
Palatino italics	non-terminals, units of syntax

Table A-1 lists the notations that are used in this grammar.

Table A-1: Notations used in the ScriptX EBNF grammar

Notation	Meaning
`::=`	is defined as
|	or
[*expr*]	*expr* is optional
[*expr*]*	repeat *expr* zero or more times
[*expr*]+	repeat *expr* one or more times

Tokens and Literals

Tokens are the basic building blocks that make up expressions and other constructs in a programming language. The ScriptX bytecode compiler interprets a stream of input characters, which it can pass through only once. As it reads these characters, the compiler interprets them, one at a time. A token is a sequence of one or more input characters that the compiler recognizes and understands as having some meaning. The ScriptX language has the following kinds of tokens: symbols, operators, reserved words, punctuation marks, and literals.

Literals are sequences of characters that are a literal representation of instances of one of a special set of classes. The value of a literal is the object it represents. Literals are tokens in the input stream that the bytecode compiler recognizes as objects. ScriptX provides string literals, name literals, numeric constants, and several kinds of collections and ranges as literal objects.

Symbols

A symbol must begin with an alpha character, and it can be up to 256 characters in length. The underscore character, which can be used interchangeably with alpha characters, is the only non-alphanumeric character that is permitted in a symbol. Symbols cannot contain blanks or other separators. Although case is remembered when a symbol is first encountered, and is saved for subsequent printing, ScriptX is not case sensitive.

symbol `::=` *initialChar* [*trailingChar*]*
initialChar `::=` *alphaChar* | *underscore*
trailingChar `::=` *alphaChar* | *underscore* | *decimalDigit*
alphaChar `::=` **`a`** | **`b`** | **`c`** | . . . | **`x`** | **`y`** | **`z`**
decimalDigit `::=` **`0`** | **`1`** | **`2`** | **`3`** | **`4`** | **`5`** | **`6`** | **`7`** | **`8`** | **`9`**
underscore `::=` _

A symbol is a lexical name, a name that is understood by the compiler. The compiler associates a symbol at compile time with the thing in a program that it names, such as a variable, constant, or function.

Operators

All ScriptX operations are performed on operands that are objects, and the result of any operation is also an object. Many operators actually serve as a shorthand for functions or generics that are defined by the ScriptX object system. For example, the `RootObject` class provides each object with a default version of the methods `isComparable`, `localLT`, `localEqual`, and `eq`. These four generic functions are visible to the scripter. A class can override these generics to provide its own version of the Comparison protocol. In this way, the number and string classes have different definitions of equality.

Table A-2 indicates the precedence and associativity of operators in the ScriptX language. In this table, precedence is ordered from highest to lowest.

Table A-2: Precedence and associativity of ScriptX operators

Operator or Token	Associativity
element access in collections (`[ ]`)	left to right
class and instance variable access (`. , '`)	left to right
coercion (`as`)	left to right
negation (`-`)	left to right
multiplication and division (`*, /`)	left to right
addition and subtraction (`+, -`)	left to right
`=, >, <, !=, ==, !==, >=, <=, <>, contains`	left to right
`not`	left to right
`and`	left to right
`or`	left to right
`select`	left to right
`where, before, after, from, to, through`	left to right
`in, by, using`	right to left
pipe operator (`|`)	left to right
`repeat, guard, exit, for, return`	left to right
thread operator (`&`)	left to right
assignment operator (`:=`)	right to left
ScriptX expression	left to right

Most ScriptX operators can be used with or without white space. Careful use of white space, which is defined on page 237, can make a program more readable. Certain operators require a separator to prevent ambiguity. Operators that are reserved words, such as `contains` or `as`, must be set off by a separator, normally a blank. The subtraction operator requires a separator, normally a blank, so that the compiler can distinguish it from the negation operator.

Reserved Words

Reserved words are words that cannot, with one exception, be used as names of things because they have special syntactic meanings in ScriptX. (Reserved words can be used as keywords, followed by a colon, in keyword argument

definitions, which are defined on page 252 of this grammar.) Table A-2 contains a list of reserved words, including several that are not currently in use in the ScriptX grammar, but are reserved for future use. Throughout this appendix, reserved words are indicated in the Courier typeface.

Table A-3: ScriptX reserved words

#key	first	my
#rest	fn	named
actual	for	nextMethod
after	from	not
and	function	object
any	get	of
anything	global	on
as	guard	or
before	if	otherwise
by	imports	prefix
case	in	readonly
catching	inclusive	reference
class	index	renames
class method	initializer	repeat
class methods	inst	return
class variables	inst methods	select
class vars	inst variables	set
collect	inst vars	settings
constant	instance	then
contains	instance methods	through
contents	instance variables	throw
continue	instance vars	throw again
continuous	into	times
do	it	to
else	its	transient
end	kind	unglobal
every	last	until
everything	local	uses
excludes	macro	using
exclusive	method	where
exit	middle	while
exports	module	with

Punctuation Marks and Meta-Characters

Separators and terminators are often classified together as a single kind of token, as punctuation. Delimiters and lead-in characters are meta-characters that the compiler uses to identify other tokens. Punctuation marks and

meta-characters have no meaning in and of themselves, except that they set off or identify other tokens. A separator indicates the end of one token and the beginning of the next. A terminator indicates the end of a complete grammatical construct, such as an expression or a clause within an expression. A delimiter or a lead-in character indicates that some group of characters it is associated with in the input stream represents a particular kind of token.

In this grammar, those punctuation marks that have a visual printed representation, such as the comma or colon, are depicted in literal form, in the same typeface and style as reserved words. Nonprinting punctuation marks, such as the end-of-line character, are indicated in certain cases.

White space is a concept that goes hand-in-hand with punctuation. Any input characters that the compiler ignores are called white space. ScriptX has two kinds of white space. First, there are comments. ScriptX identifies a sequence of characters that begins with two hyphens (`--`) or two slashes (`//`) and ends with a new line or carriage return as a comment. A comment can be inserted in the middle of an expression; interpretation of the expression will resume on the next line. ScriptX also accepts C-style inset comments, using the same delimiters. ScriptX, in contrast with ANSI C, allows inset comments to be nested.

A second kind of white space is blank space. Blank space consists of optional space, tab, and end-of-line characters that can be inserted between tokens to make code more readable. Certain punctuation marks can serve as white space, while others cannot. If a punctuation mark can serve as white space, then two or more are permitted wherever one is permitted. Table A-4 lists punctuation marks in the ScriptX language, identifying whether or not they can be used as white space.

Table A-4: Punctuation marks in ScriptX

Punctuation	White Space	Purpose
blank	yes	separator
comma (`,`)	no	separator in some lists
colon (`:`)	no	separator in some paired elements
newline	yes	separator in incomplete expressions terminator in complete expressions
carriage return	yes	separator in incomplete expressions terminator in complete expressions
semicolon (`;`)	no	terminator (newline)
stop (`!!`)	no	terminator that halts evaluation of expressions in the input stream
quotes (`" "`)	no	delimiter for string literals
@-sign	no	lead-in character for name literals
hash sign (`#`)	no	lead-in character for array and keyed list literals
function (`->`)	no	lead-in character for function body

Table A-4: Punctuation marks in ScriptX

Punctuation	White Space	Purpose
backslash (\ )	no	separator that also turns the next new line or carriage return in the input stream into white space lead-in character for escape characters
two hyphens (--)	yes	lead-in character for a comment
two slashes (//)	yes	lead-in character for a comment
comment (/* */)	yes	delimiters for ANSI C style inset comments, which can be nested
parentheses	no	delimiters for certain kinds of lists, anonymous functions, compound expressions; often required as a separator to insure that an expression is parsed
brackets	no	delimiters for access to members of collections
braces	no	delimiters for restrictions
angle brackets < >	no	delimiters for hexadecimal constants used to represent unicode characters (also used as comparison operators)

In most cases, this grammar does not explicitly indicate where end-of-line and white space characters are allowed, except where an end-of-line character is required as a separator. An incomplete sentence can be broken with a newline character; evaluation of the expression resumes on the following line. ScriptX programmers commonly break lines after a binary operator, after a separator that cannot act as white space, or anywhere where the compiler is expecting a closing delimiter such as a parenthesis.

