ScriptX
Class Reference

Kaleida Labs

December 1995

©1995 Kaleida Labs, Inc. All rights reserved.

U. S. Patent Nos. 5,430,875 and 5,475,811. Other patents pending.

This manual, as well as the software described in it, are furnished under license and may only be used in accordance with the terms of that license. Under the terms of that license: (1) this manual may not be copied in whole or in part, and (2) this manual may be used only for the purpose of using software provided by Kaleida Labs, Inc. ("Kaleida") and creating software products which run on the Kaleida Media Player. The contents of this manual is furnished for informational use only, is subject to change without notice, and should not be construed as a commitment by Kaleida of any kind. Kaleida assumes no responsibility or liability for any errors or inaccuracies that may appear in this book.

"ScriptX", "Kaleida Media Player", the "K-man" logo and "ScriptX Language Kit" are Kaleida trademarks that may be used only for the purpose of identifying Kaleida products. Your use of Kaleida trademarks for any commercial purpose without the prior written consent of Kaleida may constitute trademark infringement and unfair competition under state and federal law. All other products or services mentioned in this manual are identified by trademarks of the companies who market those products or services. Inquiries concerning such trademarks should be made directly to those companies.

This manual is a copyrighted work of Kaleida with all rights reserved. This manual may not be copied, in whole or in part without the express written consent of Kaleida. Under the copyright law, copying includes photocopying, storing electronically, or translating into another language.

The ScriptX Language and Class Library ("ScriptX") described in this manual is a copyrighted work of Kaleida. ScriptX also contains technology described in pending U.S. patent applications. You may use and copy ScriptX solely for the purpose of creating software products that run on the Kaleida Media Player by writing computer source code that is compiled into object code by software provided by Kaleida. You may not use or copy ScriptX for the purpose of writing computer source code that is compiled into object code or otherwise executed with software supplied by any other provider who has not been expressly licensed for that purpose by Kaleida.

For Defense agencies: Restricted Rights Legend. Use, reproduction or disclosure is subject to restrictions set forth in subparagraph (c)(1)(ii) of the Rights in Technical Data and Computer Software clause at 225.227-7013.

For Civilian agencies: Restricted Rights Legend. Use, reproduction or disclosure is subject to restrictions set forth in subparagraphs (a) through (d) of the Commercial Computer Software Restricted Rights clause at 52.227-19 and the limitations set forth in Kaleida's standard commercial agreement for the software described in this manual. Unpublished rights reserved under the copyright laws of the United States.

Printed in the USA.

Kaleida Labs, Inc.

c/o Apple Computer

1 Infinite Loop

Cupertino, CA 95014

Contents

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 Quick Start Guide* includes instructions for installation, a documentation roadmap, tutorials, and a list of new features. It also describes how to contact Kaleida. This is the place to begin when starting with ScriptX.
- The *ScriptX Language Guide* is a practical guide to using the ScriptX programming language. It provides complete functional descriptions of the language, but does not cover the ScriptX classes. Anyone programming in ScriptX will want to use this book.
- The *ScriptX Components Guide* provides a detailed description of the ScriptX components. For each component it describes the conceptual overview, how it works, how to use it, and complete, working sample scripts. It covers the full range of ScriptX features, from the multimedia title to the operating system services. This manual is essential to anyone designing and building multimedia titles and applications in ScriptX. It is the companion volume to the *ScriptX Class Reference*.
- The *ScriptX Class Reference* (this manual) is a detailed reference to the ScriptX classes and API (Application Programming Interface). It provides, in dictionary form, a complete specification of the classes, methods, variables, and functions available for building multimedia titles and applications in ScriptX. It is the companion volume to the *ScriptX Components Guide*.
- The *ScriptX Tools Guide* provides information about the ScriptX development process, tools, and importers, as well as how to extend ScriptX. The first part discusses how to use the Listener, debugger, browser, profiler, 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 from the three manuals *ScriptX Language Guide*, *ScriptX Components Guide*, and *ScriptX Class Reference*. 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.

Note – Release notes are included with the ScriptX software. Be sure to refer to these notes for the known restrictions and bugs in the current version of ScriptX.

Audience

This document is written for the ScriptX programmer who wants to design and build multimedia titles, applications, or tools to be delivered and run with the Kaleida Media Player.

This document describes API with the implicit understanding that, unless otherwise noted, those API are available at runtime with the Kaleida Media Player. Therefore, if a feature is not available at runtime, such as the `fileIn` instance method in the `DirRep` class, a note is included to that effect.

This document is written at the level of ScriptX, and no knowledge of the C language is necessary. (C is required only for those who want to write C extensions to ScriptX).

Titles, Applications and Tools

The terms *title*, *application*, and *tool* have specific meanings that are often mean different things to different people. While their meanings sometimes overlap, in the context of this document, they are defined as follows:

title – A program in which an end-user plays, explores, learns or is entertained.

application – A program in which an end-user gets work done, generally performing a specific, useful task.

tool – A program used to create a title or application. In general, a ScriptX tool runs only with the ScriptX executable, not with the Kaleida Media Player.

Programs do not have to be monolithic. A program that allows you to explore (a title) could contain a word processor (an application) and the means to modify the word processor (a tool). Thus, any program can contain other kinds of programs.

A title or application developer requires use of the API available in the *ScriptX Class Reference*. A tool developers also requires use of the API available in the *ScriptX Tools Guide*.

Throughout this document, in general we use the single term “title” to mean any of these three kinds of programs.

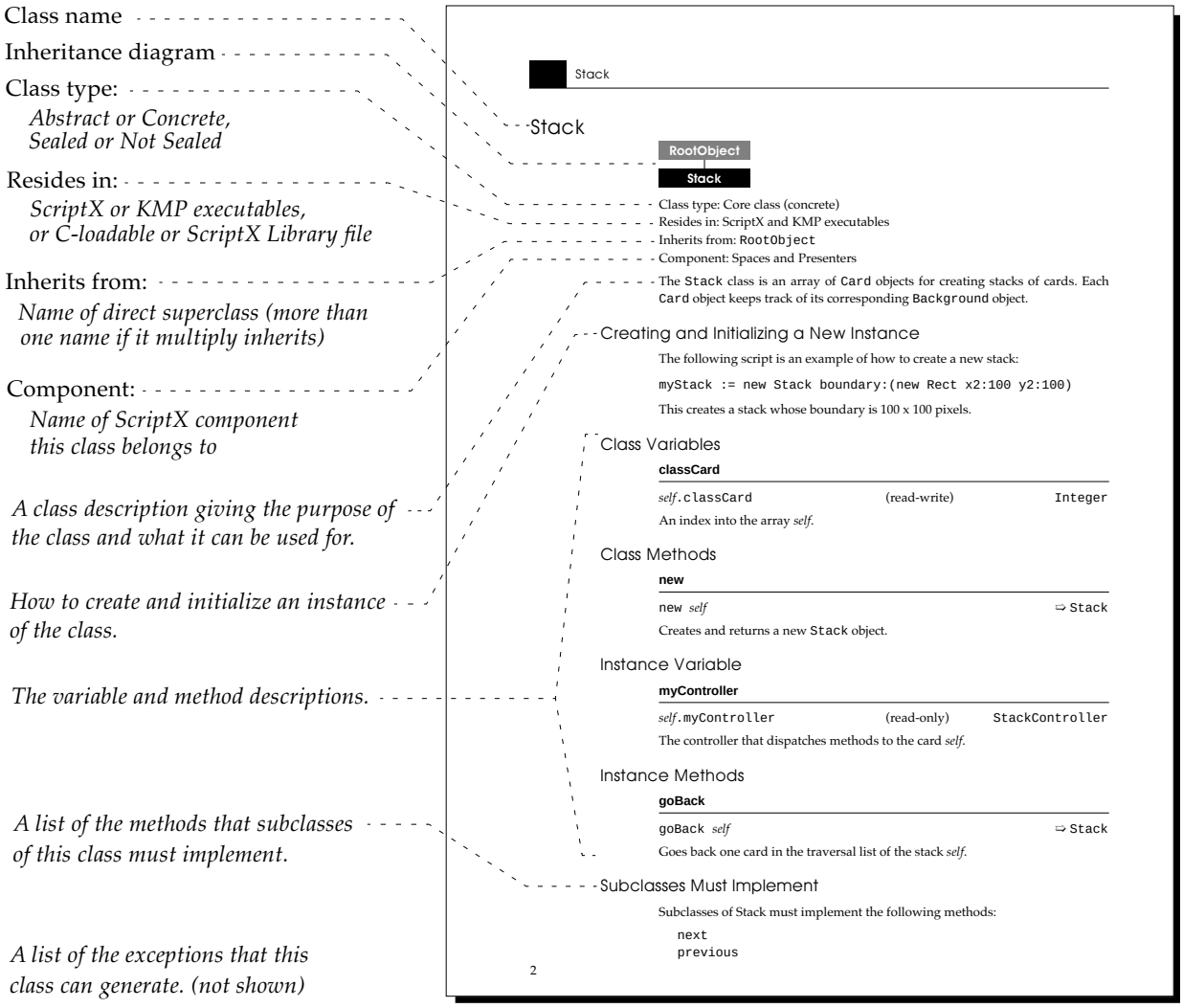
Summary of Contents

This manual contains the following sections:

- “Preface” describes what headings a typical class description contains, as well as what conventions are used throughout this manual.
- Chapter 1, “Information Common to All Classes,” includes, obviously, information that is common to all classes in this manual.
- Chapter 2, “Global Functions,” describes all of the public global functions in ScriptX.
- Chapter 3, “Global Constants and Variables,” describes all of the public global constants and variables in ScriptX.
- Chapter 4, “Class Descriptions” constitutes the bulk of this manual. It describes all of the public classes in ScriptX, in alphabetical order, from A-Z. It includes core classes, loadable classes and scripted classes. It does not include tool classes—they are in the *ScriptX Tools Guide*.

Class Descriptions

The following diagram shows how a class is described, using a fictitious class as an example. The variable and method headings are described on the following page.



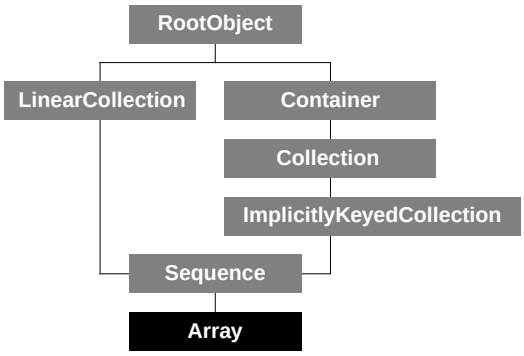
Each class description contains the headings described as follows; a heading doesn't appear if the class contains no information for that heading.

Inheritance Diagram

The inheritance diagram for each class displays all of its superclasses, including all multiple inheritance paths, up to `RootObject`. The classes are shown from left-to-right in the order that they are searched for inherited behavior and properties. When an operation involving a keyword, variable or method is performed on a class or its instance, ScriptX searches up the inheritance tree for the first class that implements the operation. As shown in the diagram, the classes are searched in this order:

From bottom-to-top, left-to-right, with `RootObject` last.

For example, the `Array` class, as shown in the following diagram, is searched in the order `Array`, `Sequence`, `LinearCollection`, `ImplicitlyKeyedCollection`, `Collection`, `Container`, and `RootObject`.



Basic Class Attributes

Class Type:

Indicates which of the following types the class is:

Core Class – Resides in the ScriptX and KMP executables. Most classes fall into this category, and they are documented in this document, the *ScriptX Class Reference*.

Tool Class – Resides only in the ScriptX executable, not in the KMP. These are documented in the *ScriptX Tools Guide*.

Loadable Class – Resides in a C-loadable. We define "loadable" here to mean only the C-loadables, not the scripted loadables. For a list of loadable classes, see the appendix "Loadable Extensions."

Scripted Class – Resides in a ScriptX library container, accessory container, title container or script. For a list of scripted classes, see the appendix "Loadable Extensions."

In addition, a class can be concrete (instantiable) or abstract (not instantiable), and sealed (subclassable) or not sealed. See the Glossary for more complete definitions of these terms.

Resides In:

Indicates which file the class is defined in. There are four places that a file can reside:

ScriptX executable – This executable has the filename "`ScriptX`" on the Macintosh, and "`scriptx.exe`" on Microsoft Windows.

KMP executable – The Kaleida Media Player, which has the filename "`KMP`".

`.lib` file – Loadable classes reside in C-libraries.

`.sxl` file – Scripted classes reside in ScriptX library containers.

By definition, core classes are defined in both the ScriptX and KMP executables, and tool classes are defined only in the ScriptX and not the KMP executable.

For loadable and scripted classes, the "Resides in" line also indicates which executables they work with. In this version of ScriptX, all loadable and scripted classes shipped from Kaleida work with both ScriptX and the KMP executables. (That is, there are no tools-only loadables at this point.)

Inherits From:

Indicates the direct superclasses of the given class. Since ScriptX supports multiple inheritance, a class can have more than one direct superclass.

Component:

Indicates which component the class is documented in. The components correspond to chapters in the *ScriptX Components Guide*.

Inherited API

For class variables, class methods, instance variables, and instance methods (the following four sections), the inherited variables and methods are also listed ahead of the variables and methods defined in that particular class. Every class also inherits the class methods from `Behavior` and instance methods from `RootObject`, as described in the "Information Common to All Classes" chapter.

Class Variables

The class variables for the specified class are listed and described. These are the variables that are kept in the class and are shared by all instances of the class. Each class variable has the same value for all instances of the class. The syntax format is as shown below under "Instance Variables," except *self* is a class, not an instance.

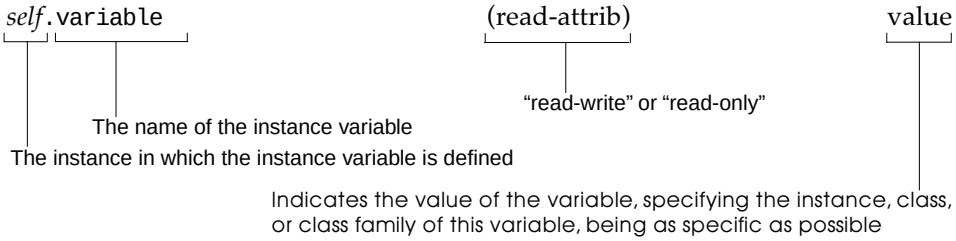
Class Methods

The class methods for the specified class are listed and described. These are the methods that operate on classes (rather than instances)—their first argument must be a class name. The class method syntax is as shown below in "Instance Methods," except the argument *self* is a class, not an instance.

Instance Variables

The instance variables for the specified class are listed and described. These are the variables whose values are kept with the instances of the class (rather than with the class itself) and therefore can vary with each instance. The variable syntax is in the following format:

variable



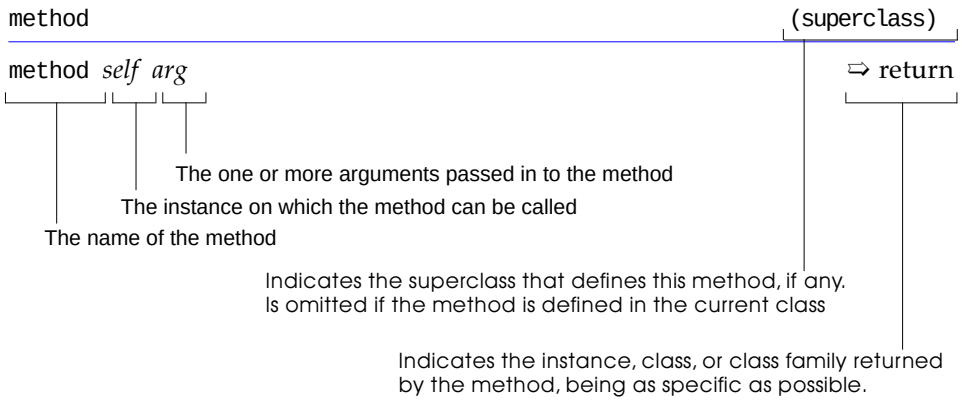
The value of the variable is specified at the right-end of the syntax line, and can be a class name, class family name, *(object)*, *(class)* or *(function)* as specified in the next section for the return value of an instance method. You can get access to an instance variable by using either of two equivalent forms: *self*`.variable` (as shown above) or *self*`'s variable`.

A variable can be either read-write, meaning you can change it, or read-only, meaning you cannot change it. In the current version, ScriptX does not always enforce read-only, so it may actually be possible for you to change a variable documented as read-only, but doing so may cause errors.

Notice that when a variable is a collection, such as `accessories` in the `TitleContainer` class, or an object with instance variables, it may not be obvious what read-only means. Read-only means that you cannot change the object assigned to the variable. However, if that object is a collection or has instance variables, it does not say whether or not you can change the members of that collection or the values of its instance variables. Those are determined by the attributes of the object itself.

Instance Methods

The instance methods for the class are listed and described. These are the methods that operate on instances of the class rather than on the class itself. The instance method syntax is in the following format. The syntax shown is the *calling* syntax rather than the *implementing* syntax. That is, we show how the method is called rather than how it would appear in a method definition.



The arrow (⇨) itself means *returns*. After the arrow is the value the method returns, giving as specific a value as possible. Every method returns an instance of some class. For many methods, the most that can be generally known is the class of the returned instance, and so that class appears here. Any of the following can appear as the return:

- A class name indicates the class of the returned instance. Two class names can appear here (separated by commas) if the value returned can be an instance of either class; for example, `String`, `Number`.
- A class family name can appear here if the value returned can be an instance of any class in that family. The class family name can be an abstract class. For example, if the return value specifies `Number`, the method can return an instance of `Integer`, `Fixed`, or possibly any other subclass of `Number`.
- *self* appears here if the object returned is the same object as the first argument *self* passed in to the method. Likewise, another method argument can appear here if that argument is returned.
- *(none)* means the return value is unspecified—don't use it for anything. (This value appears as `OK` in the ScriptX Listener window.)

- *(function)* means the return value is an instance of `AbstractFunction`, the abstract parent class to which all functions, generics, and methods belong. `AbstractFunction` and its subclasses, which include `Generic` and `ByteCodeMethod`, are not documented in the *ScriptX Class Reference*. They cannot be specialized, and they have no visible properties or methods.
- *(object)* means the return value is an instance of any class. This has the same meaning as specifying `RootObject`, since `RootObject` family includes all classes.
- *(class)* means the return value is any class. This has the same meaning as specifying `RootClass`.

Square brackets around an argument, `[optional]`, indicates that a keyword argument is optional; all other arguments are required. All positional arguments are required. See the section "Positional and Keyword Arguments in Methods" in the chapter "Information Common to All Classes" for more information about arguments.

As a convention in this document only, we omit a backslash at the end of a method that wraps to the next line, even though a backslash would be required if you were writing the method in a program.

Inherited API

Within each of the previous four headings Class Variables, Class Methods, Instance Variables, and Instance Methods, a list of inherited API can appear, listed in three columns. For example, the `TwoDShape` class inherits the following instance variables:

Inherited from `Presenter`:

`presentedBy`	`subPresenters`	`target`

Inherited from `TwoDPresenter`:

`bBox`	`globalRegion`	`transform`
`boundary`	`globalTransform`	`width`
`changed`	`height`	`window`
`clock`	`imageChanged`	`x`
`compositor`	`isVisible`	`y`
`direct`	`position`	`z`
`eventInterests`	`stationary`	
`globalBoundary`	`target`	

This information is indicates to you where else in this document those variables or methods are documented. Although in most cases, these inherited variables or methods can be used with the subclass (`TwoDShape`, in the above example), be aware that this is not always true. In other words, just because a method or variable works with a superclass does not mean that it works the same way with the subclass. Some instance variables are initialized when you create an instance of the class and at that point are read-only and not meant to be changed, even though they are documented as read-write in their superclass. Some methods will throw exceptions for subclasses because they are not seekable, not writable, or have some other constraint. This manual does not attempt to document every condition under which every inherited method or variable works.

For example, the `Pair` class is a collection that always has exactly two values—you can change these values (using `setNth`) but you cannot add or delete values. That is, although `Pair` inherits the `add` method from `Collection`, calling `add` on `Pair` reports an exception: "Collection has reached its bounds."