Parentheses serve as separators as well as delimiters in expressions that could otherwise not be parsed. The following examples demonstrate how parentheses can be used to turn an expression into a factor. Factors, indivisible syntactic units, are discussed in the section “Types of Expression” on page 240.

```
function doIt a b -> (
    print a; print b
)

doIit -1 -2
⇨  -1
   -2

doIt -(1) -(2)
⇨  no sub instance method

doIt negate(1)  negate(2)
⇨  too many arguments 4 supplied 2 allowed

doIt (negate(1))  (negate(2))
⇨  -1
   -2
```

As these examples demonstrate, ScriptX allows lists of arguments with minimal punctuation. If arguments are not separated by commas, then the arguments themselves must be factors. By enclosing an expression that is used as an argument within parentheses, it becomes a factor, a complete and indivisible unit of punctuation.

ScriptX punctuation allows for a variety of programming styles; its flexibility accounts for the fluidity of ScriptX code. Punctuation usually causes few difficulties, even for beginning scripters. For more information on punctuation, see the discussion that begins on page 29 of this volume.

Literals

Name literals represent instances of the class `NameClass`. Any valid name that is preceded by an at sign (`@`) is interned. Name literals are full-fledged `NameClass` objects that have the same value at compile time and run time. They are used as labels in programs, often to represent a state or outcome. Two name literals that have the same value are the same object. That is, the names `@insideOut`, `@insideout`, and `@INSIDEOUT` are not merely equal, they are actually the same object.

nameLiteral ::= **@** [*trailingChar*]+
trailingChar ::= *alphaChar* | *underscore* | *decimalDigit*

A string literal is a sequence of Unicode characters, enclosed in double quotes in the input stream. A string literal can extend over multiple lines, can be any length, and can include any valid Unicode character, including newline. To use certain nonprinting characters in a string, escape characters are required.

stringLiteral ::= **"** [*unicodeChar*]* **"**
unicodeChar ::= `-- any printing char`
| *escapeChar*
| **\<** *hexConst* **>**
escapeChar ::= **\n** | **\r** | **\t**
hexDigit ::= **0** | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9** | **a** | **b** | **c** | **d** | **e** | **f**
hexConst ::= [*hexDigit*]+

The compiler automatically stores a string literal as a `StringConstant` object. To modify or edit a string literal, it must first be converted to the `String` or `Text` class. Although name literals and symbols are not case sensitive in ScriptX, strings are. For more information on strings, see the “Text and Fonts” chapter of *ScriptX Components Guide*.

A numeric constant is automatically stored as an instance of one of the subclasses of `Number`: `ImmediateInteger`, `LargeInteger`, `ImmediateFloat`, or `Float`, depending on range and precision requirements. An integer constant is stored as an `ImmediateInteger` object, except when its value extends beyond the 29-bit storage range of the class. ScriptX has no unsigned data types. A floating point constant is converted to either an `ImmediateFloat` or a `Float` object depending on both range and precision requirements. For more information, see the “Numerics” chapter of *ScriptX Components Guide*.

```
numericConst    ::=   mantissa [ exponent ]
                  |   hexLiteral
mantissa        ::=   integerConst [ .decimalDigit ] [ decimalDigit ]*
exponent        ::=   e integerConst
integerConst    ::=   [ negOperator ] [ decimalDigit ]+
negOperator     ::=   -
decimalDigit    ::=   0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
hexLiteral      ::=   0x [ hexDigit ]+
```

The ScriptX language provides several other literal constructions. A literal construction is an expression that creates a new instance of a class directly. Normally, you create a new class either by calling the `new` method or by using the `object` expression. ScriptX creates instances of `Array`, `KeyedLinkedList`, `ContinuousNumberRange`, and `NumberRange` from literal expressions.

```
arrayLiteral    ::=   #( exprList )
                  |   #( keyedList )
                  |   #( )
                  |   #( : )
exprList        ::=   simpleExpr [ , exprList ]*
keyedList       ::=   factor : simpleExpr [ , keyedList ]*

rangeLiteral    ::=   factor to factor [ by factor ]
                  |   factor by factor [ by factor ]
                  |   factor to factor [ inclOption ] continuous
                  |   factor inclOption to factor [ inclOption ] continuous
inclOption      ::=   inclusive | exclusive
```

These collections and ranges can also be instantiated by normal means, by calling `new` on the appropriate class or by using an object definition expression.

Types of Expression

A ScriptX program is a sequence of ScriptX expressions. Every complete construct in the ScriptX language is an expression. Every expression yields a value that is an object, and every object is an instance of a class, either a class that is defined by the environment, or one that is created by the user.

A sentence is the basic unit of syntax that is evaluated incrementally by the ScriptX bytecode compiler, generally a single top-level expression followed by an end-of-line character.

```
sentence        ::=   [ endOfLine ]* sentence
                  |   topLevelExpr endOfStream
                  |   topLevelExpr endOfLine
                  |   endOfStream
```

An expression that can be entered and evaluated at the top level, outside of any blocks, is a top-level expression. An expression that can be entered and evaluated inside a compound expression, that is, within parentheses, is an inner-level expression. A few operations are permitted only at one level. For

example, modules can only be defined at the top level, and locals can only be declared as inner-level expressions. Most ScriptX expressions are allowed at both levels.

```
topLevelExpr    ::=  simpleExpr
                 |   globalExpr
                 |   guardExpr
                 |   assignmentExpr
                 |   repeatExpr
                 |   moduleExpr
                 |   inModuleExpr
innerLevelExpr  ::=  expr
                 |   localExpr
```

Expressions are a subcategory of inner-level expressions, a general category to which most other ScriptX expressions belong.

```
expr            ::=  simpleExpr
                 |   guardExpr
                 |   assignmentExpr
                 |   repeatExpr
                 |   blockControlExpr
```

In ScriptX, the target of a control structure is generally an expression. Examples are the `if . . . then` and `guard` constructs.

```
ifExpr          ::=  if simpleExpr then expr else expr
                 |   if simpleExpr do expr
guardExpr       ::=  guard expr [ catching catchList ] [ on exit expr ] end
```

A simple expression is a subcategory of both expression and top-level expression. An expression that forms a clause within another expression must be a simple expression.

```
simpleExpr      ::=  factor
                 |   ifExpr
                 |   caseExpr
                 |   forExpr
                 |   classDefExpr
                 |   objectDefExpr
                 |   functionDefExpr
                 |   callExpr
                 |   coercionExpr
                 |   pipeExpr
                 |   threadExpr
                 |   arithmeticExpr
                 |   pathExpr
                 |   indexExpr
                 |   rangeLiteral
```

A factor is a subcategory of simple expression. Factors are expressions, such as literals, that are indivisible units of syntax, units that usually cannot be factored into smaller units. A factor is the smallest possible piece of an expression that is itself an expression.

Any expression that is enclosed in parentheses, including a compound expression or an anonymous function, is a special kind of factor. In the surrounding scope, an expression enclosed in parentheses is recognized as a factor. This allows any inner-level expression to be used where a factor is required. Because of the precedence of factors, parentheses provide classical control over evaluation order.