Subclasses Must Implement

Subclasses of this class must implement the methods that are listed (descriptions of the methods are found elsewhere). More specifically, these methods must be implemented somewhere in the inheritance tree between this class and every instantiable subclass. For example, `Collection` is an abstract class that does not implement the `add` method, but its instantiable subclass `Sequence` implements `add`. (If `add` were not implemented, items could not be added to instances of `Sequence`.)

Global Constants

These are any special instances of the class. Examples include `empty`, `true`, `false`, `whiteBrush`, and `blackBrush`.

Exceptions Generated

The exceptions that can be generated when using this class are listed and described.

Note – Methods that a class inherits are not documented with that class in detail if their *behavior* is the same as in the superclass they inherit from; the methods are documented if this class provides different behavior.

Conventions Used in Manual

Font Conventions

In order to distinguish elements of ScriptX, this manual uses the following font conventions:

- Code samples, class names, method names, and other literal elements are in `Courier.`
- Variable argument names are set in *italic*.

As an example of both rules, here is the method syntax for creating a new instance of `Clock` object:

```
new self masterClock:clock scale:number
```

Upper/Lowercase Conventions

The following conventions are used in naming elements of ScriptX:

- Class names and module names begin with an uppercase letter.
  Examples of classes: `Clock`, `Player`, `Card`
  Examples of modules: `Scratch`, `ScriptX`
- All other names begin with a lowercase letter.
  This includes names of methods, functions, constants, and variables. Examples: `new` (method), `red` (instance variable), *obj* (user-defined variable).
- All compound names use uppercase to begin each embedded word.
  This applies to all categories: classes, methods, functions, constants and variables. Examples:

`PushButton` class (has an uppercase “B”)

`selectAll` method (has an uppercase “A”)

Class Names Used as Normal Text

Figure P-1 shows the `Space` family of classes, which is made of the `Space` class and all its subclasses. To convey that “x is an instance of `Space`, or any subclass of `Space`,” it’s a convenient shortcut to say simply “x is a space.” A space can be an instance of `TwoDSpace`, `Window`, `PageLayer`, or `GroupSpace` or class, but not the `Space` class since it is abstract.

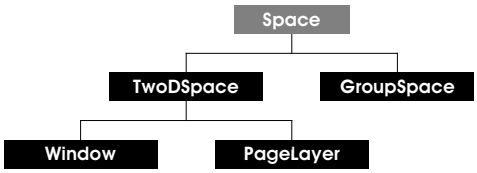


Figure Preface-1: The Space family of classes.

Notice that in the phrase “x is a space”, the word “space” does not use the conventions for indicating a class name (`Space`)—it does not have an initial uppercase letter, nor is it set in Courier. The word looks like any other regular word. This demonstrates the following convention used throughout this manual:

> *A class name that appears as normal text in lowercase letters refers to an instance that is created from the corresponding family of classes.*

Here’s another example of the word “space” in lowercase used in a sentence:

> A space is an environment where objects live and interact.

This sentence is simpler than, and yet has the same meaning as:

> An instance of the `Space` family is an environment where objects live and interact.

Throughout this manual this convention is used. An unwary reader might not catch the full meaning of such a sentence, which leads to this corollary:

> Watch out for class names formatted as regular text— such as space, action, presenter, container, and so on. They have the specific meaning described in this convention.

Note that the lowercase term “space” specifically does not refer to the `Space` class itself, or necessarily to an instance of the `Space` class. In fact, `Space` is indeed an abstract class and cannot be instantiated, so the term “space” *must* refer to an instance of a subclass of `Space`.

As another example, someone unfamiliar with the ScriptX classes might not realize that the use of the word “presenter” in the sentence “A presenter can draw itself” refers specifically to an instance from the `Presenter` family of classes.

The following table lists some of the lowercase class names that might not be obvious to the uninitiated.

Lowercase Class Name	Meaning
space	an instance of the `Space` family of classes
action	an instance of the `Action` family of classes
presenter	an instance of the `Presenter` family of classes

Lowercase Class Name	Meaning
point	an instance of the `Point` family of classes
container	an instance of the `Container` family of classes
2D space	an instance of the `TwoDSpace` family of classes
sequence	an instance of the `Sequence` family of classes
clock	an instance of the `Clock` family of classes
keyed collection	an instance of the `KeyedCollection` family of classes

Important Terms

The following terms are used throughout this manual. Also refer to the Glossary appendix for more terms.

abstract class – A type of class designed for subclassing rather than instantiating. An abstract class can range from containing no implementation (a true abstract class) to containing full implementation (a true mixin class). Contrast with *concrete.*

concrete class – A type of class that can be instantiated. Some concrete classes are instantiated by the system and cannot be instantiated by the author— `Boolean`, for example, can have only two instances, `true` and `false`. In general, a concrete class has a `new` class method for creating instances. For some concrete classes, this method is not visible at the scripter level. For example, new instances of the `Number` subclasses are automatically generated by the compiler as it encounters numbers in a script. Contrast with *abstract*.

sealed class – A type of class that cannot be subclassed. Very few classes in ScriptX are sealed. The `Number` class is an example of a sealed class. Note that sealed classes can have predefined subclasses—for example, `Number` has the subclass `Fixed`.

mixin class – A type of class that can usefully be mixed into other classes. All classes in ScriptX are technically mixin classes, although they don't all add functionality. The classes `Dragger` and `SequenceCursor` are true mixin classes in that they contain a full implementation—by mixing them in you automatically get the added functionality you want without further implementation.

method – A function that is defined and implemented in a class or an instance of a class. A *class* method requires a class as its first argument; an *instance* method requires an instance as its first argument. When used alone, the term method could apply to either—its meaning depends on the context. The methods of a class or instance are often called its *behavior.*

You can easily distinguish between class methods and instance methods by looking at whether its first argument is a class or an instance. In the following example, `new` is a class method, since it operates on `Point`, a class. However, `xSetter` is an instance method, since it operates on `pt`, an instance.

```
pt := new Point   -- 'new' is a class method
xSetter pt 100    -- 'xSetter' is an instance method
```

instance variable – A variable of a particular instance; this variable holds some state information for that instance. The value of an instance variable is kept with the instance (in contrast to a class variable, where the value is kept with the class). One instance of `Point` might have an x-location of 100, another instance might have an x-location of 50. Thus, the values of instance variables distinguish instances of the same class. In the following example, `x` is an instance variable.

```
pt := new Point   -- Creates a new instance of Point
```

```
pt.x := 100       -- Sets x of pt to 100
```

class variable – A variable of a particular class; this variable holds some state information for that class. ScriptX has very few class variables in its core classes.

C H A P T E R

Information Common to All Classes

1

The ScriptX classes represent basic building blocks of a multimedia title or application. As shown in the ScriptX Class Inheritance Tree at the front of this book, they cover a wide range of functionality, from specifying integers and arrays, to compositing video and audio on-screen. These classes form the standard for developing titles and applications using the ScriptX language.

This document contains both core classes and loadable classes; the core classes are those resident in the Kaleida Media Player file, while the loadable classes are separate loadable files that extend the ScriptX capabilities, and are written either in C, OIC (Objects in C), or in ScriptX.

This manual documents the public, platform-independent classes, methods, variables, constants, and functions that are available in version 1.1 of the Kaleida Media Player. This set is known as the ScriptX API (Application Programming Interface) targeted for the Kaleida Media Player. Notice this excludes the classes available only in the ScriptX Development Environment; these are documented in the *ScriptX Tools Guide*.

These classes are accessible at the scripter level, and they are portable to all ScriptX platforms, except where noted. Those exceptions are due to constraints or features inherent in the underlying operating systems. From the perspective of playing back a ScriptX title, *these classes are available in ScriptX players on all platforms*—currently Macintosh, Power Macintosh, Windows, and OS/2. Thus, if you write a title or application based on the classes in this document, it will run on all platforms that ScriptX runs on.

A few classes, methods, functions, and variables are available only in the development environment, not the Kaleida Media Player, since they require the ScriptX bytecode compiler, or of interest only while debugging a title, such as the `fileIn` method, `warning` method and `showChangedRegion` instance variable.

Note that substrate (private) classes, metaclasses and the C-language interface are not included in this manual.

Each class in ScriptX belongs to a component, as noted at the beginning of each class description. ScriptX is made up of the components described in the companion manual, the *ScriptX Components Guide*.

The information in this chapter is common to all classes in this reference guide, and is described in the following sections:

- Creating a Subclass
- Inherited Methods and Variables
- Setters and Getters for Instance Variables
- Positional and Keyword Arguments in Methods
- Creating and Initializing a New Instance
- Overriding the Initialization in a Subclass
- Protocols

Creating a Subclass

The core classes are defined by Kaleida Labs. If none of the core classes performs the task you need, you can define your own class by subclassing any existing core class. In this subclass you can specialize the instance methods and the setter/getter methods for its instance variables.

You should never try to directly modify the definition of a core class in any way—you should instead create a subclass and specialize that subclass. For example, if you want the `tickle` method of the `Gravity` class to operate differently, you should create a subclass of it, then specialize the method there.

To create a subclass of a class, use the `class` construct described in the *ScriptX Language Guide*. Be sure to implement the methods described in the "Subclasses Must Implement" section of the class description. For example, if you subclass `Sequence`, you must implement the methods `addNth`, `deleteNth`, `getNth`, and so forth.

Inherited Methods and Variables

Every class description includes a 3-column listing of the methods and variables that the class inherits, if any. However, the class methods and instance methods common to *every* class are listed below instead of with the class descriptions. (This saves repeating them for every class.) There are no class variables or instance variables common to all classes.

For example, `getDirectGenerics` is a useful method that you can call on any class, and `getClass` is a useful method that you can call on any instance. The complete lists follow.

Class Methods

Every class inherits the following class methods from the `Behavior` class:

Inherited from `Behavior`:

`canClassDo`	`getSupers`	`methodBinding`
`getDirectSubs`	`isDirectSub`	`new`
`getDirectSupers`	`isMemberOf`	
`getSubs`	`isSub`	

Instance Methods

Every class inherits the following instance methods from the `RootObject` class:

Inherited from `RootObject`:

`addNewToStorageContainer`	`getAllGenerics`	`localEqual`
`addSubObjects`	`getAllMethods`	`localLt`
`addToStorageContainer`	`getClass`	`morph`
`afterLoading`	`getClassName`	`prin`
`afterLoadingIfNecessary`	`getDirectGenerics`	`recurPrin`
`allIVNames`	`getDirectMethods`	`removeMethod`
`canObjectDo`	`inflate`	`removeMethods`
`comparable`	`isAKindOf`	`store`
`deflate`	`ivNames`	`traverse`
`deleteFromStore`	`ivTypes`	
`eq`	`load`	

Setters and Getters for Instance Variables

Every instance variable has a getter method (for getting its value), and those that are read-write also have a setter method (for setting its value). These methods are named *ivname*`Getter` and *ivname*`Setter`, and are not explicitly documented in this manual. For example, if an object named `myBox` has an instance variable named `width`, you can access it using instance variables:

```
myBox.width := value            -- for setting the value
myBox.width                     -- for getting the value
```

You can also use the equivalent methods:

```
widthSetter myBox value         -- for setting the value
widthGetter myBox               -- for getting the value
```

Both styles have identical access times—neither one is more direct or faster than the other. You can choose one style or the other based on which style you prefer. Kaleida has adopted the style of accessing instance variables using the instance variable style rather than the method style.

Note – In this manual we explicitly document instance variables, but not their setters or getters. You can presume that any instance variable has an "undocumented" getter and setter (if writable) that you are free to use. Their names are the instance variable name with `Getter` and `Setter` added as a suffix, respectively.

When you want an instance variable to behave differently, such as to also perform a side effect, you can override the setter and getter methods in any subclass of a core class. In the example above, you could override `widthSetter` or `widthGetter`. You might override `widthSetter` to test the supplied width value to ensure it does not exceed the screen width before accepting the value. (You should never modify any method in a core class— create a subclass and then specialize the method there.)

Important – for Current Release – For the current release, there is no direct access to an instance variable slot. A "slot" is a particular location in memory where the value of the instance variable is stored. Some instance variables store values in slots while others do not. For example, `valueEqualComparator` is an instance value with no slot. Instance variables without slots (virtual instance variables) call functions to set or get their values from the environment. The instance variable `presentedBy` is an example of a variable with a slot to hold its value (a real instance variable).

When you create a subclass, instance variables that are identified in this manual as read-only can be made read-write by including a setter method for them. For example, the `includesLower` instance variable is read-only for the immutable class `NumberRange`, but but if you create a mutable subclass of `NumberRange`, you can implement an `includesLowerSetter` to make it read-write.

In the current version, ScriptX does not always enforce read-only. Although you can actually change a variable designated as read-only, doing so may cause errors.

Positional and Keyword Arguments in Methods

ScriptX methods can have two kinds of arguments: "positional" arguments and "keyword" arguments. Most core class methods have only positional arguments; the few methods that have keyword arguments include `new`, `init`, `initAfter`, `getOneStream` and `makeOneStream`.

- Positional arguments must be supplied in the specified order, and all arguments must be supplied or an exception is reported.

  For example, the `append` method requires the first argument be the sequence object, and the second argument be the value to be appended:

  ```
  append myArray 10
  ```

- Keyword arguments can be supplied in any order; required arguments must be included, but optional arguments can be omitted.

  For example, the `new` method on `Array` has two optional keywords: `initialSize` and `growable`. When creating an array, you can specify these in either order, or omit them to accept the default. The following are all acceptable:

  ```
  new Array initialSize:100 growable:true
  new Array growable:true initialSize:100
  new Array growable:true
  new Array
  ```

Note that the first argument of every method is a positional argument (as is `myArray` in the previous example).

Creating and Initializing a New Instance

Each class description given in this manual describes how to create (instantiate) and initialize instances of that class under the heading "Creating and Initializing a New Instance." Concrete classes can be instantiated; abstract classes cannot.

Creating and initializing are related operations—the act of creating an instance automatically initializes it. Specifically, the `new` method, when called on a class, creates an instance of that class, then calls `init` to initialize the instance, and then calls `afterInit`.

Creating an Instance of a Class

The method for creating an instance is the class method `new`. To create an instance of any concrete class:

Call the `new` method specifying the class name as the first parameter, followed by the keyword arguments from the `init` and `afterInit` methods (described next). You must include the required keywords; include optional keywords at your discretion. You can enter the keyword arguments in any order. (Note that few core classes have an implementation for the `afterInit` method.)

The `new` method creates an empty instance of the class, and then it automatically initializes the instance by calling the class's `init` and `afterInit` methods with the keywords you supplied. The `new` method returns the initialized instance.

For example, the following script creates a new instance of the `Bitmap` class:

```
myPoint := new Point x:10 y:20
```

The resulting variable, called `myPoint`, contains the initialized point instance. Its `x` instance variable is set to 10, and `y` instance variable is set to 20.

The Creation and Initialization Syntax

As stated previously, the `new` method passes its arguments along for `init` and `afterInit` to use. Since most classes have no `afterInit` implementation, the syntax for the `new` and `init` methods for most classes is identical. As an example, the syntax for the `init` method of `Point` is as follows (where the square brackets indicate that the keyword argument is optional):

init

`init` *self* [`x:`*number*] [`y:`*number*] ⇨ *(none)*

self	`Point` instance
`x:`	`Number` object representing x value in pixels
`y:`	`Number` object representing y value in pixels

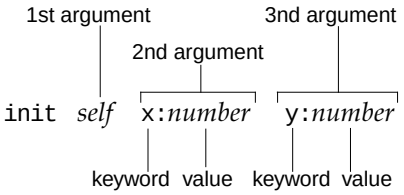
Initializes the `Point` object *self*, applying the values supplied with the keywords to the instance variables of the same names, as follows: `x` sets the x-coordinate and `y` sets the y-coordinate.

If you omit an optional keyword, its default value is used. The defaults are:

```
x:0
y:0
```

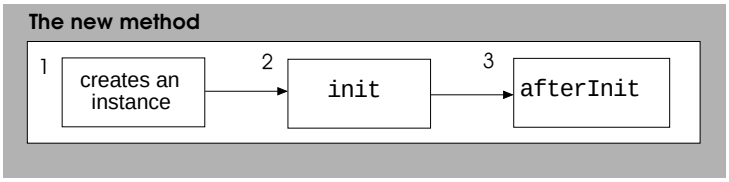
Notice that required keywords don't have default values, since the developer must always provide values.

Also notice that `init` has two different kinds of arguments—those with and without keywords. The first argument *self* has no keyword, whereas the keyword arguments are each of the form `keyword:`*value*, as shown in the following diagram:



How the new Method Works

When you call the `new` method on a class, the method creates an instance of that class, then calls `init` and `afterInit` on that instance, as shown in the following diagram:



As shown in the diagram, the `new` method has three parts:

1. Creates an instance, which allocates the appropriate amount of memory for the specified class.
2. Calls the `init` method, which initializes the variables for the instance and calls `init` on any superclasses.

3. Calls the `afterInit` method, which performs some post-initialization work, such as building internal structures dependent on settings derived from `init`, or assigning further user-supplied values to the initialized instance. This method is not implemented for most core classes; it is mainly available to be specialized in subclasses.

Notice that `new` is a class method while `init` and `afterInit` are instance methods. The `new` method is a class method defined in the `Behavior` class, and takes any number of arguments. The arguments are usually ignored when allocating memory, but are passed along for use in the `init` and `afterInit` methods.

The `init` and `afterInit` methods, on the other hand, are implemented individually in each class. When `init` is called on any instance, it initializes the instance *self*, applying the arguments to instance variables or internal states. Keywords are passed along to superclasses, as appropriate, up the inheritance tree, to ensure the inherited portions of the instance are properly initialized.

Notes About Initialization

The method for initializing an instance is `init` and the method for performing post-initialization is `afterInit`. There is no need for you to *explicitly* initialize an instance, since that happens automatically when you call `new` to create the instance. In fact, if you were to call `init` on an instance, the instance is *not* guaranteed to be initialized. There is no way to re-initialize an instance—the only way to get an initialized instance is to create a new instance.

Note – While it 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.

Since abstract classes cannot be instantiated, the `new` method is not documented for abstract classes. However, the `init` method is documented for many abstract classes because it plays an important part in the initialization of instances of concrete subclasses. That is, when you subclass an abstract (or concrete) class, the `init` method of that subclass must call its superclass's `init` method in order to instantiate the subclass properly.

Overriding the Initialization in a Subclass

Note – Do not change any method directly in any of the core classes, including the `new`, `init` or `afterInit` methods—doing so would override important internal initialization. You should create a subclass and then override the method in that subclass. You can, however, add new methods to a core class that do not override existing methods.

As stated previously, the `new` method is responsible for allocating memory required for the object, while the `init` and `afterInit` methods assign initial values to the newly created instance. You can override any of these methods in a subclass of a core class.

You would override the `new` method in some very special cases where there must be a limit to the number of instances created. For example, in a system that has only one hardware mouse, when two different concurrently-running ScriptX titles ask for a mouse, they should both get the same mouse device. When the second title calls `new`, the method should not create a new `MouseDevice` object, but should return the device already created.

You would override `init` to change how values are assigned to the created instance.

If you override the `new`, `init`, or `afterInit` method in a class, be sure to invoke its superclass's `init` (or `afterInit`) method using the `nextMethod` call. This allows the superclasses to properly initialize the instance as needed. For example:

```
method init self #rest args ->
(
    -- some specialization
    apply nextMethod self args
    -- some more specialization
)
```

For more detailed information, refer to the description of the `init` method in the *ScriptX Language Guide*.

Protocols

As used in ScriptX, the term "protocol" has a general, conceptual meaning and also has a specific implementation. The following is a description of these two senses.

Conceptual Definition

A protocol is an interface description for an object. It is most generally defined as a set of generic functions, including the setter/getter generic functions for instance variables. The purpose of any protocol is to generalize and standardize a set of behaviors for particular objects so that other objects will know how to interface to them.

For example, the Player protocol is the set of generic functions that are implemented by corresponding methods in the `Player` class. Objects that want to respond to the Player protocol can implement the Player protocol. Objects that want to use a player can call the generic functions belonging to the Player protocol.

In its simplest form, a protocol involves two objects: a "caller" and an "implementer". The protocol is the interface that the implementer makes available to the caller; it is an agreement for the caller to interface to the implementer in a certain, prescribed way. The caller agrees to call only generic functions belonging to the protocol, and the implementer agrees to provide an implementation of the methods for those generic functions.