ScriptX includes two special constructs that behave as factors. The `nextMethod` construct, which must appear within the body of a method, denotes the method you are in. Its prime use is for calling superclass methods. The question mark (`?`) is a special construct that always stands for the most recent value returned at the top level by the scripter. As a consequence, it makes sense only in scripts that are evaluated in a listener window.

factor `::=` *location*
| *numbericConstant*
| *stringLiteral*
| *nameLiteral*
| **`?`**
| **`nextMethod`**
| *anonFuncDefExpr*
| *compoundExpr*
| *arrayLiteral*

Figure A-1 demonstrates how top-level and inner-level expressions can be broken out into one of the five grammatical categories defined here: top-level expressions, inner-level expressions, expressions, simple expressions, and factors. These categories are used throughout the grammar to establish rules of syntax.

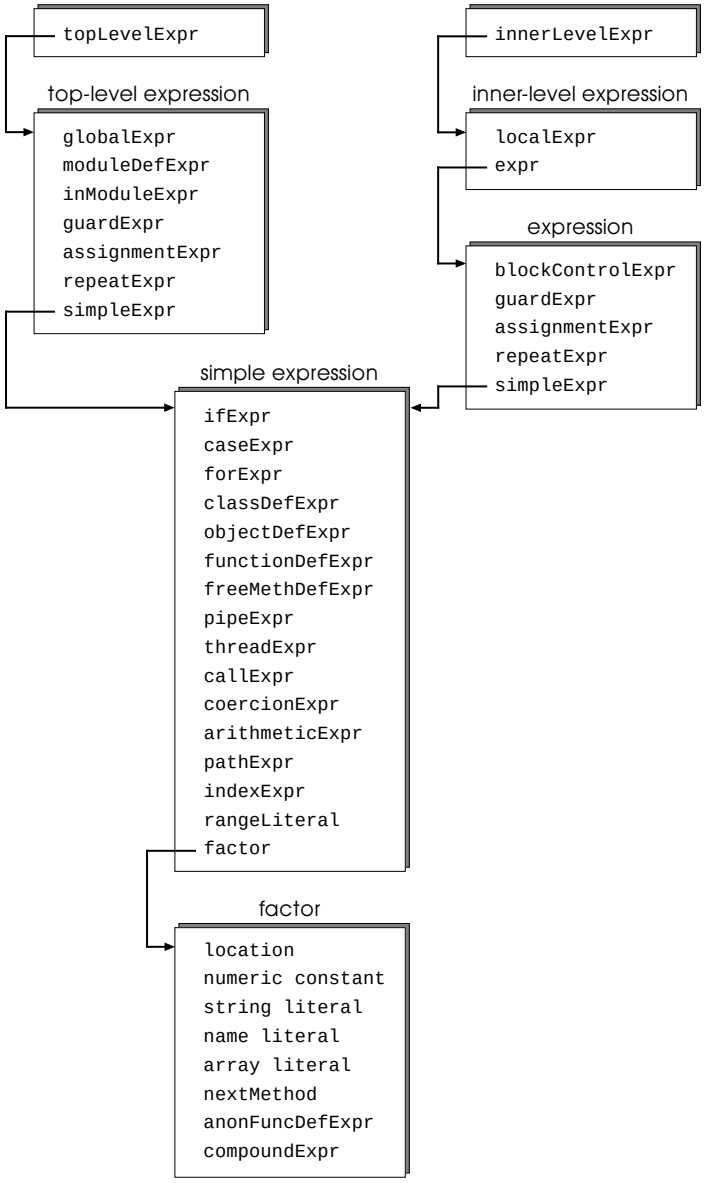


Figure A-1: Types of ScriptX expressions

Expression Syntax

This section defines some of the basic expressions that are commonly used in ScriptX programs.

A compound expression is a list of one or more inner-level expressions, enclosed in parentheses. In this respect, parentheses in ScriptX are analogous to braces in C and C++. Since end-of-line tokens can be used as white space, programmers can use parentheses in a variety of styles, just as they do in C. A compound expression is a factor.

```
compoundExpr      ::=   ( compoundExprSeq )
compoundExprSeq   ::=   innerLevelExpr moreExprs
moreExprs         ::=   endOfLine compoundExprSeq
                  |     endOfLine [ moreExprs ]*
                  |     empty
```

A call expression is a simple expression that invokes a function or method, supplying a list of arguments that matches the list of parameters in that function or method's calling sequence. (Elsewhere in this grammar, function parameters are sometimes referred to as "argument definitions.") The first token in a call expression must be a factor. This factor must yield a function object, such as a generic function, and is usually the name of a variable holding the function.

The balance of a call expression, if present, supplies a list of arguments. ScriptX supports several different formats for this list, with a choice of commas or spaces as separators.

```
callExpr          ::=   factor paramSequence
                  |     factor ( simpleExpr , paramList )
                  |     factor ( symbol : factor [ , paramList ] )
                  |     factor ( )
paramSequence     ::=   [ symbol : ] factor [ paramSequence ]
paramList         ::=   [ symbol : ] simpleExpr [ , paramList ]
```

Only the first two forms of call expression apply to method calls. The first parameter in a method call is always a positional argument; it supplies the name of the object that the method is being called on. A function that is called with no arguments requires a set of empty parentheses, depicted in the fourth form, to identify the expression as a call expression.

Note – In the call expression, and in other ScriptX expressions where a script supplies a list of elements to the compiler, ScriptX version 1.0 has a built-in limit of 200 items in a list. This restriction is defined syntactically, and applies to explicit function arguments, global and local declarations, lists of class and instance variables, and lists of expressions in a compound expression sequence.

If a call expression includes both keyword and positional arguments, positional arguments must be supplied first, and in their proper order, followed by rest and keyword arguments. Supplying improper arguments or supplying arguments in the wrong order generates an exception.

The block control expression is a general form that encompasses several types of expression. A block control expression can only be used within a control structure, such as a function, method, or loop. Within a loop, the `continue` expression is used to immediately resume at the next iteration of the loop. The `exit` expression is similar to `continue`, except that evaluation resumes outside the loop, with the next expression in the script. The optional `with` clause sets the value of the construct upon exit. The `return` expression exits from the enclosing function or method, and can set the return value, which is

otherwise `undefined`. The final form of block control expression, `throw again`, is associated with exception handling, and is used only inside guarded code.

```
blockControlExpr  ::=  continue
                   |   exit [ with simpleExpr ]
                   |   return [ simpleExpr ]
                   |   throw again
```

An arithmetic expression performs an arithmetic or logical operation on one or more operands. The coercion expression has a similar form. The negation operator and `not` are unary operators—they operate on a single operand. The compiler allows any simple expression as an operand. This expression must evaluate at run-time to an instance of an appropriate class. For example, the addition operation is defined only for instances of subclasses of `Number` and `String`. In some cases, if the second operand is incompatible with the first, it attempts to coerce it. If it is unable to coerce to an appropriate class, it generates an exception.

```
arithmeticExpr  ::=  negOperator simpleExpr
                |    simpleExpr arithOperator simpleExpr
                |    simpleExpr compOperator simpleExpr
                |    simpleExpr contains simpleExpr
                |    not simpleExpr
                |    simpleExpr and simpleExpr
                |    simpleExpr or simpleExpr
coercionExpr    ::=  factor as factor
```

Pipe and thread operators provide shorthand equivalents for common operations on objects. The pipe expression is equivalent to an invocation of the `pipe` generic function, which is implemented on the collection family of classes. The thread operator spawns a new `RegularThread` object, running the given expression asynchronously.

```
pipeExpr    ::=  | simpleExpr
threadExpr  ::=  simpleExpr &
```

Assignment and Variable Access

The ScriptX assignment expression assigns the value of a simple expression to a location. A location is either the name of a global or local variable, a specifier for some object's instance variable, or a specifier for an element in a collection.

```
assignmentExpr  ::=  location := simpleExpr
                |    location := assignmentExpr
                |    location := guardExpr
location        ::=  symbol
                |    ivAccess
                |    elementAccess
```

An assignment expression yields the value that is assigned. This allows assignment expressions to be cascaded or embedded within other expressions.

ScriptX provides a choice in syntax for instance variable access. In the possessive form, a separator is required before the symbol. Instance variable access associates left-to-right, so expressions can be nested to allow for access to instance variables defined by embedded objects.

ivAccess ::= *factor*`.`*symbol*
| *factor*`'s` *symbol*

ScriptX uses brackets to identify elements of a collection in an element access expression. Note that the brackets in the element access expression are not meta-characters of the EBNF grammar.

elementAccess ::= *factor* `[` *simpleExpr* `]`

A binding is an association between a symbol, or lexical name, and a global or local variable. A binding can be declared `local` or `global`. An assignment statement or a class or object definition expression stores a pointer to some object at that location. Objects do not necessarily have to have a binding—they can be embedded within other objects. Objects can also have more than one binding.

binding ::= *variable*
funBindings ::= *functionDefExpr* [`,` *functionDefExpr*]*
| *freeMethDefExpr* [`,` *freeMethDefExpr*]*

A local declaration is allowed only at the inner level, inside the scope for which the binding is local. An explicit global declaration is allowed only at the top level. If the scope of a declaration is not given explicitly, it is global by default. Of course, a global declaration is global only within modules that use its definition. The `unglobal` version of the `global` expression removes a global binding.

localExpr ::= `local` *binding* [`,` *binding*]*
| `local` *funBindings*
globalExpr ::= `global` *binding* [`,` *binding*]*
| `unglobal` *binding* [`,` *binding*]*
| `global` *funBindings*

A variable is a symbol that is associated with some location through a binding. When a variable is declared, it is initialized automatically. The compiler offers several options at initialization. A variable can be declared `constant`, meaning that its value cannot later be changed by assignment. A variable can be associated with a restriction. (Although the restriction syntax is legal in any variable declaration, restrictions are currently implemented only in free method definitions. Otherwise, restrictions are ignored.) A variable can be assigned an initial value. If no value is assigned to a variable when it is declared, its value is set to `undefined`, a special object.

```
variable        ::=   [ constant ] symbol [ restriction ] [ initialVal ]
restriction     ::=   { class factor }
                  |   { kind factor }
                  |   { object factor }
initialVal      ::=   : simpleExpr
                  |   := simpleExpr
```

Flow of Control

An `if` expression evaluates a simple conditional to determine flow of control in a program. In ScriptX, a conditional may yield any object, with the Boolean `false` object meaning false, and any other value meaning true. The `if` expression has three alternate forms, based on whether an `else` clause is supplied. If `then` is supplied as the second keyword, without an accompanying `else` clause, execution is delayed until the parser interprets the next complete expression. To force evaluation in a listener window after a `then` clause, enter two exclamation marks (`!!`).

```
ifExpr          ::=   if simpleExpr then expr else expr
                  |   if simpleExpr do expr
                  |   if simpleExpr then expr
```

The ScriptX `case` expression is analogous to the `switch` statement in C and C++, but the ScriptX `case` expression is more flexible. As with other control structures, a `case` expression first evaluates a simple control expression. Case tags can be any factor. These factors are evaluated in order. When a factor matches the value of the control expression (using the `equals` global function, equivalent to the "=" operator, as a test), its associated expression is executed. If no factor matches, the expression in the `otherwise` clause, if one is provided, is executed. In contrast with the C `switch` statement, no explicit break is required. Only one expression within the body of a `case` expression is executed.

```
caseExpr        ::=   case [ simpleExpr ] of taggedFormList end
taggedFormList  ::=   taggedForm [ moreTaggedForms ]
moreTaggedForms ::=   [ endOfLine ]* taggedFormList
taggedForm      ::=   factor : expr
                  |   otherwise : expr
```

The `repeat` expression is almost identical with the `if` . . . `do` form of the `if` expression, except that the target expression is repeated as a loop until its control expression returns `false`. The first two forms of this expression do not guarantee that the loop will ever be executed, since the control expression might return `false` on its first iteration. The final two forms of this expression guarantee that the loop will be executed at least once, since the control expression is evaluated only after the loop is executed.

```
repeatExpr      ::=   repeat while simpleExpr do expr
                  |   repeat until simpleExpr do expr
                  |   repeat expr while simpleExpr
                  |   repeat expr until simpleExpr
```

The ScriptX `for` expression has three parts. The first part gives the source or sources of iteration. The second part, which is optional, provides a conditional that is evaluated once before each iteration of the loop. The third part is the body of the loop.

forExpr ::= **for** *forSources* [*forTest*] *forBody*

A `for` expression can supply more than one source of iteration. The program iterates through these sources in parallel, stopping when the first of these iterations ends.

forSources ::= *forSource* [**,** *forSource*]*
forSource ::= *simpleExpr*
| *symbol* **:=** *simpleExpr*
| *symbol* **in** *simpleExpr*

There are two forms for the source of iteration in a `for` expression. The simplest takes a simple expression yielding a number. It uses this number as a count for the number of times the loop should iterate. The second associates a series of values with a local variable, accessible in the loop body. This form can be used with the discrete range literal, or with a simple expression that yields a collection.

Syntax for the optional test clause in a `for` expression is the same as for a `repeat` expression.

forTest ::= **while** *simpleExpr*
| **until** *simpleExpr*

The body of a `for` expression is a clause that is repeated on each iteration of the loop. Three variants are available for the body of a `for` expression. In the first variant, the body of the loop is simply an expression, often a compound expression, that is evaluated once for each iteration of the loop. The second and third variants result in a collection of values that is built during the iterations. The third variant selects values for this collection based on the result of a test expression.

forBody ::= **do** *expr*
| **collect** [**into** *simpleExpr*] [**by** *factor*] \
[**as** *factor*] *simpleExpr*
| **select** *factor* [**into** *simpleExpr*] \
[**by** *factor*] [**as** *factor*] **if** *simpleExpr*

The second and third variants for the body of a `for` expression allow for any of three optional clauses, which must specified in the order shown. The `into` clause allows a script to specify an existing collection as the target of `collect` or `select`. The `by` clause allows a script to specify the collection function that will be used to add values to the collection (`appendReturningSelf`, a variant of `append` that returns a collection, is the default). The `as` clause specifies which class of collection should be constructed.

Definition of Classes

The class definition expression creates a new class, which inherits the properties and behavior of a list of superclasses. Each item in the list of superclasses must be a simple expression that returns a class. The class name, which is required, must be a symbol. If this symbol is specified in the object definition expression, it is declared as a constant, and can no longer be assigned to another object.

classDefExpr ::= **class** *symbol* **(** *supersList* **)** *classBody* **end**
supersList ::= *simpleExpr* [**,** *supersList*]*

The body of a class definition expression has four clauses. These clauses are used to define class variables, class methods, instance variables, and instance methods. All four clauses are optional, but those that are present must be supplied in the specified order.

classBody ::= [*classVars*] [*instanceVars*] [*classMethods*] [*instMethods*]

Class and instance variables are defined in separate clauses within the body of the class definition expression. Class and instance variables are implemented through accessor methods, which get or set a value. Virtual instance variables, which do not correspond with any physical slot, are implemented as methods. Real instance variables, although they are associated with an actual slot in the memory, are also queried through getter and setter methods. You use the class and object definition expressions to define real variables.

classVars ::= **class var**[**iable**]**s** *varList*
instanceVars ::= **inst**[**ance**] **var**[**iable**]**s** *varList*
varList ::= [*qualifiers*] *variable moreVars*
moreVars ::= [*endOfLine*]+ *varList*
| [**,** *varList*]*

When a real class or instance variable is defined, you can specify whether that variable is `readonly`, `transient`, or `reference`. These options, if specified, must be given in order. A variable that is read-only can be retrieved from the object store, but not changed. A variable that is transient cannot be stored in the object store, so its value is discarded each time the object is saved and retrieved. If a variable is transient, you can specify an optional initialization function to restore its value each time the class or object is retrieved. A reference variable is a real variable that is not loaded from the object store until it is actually needed by the system.

qualifiers ::= [**readonly**] [*transientOption*] [**reference**]
transientOption ::= **transient** [**initializer** *factor*]

Class and instance methods can be defined within the body of a class definition expression. Each method definition must begin on a new line.

classMethods ::= **class methods** *methods*
instMethods ::= **inst**[**ance**] **methods** *methods*
methods ::= *methodDefExpr moreMethods*
moreMethods ::= [*endOfLine*]* *methods*

Note that both class and instance methods can be defined outside the class body, as free methods. Discussion of method definition expressions, and of syntax for free methods, begins on page 251.