The implementer must provide the implementation (that is, a method) for each generic function in the set defined by the protocol. It can do so either directly by method definition in its instance or class, or indirectly by inheriting an implementation from a superclass. These methods allow the implementer object to respond to the protocol.

Different families of classes can independently implement the same protocol. In other words, it is not necessary that two implementers that respond to the same particular protocol share any classes in common. The only requirement is that if the caller invokes a generic function that is within the protocol definition, that the implementer have a corresponding method. Independent implementations of protocols makes the generic functions more polymorphic and general-purpose.

In the following figure, the implementer (object B) provides a protocol that the caller (object A) uses. For example, if object B implements the Player protocol, which includes the generic function `play`, then object A can call the `play` generic function on object B, invoking its `play` method and causing it to begin playing.

From the standpoint of object A, the caller, when A calls a generic function on object B, object B can implement the corresponding method either by inheriting the implementation from the `Player` class, or by instead directly defining the method. It is important only that the protocol is somehow implemented by object B.

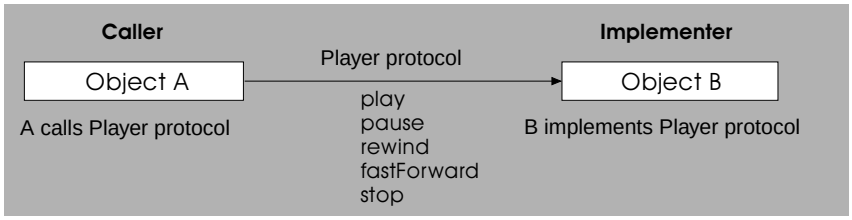


Figure 1-1: Object A calls the Player protocol on object B. Object B implements and responds to the Player protocol.

Here are some of the basic protocols in ScriptX:

- The Space protocol – for containment of objects
- The TwoDPresenter protocol – for drawing objects to the screen
- The Player protocol – for media playback
- The Clock protocol – for real-time control
- The Thread protocol – for execution control
- The Collection protocol – for data management

In ScriptX version 1.0 we do not provide an explicit representation of a protocol separate from that of a class. Implicitly, every class represents a protocol defined by the set of methods that the class and its superclasses implement. In a later version of ScriptX, we plan to explicitly represent protocols and be able to name them, test for conformance, compose over them and so on.

Implementation of Protocols

In version 1.0 of ScriptX, the space and controller families of classes are the only classes that use protocols by name and do protocol checking—in this case, every class represents a protocol. Both the `Space` and `Controller` classes define a `protocols` instance variable, which is a list of classes. This list of classes forms the necessary protocols for objects added to the space or controller. You can add or remove classes from this list to raise or lower the admissions requirements.

Saying a class is a protocol is a shorthand way of saying the generic functions of that class (including setters/getters for instance variables) define the protocol. Indirectly, the `protocols` list defines the generic functions that objects in the space or controller are capable of responding to—hence, you can safely call those generic functions on objects in the space or controller.

In the case of a space, before allowing an object into the space, the space checks to see if the object implements all the protocols. It does this by iteratively calling `isAKindOf` on the object and each class in the protocols list. The object is added only if every test returns `true`.

The current means of protocol-checking in spaces and controllers is by testing against the list of classes. However, if you need to perform your own way of protocol-checking, you can implement it yourself by checking against a list of generic functions. If you need to do protocol-checking, you can implement your own way of doing it.

Some of the core classes have a default protocols list. For example, an instance of the `TwoDSpace` class has the `TwoDPresenter` class in its protocols list. This means only instances of `TwoDPresenter` or its subclasses may enter a 2D space.

Likewise, the Bounce class is a controller that has TwoDPresenter and Projectile in its protocols list, as shown in the following diagram. Thus, an instance of Bounce can control only objects that are both a kind of TwoDPresenter and Projectile.

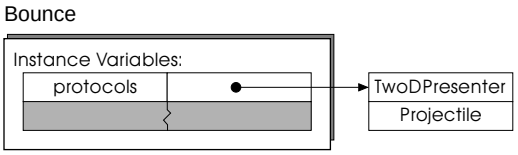


Figure 1-2: The Bounce class has two built-in protocols: TwoDPresenter and Projectile.

C H A P T E R

Global Functions

2

This chapter lists global functions that are defined by ScriptX. These functions are available in any module that uses the ScriptX module. Some of these functions are associated closely with one component and the classes that component defines. The component reference at the end of each description indicates which chapter in the *ScriptX Components Guide* to refer to for more information.

Global Functions

addManyValues (global function)

`addManyValues` *collection value1 value2 ...* ⇨ `Collection`

collection	`Collection` object
value1	Any object
value2	Any object
...	Any object

Adds to *collection* each of the values as if by "`add` *self* `empty` *value*" in the given order (notice the key is passed in as `empty`). The newly extended collection is returned.

addressOf (global function)

`addressOf` *object* ⇨ `Integer`

object	Any object

Returns the address in memory of the given *object*. This function is used in debugging tools. *(Memory Management component)*

appendReturningSelf (global function)

`appendReturningSelf` *self value* ⇨ Sequence

self	Sequence object
value	Any object

Appends the given *value* to the sequence *self*, returning the sequence. Similar to the generic function `append`, which it calls implicitly, `appendReturningSelf` is used internally by ScriptX language constructs that append values to a sequence. *(Collection component)*

callInThread (global function)

`callInThread` *func arg priority* ⇨ `Thread`

func	`Function` object
arg	Any object, used as an argument to the function
priority	`NameClass` object, either `@system` or `@user`

Creates and returns a new `Thread` object with the given *priority*. This thread calls the function *func* with the argument *arg* when it runs. It applies default values for the other parameters. This thread is immediately active (runnable), but it won't actually run until its turn, as determined by the scheduler. A high priority thread is guaranteed to run at least once before the scheduler allows any before normal priority threads to run again.

The global function `callInThread` is an alternative to calling `new` on the `Thread` class. You can assign any thread properties, such as `label` and `preemptibility`, to a thread created with this function. For more information on creating a thread, see the class definition for `Thread`.

This global function gets it name `callInThread` because it calls a function "in a thread" as opposed to the way functions are normally called.

All three arguments are required. If the thread function is to be called with no argument, use `undefined` as a value for *arg*. Threads have 2 priority levels, `@system` and `@user`; system priority is intended only for critical tasks that run for a short time without blocking. *(Threads component)*

canRequestPurge (global function)

`canRequestPurge` *object* ⇨ `Boolean`

object	Any object

Returns `true` if *object* has been added to a storage container. *(Title Management component)*

closeMidiDriver (global function)

`closeMidiDriver` *midiDriver* ⇨ `OK`

This function closes a MIDI driver, which is specified as a pair whose first element is a string of the driver's name, and the second element is a name such as `@internal` or `@external`.

cmp (global function)

`cmp` *x y* ⇨ `NameClass`

Compares the objects *x* and *y*, returning either `@same`, `@before`, or `@after`. *(Object System Kernel)*

The `cmp` function is equivalent to the following:

```
function cmp x y ->
    if (isComparable x y) then (
        if (localEqual x ) then
                @same
            else if (localLT x y) then
                @before
            else
                @after
        )
    else -- report an exception
        report unordered (#(x, y) as Pair)
```

coerce (global function)

`coerce` *objectOrClass targetClass* ⇨ *(object)*

objectOrClass	Any object or class
targetClass	Any class

Returns a copy of *objectOrClass* coerced to an instance of the class *targetClass*. The global function `coerce` is the basis for the ScriptX language construction `as`. Coerce is defined to call `morph` and `newFrom`, two generics that implement the Coercion protocol.*(Object System Kernel)*

The `coerce` is equivalent to the following:

```
function coerce x y ->
    guard
        morph(x, y, @normal)
    catching
        cantCoerce : (newFrom (classOf x) y)
    end
```

For more information, see the discussion of coercion in the "Object System Kernel" chapter of *ScriptX Components Guide*.

currentModule (global function)

`currentModule()` ⇨ `ModuleClass`

Returns a `ModuleClass` object, the module in which the program is currently running. See also the global function `getModule`. *(Modules)*

defaultDeflate (global function)

`defaultDeflate` *object class stream* ⇨ *(object)*

object	Object to be deflated by default mechanism
class	Class of object to be deflated
stream	A storage stream containing the object reference

Causes *object* to be deflated by its default mechanism. *(Title Management component)*

defaultInflate (global function)

`defaultInflate` *object class stream* ⇨ *(object)*

object	Object to be inflated by default mechanism
class	Class of object to be inflated
stream	A storage stream containing the object reference

Causes *object* to be inflated by its default mechanism. *(Title Management component)*

deflateSubObjectReference (global function)

`deflateSubObjectReference` *theSubObject stream* ⇨ *(object)*

theSubObject	An object to be deflated by default mechanism
stream	A storage stream containing the subobject reference

Call this function within a specialized `deflate` method to perform default deflation on any subobjects such as instance variables. *(Title Management component)*

deInstallQuitQuery (global function)

`deInstallQuitQuery` *queryID* ⇨ `Function`

queryID	`Integer` object

Removes the quit query with ID number *queryID* from the list of quit queries, returning the function that was maintained as a quit query. This query ID, the result of `installQuitQuery`, should be stored in a program that may later need to remove a quit query.

A quit query is a ScriptX function that returns `true` or `false`. If the query returns `true`, execution of quit queries continues until all queries return `true` or until one turns `false`. For more information, see the "Title Management" chapter of the *ScriptX Components Guide*. *(Title Management component)*

deInstallQuitTask (global function)

`deInstallQuitTask` *taskID* ⇨ `Function`

taskID `Integer` object

Removes the quit task with ID number *taskID* from the list of quit tasks, returning the function that was maintained as a quit task. This task ID, the result of `installQuitTask`, should be stored if a program may need to remove a quit task later in its execution.

A quit task is a ScriptX function that has no return value. Once the quit queries have executed successfully, All quit tasks are guaranteed to run. For more information, see the "Title Management" chapter of the *ScriptX Components Guide*. *(Title Management component)*

deleteModule (global function)

`deleteModule` *moduleName* ⇨ *(none)*

The global function `deleteModule` can be called with either the name of the module (a `NameClass` object) 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 as well.

disassemble (global function)

`disassemble` *funcObj* ⇨ `Boolean`

funcObj Scripted `Function` or `ByteCodeMethod` to disassemble

See the "Title Analysis API" chapter in the *ScriptX Tools Guide*. *(Tools component)*

enableHeapGrowth (global function)

`enableHeapGrowth` *option* ⇨ *option*

option `Boolean` object

If *option* is `true`, the heap is allowed to grow in small increments as memory is required. If *option* is `false`, then the system calls `garbageCollect` before heap growth is allowed. In either case, the function returns the value of *option* passed in. Note that setting *option* to `false` does not prevent growth of the heap; it only forces the system to recover unused memory before trying to grow the heap. This function is considered part of the ScriptX API and can be used in ScriptX titles and code libraries. *(Memory Management component)*

eq (global function)

`eq` *x y* ⇨ `Boolean`

Returns `true` if *x* and *y* are exactly the same object. This function is the same as using the double-equal operator (==). In ScriptX 1.0, `eq` was implemented as a generic function.

For `ImmediateInteger` and `ImmediateFloat` objects, `eq` returns `true` if two objects have the same value, even though they may be different objects. For more information, see the discussion of immediate objects in the "Numerics" chapter of *ScriptX Components Guide*. *(Object System Kernel)*

equal (global function)

`equal` *x y* ⇨ `Boolean`

Compares the objects *x* and *y*, returning `true` if they are comparable and have the same values. This function is the same as using the single equal operator (=). Thus, the follow two expressions are equivalent:

```
equal a b
a = b
```

The `equal` function is also equivalent to the following definition:

```
function equal x y -> (isComparable x y) and (localEqual x y)
```

The generic functions `isComparable` and `localEqual` are defined by `RootObject` and implemented by many classes in the system. *(Object System Kernel)*

eventCriticalDown (global function)

`eventCriticalDown()` ⇨ *(none)*

Allows the event system to resume processing user input. This function is always paired with a prior call to `eventCriticalUp`, which suspends processing of keyboard and mouse events. A call to `eventCriticalDown` balances a call to `eventCriticalUp`, enabling the system to resume processing of these events. See `eventCriticalUp`, a global function defined below. *(Events component)*

eventCriticalUp (global function)

`eventCriticalUp()` ⇨ *(none)*

Suspends processing of keyboard and mouse events until the next call to `eventCriticalDown`. A call to `eventCriticalUp` precedes a critical segment of a code in which the system discards mouse and keyboard events. Of course, `eventCriticalUp` affects any class that maintains an interest in mouse and keyboard events, including `Menu`, `ScrollBar`, and `TextEdit`.

Within an event-critical segment, input devices continue to receive events from the underlying operating system, but they "swallow" these events rather than convert them into ScriptX events. An event-critical segment must be followed by a call to `eventCriticalDown`, which allows the event system to resume processing of these events. These two functions are designed so that a program can insure that the system finishes responding to a keyboard or mouse event before the next event is received.

The paired functions `eventCriticalUp` and `eventCriticalDown` are analgous to `threadCriticalUp` and `threadCriticalDown`. The system maintains a count of calls to `eventCriticalUp`, and each call must be matched by a corrresponding call to `eventCriticalDown` before the system resumes processing user input. If the system is likely to be in an event-critical state for a long time, set the value of `pointerType`, an instance variable defined by `MouseDevice`, to `@wait`.

Use `eventCriticalUp` and `eventCriticalDown` cautiously and sparingly. An unbalanced call to `eventCriticalUp` can leave the system in a suspended state, from which the user may be unable to regain control of the system. The keyboard combinations command-period (MacOS) and control-break (Windows and OS/2) allow the user to escape from an event-critical state. *(Events component)*

Note – This function affects the entire runtime or authoring environment, including any other title or tool that is currently open. If a program requires user input, it must insure that event processing is restored.

---

findNthContext (global function)

`findNthContext` *args context* ⇨ `Boolean`

args is an array with these elements: *#(string, n, startOffset, endOffset)* `as Quad`

string	`String` object (*string* to select from)
n	`Integer` object (*n*th item)
startOffset	`Integer` object (cursor position to start searching in *string*)
endOffset	`Integer` object (cursor position to end searching in *string*)

context one of (`@word`/`@sentence`/`@paragraph`) or a supplied delimiter function

If the search is successful, returns `true` and sets *startOffset* and *endOffset* to be the range of the *context* found; otherwise, returns `false` and sets *startOffset* and *endOffset* to –1. (It is not required that *args* be coerced to a `Quad`, but it is more efficient to do so.)

The delimiters used by `findNthContext` to determine where a context begins and ends are listed below:

`@word`

– anything that is not alphanumeric (“high-order” or “text_Matrix” would be considered two words)

`@sentence`

– period (“.”), question mark (“?”), and exclamation mark (“!”)

– new line character (“\n”)

`@paragraph`

– new line character (“\n”)

In addition, the end of a string will serve as the end delimiter for any context.

You may specify your own delimiter by supplying an anonymous function indicating what the delimiter is to be. The argument to this function is the element to be used as the delimiter. It must be given as a Unicode character, which means that it must be in integer form. Below are two ways of specifying `x` as the delimiter:

```
findNthContext args (r -> r == "x"[1]) --”x”[1] is 120
findNthContext args (r -> r == 120)
```

If the context is `@sentence` or `@paragraph`, or if you define your own delimiter, the delimiter is included as part of the text in the range defined by the third and fourth elements of `args`. This is so because sometimes you want the delimiter included, as in the case of a period to end a sentence. If you do not want the delimiter included, however, you need to remove it yourself. (Example 3 demonstrates removing a delimiter from text.) Delimiters are not included if the context is `@word`.

Example 1:

To find the second word in the `StringConstant` object `s`, starting the search at cursor position 0 and ending the search at curosr position 999, enter the following code:

```
global s := "a quick brown fox. First there was the word"
global args := #(s, 2, 0, 999) as Quad
findNthContext args @word
```

The function returns `true`, and `args` becomes `#("a quick brown fox. First there was the word", 2, 2, 7)`.

Example 2:

To find the first sentence in `s` starting at cursor position 0 and ending at the end of the string, enter the following code:

```
global args := #(s, 1, 0, s.size) as Quad
findNthContext args @sentence
```

The function returns `true`, and `args` becomes `#("a quick brown fox. First there was the word", 1, 0, 18)`.

Example 3:

The following code uses the letter "A" as the delimiter. Note that the code sample is enclosed in parentheses so that variables can be local.

```
(
local t := "oneAtwoAthreeAfourAfive"
--assigne t.size to a variable for efficiency
local lengthOfT := t.size
local args := #(t, 1, 0, lengthOfT) as Quad
repeat while (findNthContext args (r -> r == "A"[1])) do
    (
    print ("(" + (args[3] as String) + ", " + (args[4] as String) + \
        "):" + "'" + (copyFromTo t args[3] args[4]) + "'")
    args[2] := args[2] + 1 --reset args for next search
    args[3] := 0
    args[4] := lengthOfT
    )
)
```

Here is the output from this script:

```
"oneAtwoAthreeAfourAfive"
#("oneAtwoAthreeAfourAfive", 1, 0, 23) as Quad
1
"(0, 4):'oneA'"
"(4, 8):'twoA'"
"(8, 14):'threeA'"
"(14, 19):'fourA'"
"(19, 23):'five'"
OK
```

The following code demonstrates first deleting the delimiter, and then replacing the delimiter with a space:

```
global t := t as String --coerce t to a String so you can modify it
global myCopyT := copy t
deleteAll myCopyT "A"[1] --the value of "A"[1] is 65
⇨  4
myCopyT
⇨  "onetwothreefourfive"

--instead of just deleting the delimiter, the following code replaces
--the delimiter with a space
```

```
for i in 1 to t.size do
   if t[i] = "A"[1] do t[i] := 32 --32 is the Unicode value for " "
⇨  undefined
t
⇨  "one two three four five"
```

findParents (global function)

`findParents` *address maxLevel* ⇨ *(none)*

address	Address of object to find the parents of
maxLevel	`Integer` indicating how far up the reference paths to go

See the "Title Analysis API" chapter in the *ScriptX Tools Guide*. *(Tools component)*

format (global function)

`format` *stream formatString objectToPrint arg* ⇨ *(object)*

stream	`ByteStream` object that is writable
formatString	`String` object
objectToPrint	An object to print
arg	`NameClass` object indicating style

Allows you to specify which item in an object to print, where *stream* is a writable byte stream, typically the `debug` stream, to which the function will print, and *formatString* is a `String` object. If *formatString* contains `%n` or `%n%`, where `n` is a nonnegative integer, then the function is equivalent to:

```
prin (getNth (objectToPrint, n)) (getNth (printArg, n)) self
```

If *formatString* contains `%*`, then the function is equivalent to

```
prin (objectToPrint, printArg, stream)
```

The target *objectToPrint* is typically an object, such as an instance of `Array`, that has several elements. Use `format` to selectively print one of those elements.

The final argument, *arg*, represents a ScriptX printing style such as `@normal` or `@debug`. For more examples of `format`, and for an explanation of printing styles, see "Formatted Output" in the *ScriptX Language Guide*. *(Streams component)*

fps (global function)

`fps()` ⇨ `Float`

See the "Title Analysis API" chapter in the *ScriptX Tools Guide*. *(Tools component)*

garbageCollect (global function)

`garbageCollect()` ⇨ `Integer`

Runs the garbage collector to completion (without yielding) from the present point in its execution cycle. Returns the number of complete cycles since ScriptX began running. This function could be used to clear all available memory, perhaps in preparation for a scene transition. *(Memory Manager component)*

gateIsOpen (global function)

`gateIsOpen` *gate* ⇨ `Boolean`

gate	`Gate` object

Returns `true` if the given *gate* is currently open. This function never returns `true` on `Condition` objects, since they are only open instantaneously. *(Threads component)*

gateOpen (global function)

`gateOpen` *gate* ⇨ *(none)*

gate	`Gate` object

Opens the given *gate*. Note the restrictions placed on particular gates. For example, `Lock` objects may be opened only by the thread that successfully passed through or waited on them. *(Threads component)*

gateWait (global function)

`gateWait` *gate* ⇨ *(none)*

gate	`Gate` object

Causes the current thread to block, if the given *gate* is not open, and the thread waits for the gate to open. If the thread succeeds in acquiring the gate, execution of the process continues. If the thread does not succeed in acquiring the gate, it blocks. When a thread blocks, the value of `status` is `@waiting`. *(Threads component)*

gateWaitAfterOpening (global function)

`gateWaitAfterOpening` *waitOnMe openMe* ⇨ *(none)*

waitOnMe	`Gate` object
openMe	`Gate` object

Opens the gate *openMe* and then waits for the gate *waitOnMe* to open. The open and wait are performed atomically with respect to the thread system, allowing you to prevent race conditions in typical lock/condition pairs. *(Threads component)*

ge (global function)

`ge` *x y* ⇨ `Boolean`

Compares the objects *x* and *y*, returning `true` if *x* is greater than or equal to *y*. *(Object System Kernel)*

The `ge` function is equivalent to the following:

```
function ge x y -> not (lt x y)
```

getMidiDriverList (global function)

`getMidiDriverList()` ⇨`Array`

This function returns a list of pairs, where each pair corresponds to an installed MIDI driver. In each pair, the first element is a string of the driver's name, and the second element is a name such as `@internal` or `@external`.

getModule (global function)

`getModule` *moduleName* ⇨ `ModuleClass`

moduleName	`NameClass` object representing a module name

Returns a `ModuleClass` object with the name *moduleName*, or `false` if no such module exists. This name does not have to be interned. The function returns a module, which can be coerced to an array to see its contents. If no such module exists, it returns `false`. See also the global function `currentModule`. *(Modules)*

The following example uses coercion and the pipe operation to print a sorted list of names defined in the substrate module, the definition module for the ScriptX core classes. (Since this example takes a long time to run, you might run it in a separate thread by opening a new listener window from the **File** menu.)

```
global myModule := getModule @substrate
(myModule as SortedArray) | print
```

getPrinterNameList (global function)

`getPrinterNameList()` ⇨ `Array`

Returns an array of `String` objects that represent the names of printer devices available. These names can be used to create `PrinterSurface` and `PrinterSpace` instances. On some platforms, printer selection is carried out outside the scope of an application and only the currently chosen printer is listed. *(Printing component)*

getStorageContainer (global function)

`getStorageContainer` *object* ⇨ `StorageContainer`

object `RootObject` object

Returns the `StorageContainer` in which *object* resides. *(Title Management component)*

getStorageCacheSize (global function)

`getStorageCacheSize()` ⇨ `ImmediateInteger`

Prints the number of 4K blocks currently being used for storage cache. See also `setStorageCacheSize` below. *(Title Management component)*

getThreadList (global function)

`getThreadList` *desiredStatus* ⇨ Sequence

desiredStatus `NameClass` object

Returns a list of currently existing threads with the status *desiredStatus*. Refer to the instance variable `status`, defined by the `Thread` class, for a list of available status names. If *desiredStatus* is passed as `undefined`, this method returns a list of all threads. For technical reasons, threads whose status is `@done` or `@killed` are not tracked. *(Threads component)*

growHeap (global function)

`growHeap` *byteCount* ⇨ `Integer`

byteCount `Integer` object

Attempts to grow the heap by *byteCount* bytes. Normally the heap grows in small increments as required, but you might be able to make your title run more efficiently if you know in advance by how much you need to grow the heap. Returns the number of bytes actually added, which may be larger than `byteCount`, since the heap grows in fixed increments. This function is considered part of the ScriptX API and can be used in ScriptX titles and code libraries. *(Memory Management component)*

gt (global function)

`gt` *x y* ⇨ `Boolean`

Compares the objects *x* and *y*, returning `true` if *x* is greater than *y*. *(Object System Kernel)*

The `gt` function is equivalent to the following:

```
function gt x y -> not (le x y)
```

ignoreRefreshRegion (global function)

`ignoreRefreshRegion` *compositor boolean* ⇨ `Boolean`

compositor	`TwoDCompositor` object
boolean	`Boolean` object, indicating desired state

Tells the `TwoDCompositor` object *compositor* whether or not to ignore requests to refresh the screen. If the value of the second argument is `true`, it ignores requests. Otherwise, it restores the compositor to normal operation. Although the compositor has no visible state variable, it has an implicit state. At any given time, it is either ignoring refresh operations (`true`) or not ignoring them (`false`).

The return value indicates the previous "state" of the compositor—whether it was previously ignoring requests to refresh. Typically, you call `ignoreRefreshRegion` to turn refreshing off, saving the state to a variable. Then, when you no longer want to ignore refreshing, call `ignoreRefreshRegion` again, passing the saved state as the second argument to restore the compositor to its original state. This allows calls to `ignoreRefreshRegion` to be nested, as might happen in recursive structures.

The following code demonstrates a standard framework for using `ignoreRefreshRegion` to prevent unnecessary redrawing of a presenter:

```
global saveState
myWindow.compositor.enabled := false
notifyChanged myPresenter false
saveState := ignoreRefreshRegion myPresenter true -- start ignoring
-- do some things to the presenter myPresenter here
ignoreRefreshRegion myPresenter saveState -- restore compositor
notifyChanged myPresenter false
myWindow.compositor.enabled := true
```

Compare `ignoreRefreshRegion` with simply disabling the compositor. When you disable the compositor, calls to `notifyChanged` still accumulate, but changes are not drawn to the current display surface. When you enable the compositor, the display surface is updated to the state it would have been in if the compositor had never been disabled. (The generic function `notifyChanged` is defined by `TwoDPresenter` and specialized by many of its subclasses.)

When you call `ignoreRefreshRegion`, calls to `notifyChanged` are ignored. The compositor "stops listening" until refreshing is restored. Changes to the display surface do not accumulate. *(Spaces and Presenters component)*

Calls to `ignoreRefreshRegion` should always be paired, and they should be used only for very brief periods of time. Since `ignoreRefreshRegion` suspends the thread system by calling `threadCriticalUp` and `threadCriticalDown`, it should be used only briefly, so as not to suspend essential system services such as garbage collection and callbacks.

---

inflateSubObjectReference (global function)

`inflateSubObjectReference` *stream* ⇨ *(object)*

stream	A storage stream containing the subobject reference

Call this function with a specialized `inflate` method to perform default inflation on any subobjects such as instance variables. *(Title Management component)*

initialSearchContext (global function)

`initialSearchContext` *strIndex startOffset* ⇨ `SearchContext`

strIndex	`StringIndex` object.
startOffset	`Integer` object

Returns an instance of `SearchContext` which can be used as the third argument to `searchIndex` to start a search of the string associated with *strIndex* at the ordinal position designated by *startOffset*.

This function is used only in connection with the global function `searchIndex`, which is in turn used only in connection with the classes `StringIndex` and `SearchContext`. See the entries for those classes for a more complete definition and code examples.

installQuitQuery (global function)

`installQuitQuery` *func arg* ⇨ `Integer`

func	`Function` object
arg	Any object

Adds the quit query, with the function *func* and the argument *arg*, to the list of quit queries maintained by the Kaleida Media Player, returning a unique integer as an ID. This ID can be used later to remove a quit task, by calling `deInstallQuitQuery`. When `quit` is called, the function *func* is called with *arg* as its only argument:

func arg

A quit query is a ScriptX function that should be written to return `true` or `false`. If a query returns `true`, execution continue with the next quit query; if all quit queries return `true`, the quit tasks are then executed. If a quit query returns `false`, the quit process is immediately aborted and no more quit queries or quit tasks are executed. For more information, see the "Title Management" chapter of the *ScriptX Components Guide*. *(Title Management component)*

installQuitTask (global function)

`installQuitTask` *func arg* ⇨ `Integer`

func	`Function` object
arg	Any object

Adds the quit task, with the function *func* and the argument *arg*, to the list of quit tasks maintained by the Kaleida Media Player, returning a unique integer as its ID. This ID can be used later to remove a quit task, by calling `deInstallQuitTask`. When `quit` is called, after all quit queries have returned `true`, the quit tasks are run. Each quit task calls its function *func* with *arg* as its only argument:

func arg

You should write this function to perform a quit task, such as releasing an operating system resource. The return value from this function is ignored. For more information, see "Quitting ScriptX" in the "Title Management" chapter of the *ScriptX Components Guide*. *(Title Management component)*

isDefined (macro)

`isDefined` *variable* ⇨ `Boolean`

variable	a valid ScriptX lexical name

Indicates whether the given name has a global name binding in the current module. Returns `false` if no binding exists, or if one exists but has the value `undefined`. Returns `true` if a global binding exists, and if its value cell points to an object other than the `undefined` system object. Since `isDefined` only checks global names (in the current module), it cannot tell when a local binding is overriding a global binding. In the following code example, `isDefined` returns `false`.

```
global foo := undefined
if foo != true do (
    local foo := 3
    if foo = 3 do print "I know about foo. Do you too?"
    if isDefined foo then format debug "My foo is %*\n" foo @normal
    else print "boo hoo, I know no foo!"
)
⇨  "I know about foo. Do you too?"
"boo hoo, I know no foo!"
OK
```

You can use `isDefined` to determine if a script is operating in the Kaleida Media Player or the ScriptX Language and Class Library by checking to see if a name binding that exists only in the ScriptX Language and Class Library, such as `fileIn`, is currently defined. Note that `isDefined` is a macro, not a function. Thus, it cannot be passed as an argument to a function or generic that expects an instance of `AbstractFunction`. *(Modules)*

isInMemory (global function)

`isInMemory` *object* ⇨ `Boolean`

object	Any object

Returns `true` if *object* currently occupies memory. Returns `false` if *object* is not in memory. *(Title Management component)*

isPurgeRequested (global function)

`isPurgeRequested` *object* ⇨ `Boolean`

object	Any object added to a storage container

Returns `true` if `requestPurge` has been called on *object*, where *object* has been added (but not necessarily saved) to a storage container. Note that `isPurgeRequested` and `isInMemory` could both be true for *object* at the same time, since *object* may not yet have been garbage collected. *(Title Management component)*

largestFreeHeapBlock (global function)

`largestFreeHeapBlock()` ⇨ `Integer`

Reports the largest free, contiguous block of memory that is available to the ScriptX runtime environment in the ScriptX heap. This function is independent of the underlying operating system. See also the global function `totalFreeHeapSpace`. *(Memory Manager component)*

largestFreeSystemBlock (global function)

`largestFreeSystemBlock()` ⇨ `Integer`

Reports the largest free, contiguous block of memory that is available to the ScriptX runtime environment outside of the ScriptX heap. See also the global function `totalFreeSystemSpace`.

- Windows allocates memory to both system and applications from the Windows global heap. The function `largestFreeSystemBlock` reports the size of the largest continuous block in the global heap that could be allocated to ScriptX. (Windows systems may run with virtual memory.)
- The Macintosh System implements separate heaps for the system and applications. It allocates a fixed, contiguous zone in main memory to each application. On the Macintosh, `largestFreeSystemBlock` reports the size of the largest contiguous block available in the zone currently allocated to ScriptX. (Macintosh systems may run with virtual memory.)
- OS/2 does not distinguish between real and virtual memory, so `largestFreeSystemBlock` reports the largest contiguous zone of real or virtual memory that is available. Since OS/2 systems generally run with virtual memory, this is typically a very large number. OS/2 is implemented with fully preemptive multitasking, so this number changes instantaneously. Under OS/2, applications run in memory that is protected and paged by the operating system, and the system is free to swap pages between real and virtual memory. OS/2 behaves as if it had virtually unlimited memory, however performance slows markedly as main memory is used up by the system and open applications, forcing applications to swap between real and virtual memory.

This function is useful in tracing the source of memory management conflicts with certain platforms, however results are dependent on the underlying operating system. It is recommended strictly as a diagnostic tool. *(Memory Manager component)*

le (global function)

`le` *x y* ⇨ `Boolean`

Compares the objects *x* and *y*, returning `true` if *x* is less than or equal to *y*. *(Object System Kernel)*

The `le` function is equivalent to the following:

```
function le x y ->
    if (isComparable x y) then
        (localLt x y) or (localEqual x y)
    else
        (report unordered (#(x, y) as Pair))
```

loadDeep (global function)

`loadDeep` *object* ⇨ *self*

object `RootObject` object

Loads the object *object* along with all subobjects of *object*, which are any objects reachable from *object* through instance variables, function arguments, or any other means. If *object* is not a persistent object (has not been added to any storage container), then this method does nothing.

Use the `loadDeep` method with caution. In some cases you may be able to get a performance gain from using `loadDeep`, for example to preload a scene, but you need to make sure you know what you are loading and make sure it is well constrained. If a subobject of a subobject of the object you called `loadDeep` on is an object that is stored in a different container, then that container will be opened. Or you might simply be loading many more objects than you need to, even if they are all in the same container. You could end up taking a performance hit instead of making a performance gain. Since objects are loaded from storage in blocks, you are almost always loading some objects you may not need, even when you are very careful about what you load. If you are less cautious about what you load, you could end up loading quite a few unneeded objects.

lockMany (global function)

`lockMany` *lock1 lock2 . . .* ⇨ *(none)*

lock1	`Lock` object
lock2	`Lock` object
. . .	

Used when a thread must acquire several locks simultaneously. This prevents "deadly embrace," a scenario where two or more threads are blocked because they are each unable to acquire a lock that another thread owns. *(Threads component)*

lockNowOrFail (global function)

`lockNowOrFail` *lock* ⇨ `Boolean`

Locks (waits successfully on) the given *lock* and returns `true`, or leaves the lock in its previous state and returns `false`. *(Threads component)*

lt (global function)

`lt` *x y* ⇨ `Boolean`

Compares the objects *x* and *y*, returning `true` if *x* is less than *y*. *(Object System Kernel)*

The `lt` function is equivalent to the following:

```
function lt x y ->
    if (isComparable x y) then
        (localLt x y)
    else
        (report unordered (#(x, y) as Pair))
```

mciCommand (global function)

`mciCommand` *cmdString* ⇨ *(none)*

cmdString	`String` representing the command

Loads the Multimedia Command Interface (MCI) command represented by the string *cmdString*.

Note – This function works only in Microsoft Windows, and resides in a loadable extension that must be explicitly loaded. To load it, select **Open Title** from the **File menu**, go to the `LOADABLE/MCICMD` directory and select the "`loadme.sx`" file.

Once the MCI command is loaded into the ScriptX runtime environment, you can compile code that mixes MCI commands with other ScriptX commands. Note that mixing MCI commands with ScriptX commands makes your code platform specific. If you want your title to run on any platform, you should conditionally execute this platform-specific code and provide alternative functionality for systems other than Windows.

The `mciCommand` function is contained in the MCI loadable extension. This function The MCI extension provides the interface to Microsoft Windows MCI commands. Loading an MCI command provides the same syntax defined for that command through the standard ScriptX interface. In other words, the commands and syntax for the MCI command can be mixed freely with ScriptX expressions in the source code for a title. The entire MCI interface is provided through the single function `mciCommand`.

For more information, such as how to load this extension, refer to the appendix "Loadable Extensions" in the *ScriptX Components Guide*. *(Loadable extensions component)*

memoryDiff (scripted global function)

`memoryDiff` *memorySnap* ⇨ *(none)*

memorySnap `RamStream` object returns from `memorySnap` function

See the "Title Analysis API" chapter in the *ScriptX Tools Guide*. *(Tools component)*

memorySnap (scripted global function)

`memorySnap()` ⇨ `RamStream`

See the "Title Analysis API" chapter in the *ScriptX Tools Guide*. *(Tools component)*

memoryUsage (global function)

`memoryUsage` *items* ⇨ `Integer`

items `Integer` object

See the "Title Analysis API" chapter in the *ScriptX Tools Guide*. *(Tools component)*

moo (scripted global function)

`moo()` ⇨ *(none)*

See the "Title Analysis API" chapter in the *ScriptX Tools Guide*. *(Tools component)*

ne (global function)

`ne` *x y* ⇨ `Boolean`

Compares the objects *x* and *y*, returning `true` if *x* is not the same object as *y*. *(Object System Kernel)*

The `ne` function is equivalent to the following:

```
function ne x y -> not (eq x y)
```

nequal (global function)

`nequal` *x y* ⇨ `Boolean`

Compares the objects *x* and *y*, returning `true` is they are not comparable or do not have the same values. *(Object System Kernel)*

The `nequal` function is equivalent to the following:

```
function nequal x y -> not (equal x y)
```

objectify (global function)

`objectify` *address* ⇨ *object*

address `Integer` object

Returns the object at a given *address* in memory. The value of *address* must be the address of a legal object. Note that `objectify` is the inverse of `addressOf`, also a global function. This function is used by debugging tools. *(Memory Management component)*

objectSize (global function)

`objectSize` *object* ⇨ `Integer`

object Any object

Reports how much memory an object is using. This amount is a shallow count of memory, and does not report how much memory is held onto by internal structures or embedded objects. Note that the amount of memory used at the top level by the array defined below does not change even as 1024 objects are added to the array.

```
global myObj := new Array
objectSize myObj -- it is empty
⇨32
for i in 1 to 1024 collect into myObj i
⇨#(1, 2, 3, 4, 5, 6, 7, 8, 9, 10, ...)
objectSize myObj -- its size is now 1024
⇨32
```

This function should be used strictly as a diagnostic tool. See also the global function `treeSize`. *(Memory Management component)*

openMidiDriver (global function)

`openMidiDriver` *midiPair* ⇨ `MIDIDriver` or `false`

This function opens a MIDI driver, which is specified as a pair whose first element is a string of the driver's name, and the second element is a name such as `@internal` or `@external`. The function returns the opened `MIDIDriver`, or `false` if the open operation was unsuccessful.

See the introduction to this chapter, or the Media Players chapter in the *ScriptX Components Guide* for more information on finding and opening MIDI drivers.

presentMessagePanel (global function)

`presentMessagePanel` *message icon button-list default-ix cancel-ix*
⇨ `ImmediateInteger`

message	`String` object, the text to display
icon	`NameClass` object, one of `@warning`, `@critical`, `@information`, or `@none`
button-list	`Array` object containing 1, 2, or 3 `String` objects to be used as button names
default-ix	`ImmediateInteger` object, the index of the button that is the default
cancel-ix	`ImmediateInteger` object, the index of the button that is to be returned when the user clicks the cancel key

Presents a native system dialog box containing the user-suppplied message specified in *message*, the icon specified in *icon*, and buttons labeled with the strings specified in *button-list*; returns the index into *button-list* of the button that was pressed. One button must be the default (the button selected if the user hits the return key), and it is specified by giving its index into *button-list*. There must also be a button designated as the one to be returned if the user hits command -. (the command key plus a period) on a Macintosh or Ctrl-Break on a Windows machine. This button can be the same as the default key, but that is not necessarily the case.

The icons displayed in the message window are machine-dependent.

Example 1

```
presentMessagePanel "Nyuk, nyuk" @critical #("Moe", "Larry",
    "Curly") 2 1
```

Example 1 presents a dialog box with the message "Nyuk, nyuk." The dialog box contains an icon indicating that the message is "critical", and it has three buttons labeled "Moe", "Larry", and "Curly". "Larry" is the default button, and "Moe" is the cancel button.

Example 2

```
presentMessagePanel "Format your disk" @warning #("Format",
   "No") 2 2
```

Example 2 presents a dialog box with the icon that indicates a “warning” message and the message “Format your disk”. It has two buttons, one labeled “Format” and one labelled “No”. “No” is both the default button and the cancel button.

presentOpenFilePanel (global function)

`presentOpenFilePanel` *typelist* ⇨ `Array` or `undefined`

typelist	`Array` object containing `NameClass` objects, which may include the following file types: `@title` - A ScriptX file containing a `TitleContainer` object `@library` - A ScriptX file containing a `LibraryContainer` object `@accessory` - A ScriptX file containing an `AccessoryContainer` object `@binary` - A file containing binary data `@text` - A file containing ASCII data `@unknown` - The file type is not specified, and any file type may be selected

Presents a platform-specific **Open** dialog box where the user can select a file; returns an array of strings representing the full path of the file selected, or `undefined` if no file is selected. The dialog box displays as options the file types specified in *typelist*. If *@unknown* or any unknown value is specified in *typelist*, the **Open** dialog box allows any file type to be selected. If the user clicks **Open** or **OK** (depending on the platform), the filename is returned; if the user clicks **Cancel**, the function returns `undefined`. In any case, it is up to the title to actually open the file.