Definition of Objects

The object definition expression creates a new object, whose properties and behavior are derived from a given list of classes. This list has the same form and syntax as the list of superclasses in a class definition expression. If the object inherits from one class, it is an instance of that class. A script can also define an object that inherits from more than one class, and is a special instance of all of them.

The object's name, which is optional, must be a symbol. If this symbol is specified in the object definition expression, it is declared as a constant global variable into which the object is stored.

objectDefExpr ::= **object** [*symbol*] **(** [*classesList*] **)** *objectBody* **end**
classesList ::= *simpleExpr* [**,** *classesList*]

The body of an object definition expression has five optional clauses. These clauses are used to set keyword arguments, to define instance variables and methods, to initialize instance variables, and to insert a series of initial values into certain kinds of objects, such as collections. All five clauses are optional, but those that are present must be supplied in the specified order. The instance variables and instance methods clauses are exactly as defined under the class definition expression, on page 249. The others are defined below.

objectBody ::= [*initialization*] [*instVars*] [*instMethods*] \
[*settings*] [*contents*]

Every ScriptX object defines or inherits a version of the `init` method, which takes keyword arguments and generally performs initialization tasks. The initialization section supplies the object's `init` method with a list of keyword-value pairs. Although the initialization section is optional, many classes require some keyword arguments when a new object is created.

initialization ::= *initList* | *empty*
initList ::= *initForm moreInitForms*
initForm ::= *symbol* **:** *simpleExpr*
moreInitForms ::= [*endOfLine*]+ *initList*
| [**,** *initList*]*

The `settings` clause in the object definition expression is in the same form as the initialization section. It allows a script to set initial values for instance variables that are writable, but are not set by the object's `init` method. Any instance variables that are not initialized are set to `undefined`.

settings ::= **settings** *initList*

The `contents` clause in the object definition expression applies only to objects that implement the `addToContents` method. In the core classes, `addToContents` is defined as part of the Collection protocol. Use the `contents` clause to add a series of values to a new `Collection` object. Values are supplied as a comma or return separted list of simple expressions, and are added to the collection by the `addToContents` method in the order in which they are supplied.

contents ::= **contents** *contentsList*
contentsList ::= *simpleExpr moreContentForms*
moreContentForms ::= [*endOfLine*]+ *contentsList*
| [**,** *contentsList*]*

Definition of Functions and Methods

Expressions that define functions, methods, and free methods are closely related in syntax. Each definition has three sections: a symbol, a set of parameter definitions, and a body.

functionDefExpr ::= **f**[**unctio**]**n** *symbol* [*argSeq*] *body*
| **f**[**unctio**]**n** *symbol* **(** [*argList*] **)** *body*
methodDefExpr ::= **method** [*getterOrSetter*] *symbol argSeq body*
freeMethDefExpr ::= [**class**] **method** [*getterOrSetter*] *symbol argSeq body*
body ::= **->** *expr*

The reserved words `get` and `set` are semantically equivalent to naming a method with "getter" and "setter" as suffixes. A method that is defined as a getter or setter acts as a virtual instance variable. For more information on getter and setter methods, see page 142.

getterOrSetter ::= **get** | **set**

A ScriptX function or method can accept three different kinds of arguments. Positional arguments are conventional, unkeyed arguments, and they must be supplied in the correct order when the function or method is called. Rest arguments, which are also unkeyed, provide a mechanism for passing in a variable number of arguments. Key arguments are passed with a keyword identifier, as a keyword-value pair, so they can be supplied in any order.

argSeq ::= [*positionalArgSeq*] [*restArg*] [*keywordArgSeq*]
argList ::= [*positionalArgList*] [*restArg*] [*keywordArgList*]

A function's argument definitions (parameters) are matched with arguments in the call expression's argument sequence. Call expressions are discussed on page 244. Positional arguments are matched first. A function or method accepts a set of positional arguments, separated by commas or spaces.

positionalArgSeq ::= *argument* [*argument*]*
positionalArgList ::= *argument* [**,** *positionalArgList*]*

An argument definition is a symbol. It resembles a local variable definition, naming a variable that is defined within its own body. Like a local variable declaration, an argument definition overrides a name that is defined in a surrounding scope. Restrictions are permitted as part of the argument syntax, but they are not implemented in the current version of ScriptX.

argument ::= *symbol* [*restriction*]

The rest argument clause supplies a symbol, which the call expression uses to pass a variable-length sequence of unkeyed arguments. When the function or method is called, this symbol is bound to an array, and all the arguments from the call expression's argument list are placed in sequence in that array. Rest arguments are useful when a function wants to accept a variable-length sequence of arguments.

restArgs ::= **#rest** *symbol*

The keyword argument sequence begins with the reserved word #key, which serves as a lead-in for the sequence or list that follows. Keyword-argument pairs can be separated by blanks if parentheses are not used, or by commas if parentheses are used.

keywordArgSeq ::= **#key** *keyArgPair* [*keyArgPair*]*
keywordArgList ::= **#key** *keyArgPair* [**,** *keyArgPair*]*

Each keyword-argument definition can have three parts. Only the first part, the name of the keyword itself, is actually required. This argument defintion is the key that a calling function uses to specify which argument it is supplying. A function or method definition can supply an optional symbol, which will be the name of the argument inside the function body. Otherwise, the argument's name inside the function is the name of the keyword itself.

keyArgPair ::= *argument* **:** [*symbol*] [**(** *simpleExpr* **)**]

A function or method definition can also supply an optional expression, enclosed in parentheses, which is used to set a default value if a keyword argument is not supplied by the caller. It is up to the function or method itself to test for required keyword arguments, and report an exception if any are missing, or are supplied incorrectly.

An anonymous function, also called an in-line function definition, is a function definition that is not assigned to a lexical name. Unlike other functions and methods, it is a factor. Its argument definition list is in the same form as the argument definition list for a regular function definition. The body of an anonymous function must be a compound expression sequence. This sequence can be as simple as one inner-level expression.

anonFuncDefExpr ::= **(** *anonArgs* **->** *compoundExprSeq* **)**
anonArgs ::= [*paramSequence*] [*restArg*] [*keywordArgSeq*]
| *empty*

Modules

Expressions that define modules, or that set the current working module, are only allowed at the top level.

```
inModuleExpr      ::=   in module symbol
moduleDefExpr     ::=   module symbol [ moduleOptions ] end
```

Each module option can be repeated any number of times, and in any order, within the module definition expression.

```
moduleOptions     ::=   [ moduleOption ]*
moduleOption      ::=   exports [ readonly ] \
                              [ inst[ance] var[iable]s ] symbolSeq
                  |     uses symbolSeq
                  |     uses symbol [ with usesOptions ] end
symbolSeq         ::=   symbol [ , symbol ]*
```

A module must explicitly list each name that it exports. Names of classes, objects, global variables, global functions, and generic functions, including getter and setter methods, must be explicitly listed in an `exports` clause.