NOTE — The global function `presentOpenFilePanel` replaces the class `OpenPanel`.

presentSaveFilePanel (global function)

`presentSaveFilePanel` *prompt defaultName* ⇨ `Array` or `undefined`

prompt	String object
defaultName	String object

Presents a platform-specific **Save As** file dialog box , where the user can type in a filename, and which contains the specified prompt and default filename. If the user clicks **OK** or **Save**, the function returns an array containing the full pathame of the file named in the file entry field. If the user clicks **Cancel**, the function returns `undefined`. In any case, it is up to the title to actually save the file. The string of text supplied for *prompt* will appear next to the file entry field as a prompt; the string of text supplied as the default file name will appear in the dialog box’s file entry field when it is first opened.

NOTE — The global function `presentSaveFilePanel` replaces the class `SavePanel`.

prin1 (global function)

`prin1` *object stream* ⇨ *(none)*

object	An object to print
stream	`ByteStream` object that is writable

Prints *object* to *stream* in the same way that the `prin` method would with `@normal` as the printing style. If *stream* isn't supplied, this function prints to the debugging stream. *(Object System Kernel)*

prinLn (global function)

`prinLn` *object arg stream* ⇨ *(none)*

object	An object to print
arg	`NameClass` object indicating printing style
stream	`ByteStream` object that is writable

Prints *object* to *stream* in the same way that the `prin` method would, and with the same arguments, but with a newline character added at the end of the printed string. The second argument, *arg* represents a ScriptX printing style such as `@normal` or `@debug`. If *stream* isn't supplied, this function prints to the debugging stream. *(Object System Kernel)*

prinString (global function)

`prinString` *object arg* ⇨ `String`

object	An object to print
arg	`NameClass` object indicating printing style

Creates and returns a new instance of `String` containing the printed representation of *object*, formatted by *arg*, where *arg* represents a ScriptX printing style such as `@normal` or `@debug`. *(Object System Kernel)*

```
str := new String
prin object arg str
return str
```

print (global function)

`print` *object stream* ⇨ *(none)*

object	An object to print
stream	`ByteStream` object that is writable

Prints *object* to *stream* in the same way as the `prin` function would if `@normal` were supplied as its printing style, but adding a newline character at the end of the printed string. If *stream* isn't supplied, this function prints to the debugging stream. *(Object System Kernel)*

printRecursively (global function)

`printRecursively` *object arg stream* ⇨ *(none)*

object	An object to be printed
arg	`NameClass` object indicating printing style
stream	`ByteStream` object that is writable

Redirects the `prin` method to call `recurPrin`, a primitive defined in `RootObject` that handles the printing of objects that contain recursive structures. Use this function to specialize the `prin` method to handle objects that contain recursive structures. In a scripted class that contains recursive structures, define the `prin` method to call `printRecursively`, passing on its own arguments.

The function `printRecursively` calls the method `recurPrin` on the *object* to be printed, using the printing style *arg*, to print to *stream*. It supplies a final argument that represents a recursive print state. See the method `recurPrin`, defined by `RootObject`. This fourth argument is passed on to resolve references in recursive structures. *(Object System Kernel)*

printStorageStats (global function)

`printStorageStats()` ⇨ *(none)*

Prints information about where the object storage system is spending its time (seeking, reading, caching). You can use this information to identify portions of your title where you are doing too much seeking and reading or where you are recaching an object that you just cached a couple of reads ago. You can also use this information to quantify the improvement (or lack of improvement) in your title's performance as a result of using the storage reorganizer (see the `storageFileRelocate` global function) or making other changes to your title. To collect information on discrete portions of your title, use the `resetStorageStats` global function before you enter the portion of your title for which you want to collect a separate set of storage statistics. See the "Title Management" chapter of the *ScriptX Components Guide* for a detailed description of the information printed by `printStorageStats`. *(Title Management component)*

printString (global function)

`printString` *object* ⇨`String`

Creates and returns a new instance of `String` containing the printed representation of *object*. *(Object System Kernel)*

The function `printString` is equivalent to

```
return (prinString (object @normal))
```

ptt (global function)

`ptt` *filename* ⇨ *(none)*

filename	String representing name of file

See the "Title Analysis API" chapter in the *ScriptX Tools Guide*. *(Tools component)*

purgeModuleContents (global function)

`purgeModuleContents` *module* ⇨ *(none)*

module	`ModuleClass` object

Iterates through all named objects in a module, calling `requestPurge` on named objects that are purgeable. *(Modules)*

quit (global function)

`quit` *object* ⇨ *(none)*

ignoreMe	any object

Initiates the Quit Manager, quitting ScriptX. The single parameter can be any object, and is ignored. A parameter is required for compatibility with the system menu bar. The Quit Manager passes through three phases as it shuts down the system. First it runs quit queries. If a quit query returns `true`, execution continues with the next quit query. If any quit query returns `false`, the quit process is immediately aborted, no more quit queries or quit tasks are executed, and execution of the main program resumes.

If all quit queries return `true`, the Quit Manager goes on to the second phase, running quit tasks. Once the quit tasks begin running, system shutdown is assured.

After all quit tasks have run, the Quit Manager closes the ScriptX object system and relinquishes control of resources it was using, such as memory, to the underlying operating system. This final phase of shutdown is not visible to the user or the scripter.

By default, there are no quit queries defined, so calling `quit` closes the Kaleida Media Player without any user interaction. If you want to ask the user about saving work they have done, or asking if they are sure they want to close a title, you would do that in a quit query.

For more information, see the global functions `installQuitQuery` and `installQuitTask`, defined in this chapter. *(Title Management component)*

requestPurge (global function)

`requestPurge` *object* ⇨ *(none)*

object	Any object added to a storage container

If *object* is in memory, this method flags *object* for garbage collection. This *object* can be any object that has been added (but not necessarily saved) to a storage container.

The object in memory is marked for garbage collection and its memory is recovered in the next garbage collection cycle as long as no methods are called on it. Any subobject of *object* also is marked for garbage collection. All references to *object* are maintained if *object* has been saved to its storage container. (References are maintained by keeping a handle to the object in memory, and purging only the body of the object. If all references are removed, the handle will also be garbage collected.) Any future method call on *object* reloads the object into memory from its storage container. See also the `isInMemory`, `canRequestPurge`, and `isPurgeRequested` global functions.

Calling `requestPurge` on an object that has never been added to a storage container does nothing. Calling `requestPurge` on an object that has been added but has never been updated (saved to its storage container), displays a warning that you are requesting a premature purge; once the next garbage collection cycle has run, that storage container will no longer contain that object, and future references to the object will result in errors. If you call `requestPurge` on an object that has been modified since the last time it was updated, then changes you made to the object since the last time you updated it will be lost; you may want to use this as a way to "revert to the last saved" version of the object. Calling `requestPurge` on a persistent container is illegal and results in an exception. *(Title Management component)*

resetStorageStats (global function)

`resetStorageStats()` ⇨ *(none)*

Resets or clears the storage statistics that are being gathered by the `printStorageStats` global function. You may want to use the `resetStorageStats` function between transitions or at other strategic places (so that you can gather statistics for a discrete portion of your title) when you are running your title with `printStorageStats` turned on. *(Title Management component)*

rtt (global function)

`rtt` *arg* ⇨ *(none)*

arg	`Integer` value of `1` or `0`

See the "Title Analysis API" chapter in the *ScriptX Tools Guide*. *(Tools component)*

safeRecurPrin (global function)

`safeRecurPrin` *object arg stream state* ⇨ *(object)*

object	Object to be printed
arg	`NameClass` object indicating printing style
stream	`ByteStream` object that is writable
state	Object that represents a recursive print state

This function is called by the `recurPrin` method, a variation on the `prin` method that is used to print out objects that contain recursive structures. The `recurPrin` method calls `safeRecurPrin`, passing its own arguments. The arguments to `recurPrin` and `safeRecurPrin` are the same as the arguments to `prin`, with the addition of a final argument that represents a recursive print state. This argument, supplied by the global function `printRecursively`, is passed on to resolve references in recursive structures. *(Object System Kernel)*

searchIndex (global function)

`searchIndex` *strIndex match searchContext whole* ⇨ `SearchContext`

strIndex	`StringIndex` object
match	`String` object
searchContext	`SearchContext` object
whole	`Boolean` object

This function searches *string* for *match* using *searchContext* to tell it where to start or resume searching and using*whole* to tell it whether or not a match must be a whole word. Returns `empty` if *match* is not found; otherwise returns a `SearchContext` object.

Because `searchIndex` is used only in connection with `StringIndex` and `SearchContext` objects, the entries for those classes contain more complete information.

setGCIncrement (global function)

`setGCIncrement` *incr* ⇨ *incr*

incr	`Integer` object between 1 and 1000

Sets the time increment *incr*, measured in milliseconds, for which the garbage collector will run before yielding, returning the value that has been set. Developers can use `setGCIncrement` to increase or decrease the amount of time spent in the garbage collector.

Time spent in the garbage collector is also a function of the number and priorities of other active threads. Setting too low a value for *incr* can starve the garbage collector. Setting too high a value can interrupt other processes. If the system is thrashing, setting a higher value may even have a negative effect on the overall performance of a title. *(Memory Management component)*

setStorageCacheSize (global function)

`setStorageCacheSize` *num* ⇨ `ImmediateInteger`

num	`ImmediateInteger` object, 0<*num*<256

Sets the size of the storage cache to *num* 4K blocks, where *num* must be greater than zero and less than 256. The default storage cache size is 8 4K blocks. You may want to increase this cache size to improve your object loading time (and therefore your title's performance). *(Title Management component)*

shortPrin (global function)

`shortPrin` *object arg stream maxLength* ⇨ *(none)*

object	Object to be printed
arg	`NameClass` object indicating printing style
stream	`ByteStream` object that is writable
maxLength	`Integer` object greater than zero

Identical to the `prin` method (defined on `RootObject`) except that `shortPrin` will never print out more than the number of characters specified by *maxLength*. *(Object System Kernel)*

storageFileRelocate (global function)

`storageFileRelocate` *inputTitleStream outputTitleStream logStream* ⇨ `ByteStream`

inputTitleStream	any readable `ByteStream` object
outputTitleStream	any writable `ByteStream` object
logStream	any readable `ByteStream` object

Reorganizes the objects are stored in the title file or files represented by *inputTitleStream* into a physical layout that will be more optimal when the title is run. The *logStream* is the record of object access that was saved in the `storageProfileLog` when you ran your title with storage profiling turned on. This object access information is used by storageFileRelocate to optimize the physical layout of objects in the container. The title file or files represented by *outputTitleStream* are the newly reorganized files.

The following is an example script to reorganize a title, `title.sxt`, into a more optimal physical layout:

```
global src := getStream (buildDir, "title.sxt", @readable)
global tgt := getStream (buildDir, "reloc.sxt", @writable)
global log := getStream (buildDir, "osprofile.log", @readable)
storageFileRelocate src tgt log
plug src
plug tgt
plug log
```

To reorganize a more realistic title that consists of many files, use collection objects for the input and output title arguments and operate on the collection members in a loop. See the "Title Management" chapter of the *ScriptX Components Guide* for an example script. *(Title Management component)*

systemQuery (global function)

`systemQuery` *name* ⇨ *(object)*

name	`NameClass` object representing attribute

Queries the operating system for information about attributes of the underlying platform on which ScriptX or the Kaleida Media Player is running. The *name* argument represents the attribute being queried, the return values represent the attributes for the particular platform. The following name arguments are used:

- `@monitorRects` – Returns an `Array` object containing `Rect` objects that represent the pixel area of all monitors; the first entry represents the main monitor.*
- `@monitorDepths` – Returns an `Array` object containing integers representing the depths of all monitors; the first entry represents the main monitor.*
- `@cpuType` – Returns an instance of `String` such as `"MC68030"` and `"i486 compatible"` (returns `"MC68020"` in emulation mode on a PowerMacintosh).

- `@physicalMemory` – Returns an instance of `Integer` representing physical memory in the machine, in 1024 byte units.
- `@osVersion` – Returns an instance of `Number` representing the version of the operating system, which has both a major and a minor version number, such as “7.1” for Macintosh System 7.1, or “3.1” for Microsoft Windows 3.1.
- `@osName` – Returns a `String` object representing the operating system name (currently `"Macintosh"`, `"OS/2"`, or `"Windows"`).
- `@sxBuild` – Returns a string indicating the version, build, and date compiled.
- `@sxBuildNumber` – Returns an integer representing the ScriptX build, for version 1.5 only. For earlier versions, `@sxBuildNumber` returns `undefined`.
- `@sxVersion` – Returns an instance of `Float` representing the ScriptX version (a value such as 1.0 or 1.5).
- `@sxName` – Returns an instance of `String` representing the ScriptX product name (currently `"ScriptX Builder"` or `"ScriptX Player"`)

* Note that `@monitorRects` and `@monitorDepth` return arrays with entries in the same order, so that a size in the array of rectangles corresponds with a pixel depth in the array of integers.

threadCriticalDown (global function)

`threadCriticalDown()` ⇨ *(none)*

Allows the scheduler to resume, so that it can switch threads. This function is always paired with a prior call to `threadCriticalUp`, which suspends the scheduler and prevents the running thread from being preempted. A call to `threadCriticalDown` balances a call to `threadCriticalUp`, enabling the scheduler to preempt the running thread. See `threadCriticalUp`, a global function defined below. *(Threads component)*

threadCriticalUp (global function)

`threadCriticalUp()` ⇨ *(none)*

Suspends the scheduler, preventing it from switching threads until the next call to `threadCriticalDown`. A call to `threadCriticalUp` precedes a critical segment of a code in which the scheduler is not allowed to switch threads. This segment must be followed by a call to `threadCriticalDown`, which allows the scheduler to resume. These two functions are designed to insure that short segments of code can be executed atomically with respect to the scheduler.

Use `threadCriticalUp` and `threadCriticalDown` cautiously and sparingly. An unbalanced call to `threadCriticalUp` can leave the system in a suspended state, from which the user will be unable to recover. Two other global functions, `gateOpenThenWait` and `lockMany`, can be used to assure that common operations with gates are performed atomically. *(Threads component)*

Note – In the future, your ScriptX title or tool may run on systems that use more than one processor. When the ScriptX runtime environment is implemented to run with multiple processors, `threadCriticalUp` will suspend all threads that are running on other processors. Keep in mind that `threadCriticalUp` and `threadCriticalDown` are meant to enclose only brief segments of code. To assure that a title runs smoothly, make sure that any threads can be preempted whereever possible.

threadExit (global function)

`threadExit` *result* ⇨ `Boolean`

result The object that is to be returned by the thread

Exits the currently running thread and returns *result*. This is equivalent to calling the `Thread` instance method `threadReturn`: *(Threads component)*

```
threadReturn  theRunningThread  result)
```

threadIdle (global function)

`threadIdle()` ⇨ `Boolean`

Allows the thread system to do idling activities, and then yields to let other threads run. Returns `true` if a yield took place. *(Threads component)*

threadYield (global function)

`threadYield()` ⇨ `Boolean`

Causes the running thread to yield for one cycle to the next thread scheduled to run. Returns `true` if in fact there was such a thread and a yield took place. *(Threads component)*

threadYieldTo (global function)

`threadYieldTo` *nextThread* ⇨ `Boolean`

nextThread `Thread` object

Causes the running thread to yield to the thread *nextThread*, but only if that thread is in fact runnable. If it isn't, this function is equivalent to `threadYield`, yielding to the next thread scheduled to run. Returns `true` if there was such a thread and a yield took place. *(Threads component)*

totalFreeHeapSpace (global function)

`totalFreeHeapSpace()` ⇨ `Integer`

Reports the total amount of free memory that is available to the ScriptX runtime environment in the ScriptX heap. See also the global functions `largestFreeHeapBlock` and `totalHeapSpace()`. *(Memory Management component)*

totalFreeSystemSpace (global function)

`totalFreeSystemSpace()` ⇨ `Integer`

Reports the total amount of free memory that is available to the ScriptX runtime environment outside of the ScriptX heap. See also the global function `largestFreeSystemBlock` for an explanation of how to interpret `totalFreeSystemSpace` on each ScriptX platform.

This function is useful in tracing the source of memory management conflicts with certain platforms, however results are dependent on the underlying operating system. It is recommended strictly as a diagnostic tool. *(Memory Manager component)*

totalHeapSpace (global function)

`totalHeapSpace()` ⇨ `Integer`

Returns the total number of bytes in the ScriptX heap. Note that the amount of heap space currently in use can always be determined as follows:

```
x := totalHeapSpace() - totalFreeHeapSpace().
```

This function should be used strictly as a diagnostic tool. *(Memory Manager component)*

treeSize (global function)

`treeSize` *object verbose maxDepth* ⇨ `Integer`

object	Any object
verbose	`Integer` object
maxDepth	`Integer` object

Reports how much memory the given *object* is using, treating this object as the root of a possible tree of objects. The function `treeSize` adds up the memory used by all objects that the given object is pointing to. It detects recursion, counting recursive structures only once. If the value of *verbose* is `0`, `treeSize` returns a value, but does not print a report. If the value of *verbose* is non-zero, `treeSize` prints out a full report, showing by indentation how objects are laid out in memory.

```
myObj := new Array
treeSize myObj 0 10
⇨32
for i in 1 to 1024 collect into myObj i
⇨#(1, 2, 3, 4, 5, 6, 7, 8, 9, 10, ...)
treeSize myObj 0 10
⇨8224
```

Note that `treeSize` only follows pointers in the ScriptX heap. Some objects, such as `Window` objects, contain pointers to structures in system memory.

This function should be used strictly as a diagnostic tool. See also the global function `objectSize`. *(Memory Management component)*

ucmp (global function)

`ucmp` *x y* ⇨ `NameClass`

Compares the objects *x* and *y* universally, returning either `@same`, `@before`, or `@after`. *(Object System Kernel)*

The `ucmp` function is equivalent to the following:

```
function ucmp x y ->
if (ueq x y) then @same
    else if (ult x y) then @before
    else @after
```

ueq (global function)

`ueq` *x y* ⇨ `Boolean`

Compares the objects *x* and *y*, returning `true` if they are universally equal. *(Object System Kernel)*

The `ueq` function is equivalent to the following:

```
function ueq x y -> ((getClass x) == (getClass y)) and (localEqual x y)
```

uge (global function)

uge *x y* ⇨ Boolean

Compares the objects *x* and *y*, returning `true` if *x* is universally greater than or equal to *y*. *(Object System Kernel)*

The uge function is equivalent to the following:

```
function uge x y -> not (ult x y)
```

ugt (global function)

ugt *x y* ⇨ Boolean

Compares the objects *x* and *y*, returning `true` if *x* is universally greater than *y*. *(Object System Kernel)*

The ugt function is equivalent to the following:

```
function ugt x y -> not (ule x y)
```

ule (global function)

ule *x y* ⇨ Boolean

Compares the objects *x* and *y*, returning `true` if *x* is universally less than or equal to *y*. *(Object System Kernel)*

The ule function is equivalent to the following:

```
function ule x y ->
    if ((getClass x) == (getClass y)) then
        (localLt x y) or (localEqual x y)
    else
        (localLT (getClass x) (getClass y))
```

ult (global function)

ult *x y* ⇨ Boolean

Compares the objects *x* and *y*, returning `true` if *x* is universally less than *y*. *(Object System Kernel)*

The ult function is equivalent to the following:

```
function ult x y ->
    if ((getClass x) == (getClass y)) then
        (localLt x y)
    else
        (localLT (getClass x) (getClass y))
```

une (global function)

une *x y* ⇨ Boolean

Compares the objects *x* and *y*, returning `true` if they are universally not equal. *(Object System Kernel)*

The une function is equivalent to the following:

```
function une x y -> not (ueq x y)
```

updateIndex (global function)

`updateIndex` *strIndex* ⇨ `StringIndex`

strIndex `StringIndex` object

This function rebuilds *strIndex*'s signature index based on *strIndex*'s `string` instance variable. In order to keep the index valid, `updateIndex` should be used any time *strIndex*'s `string` is modified.

See the entry for `StringIndex` for a more complete description and code examples.

warnings (global function)

`warnings` *flag* ⇨ *(object)*

flag `Boolean` object

When flag is *true*, sets ScriptX to display warnings in the Listener when it encounters harmful—though not fatal—situations. In the current release, this function causes a warning when a bitmap with one color map is transferred to a surface with another color map, which diminishes drawing performance.

This function is provided only in ScriptX. It is not included in the Kaleida Media Player.

xor (global function)

`xor` *boolean1 boolean2* ⇨ `Boolean`

Performs the "logical exclusive or" operation on *boolean1* and *boolean2*, two `Boolean` objects. Returns `true` if and only if the value of *boolean1* or *boolean2* (but not both) is `true`. (Note that in ScriptX 1.0, `eq` was defined as a generic function.)

The following example demonstrates how a function can be rewritten in a more efficient (although perhaps less readable) manner using `xor`. Note that `round1` and `round2` are both equivalent to `round`, a generic that is implemented by `Number` objects.

```
fn round1 x -> (
    if x > 0 then
        if (abs (frac x)) >= 0.5 then ceiling x
        else floor x
    else
        if (abs (frac x)) >= 0.5 then floor x
        else ceiling x
)
-- a more efficient but less readable version of round1
fn round2 x ->
    if xor (abs(frac x) >= 0.5) (x > 0) then floor x else ceiling x
```

C H A P T E R

Global Constants and Variables

3

This chapter lists global constants and global variables that are defined by ScriptX. These objects are available to the scripter with these name bindings, in any module that uses the ScriptX module. Some of these constants and variables are associated closely with one component and the classes that component defines. The component reference at the end of each description indicates which chapter in the *ScriptX Components Guide* to refer to for more information.

blackBrush (global constant)

`blackBrush` `Brush`

Represents a global instance of the `Brush` class with `color` set to `blackColor`, `pattern` set to `blackPattern`, and `inkMode` set to `srcCopy` *(2D Graphics component)*.

blackColor (global constant)

`blackColor` `RGBColor`

Represents an `RGBColor` instance whose `red`, `green`, and `blue` instance variables are all set to 0 *(2D Graphics component)*.

blueColor (global constant)

`blueColor` `RGBColor`

Represents an `RGBColor` instance whose `blue` instance variable is set to 255 and whose `red` and `green` instance variables are set to 0 *(2D Graphics component)*.

console (global constant)

`console` `MDTextWindow`

Available only in the ScriptX executable, not in the Kaleida Media Player. The `console` is the Listener window, which is an instance of `MDTextWindow` (machine-dependent). *(Tools Component)*

You can change the font size in the Listener window by setting the `fontSize` instance variable, which is quite useful for relieving eye strain on large monitors or when demonstrating ScriptX to large audiences:

```
console.fontSize := 16
```

cyanColor (global constant)

`cyanColor` `RGBColor`

Represents an `RGBColor` instance whose `green` and `blue` instance variables are set to 255 and whose `red` instance variable is set to 0 *(2D Graphics component)*.

defaultBrush (global constant)

`defaultBrush` `Brush`

Represents a global instance of the `Brush` class; also, a synonym for the global instance `whiteBrush`. See `whiteBrush`, defined in this section. *(2D Graphics component)*

e (global constant)

`e` `Float`

Represents the mathematical constant e (2.1718...) as a float. *(Numerics component)*

empty (global constant)

`empty` `EmptyClass`

The `empty` object is an instance of `EmptyClass`. It has several different uses:

In an implicitly keyed collection, when there's no need to specify a key with a method, use `empty` as a placeholder. For example, the key is ignored when adding to a sorted array: *(Collections component)*

```
ar := new SortedArray
add ar empty "Hello"  -- adds the value "Hello" to the array
```

In explicitly keyed collections, using `empty` as the key for add means "use the given value as the key". For example:

```
kl := new KeyedLinkedList
add kl empty 1  -- adds the key-value pair 1 1 to the linked list
```

The `empty` object is returned from a collection when the thing you asked for isn't there. For example, you get `empty` if you try to get the third item of a `Pair` object:

```
pr := new Pair 1 2
getNth pr 3  -- returns empty
```

Note – Never insert the `empty` object into a collection. A number of collection methods return `empty` as the value that signals "no matching elements in the collection." Placing the `empty` object in the collection would cause these methods to behave erratically, and possibly fail altogether. Some collections may report the `badValue` exception if you try to put `empty` in them.

false (global constant)

`false` `Boolean`

Represents a constant that can be used in Boolean expressions. The global constant `false` is one of two global instances of the `Boolean` class. *(Numerics component)*

greenColor (global constant)

`greenColor` `RGBColor`

Represents an `RGBColor` instance whose `green` instance variable is set to 255 and whose `red` and `blue` instance variables are set to 0. *(2D Graphics component)*

loadableUnitIdNull (global constant)

`loadableUnitIdNull` `LoadableUnitId`

Returned by the `Loader` method `loadModule` when the loadable unit contains unloadable data. *(Loader component)*

loadableUnitIdError (global constant)

`loadableUnitIdError` `LoadableUnitId`

Returned by the `Loader` method `loadModule` when loading a loadable unit generates an error. *(Loader component)*

loadableUnitIdLoading (global constant)

`loadableUnitIdLoading` `LoadableUnitId`

Returned by the `Loader` method `loadModule` when a loadable unit is already loading. *(Loader component)*

loadableUnitIdInitErr (global constant)

`loadableUnitIdInitErr` `LoadableUnitId`

Returned by the `Loader` method `loadModule` when a loadable unit's entry point function fails to return properly. *(Loader component)*

loaderCodeBad (global constant)

`loaderCodeBad` `LoaderCode`

Represents a global instance of the class `LoaderCode`. *(Loader component)*

loaderCodeError (global constant)

`loaderCodeError` `LoaderCode`

Represents a global instance of the class `LoaderCode`. *(Loader component)*

loaderCodeOk (global constant)

`loaderCodeOk` `LoaderCode`

Represents a global instance of the class `LoaderCode`. *(Loader component)*

magentaColor (global constant)

`magentaColor` `RGBColor`

Represents an `RGBColor` instance whose `red` and `blue` instance variables are set to 255 and whose `green` instance variable is set to 0. *(2D Graphics component)*

nan (global constant)

`nan` `Number`

Represents an uncountable number of items, such as all numbers in the range between 1 and 2. (This example is uncountable because it includes an infinite number of irrational numbers, such as the square root of 2, with no way to find them all.) By "uncountable," we mean there is no way to map the positive integers (1, 2, 3 ...) to all numbers. The term `nan` is from computer science and stands for "not a number." This constant is returned as the result of certain operations on numbers, such as dividing by zero. *(Numerics component)*

negInf (global constant)

`negInf` `Number`

Represents negative infinity. This constant is useful for expressing a range with any `Number` class. The following example creates the range of negative numbers: *(Numerics component)*

```
new NumberRange lowerBound:negInf upperBound:0
```

nullStream (global constant)

`nullStream` `NullStreamClass`

The single, global instance of `NullStreamClass`, a subclass of `ByteStream`, representing a stream containing no data. *(Streams component)*

objectStoreMessages (global variable)

`objectStoreMessages` (read-write) `ImmediateInteger`

Prints messages to the Listener whenever a storage container is loaded from or saved to. If you want the messages written to some other stream and not to the Listener, redirect the output by setting `objectStoreMessagesStream`. Which messages are written depends on which flags are set, where each flag is a separate bit as shown in the table below. Set the value of `objectStoreMessages` to an integer from 1 to 7. To set more than one flag, add their values. For example, if you set `objectStoreMessages` to 3, a message will be printed to the Listener when an object is inflated (1) and when it is deflated (2). Notice that only the first flag is available in the KMP. *(Title Management component)*

Flag Bit	Messages Written
`1`	Write info on inflate of object.
`2`	Write info on deflate of object.
`4`	Write info on deflate/inflate for every class slice of object.

Note: Only output level 1 is available in the KMP.

objectStoreMessagesStream (global variable)

`objectStoreMessagesStream` (read-write) `ByteStream`

Stores messages written by `objectStoreMessages`. If `objectStoreMessagesStream` is undefined, messages written by `objectStoreMessages` are printed to the Listener. *(Title Management component)*

The following is an example of using `objectStoreMessagesStream`:

```
objectStoreMessagesStream := debug
```

OK (global constant)

`OK` `OKClass`

Represents the value returned by functions or methods when they have no relevant return value (similar to `void`-returning functions in C). Functions that return OK always return OK regardless of the outcome of the function.

This global constant is useful for developers writing tools. For example, if you write a function that prints a list of classes in the Listener window. When done printing the list, the function also returns the value of the function, which would normally be the last class in the list—therefore, the last class would print twice. To avoid this, you can include OK as the return value for the function; then the function would print the list of classes, then the value OK. *(Object System Kernel)*

pi (global constant)

`pi` `Float`

Represents pi (3.14159...) as a float. *(Numerics component)*

piDiv2 (global constant)

`piDiv2` `Float`

Represents pi/2 as a float. *(Numerics component)*

posInf (global constant)

`posInf` `Number`

Represents positive infinity. This constant is useful for expressing a range with any `Number` class. The following example creates the range of positive numbers:

```
new NumberRange lowerBound:0 upperBound:posInf
```

(Numerics component)

printTruncateStringSize (global variable)

`printTruncateStringSize` (read-write) `Integer`

Sets the size in characters of the longest possible string returned in the Listener window. Its initial value is 512 characters. To allow unlimited characters, set this variable to –1.

Available only in the ScriptX development environment, not in the Kaleida Media Player, since the Listener window does not exist in the KMP. *(Tools component)*

rarelyInflatedClasses (global variable)

`rarelyInflatedClasses` (read-write) `Array`

Allows coarse-grained developer control of physical storage layout by allowing you to specify classes that are rarely used in your title. The `rarelyInflatedClasses` variable is a collection of class names; it contains `DebugInfo` by default. Instances of classes listed in `rarelyInflatedClasses`, and children of these instances, are moved to the end of the storage file. This can slow deflation of these objects, but it can improve overall inflation performance substantially. *(Title Management component)*

redColor (global constant)

`redColor` `RGBColor`

Represents an `RGBColor` instance whose `red` instance variable is set to 255 and whose `green` and `blue` instance variables are set to 0 *(2D Graphics component)*.

showCode (global variable)

`showCode` (read-write) `Boolean`

See the "Title Analysis API" chapter in the *ScriptX Tools Guide*. *(Tools component)*

sqrt2 (global constant)

`sqrt2` `Float`

Represents the square root of 2 as a float.

storageProfileLog (global variable)

`storageProfileLog` (read-write) `ByteStream`

Stores a record of how objects are accessed in your title when you run your title with storage profiling turned on. This record can then be used by the `storageFileRelocate` global function to reorganize the objects stored in your title so that they are stored in a more optimal physical layout. To turn on storage profiling, simply define this variable and then open and run the title you want to profile. When you are finished profiling, plug the `storageProfileLog` stream to ensure all data is flushed to disk. Then set `storageProfileLog` to `undefined` to turn profiling off. *(Title Management component)*

The following is an example of how to profile your title:

```
-- Turn on profiling.
storageProfileLog := getStream (buildDir, "osprofile.log", @readable)
-- Open and run your title.
-- Close the profile log and turn off profiling.
plug storageProfileLog
storageProfileLog := undefined
```

theCalendarClock (global constant)

`theCalendarClock` `CalendarClock`

This global constant is created by the ScriptX runtime environment at startup, and is the single instance of the class `CalendarClock`. If you attempt to create an instance of `CalendarClock` using the `new` method, this global is returned *(Clocks component)*.

theClipboard (global constant)

`theClipboard` `Clipboard`

Specifies the clipboard into which objects can be cut or copied, and from which objects can be pasted. For more details, see the `Clipboard` class. *(Title Management component)*

theContainerSearchList (global variable)

`theContainerSearchList` (read-write) `Collection`

The global variable `theContainerSearchList` provides a collection of `DirRep` instances that represent directory search paths for resolving references between objects stored in separate `StorageContainer` files. At ScriptX startup, this collection contains only the global `DirRep` instance `theStartDir`. When any title, library, or accessory container is opened, its directory is automatically added to this list, to facilitate finding other files. A title can add items to and remove items from this collection as it needs to.

To see how this global variable is used, refer to “Opening a Title Container” in the *ScriptX Components Guide*. *(Title Management component)*

theDefault1ColorMap (global constant)

`theDefault1ColorMap` `ColorMap`

Represents the platform-specific default color maps for 1-bit screen depths. See note for `theDefault8ColorMap` *(2D Graphics component)*.

theDefault2ColorMap (global constant)

`theDefault2ColorMap` `ColorMap`

Represents the platform-specific default color maps for 2-bit screen depths. See note for `theDefault8ColorMap` *(2D Graphics component)*.

theDefault4ColorMap (global constant)

`theDefault4ColorMap` `ColorMap`

Represents the platform-specific default color maps for 4-bit screen depths. See note for `theDefault8ColorMap` *(2D Graphics component)*.

theDefault8ColorMap (global constant)

`theDefault8ColorMap` `ColorMap`

Represents the platform-specific default color maps for 8-bit screen depths. The actual default color map used on a specific platform depends on the pixel depth of the underlying hardware and the actual color palettes supported by the underlying graphics system *(2D Graphics component)*.

theEventTimeStampClock (global constant)

`theEventTimeStampClock` `Clock`

Represents the `Clock` instance used for generating time stamps supplied with events. Its `scale` instance variable is set to 1000 and its `rate` instance variable is set to 1 *(Events component)*.

theImportExportEngine (global constant)

`theImportExportEngine` `ImportExportEngine`

Specifies the instance of `ImportExportEngine` that the ScriptX system created at startup. *(Import and Export component)*

theMainThread (global variable)

`theMainThread` (read-only) `Thread`

Specifies the main thread, which is the original thread of control in the system and is not necessarily currently running. The thread labeled `theMainThread` has no special priority in the scheduler. This thread cannot be restarted. Killing it is equivalent to quitting ScriptX. *(Threads component)*

theOpenContainers (global variable)

`theOpenContainers` (read-only) `Array`

Specifies all open library, title and accessory containers. This list is automatically updated any time such a container is opened or closed. The order of containers is the order in which they are opened, with the most recently opened one first—not the order they appear on-screen. *(Title Management component)*

theOpenTitles (global variable)

`theOpenTitles` (read-only) `Array`

Specifies the open title containers, generally ordered by focus priority, which is the same order they appear visually on-screen. (This order is not guaranteed.) This list is updated automatically any time a title container is opened or closed. *(Title Management component)*

theRootDir (global constant)

`theRootDir` `RootDirRep` or `DirRep`

Represents the root directory of the host platform. This constant is an instance of the system-specific subclass of `RootDirRep` on operating systems, such as the Macintosh OS and DOS, that represent each storage device as a volume or drive. It is an instance of the system-specific subclass of `DirRep` on other operating systems, such as UNIX. *(Files component)*

theRunningThread (global variable)

`theRunningThread` (read-only) `Thread`

Specifies the currently executing thread. In ScriptX, only one thread runs at a time; the scheduler apportions execution time to active threads. *(Threads component)*

theScratchTitle (global constant)

`theScratchTitle` `TitleContainer`

Specifies the default title container to hold windows, clocks, and players created outside a title. That is, when you create a new window, clock, or player and omit the `title` keyword, the object is added to the title container specified by `theScratchTitle`. This title container is created automatically at startup and cannot be saved.

You can move a window, clock, or player from the container specified by `theScratchTitle` to any other library, title, or accessory container either by setting its `title` instance variable or by appending it directly to the new title container, which causes it to automatically be removed from its previous title container. *(Title Management component)*

theScriptDir (global variable)

`theScriptDir` (read-write) `DirRep`

Not available in the Kaleida Media Player. This variable represents the source directory of the most recently opened and compiled ASCII text script file. It can be useful for specifying the path to files needed by a script when that script is shared from a server by mulitple developers. Using `theScriptDir` instead of `theStartDir` enables the script to be successfully compiled regardless of the location of the ScriptX development environment. *(Files component)*

Note – The global variable `theScriptDir` is defined only in the ScriptX executable, and not in the Kaleida Media Player. Using this global in a KMP title may cause the title to fail. The `fileIn` method that sets this value requires the ScriptX bytecode compiler, which is present only in ScriptX. *(Files component)*

theStartDir (global constant)

`theStartDir` `DirRep`

Specifies the directory representative instance representing the directory where the currently running ScriptX runtime environment is located. To see the array of strings representing the directories, coerce `theStartDir` to an array, as follows:

```
theStartDir as Array
```

When a title or library is specifying the path to another title or library, it is usually preferable to specify it relative to the original title, by using the `directory` instance variable of the title container (rather than `theStartDir`). This frees the Kaleida Media Player to be located anywhere relative to the title. *(Files component)*

theTempDir (global constant)

`theTempDir` `DirRep`

Represents a directory that can be used for temporary storage in the file system. *(Files component)*

theTitleContainer (global variable)

`theTitleContainer` (read-only) `TitleContainer`

Specifies the title container that currently has user focus. This means the system menu bar for this title container, if set to showing, is visible, and the title's frontmost window, if any, has user focus.

This value is updated when the user (or a script) selects another title, making it active. *(Title Management component)*

theUIEventDispatchQueue (global variable)

`theUIEventDispatchQueue` (read-only) `EventDispatchQueue`

Contains the name of the primary dispatch queue, an `EventDispatchQueue` object, through which system-defined queued events are processed, including mouse and keyboard events. *(Events and Input Devices component)*

throwArg (global variable)

`throwArg` (read-only) *(object)*

Contains the argument that was reported with the most recent exception. See also `throwTag`. *(Exceptions)*

throwTag (global variable)

`throwTag` (read-only) `Exception`

Contains the most recent exception that was reported. See also `throwArg`. *(Exceptions)*

true (global constant)

`true` `Boolean`

Represents a constant that can be used in Boolean expressions. The global constant `true` is one of two global instances of the `Boolean` class. *(Numerics component)*

whiteBrush (global constant)

`whiteBrush` `Brush`

Represents a global instance of the `Brush` class with `color` set to `whiteColor`, `pattern` set to `blackPattern`, and `inkMode` is `@srcCopy`. *(2D Graphics component)*

whiteColor (global constant)

`whiteColor` `RGBColor`

Represents an `RGBColor` instance whose `red`, `green`, and `blue` instance variables are all set to 255 *(2D Graphics component)*.

yellowColor (global constant)

`yellowColor` `RGBColor`

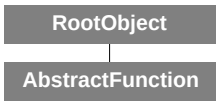
Represents an `RGBColor` instance whose `red` and `green` instance variables are set to 255 and whose `blue` instance variable is set to 0 *(2D Graphics component)*.

C H A P T E R

Class Descriptions

4

AbstractFunction



Class type: Core class (abstract, sealed)
Resides in: ScriptX and KMP executables
Inherits from: `RootObject`
Component: Object System Kernel

`AbstractFunction` is the superclass of all ScriptX functions and generic functions. In the current version of ScriptX, subclasses of `AbstractFunction` include `ByteCodeMethod`, `Primitive`, `PrimitiveMethod`, and `Generic`.

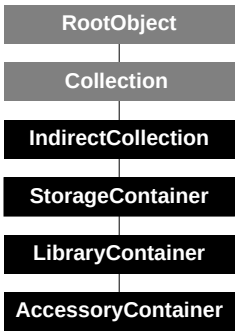
Although all ScriptX functions and generic functions are objects (a method is an implementation of a generic function for a particular class or object), you never create an instance of `AbstractFunction` directly. ScriptX functions are created automatically when you use the function, method, and anonymous function definition expressions in the ScriptX language.

To test whether an object is a function, use the generic `isAKindOf` with the class `AbstractFunction` as an argument.

```
global cubeMe := (x -> x * x * x) -- creates an anonymous function
⇨  #<ByteCodeMethod anonymous of 1 argument>
isAKindOf cubeMe AbstractFunction
⇨  true
```

Developers can assume that future ScriptX function classes will inherit from `AbstractFunction`. In other respects, do not assume that the implementation of functions and generics will be the same in future releases of ScriptX.