Two syntactic choices are available in a module definition expression for using other modules. A `uses` clause lists a module or modules that are imported as a whole into the module. A `uses . . . with` clause allows for additional restrictions on how names are imported from another module. A `uses . . . with` clause contains a list of options. Each `uses . . . with` option can be repeated any number of times, and in any order.

```
usesOptions       ::=   [ usesOption ]*
usesOption        ::=   imports everything
                  |     imports [ readonly ] \
                              [ inst[ance] var[iable]s ] symbolSeq
                  |     exports everything
                  |     exports [ readonly ] \
                              [ inst[ance] var[iable]s ] symbolSeq
                  |     prefix symbol
                  |     excludes [ readonly ] \
                              [ inst[ance] var[iable]s ] symbolSeq
                  |     renames [ readonly ] \
                              [ inst[ance] var[iable]s ] renameSeq
renameSeq         ::=   symbol : symbol [ , symbol : symbol ]*
```

Each `uses . . . with` option provides functionality for handling names that are imported into modules. Names that are imported from another module can be specified individually, or a module can import every name. Names that are imported can be exported again to other modules that use the current module. Names can also be excluded or renamed. An importing module can prefix all names from a given module that it uses.

Exception Handling

The `guard` expression is used for catching and handling exceptions. It provides an extra layer of exception handling while ScriptX executes the guarded expression, which is often a compound expression. If this expression is a compound expression, other `guard` expressions can be nested dynamically inside it.

The `guard` expression has two optional clauses. The `catching` clause is executed only if an exception occurs. It supplies a list of exception tags, known as catchers, that are paired with expressions. These catchers can be exception classes, or they can be instances of those classes.

guardExpr ::= **guard** *expr* [**catching** *catchList*] [**on exit** *expr*] **end**

The `on exit` clause allows a script to supply an expression that will be executed automatically when the flow of control leaves the `guard` expression. ScriptX is guaranteed to execute the expression in the `on exit` clause, whether or not an exception is reported.

The `catching` clause resembles the `case` expression, but it actually functions quite differently. Each entry in the catch list is called a catcher. When an exception occurs, ScriptX matches the exception with every catcher on the catch list. After each entry in the catch list has been tested, ScriptX executes the expression in the `on exit` clause, if one is present, and throws the exception again.

catchList ::= *catcher moreCatchers*
moreCatchers ::= [*endOfLine*]* *catchList*
| [*endOfLine*]* *moreCatchers*
| *empty*
catcher ::= *symbol* [*symbol*] [: *expr*]

The following constructs apply to the `catching` clause. The reserved word `all`, used as a symbol in the catch list, catches all exceptions. Note that `all` is not analogous with `otherwise` in the `case` expression. Evaluation of the script resumes with the next item in the catch list after the expression associated with `all` has been executed. An `all` catcher can be at the beginning, at the end, or anywhere in the middle of a catch list, and there can even be multiple `all` catchers in a catch list.

The reserved word `caught` acts as a function, and catches the exception. It serves as a break point, causing the flow of control to leave the catch list. `caught` requires an expression, which is evaluated and serves as the return value of the `guard` expression. If the expression is not caught, evaluation continues until the end of the catch list is reached.

The reserved word `throw again`, defined as a block control expression, causes the flow of control to leave the current catch list, but it reports the exception again. If the `guard` expression is called within another `guard` expression, evaluation switches to the `catching` clause in the next surrounding `guard` expression.

ScriptX EBNF Grammar

Tokens and Literals

```
symbol          ::=   initialChar [ trailingChar ]*
initialChar     ::=   alphaChar | underscore
trailingChar    ::=   alphaChar | underscore | decimalDigit
nameLiteral     ::=   @ [ trailingChar ]+

arithOperator   ::=   + | - | * | /
negOperator     ::=   -
compOperator    ::=   = | == | <> | != | !== | < | > | <= | >=
integerConst    ::=   [ negOperator ] [ decimalDigit ]+
mantissa        ::=   integerConst [ .decimalDigit ] [ decimalDigit ]*
exponent        ::=   e integerConst
numericConst    ::=   mantissa [ exponent ]
                  |   hexLiteral
hexConst        ::=   [ hexDigit ]+
hexLiteral      ::=   0x [ hexDigit ]+

stringLiteral   ::=   " [ unicodeChar ]* "
unicodeChar     ::=   -- any printing char
                  |   escapeChar
                  |   \< hexConst >

alphaChar       ::=   a | b | c | . . . | x | y | z
decimalDigit    ::=   0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
hexDigit        ::=   0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | a | b | c | d | e | f
underscore      ::=   _
escapeChar      ::=   \n | \r | \t

endOfLine       ::=   \n\r -- new line plus carriage return (OS/2 and Windows)
                  |   \r -- carriage return (Macintosh)
                  |   ;
endOfStream     ::=   -- end of stream

arrayLiteral    ::=   #( exprList )
                  |   #( keyedList )
                  |   #( )
                  |   #( : )
exprList        ::=   simpleExpr [ , exprList ]*
keyedList       ::=   factor : simpleExpr [ , keyedList ]*

rangeLiteral    ::=   factor to factor [ by factor ]
                  |   factor by factor [ by factor ]
                  |   factor to factor [ inclOption ] continuous
                  |   factor inclOption to factor [ inclOption ] continuous
inclOption      ::=   inclusive | exclusive
```

Types of Expression

```
sentence        ::=   [ endOfLine ]* sentence
                  |   topLevelExpr endOfStream
                  |   topLevelExpr endOfLine
                  |   endOfStream
```

```
topLevelExpr      ::=  simpleExpr
                     |  globalExpr
                     |  guardExpr
                     |  assignmentExpr
                     |  repeatExpr
                     |  moduleExpr
                     |  inModuleExpr

innerLevelExpr    ::=  expr
                     |  localExpr

expr              ::=  simpleExpr
                     |  guardExpr
                     |  assignmentExpr
                     |  repeatExpr
                     |  blockControlExpr

simpleExpr        ::=  factor
                     |  ifExpr
                     |  caseExpr
                     |  forExpr
                     |  classDefExpr
                     |  objectDefExpr
                     |  functionDefExpr
                     |  callExpr
                     |  coercionExpr
                     |  pipeExpr
                     |  threadExpr
                     |  arithmeticExpr
                     |  pathExpr
                     |  indexExpr
                     |  rangeLiteral

factor            ::=  location
                     |  numericConstant
                     |  stringLiteral
                     |  nameLiteral
                     |  ?
                     |  nextMethod
                     |  anonFuncDefExpr
                     |  compoundExpr
                     |  arrayLiteral
```

Expression Syntax

```
compoundExpr      ::=  ( compoundExprSeq )
compoundExprSeq   ::=  innerLevelExpr moreExprs
moreExprs         ::=  endOfLine compoundExprSeq
                     |  endOfLine [ moreExprs ]*
                     |  empty

callExpr          ::=  factor paramSequence
                     |  factor ( simpleExpr , paramList )
                     |  factor ( symbol : factor [ , paramList ] )
                     |  factor ( )
paramSequence     ::=  [ symbol : ] factor [ paramSequence ]
paramList         ::=  [ symbol : ] simpleExpr [ , paramList ]
```

```
blockControlExpr    ::=   continue
                      |   exit [ with simpleExpr ]
                      |   return [ simpleExpr ]
                      |   throw again

coercionExpr        ::=   factor as factor
pipeExpr            ::=   | simpleExpr
threadExpr          ::=   simpleExpr &

arithmeticExpr      ::=   negOperator simpleExpr
                      |   simpleExpr arithOperator simpleExpr
                      |   simpleExpr compOperator simpleExpr
                      |   simpleExpr contains simpleExpr
                      |   not simpleExpr
                      |   simpleExpr and simpleExpr
                      |   simpleExpr or simpleExpr
```