AccessoryContainer



Class type: Core class (concrete)
Resides in: ScriptX and KMP executables
Inherits from: `LibraryContainer`
Component: Title Management

The `AccessoryContainer` class represents a file on disk that contains a set of classes and instances that can be dynamically added to a running title. This overall set of classes and instances is known as an "accessory". An accessory incrementally adds data or behavior to a title, and may be used by any number of titles. An accessory supplements and is subordinate to a title. The title container represented by `theScratchTitle` is always open and can serve as the title container for accessories that stand alone.

An accessory container is a collection: It inherits from `IndirectCollection`. Its default target collection is an `Array` object. You should add to an accessory container objects that you want the `getAccessory` method to access. The `getAccessory` method determines which objects in the accessory are passed to the title. The title's `addAccessory` method receives those objects and performs whatever is appropriate for that title, such as instantiating accessory classes, slaving an accessory clock off its clocks, or putting accessory presenters into its windows.

If an accessory container creates an instance of a window, clock, or player, that instance has its `title` instance variable set to `theScratchTitle`. It is then up to the title to reassign this value to itself when it adds the accessory. The window will then close when the title closes, and the clock or player will pause when the title pauses.

The responsibility for determining the whether a particular accessory is suitable for a title rests on the title to which it is added. The title's `isAppropriateAccessory` method is automatically called when the user opens a title container from within ScriptX using the **Open Title** menu command. When calling the `open` method from a script, it is up to the script to call `isAppropriateAccessory`.

Creating and Initializing a New Instance

The following script is an example of how to create a new instance of the `AccessoryContainer` class:

```
myAcc := new AccessoryContainer \
          dir:myTitle.directory \
          path:"MyAcc.sxa" \
          name:"My Accessory" \
          user:myTitle
```

The variable myAcc contains the initialized accessory container, which is stored in a new file called MyAcc.sxa in the directory given by myTitle.directory. Its name when asked to print is My Accessory. The accessory container will be used by the myTitle title container, meaning the accessory container will normally remain open as long as the title container is open. The new method uses the keywords defined in init.

Note – The convention for naming accessory container files, valid across all platforms, is to use the .sxa extension, as shown above (meaning "ScriptX Accessory").

init

init *self* [dir:*dirRep*] path:*collectionOrString* [name:*string*]
[user:*titleContainer*] [targetCollection:*collection*] ⇨ *(none)*

This method is inherited from LibraryContainer with no change in keywords and defaults, except the user can only be a title container, not a library container—refer to that class for details. Do not call init directly on an instance—it is automatically called by the new method.

Class Methods

Inherited from Collection:

pipe

Inherited from StorageContainer:

open

Inherited from LibraryContainer:

open

The following class methods are defined in AccessoryContainer:

open (LibraryContainer)

open *self* [dir:*dirRep*] path:*collection* [mode:*name*]
[user:*titleContainer*] ⇨ AccessoryContainer

Similar to the open method in LibraryContainer, except it opens and returns an accessory container.

To summarize, this method opens the accessory, calls the function in preStartupAction, loads its libraries instance variable, and makes itself a user of each library. If the user keyword is supplied, this method then calls addUser on the supplied title container with *self* as its new user. This method then adds the accessory to theOpenContainers global variable, and finally calls the function in startupAction.

Most of the implementation is inherited from LibraryContainer, with no change in keywords or defaults. Refer to open in that class for more details.

Note – When a user chooses **Open Accessory** from the menu and chooses an accessory, the open method is called on the accessory with theTitleContainer as its user. It then calls isAppropriateAccessory: If true is returned, it calls addAccessory; if false, the accessory is not added to the title, but is left open.

Instance Variables

Inherited from Collection:

bounded	maxSize	size
iteratorClass	minSize	uniformity
keyEqualComparator	mutable	uniformityClass
keyUniformity	mutableCopyClass	valueEqualComparator
keyUniformityClass	proprietored	

Inherited from IndirectCollection:

targetCollection

Inherited from LibraryContainer:

copyright	preStartupAction	users
directory	startupAction	version
libraries	terminateAction	
name	type	

Instance Methods

Inherited from Collection:

add	forEach	iterate
addMany	forEachBinding	localEqual
addToContents	getAll	map
chooseAll	getAny	merge
chooseOne	getKeyAll	pipe
chooseOneBinding	getKeyOne	prin
deleteAll	getMany	removeAll
deleteBindingAll	getOne	removeOne
deleteBindingOne	hasBinding	setAll
deleteKeyAll	hasKey	setOne
deleteKeyOne	intersects	size
deleteOne	isEmpty	
emptyOut	isMember	

Inherited from IndirectCollection:

isAppropriateObject	objectAdded	objectRemoved

Inherited from StorageContainer:

close	update
requestPurgeForAllObjects	

Inherited from LibraryContainer:

addUser	objectAdded	terminate
close	recurPrin	
isAppropriateObject	removeUser	

Since an AccessoryContainer object is an indirect collection, you can also use any methods defined in the class specified by targetCollection. The target collection is by default an instance of Array, which inherits from Sequence; in this case, the following instance methods are accessible:

Accessible from LinearCollection:

chooseOneBackwards	findRange	getNthKey
chooseOrdOne	forEachBackwards	getOrdOne
deleteFirst	getFirst	getRange
deleteLast	getLast	localEqual
deleteNth	getMiddle	localLT
deleteRange	getNth	pop

Accessible from Sequence by redirection:

addFifth	moveBackward	setFourth
addFirst	moveForward	setLast
addFourth	moveToBack	setNth
addNth	moveToFront	setSecond
addSecond	prepend	setThird
addThird	prependNew	sort
append	setFifth	
appendNew	setFirst	

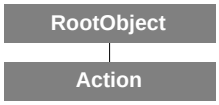
The following instance method is defined in AccessoryContainer:

getAccessory

`getAccessory` *self* ⇨ `Collection`

Returns objects from the accessory *self*. This method is automatically called by `addAccessory` (defined in `TitleContainer`). This method has an empty implementation in `AccessoryContainer`; you must override it in a subclass to do something useful. You should implement this method to return a collection of whatever objects the accessory should pass to the title when `addAccessory` is called. It might typically return an array containing clocks, players, and presenters. For example, a tape measure accessory might create an array of all the objects that make up a new tape measure and return that array.

Action



Class type: Core class (abstract)
Resides in: ScriptX and KMP executables
Inherits from: `RootObject`
Component: Animation

`Action` is a noninstantiable superclass for the `Action` family of classes. All objects in the `Action` family have a target object (`targetNum`), a time when the action occurs (`time`), and a `trigger` method to begin the action.

Subclasses of `Action` provide a variety of possible actions to change an object's bitmap, position, or path, or to run a script.

Instance Variables

authorData

self`.authorData` (read-write) *(object)*

You may put whatever data you desire here. Otherwise, the value is `undefined`.

playOnly

self`.playOnly` (read-write) `Boolean`

If `true`, the action *self* is triggered only while playing, not while "fast forwarding" or otherwise jumping in time. It is important to set `playOnly` to `true` for actions that you want to happen only while playing, and not while "fast forwarding".

For example, let's say you have an action list with an action at time `t1` that causes the player to jump back to an earlier time. With `playOnly` set to `false`, doing a time jump forward beyond time `t1` would be impossible, because trying to jump past `t1` would trigger the backward time jump and send the player backward. However, with `playOnly` set to `true`, such an action would be ignored during the "fast forward" phase.

targetNum

self`.targetNum` (read-write) `Integer`

The index of the target object to perform the action *self* on. This index is within the list of targets held by the `ActionListPlayer` object.

time

self`.time` (read-write) `Integer`

The time at which the action *self* should happen, in ticks of the `ActionListPlayer` object. A time of 0 ticks corresponds to the start of the action list player. Converting the time to seconds depends on whether the action list player is slaved to other clocks. In the simplest case, if the action list player has no master clock (its `masterClock` is set to `undefined`), the time in seconds is calculated by `time`/`scale`, where `scale` is the scale of the action list player.

Because the time relative to the start of the action list player, the time value of an action on an `ActionList` object does not change if an action is added ahead of it, or if the scale is changed.

Instance Methods

trigger

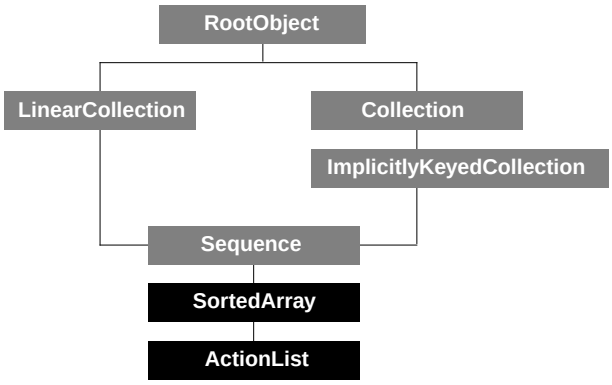
`trigger` *self target player* ⇨ *self*

self	`Action` object
target	Any object
player	`ActionListPlayer` object

Causes the action *self* to happen on the given *target* object. This method is called by the `ActionListPlayer` object *player* when it's time to execute this action.

The action should affect the *target* object. When the action list player calls `trigger`, the value for *target* is automatically determined by taking the `targetNum` instance variable and finding the object in the corresponding slot in the player's target list. If the `targetNum` instance variable is out of range (that is, less than 1 or greater than the size of the target list), the value of the *target* is `empty`. Also, if there is no target in a certain slot in the target list yet, the value of *target* is `undefined`.

ActionList



Class type: Core class (concrete)
Resides in: ScriptX and KMP executables
Inherits from: `SortedArray`
Component: Animation

`ActionList` class represents a collection of actions used by the `ActionListPlayer` object to control target objects over time. The action is any instance of the `Action` family of classes. Actions in an action list are sorted and keyed by time.

Creating and Initializing a New Instance

The following script is an example of how to create a `new` instance of the `ActionList` class:

```
myActionList:= new ActionList
```

The variable `myActionList` contains the initialized action list. The `new` method uses the keywords defined in `init`.

init

`init` *self* [`initialSize:`*integer*] [`growable:`*boolean*] ⇨ *(none)*

self `ActionList` object

Superclass `SortedArray` uses the following keywords:

`initialSize:`	`Integer` object
`growable:`	`Boolean` object

Initializes the `ActionList` object, which is a `SortedArray` object with an initial size of 300. Do not call `init` directly on an instance—it is called automatically by the `new` method.

If you omit an optional keyword, its default value is used. The defaults are:

```
initialSize:300
growable:true
```

Class Methods

Inherited from `Collection`:

`pipe`

Instance Variables

Inherited from `Collection`:

`bounded`	`maxSize`	`size`
`iteratorClass`	`minSize`	`uniformity`
`keyEqualComparator`	`mutable`	`uniformityClass`
`keyUniformity`	`mutableCopyClass`	`valueEqualComparator`
`keyUniformityClass`	`proprietored`	

The following instance variables are defined in `ActionList`:

duration

self`.duration` (read-only) `Integer`

Specifies the total time to play all the actions on the list, in the action list player's units of time. For example, if the action list player's master clock is set to `undefined`, then to get the time in seconds, you calculate `time/scale`, where `scale` is the scale of the action list player.

Instance Methods

Inherited from `Collection`:

`add`	`forEach`	`iterate`
`addMany`	`forEachBinding`	`localEqual`
`addToContents`	`getAll`	`map`
`chooseAll`	`getAny`	`merge`
`chooseOne`	`getKeyAll`	`pipe`
`chooseOneBinding`	`getKeyOne`	`prin`
`deleteAll`	`getMany`	`removeAll`
`deleteBindingAll`	`getOne`	`removeOne`
`deleteBindingOne`	`hasBinding`	`setAll`
`deleteKeyAll`	`hasKey`	`setOne`
`deleteKeyOne`	`intersects`	`size`
`deleteOne`	`isEmpty`	
`emptyOut`	`isMember`	

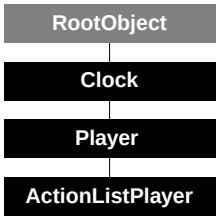
Inherited from `LinearCollection`:

`chooseOneBackwards`	`findRange`	`getNthKey`
`chooseOrdOne`	`forEachBackwards`	`getOrdOne`
`deleteFirst`	`getFirst`	`getRange`
`deleteLast`	`getLast`	`localEqual`
`deleteNth`	`getMiddle`	`localLT`
`deleteRange`	`getNth`	`pop`

Inherited from `Sequence`:

`addFifth`	`moveBackward`	`setFourth`
`addFirst`	`moveForward`	`setLast`
`addFourth`	`moveToBack`	`setNth`
`addNth`	`moveToFront`	`setSecond`
`addSecond`	`prepend`	`setThird`
`addThird`	`prependNew`	`sort`
`append`	`setFifth`	
`appendNew`	`setFirst`	

ActionListPlayer



Class type: Core class (concrete)
Resides in: ScriptX and KMP executables
Inherits from: `Player`
Component: Animation

The `ActionListPlayer` class represents a player that plays the actions on an action list over time. The "media" of this player is the `ActionList` object. At the time indicated in each `Action` object, the `ActionListPlayer` object calls the `trigger` method on the action, which then causes the action to affect its target object. The action can be an instance of any subclass of `Action`, such as `ScriptAction`, `PathAction`, or `ShapeAction`.

The `ActionListPlayer` object also has a list of target objects. Since actions refer only to an index into the list of target objects, an action list can be reused for different target objects.

You can fast forward or rewind an `ActionListPlayer` object. Fast forward simply plays all actions quickly up to the specified time. However, rewinding is not so simple. Rewinding is accomplished by rewinding to the very beginning of the action list and fast forwarding to the specified time; this process can be slow for long action lists, but is necessary to reconstruct the state. If you can rewind to a time when a target object should not exist, you must make sure that the object is disposed of during the rewind. You can do this by adding objects to the target list using a `TargetListAction` object and specifying a dispose function for `rewindScript` (defined in `TargetListAction`). Then, during any rewind, all of the objects on the target list are immediately disposed of, before playing the action list player from the beginning of the action list.

Note that an `ActionListPlayer` object can operate inside of a `GroupPresenter` object, so that the objects being controlled by the `ActionListPlayer` object can be moved as a contiguous group, or can, for example, respond as a group to a mouse down event. In this case, the actions on the action list continue to operate in the local coordinate system of the `GroupPresenter` object.

Creating and Initializing a New Instance

The following script is an example of how to create a new instance of the `ActionListPlayer` class:

```
alp:= new ActionListPlayer \
       ActionListPlayer ActionList) \
       targetCount:10
```

The variable `alp` contains an initialized instance of the action list player with room for 10 targets that it can act on. The `new` method uses the keywords defined in `init`.

init

`init` *self* `actionList:`*actionList* [`targetCount:`*integer*]
[`targets:`*sequence*] [`masterClock:`*clock*] [`scale:`*number*] ⇨ *(none)*

self	`ActionListPlayer` object
`actionList:`	`ActionList` object
`targetCount:`	Maximum number of targets in the target list. Note that if you supply the target list you do not also need to supply this number
`targets:`	`Array` object listing the targets for the player

Superclass `Clock` uses the following keywords:

`masterClock:`	`Clock` object to use as the master clock for the player
`scale:`	`Integer` object to use as the scale for the player

Initializes the `ActionListPlayer` object *self*. If no `actionList` is supplied, this method reports an exception. If `targets` is supplied, sets the `targets` instance variable to the given target list.

If `targets` is not supplied, the method uses `targetCount` to determine the number of items in the target list. If both `targets` and `targetCount` are not supplied, the method reports an exception. Do not call `init` directly on an instance—it is automatically called by the `new` method.

If you omit an optional keyword, its default value is used. The defaults are:

```
targetCount:50
targets:(new Array initialSize:50 growable:true)
masterClock:undefined
scale:1
```

Instance Variables

Inherited from `Clock`:

`callbacks`	`rate`	`ticks`
`effectiveRate`	`resolution`	`time`
`masterClock`	`scale`	`title`
`offset`	`slaveClocks`	

Inherited from `Player`:

`audioMuted`	`globalContrast`	`globalVolumeOffset`
`dataRate`	`globalHue`	`markerList`
`duration`	`globalPanOffset`	`status`
`globalBrightness`	`globalSaturation`	`videoBlanked`

The following instance variables are defined in `ActionListPlayer`:

actionList

self`.actionList` (read-write) `ActionList`

Specifies the action list being played back (this player's media).

authorData

self`.authorData` (read-write) *(object)*

The author may put whatever data they desire here. Otherwise, the value is `undefined`.

rewindScripts

self.`rewindScripts` (read-write) `Array`

Specifies an initially empty array that gets filled up with functions from the `rewindScript` instance variable of instances of `TargetListAction`, as the action list player plays. The `rewindScripts` instance variable is automatically maintained—you shouldn't directly add or remove items from it.

targets

self.`targets` (read-write) `Sequence`

Specifies an array of target objects. Actions refer to targets on this list by specifying an index into the list.

Instance Methods

Inherited from `Clock`:

`addPeriodicCallback`	`clockAdded`	`pause`
`addRateCallback`	`clockRemoved`	`resume`
`addScaleCallback`	`effectiveRateChanged`	`timeJumped`
`addTimeCallback`	`forEachSlave`	`waitTime`
`addTimeJumpCallback`	`isAppropriateClock`	`waitUntil`

Inherited from `Player`:

`addMarker`	`goToBegin`	`playPrepare`
`eject`	`goToEnd`	`playUnprepare`
`fastForward`	`goToMarkerFinish`	`playUntil`
`getMarker`	`goToMarkerStart`	`resume`
`getNextMarker`	`pause`	`rewind`
`getPreviousMarker`	`play`	`stop`

The following instance methods are defined in `ActionListPlayer`:

getMuteChannel

`getMuteChannel` *self targetNum* ⇨ `Boolean`

self	`ActionListPlayer` object
targetNum	`Number` object

Returns `true` if the target channel specified by *targetNum* is currently muted; otherwise, it returns `false`. When a channel is muted, then all actions for that sprite are ignored (that is, not executed).

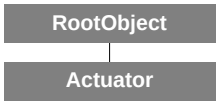
setMuteChannel

`setMuteChannel` *self targetNum onOff* ⇨ *onOff*

self	`ActionListPlayer` object
targetNum	`Number` object
onOff	`Boolean` object

Sets the target channel given by *targetNum* to be muted if the value of *onOff* is `true`. Returns the same Boolean value that was passed in.

Actuator



Class type: Core class (abstract)
Resides in: ScriptX and KMP executables
Inherits from: `RootObject`
Component: User Interface

The `Actuator` class provides the basic protocol for interacting with `PushButton` and `Toggle` objects. Think of an actuator as an object that passes through different states, such as *pressed*, *released*, and *disabled*.

`Actuator` defines six methods which are generally specialized by its subclasses: `activate`, `multiActivate`, `press`, `release`, `toggleOn`, and `toggleOff`. These methods, which correspond to changes in the actuator's state, are meant to be called as actions from a controller. As defined by `Actuator`, these methods do nothing but manage the value of the `pressed` and `toggledOn` instance variables.

The `activate`, `press`, and `release` methods are called automatically by a controller whenever there is a corresponding change in state. The `multiActivate` method is really a special case of `activate`, and is called automatically when multiple mouse clicks occur within the double-click time interval. The `toggleOn` and `toggleOff` methods are called indirectly by the `activate` method in the `Toggle` class. Since they are defined by `Actuator`, a developer can create an actuator with toggle-like behavior that does not inherit from `Toggle`.

An actuator is associated with a controller, generally an instance of `ActuatorController`, from which it derives its behavior. This controller is associated with a set of interests in mouse events. It actively manages these interests, receiving and processing mouse events, and responds by calling the `activate`, `multiActivate`, `press`, and `release` methods on the actuators it controls. For more information, see the definition of `ActuatorController`.

To be controlled by an `ActuatorController` object, an actuator must be attached to that controller, and to the space that it controls. A controller is always attached to a space—it can control only objects that are contained in its space. To be controlled by an actuator controller, an actuator must first be added to that space. Then, depending on the value of `wholeSpace`, it must be added to the controller as well.

If the value of `wholeSpace` is `true`, an actuator controller automatically controls all actuators that are added to its space. If the value of `wholeSpace` is `false`, then the actuator must be explicitly added to the actuator controller. The controller classes, including `ActuatorController`, inherit from `IndirectCollection`. A controller uses this collection behavior to maintain a list of objects that it controls.