Assignment and Variable Access

```
assignmentExpr      ::=   location := simpleExpr
                      |   location := assignmentExpr
                      |   location := guardExpr

location            ::=   symbol
                      |   ivAccess
                      |   elementAccess

localExpr           ::=   local binding [ , binding ]*
                      |   local funBindings
globalExpr          ::=   global binding [ , binding ]*
                      |   unglobal binding [ , binding ]*
                      |   global funBindings

binding             ::=   variable
funBindings         ::=   functionDefExpr [ , functionDefExpr ]*
                      |   freeMethDefExpr [ , freeMethDefExpr ]*

variables           ::=   variable [ , variable ]*
variable            ::=   [ constant ] symbol [ restriction ] [ initialVal ]

restriction         ::=   { class factor }
                      |   { kind factor }
                      |   { object factor }
initialVal          ::=   : simpleExpr
                      |   := simpleExpr

ivAccess            ::=   factor.symbol
                      |   factor's symbol
elementAccess       ::=   factor [ simpleExpr ]
```

Flow of Control

```
ifExpr              ::=   if simpleExpr then expr else expr
                      |   if simpleExpr then expr
                      |   if simpleExpr do expr
```

```
caseExpr          ::=   case [ simpleExpr ] of taggedFormList end
taggedFormList    ::=   taggedForm moreTaggedForms
moreTaggedForms   ::=   [ endOfLine ]* taggedFormList
taggedForm        ::=   factor : expr
                  |     otherwise : expr

repeatExpr        ::=   repeat while simpleExpr do expr
                  |     repeat until simpleExpr do expr
                  |     repeat expr while simpleExpr
                  |     repeat expr until simpleExpr

forExpr           ::=   for forSources [ forTest ] forBody
forSources        ::=   forSource [ , forSource ]*
forSource         ::=   symbol in simpleExpr
                  |     symbol := simpleExpr
                  |     simpleExpr
forTest           ::=   while simpleExpr
                  |     until simpleExpr
forBody           ::=   do expr
                  |     collect [ into simpleExpr ] [ by factor ] [ as factor ] simpleExpr
                  |     select factor [ into simpleExpr ] \
                              [ by factor ] [ as factor ] if simpleExpr
```

Definition of Classes

```
classDefExpr      ::=   class symbol ( supersList ) classBody end
supersList        ::=   simpleExpr [ , supersList ]*
classBody         ::=   [ classVars ] [ instanceVars ] [ classMethods ] [ instMethods ]

classVars         ::=   class var[iable]s varList
instanceVars      ::=   inst[ance] var[iable]s varList
varList           ::=   [ qualifiers ] variable moreVars
moreVars          ::=   [ endOfLine ]+ varList
                  |     [ , varList ]*
qualifiers        ::=   [ readOnly ] [ transientOption ] [ reference ]
transientOption   ::=   transient [ initializer factor ]

classMethods      ::=   class methods methods
instMethods       ::=   inst[ance] methods methods
methods           ::=   methodDefExpr moreMethods
moreMethods       ::=   [ endOfLine ]* methods
```

Definition of Objects

```
objectDefExpr     ::=   object [ symbol ] ( [ classesList ] ) objectBody end
classesList       ::=   simpleExpr [ , classesList ]
objectBody        ::=   [ initialization ] [ instanceVars ] [ instMethods ] \
                              [ settings ] [ contents ]

initialization    ::=   initList | empty
initList          ::=   initForm moreInitForms
initForm          ::=   symbol : simpleExpr
moreInitForms     ::=   [ endOfLine ]+ initList
                  |     [ , initList ]*
```

```
settings          ::=   settings initList
contents          ::=   contents contentsList
contentsList      ::=   simpleExpr moreContentForms
moreContentForms  ::=   [ endOfLine ]+ contentsList
                  |     [ , contentsList ]*
```

Definition of Functions and Methods

```
functionDefExpr   ::=   f[unctio]n symbol [ argSeq ] body
                  |     f[unctio]n symbol ( [ argList ] ) body
methodDefExpr     ::=   method [ getterOrSetter ] symbol argSeq body
freeMethDefExpr   ::=   [ class ] method [ getterOrSetter ] symbol argSeq body
anonFuncDefExpr   ::=   ( anonArgs -> compoundExprSeq )
body              ::=   -> expr
getterOrSetter    ::=   get | set

argSeq            ::=   [ positionalArgSeq ] [ restArg ] [ keywordArgSeq ]
argList           ::=   [ positionalArgList ] [ restArg ] [ keywordArgList ]
anonArgs          ::=   [ paramSequence ] [ restArg ] [ keywordArgSeq ]
                  |     empty
positionalArgSeq  ::=   argument [ argument ]*
positionalArgList ::=   argument [ , positionalArgList ]*
restArgs          ::=   #rest symbol
keywordArgSeq     ::=   #key keyArgPair [ keyArgPair ]*
keywordArgList    ::=   #key keyArgPair [ , keyArgPair ]*
argument          ::=   symbol [ restriction ]
keyArgPair        ::=   argument : [ symbol ] [ ( simpleExpr ) ]
```

Modules

```
inModuleExpr      ::=   in module symbol
moduleDefExpr     ::=   module symbol [ moduleOptions ] end
moduleOptions     ::=   [ moduleOption ]*
moduleOption      ::=   exports [ readonly ] [ inst[ance] var[iable]s ] symbolSeq
                  |     uses symbolSeq
                  |     uses symbol [ with usesOptions ] end
symbolSeq         ::=   symbol [ , symbol ]*

usesOptions       ::=   [ usesOption ]*
usesOption        ::=   imports everything
                  |     imports [ readonly ] [ inst[ance] var[iable]s ] symbolSeq
                  |     exports everything
                  |     exports [ readonly ] [ inst[ance] var[iable]s ] symbolSeq
                  |     prefix symbol
                  |     excludes [ readonly ] [ inst[ance] var[iable]s ] symbolSeq
                  |     renames [ readonly ] [ inst[ance] var[iable]s ] renameSeq
renameSeq         ::=   symbol : symbol [ , symbol : symbol ]*
```

Exception Handling

guardExpr	`::=`	**`guard`** *expr* [**`catching`** *catchList*] [**`on exit`** *expr*] **`end`**
catchList	`::=`	*catcher moreCatchers*
moreCatchers	`::=`	[*endOfLine*]* *catchList*
	`|`	[*endOfLine*]* *moreCatchers*
	`|`	*empty*
catcher	`::=`	*symbol* [*symbol*] [**`:`** *expr*]

APPENDIX

Unicode Escape Characters

B

ScriptX maps Unicode values into the character set of the underlying platform. Most platforms on which the Kaleida Media Player runs do not yet support Unicode. The Macintosh and Windows have their own internal character sets, which are incompatible. ScriptX currently depends on the underlying platform for screen display of fonts. The font, in turn, is responsible for the glyph.

Table B-1 lists selected Unicode characters that can be mapped to the character sets of current Kaleida Media Player platforms. For a complete listing of Unicode characters, see *The Unicode Standard*, published by Addison-Wesley.

The Unicode character encoding standard is developed by The Unicode Consortium, incorporated as Unicode, Inc. The Unicode Consortium is a non-profit organization whose charter is to maintain and promote the Unicode standard worldwide. Founding members include major companies and institutions involved in international computing.

Although you can specify any valid Unicode character in ScriptX, a character can only be displayed if it can be mapped to the character set of the underlying operating system. The operating system, in turn, must supply a font that renders the appropriate glyph.

Table B-1: Unicode characters

Character	Unicode	Character	Unicode
¡	00A1	å	00E5
¢	00A2	æ	00E6
£	00A3	ç	00E7
¤	00A4	è	00E8
¥	00A5	é	00E9
§	00A7	ê	00EA
¨	00A8	ë	00EB
©	00A9	ì	00EC
ª	00AA	í	00ED
«	00AB	î	00EE
¬	00AC	ï	00EF
®	00AE	ñ	00F1
¯	00AF	ò	00F2
±	00B1	ó	00F3
´	00B4	ô	00F4
µ	00B5	õ	00F5
¶	00B6	ö	00F6
·	00B7	÷	00F7

Table B-1: Unicode characters *(Continued)*

Character	Unicode	Character	Unicode
¸	00B8	ø	00F8
º	00BA	ù	00F9
«	00BB	ú	00FA
¿	00BF	û	00FB
À	00C0	ü	00FC
Á	00C1	ÿ	00FF
Â	00C2	ı	0131
Ã	00C3	Œ	0152
Ä	00C4	œ	0153
Å	00C5	Ÿ	0178
Æ	00C6	*ƒ*	0192
Ç	00C7	˚	02DA
È	00C8	¸	02DB
É	00C9	˜	02DC
Ê	00CA	ˇ	0306
Ë	00CB	˙	0307
Ì	00CC	–	2013
Í	00CD	—	2014
Î	00CE	‘	2018
Ï	00CF	’	2019
Ñ	00D1	‚	201A
Ò	00D2	“	201C
Ó	00D3	”	201D
Ô	00D4	„	201E
Õ	00D5	†	2020
Ö	00D6	‡	2021
¥	00D7	•	2022
Ø	00D8	‰	2030
Ù	00D9	‹	2039
Ú	00DA	›	203A
Û	00DB	∂	2202
Ü	00DC	™	2122
ß	00DF	⁄	2215
à	00E0	∞	221E
á	00E1	≠	2260
â	00E2	≤	2264
ã	00E3	≥	2265
ä	00E4		

Index

D

E

F

L

M

N

O

T

U

V

W

Colophon

PRODUCT DEVELOPMENT

VP Engineering • Chris Jette

Chief Architect • John Wainwright

Kaleida Founder • Erik Neumann

Kaleida Fellow • Andrew Nicholson

ScriptX Language Team • Wade Hennessey (mgr), Mike Agostino, Eric Benson, Ross Nelson, Chris Richardson, David Williams

ScriptX Media Team • Erik Neumann (mgr), Vidur Apparao, Ikko Fushiki, Jennifer Jacobi, Chih Chao Lam, Michael Papp, Ken Tidwell, Ken Wiens

Cross-Platform Team • Elba Sobrino (mgr), Yukari Huguenard, Alan Little, Jeanne Mommaerts, Charlie Reiman, Richard Roth, Vladimir Solomonik, Clayton Wishoff, Wanmo Wong

Quality Engineering Team • Ermalinda Horne (mgr), William Africa, Adela Bartl, Ron Decker, Suzan Ehdaie, Rajiv Joshi, Tony Leung

Technical Publications Team • Douglas Kramer (mgr), Jocelyn Becker, Alta Elstad, Maydene Fisher, Howard Metzenberg, Sandra Ware

Application Support Engineering Team • Ray Davis, Rob Lockstone, Felicia Santelli, Su Quek

AND ALL OUR FELLOW KALEIDANS

Masumi Abe, Harvey Alcabes, Rob Barnes, Amy Benesh, Fred Benz, Alison Booth, Mike Braun, Mark Bunzel, Janet Byler, John Cummerford, Shannon Garrow, Marylis Garthoeffner, Norman Gilmore, Bill Grotzinger, Sue Haderle, Diana Harwood, Don Hopkins, Bill Howell, John Hudson, Pat Ladine, Fritzi Lareau, Deb Lyons, Karl May, Steve Mayer, Victor Medina, Gabe Mont-Reynaud, Tom Morton, Randy Nelson, Christy O'Connell, Karen O'Such, Christian Pease, David Rosnow, Molly Seamons, Ken Smith, Michelle Smith, Ivan Vazquez, Greg Womack

THANKS TO KALEIDA ENGINEERING ALUMNI, INCLUDING:

Sarah Allen, Dan Bornstein, Jim Inscore, David Kaiser, Shel Kaphan, Laura Lemay, Dave Lundberg, Leslie Lundquist, Fred Malouf, Dmitry Nasledov, Steve Riggins, Steve Shaw, Cheng Tan, Phil Taylor

Special Thanks To...

Lady, Nikki, Boots, Ella, Iggy, Kiri, Frisky, Tyler and Rufus

THIS DOCUMENT

Writing • Howard Metzenberg, Maydene Fisher, Jocelyn Becker

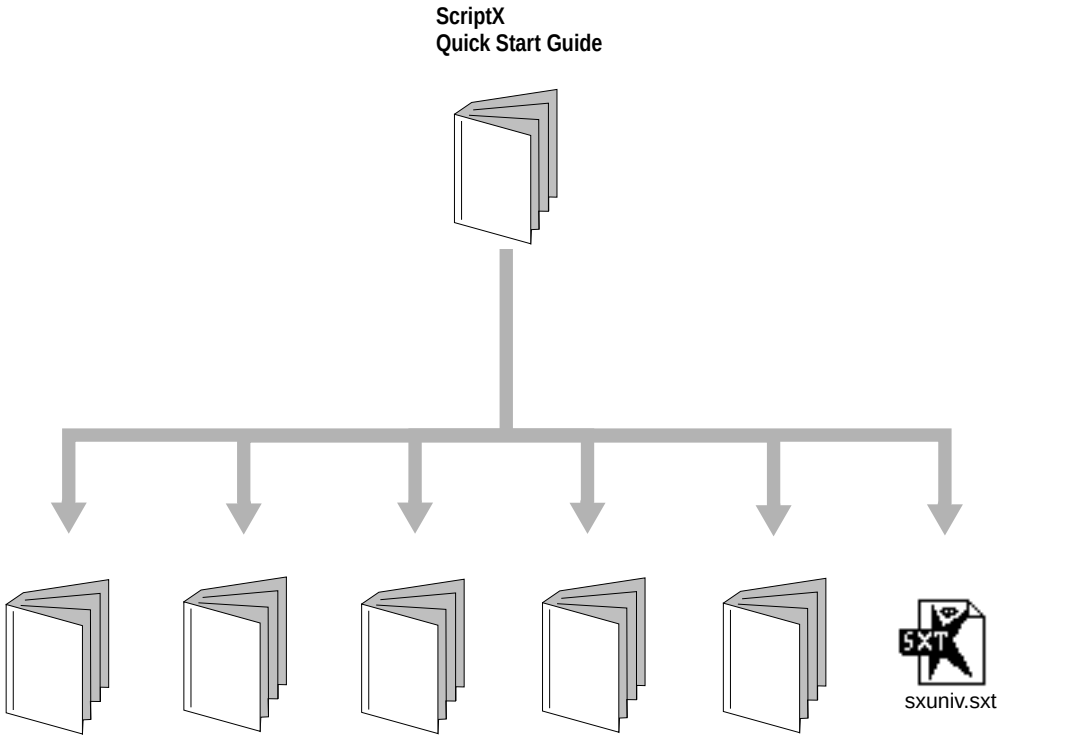
Original author • Laura Lemay

Book production • Sandra Ware, Jacki Dudley, Diana Harwood

Publication management • Doug Kramer, Jim Inscore

This book was created electronically using Adobe FrameMaker on Macintosh Quadra computers.

Documentation Roadmap



Acrobat versions are available for most manuals