Although the `PushButton` and `Toggle` classes both perform hit testing within a rectangular region, it is possible to define a subclass of `Actuator` that performs more precise hit testing on non-rectangular objects such as bitmaps. For example, the author could define a subclass of `Actuator` mixed with `TwoDShape` using a `Bitmap` as a target. The boundary of the resulting object is based on the bitmap. For more information, see the discussion of hit testing in the "User Interface" chapter of the *ScriptX Components Guide*.

`Actuator` is not a `TwoDPresenter` class. An author can create a concrete subclass of `Actuator`, but it must inherit from both `Actuator` and a `TwoDPresenter` class if it is to be controlled by an `ActuatorController` object.

Creating and Initializing a New Instance

You cannot create an instance of `Actuator`—it is an abstract class. However, `Actuator` has an `init` method for initializing instances of its concrete subclasses.

init

`init` *self* [`enabled:`*boolean*] [`menu:`*menu*] ⇨ *(none)*

self	`Actuator` object
`enabled:`	`Boolean` object
`menu:`	`Menu` object to present when the actuator is activated

Initializes the `Actuator` object *self*, applying values supplied with the keyword arguments to the instance variables of the same name. Do not call `init` directly on an instance—it is automatically called by the `new` method.

If you omit an optional keyword, its default value is used. The defaults are:

```
enabled:true
menu:undefined
```

Instance Variables

enabled

self`.enabled` (read-write) `Boolean`

Specifies whether the actuator *self* can respond to user input. If the value of `enabled` is `false`, then `press`, `release`, `activate`, and `multiActivate` should return without taking action.

Subclasses of `Actuator` may take some action to indicate that an actuator is not enabled. For example, the `Pushbutton` class specifies a presenter, stored in its `disabledPresenter` instance variable, that is displayed whenever the value of `enabled` is `false`.

menu

self`.menu` (read-write) `Menu`

Specifies the menu to present when the actuator *self* is activated.

pressed

self`.pressed` (read-only) `Boolean`

Specifies whether the actuator *self* is currently pressed.

toggledOn

self`.toggledOn` (read-write) `Boolean`

Specifies whether the actuator *self* is currently toggled on.

Instance Methods

activate

`activate` *self* ⇨ *self*

Sets the value of `pressed` on the actuator *self* to `false`. Subclasses of `Actuator` generally specialize this method. Although `activate` can be called directly from a script, it is usually the controller associated with an actuator that calls `activate`.

multiActivate

`multiActivate` *self numberOfClicks* ⇨ *self*

self	`Actuator` object
numberOfClicks	`Integer` object indicating number of clicks

Sets the value of `pressed` on the actuator *self* to `false`. Subclasses of `Actuator` may specialize this method to perform an action associated with multiple mouse clicks. In a class of actuators that does not define a separate and distinct response to multiple mouse clicks, `multiActivate` can be redirected to call `activate` one or more times. Although `multiActivate` can be called directly from a script, it is usually the controller associated with an actuator that calls `multiActivate`.

A call to `multiActivate` tells the actuator to take the action defined by that actuator if it receives *numberOfClicks* mouse clicks, each within the double-click interval of the last click. (`ActuatorController` class queries the system to obtain the system defined double-click threshold.)

press

`press` *self* ⇨ *self*

Sets the value of `pressed` on the actuator *self* to `true`. Subclasses of `Actuator` generally specialize this method. Although `press` can be called directly from a script, it is usually the controller associated with an actuator that calls `press`.

release

`release` *self* ⇨ *self*

Sets the value of `pressed` on the actuator *self* to `false`. Subclasses of `Actuator` generally specialize this method. Although `release` can be called directly from a script, it is usually the controller associated with an actuator that calls `release`.

A call to `release` tells an actuator, formerly in its pressed state, that it has been released without being activated. This can happen, for example, if the user clicks the mouse over an actuator and then slides the mouse away before releasing the mouse button, so that the actuator is released without being activated.

toggleOff

`toggleOff` *self* ⇨ *self*

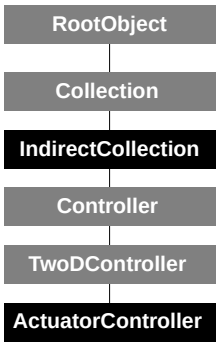
Sets the value of `toggledOn` on the actuator *self* to `false`. Any subclass of `Actuator` (such as `Toggle`) that maintains an on or off state should specialize `activate` to call `toggleOn` and `toggleOff`, as appropriate.

toggleOn

`toggleOn` *self* ⇨ *self*

Sets the value of `toggledOn` on the actuator *self* to `true`. Any subclass of `Actuator` (such as `Toggle`) that maintains an on or off state should specialize `activate` to call `toggleOn` and `toggleOff`, as appropriate.

ActuatorController



Class type: Core class (concrete)
Resides in: ScriptX and KMP executables
Inherits from: `TwoDController`
Component: User Interface

The `ActuatorController` class manages actuators (objects subclassed from `Actuator`) in the space it controls, including instances of `PushButton` and `Toggle` and many of the Widget Kit classes. This controller is partially responsible for the "feel" aspect of "look and feel" in these classes.

An `ActuatorController` object is a collection of the actuators it controls. These actuators must also be in the space that the controller is controlling. Actuators are either automatically or manually added to the actuator controller, according to the `wholeSpace` instance variable. If `wholeSpace` is `false`, you can use the methods defined by `Collection` to add and remove objects from the actuator controller. To ensure that only actuators are added to an actuator controller, the `protocols` instance variable is set to the `Actuator` class. See the `Controller` class for descriptions of `wholeSpace`, `protocols`, and other general properties of controllers. See also the section "Hit Testing" in the User Interface chapter in the *ScriptX Components Guide*.

An actuator controller maps mouse-down events to the `press` method on `Actuator` objects. It also maps mouse-move events to the `release` and `press` methods as the mouse moves in and out of a button while the mouse button remains down, and it maps mouse-up events to the `activate` method when appropriate.

`ActuatorController` defines three instance variables: `activateInterest`, `pressInterest`, and `releaseInterest`. These variables store event interests—the `ActuatorController` class is interested only in mouse events.

Corresponding with these interests, the class defines three instance methods: `processActivate`, `processPress`, and `processRelease`. These methods are event receivers, and they are associated automatically as receivers for their associated event interests. For more information, see the "Events and Input Devices" chapter in the *ScriptX Components Guide*.

Table 1 shows the correspondence of instance variables and methods defined by actuator controllers and actuators.

Table 1: How ActuatorController and Actuator objects work together

ActuatorController		Actuator		
(instance variable)	(instance method)	(instance method)	(instance method)	(instance variable)
`activateInterest`	`processActivate`	`activate`	`handleActivate`	`activateAction`
		`multiActivate`	`handleMultiActivate`	`multiActivateAction`
`pressInterest`	`processPress`	`press`	`handlePress`	`pressAction`
`releaseInterest`	`processRelease`	`release`	`handleRelease`	`releaseAction`

Note that the ActuatorController class manages both the PushButton and Toggle classes and many of the Widget Kit classes such as the GenericButton classes. The Actuator class defines the toggledOn instance variable, as well as the toggleOn and toggleOff methods. However, it is the Toggle class that redefines these methods to behave like toggle buttons. When an actuator controller calls activate on a toggle, it is not concerned with whether the toggle is currently on or off. The Toggle class specializes activate to support these two states.

Table 2 shows the correspondence of instance variables and methods defined by actuator controllers and toggles, which are a specialized class of actuators.

Table 2: How ActuatorController and Toggle objects work together

ActuatorController		Toggle		
(instance variable)	(instance method)	(instance method)	(instance method)	(instance variable)
activateInterest	processActivate	activate	handleActivate	activateAction
			toggleOn	
			toggleOff	
		multiActivate	handleMultiActivate	multiActivateAction
pressInterest	processPress	press	handlePress	pressAction
releaseInterest	processRelease	release	handleRelease	releaseAction

Toggle inherits its treatment of multiActivate from PushButton, although toggle buttons do not respond to multiple clicks in most conventional user interface designs. A developer might decide to ignore multiple clicks, or to interpret a double click as two single clicks.

The ActuatorController class has the same interface with Toggle as with PushButton. The Toggle class defines the methods toggleOn and toggleOff, and it specializes the activate and handleActivate methods, which are defined by Actuator or PushButton.

Creating and Initializing a New Instance

The following script creates a new instance of the ActuatorController class. First create the control space, and then create the actuator controller:

```
mySpace := new TwoDSpace boundary:(new Rect x2:600 y2:400)
myController := new ActuatorController \
                    space:mySpace
```

The variable myController contains an initialized instance of an actuator controller that operates on actuators in mySpace. The new method uses keywords defined in init.

init

`init` *self* [`space:`*space*] [`wholeSpace:`*boolean*] [`enabled:`*boolean*]
[`targetCollection:`*sequence*] [`menu:@unsupplied`] ⇨ *(none)*

self	ActuatorController object
menu	Menu object

The superclass Controller uses the following keywords:

space:	Space object
wholeSpace:	Boolean object
enabled:	Boolean object

The superclass `TwoDController` defines the following keyword:

`targetCollection:` `Sequence` object (use with caution)

Initializes the `ActuatorController` object *self*, applying the keyword arguments to instance variables of the same name. Use discretion in changing the target collection; for more information, see the definition of the `TwoDController` class. Do not supply a value for the `menu` keyword; when you create a new `Menu` instance, the new `Menu` object automatically creates a new `ActuatorController` object and passes itself as the value of the `menu` keyword. Do not call `init` directly on an instance — it is automatically called by the `new` method.

If you omit an optional keyword, its default value is used. The defaults are:

```
space:undefined
wholeSpace:false
enabled:true
targetCollection:(new Array initialSize:14 growable:true)
menu:@unsupplied
```

Class Methods

Inherited from `Collection`:

`pipe`

Instance Variables

Inherited from `Collection`:

`bounded`	`maxSize`	`size`
`iteratorClass`	`minSize`	`uniformity`
`keyEqualComparator`	`mutable`	`uniformityClass`
`keyUniformity`	`mutableCopyClass`	`valueEqualComparator`
`keyUniformityClass`	`proprietored`	

Inherited from `IndirectCollection`:

`targetCollection`

Inherited from `Controller`:

`enabled`	`space`	`wholeSpace`
`protocols`		

The following instance variables are defined in `ActuatorController`:

activateInterest

self`.activateInterest` (read-write) `MouseEvent`

Specifies an event interest that is registered by the actuator controller *self*. Its value must be an instance of `MouseEvent`. This variable defaults to an instance of `MouseUpEvent` that is associated with the first mouse button (`@mousebutton1`). The release interest is the matched interest of the activate interest. For more information on matching interests and mouse events, see the discussion of the `matchedInterest` instance variable, as specialized by `MouseUpEvent`, in the "Events and Input Devices" chapter in the *ScriptX Components Guide*. The `processActivate` method is the receiver for this interest.

doubleClickTime

self.doubleClickTime (read-write) Integer

Specifies the threshold of time within which the actuator controller *self* interprets two mouse clicks as a multiple click. Each actuator controller stores its own double-click time. An actuator controller interprets two mouse clicks within the double-click time as a double click, three mouse clicks within two double-click times as a triple click, and so forth. When an actuator controller is instantiated, the `ActuatorController` class queries the system for the system's double-click time. Thereafter, a script may modify this value, but it will not be reflected in the system's double-click time. Nor does a change to the system's double-click time update the value stored by an actuator controller.

pressInterest

self.pressInterest (read-write) MouseEvent

Specifies an event interest that is registered by the actuator controller *self*. The value must be an instance of `MouseEvent`. This variable defaults to an instance of `MouseDownEvent` that is associated with the first mouse button (`@mousebutton1`). The `processPress` method is the receiver for this interest.

protocols (Controller)

self.protocols (read-write) Array

Specifies the required protocols for the actuator controller *self*. For instances of `ActuatorController`, this array contains the classes `Actuator` and `TwoDPresenter`. Any object added to an actuator controller must include these two classes among its superclasses. See the `Controller` class for a further description of `protocols`.

releaseInterest

self.releaseInterest (read-write) MouseEvent

Specifies an event interest that is registered by the actuator controller *self*. The value must be an instance of `MouseEvent`. This variable defaults to an instance of `MouseMoveEvent` that is associated with the first mouse button (`@mousebutton1`). The `processRelease` method is the receiver for this interest.

Instance Methods

Inherited from `Collection`:

add	forEach	iterate
addMany	forEachBinding	localEqual
addToContents	getAll	map
chooseAll	getAny	merge
chooseOne	getKeyAll	pipe
chooseOneBinding	getKeyOne	prin
deleteAll	getMany	removeAll
deleteBindingAll	getOne	removeOne
deleteBindingOne	hasBinding	setAll
deleteKeyAll	hasKey	setOne
deleteKeyOne	intersects	size
deleteOne	isEmpty	
emptyOut	isMember	

Inherited from `IndirectCollection`:

isAppropriateObject	objectAdded	objectRemoved

Inherited from `Controller`:

`isAppropriateObject`	`tickle`

Since an `ActuatorController` object is an indirect collection, you can also use any methods defined in the class specified by `targetCollection`. The target collection is typically an instance of `Array`, which inherits from `Sequence`, so the following instance methods are redirected to this controller.

Accessible from `LinearCollection`:

`chooseOneBackwards`	`findRange`	`getNthKey`
`chooseOrdOne`	`forEachBackwards`	`getOrdOne`
`deleteFirst`	`getFirst`	`getRange`
`deleteLast`	`getLast`	`localEqual`
`deleteNth`	`getMiddle`	`localLT`
`deleteRange`	`getNth`	`pop`

Accessible from `Sequence`:

`addFifth`	`moveBackward`	`setFourth`
`addFirst`	`moveForward`	`setLast`
`addFourth`	`moveToBack`	`setNth`
`addNth`	`moveToFront`	`setSecond`
`addSecond`	`prepend`	`setThird`
`addThird`	`prependNew`	`sort`
`append`	`setFifth`	
`appendNew`	`setFirst`	

The following instance methods are defined in `ActuatorController`:

processActivate

`processActivate` *self interest event* ⇨ *self*

self	`ActuatorController` object
interest	`MouseEvent` object, the `activateInterest` of *self*
event	`MouseEvent` object that matches *interest*

If `wholeSpace` is `true`, `processActivate` looks through the collection of actuators controlled by the actuator controller *self* to determine which actuator was hit. If `wholeSpace` is `false`, `processActivate` determines which actuator was hit by examining the `presenter` instance variable of the *event* that matched its event interest.

If an actuator was hit, `processActivate` calls `activate` or `multiActivate` on that actuator. This invokes the associated menu or triggers the associated action. An actuator is considered hit if an *event* that matches the corresponding *interest* occurs within its boundary. If no actuator was hit, `processActivate` rejects the event.

The `processActivate` method interprets a set of clicks that occur within the double-click time as a multiple click, a gesture that is distinct from a series of single clicks. When `processActivate` receives the first `MouseUpEvent` instance, it calls `activate` on the actuator. For the second mouse-up, it calls either `activate` or `multiActivate`, depending on the time that has elapsed. In this way, two mouse clicks are interpreted as a single gesture if they are close together in time. For more information on multiple clicking, see the "User Interface" chapter of *ScriptX Components Guide*.

Do not call `processActivate` from the scripter. It is triggered automatically, and is visible to the scripter so that it can be specialized. The `processActivate` method is the event receiver for `activateInterest`.

processPress

`processPress` *self interest event* ⇨*self*

self	`ActuatorController` object
interest	`MouseEvent` object, the `pressInterest` of *self*
event	`MouseEvent` object that matches *interest*

If `wholeSpace` is `true`, `processPress` looks through the collection of actuators controlled by the actuator controller *self* to determine which actuator was hit. If `wholeSpace` is `false`, `processPress` determines which actuator was hit by examining the `presenter` instance variable of *event*.

If an actuator was hit, `processPress` calls `press` on that actuator. This invokes the associated menu or triggers the associated action. An actuator is considered hit if an *event* occurs within its boundary that matches the corresponding *interest*. If no actuator was hit, then `processPress` rejects the event.

Do not call `processPress` from the scripter. It is triggered automatically, and is visible to the scripter so that it can be specialized. The method `processPress` is the event receiver for `pressInterest`.

processRelease

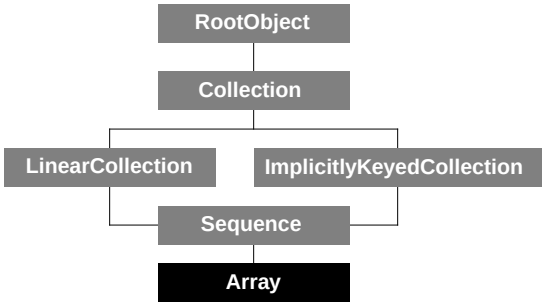
`processRelease` *self interest event* ⇨*self*

self	`ActuatorController` object
interest	`MouseEvent` object, the `releaseInterest` of *self*
event	`MouseEvent` object that matches *interest*

Whatever the value of `wholeSpace` may be, `processRelease` looks through the collection of actuators controlled by the actuator controller *self* to determine if an actuator was hit. If an actuator was hit, `processRelease` calls `release` on that actuator. This releases the associated menu or triggers the associated action. An actuator is considered hit if an *event* occurs that matches the corresponding *interest*. If no actuator was hit, then `processRelease` rejects the event.

Do not call `processRelease` from the scripter. It is called automatically, and is visible to the scripter so that it can be specialized. The `processRelease` method is the event receiver for `releaseInterest`.

Array



Class type: Core class (concrete)
Resides in: ScriptX and KMP executables
Inherits from: Sequence
Component: Collections

The `Array` class provides a collection that stores a list of values as contiguous elements in one continuous data structure. Since variable assignment in ScriptX is pointer-based, an array actually contains a list of pointers to objects. With the exception of immediate objects, elements of an array are stored elsewhere.

The ScriptX language creates an instance of `Array` automatically when it encounters a list of elements enclosed by the array literal. For more information, see the *ScriptX Language Guide*.

When an `Array` object is created (with either the `new` method or the `object` expression), memory is allocated to support the number of elements designated by `initialSize`. Note that `initialSize` is a keyword, not an instance variable. Once an array is initialized, `initialSize` becomes a fixed property of that array, and is invisible to the scripter.

Distinguish between `size` and `initialSize`. The `size` property is a virtual instance variable that returns the number of elements currently in the array. Although `initialSize` may be large, `size` returns 0 on a newly allocated array, since there are initially no elements stored in the array.

As you add elements to an array, the array becomes filled. Once it is full, the `growable` keyword determines what happens next. If `growable` was set `true` when the array was instantiated, then adding elements beyond the number specified by `initialSize` extends the array to encompass a larger block of memory. If `growable` is `false`, the array reports the `bounded` exception. Think of the growable keyword as an option that creates a bounded array, with `maxSize` set to `initialSize`.

```
myArray := new Array initialSize:10 growable:false
myArray.bounded
true
myArray.maxSize
10
```

Typically, appending new elements to the end of an array is very fast. However, when an array is extended beyond its initial size, the Kaleida Media Player may be forced to allocate a new memory block and copy the elements of the array into the new block. This may have implications for performance. As an alternative to large arrays, consider specifying the `ArrayList` class, a hybrid of `Array` and `LinkedList`.

Array specializes several iterative methods that are defined by `Collection` and `LinearCollection` to allow for insertion and deletion of items without compromising the integrity of the iterator. Wholesale rearrangement of the collection is not supported. Although insertion and deletion will not create an exception, performance can be

expected to suffer if there are a large number of changes to the collection during any iterative process. Developers can be warned, through messages to the console stream, of changes to the array. Set the global function `warnings` to `true`. (The global function `warnings` is provided only in ScriptX, and is not included in the Kaleida Media Player.)

Creating and Initializing a New Instance

Instances of the class `Array` are created in two ways. The first way is to use the method `new`, which is exemplified in the following script: :

```
myArray := new Array \
              initialSize:100 \
              growable:true
```

The variable `myArray` contains the initialized array. The instance has an initial size of 100 items and can grow beyond 100 items, since it is `growable`. The `new` method uses the keywords defined in `init`.

The second way to create an instance of `Array` is to use the hash sign (#) to create an array literal:

```
anotherArray := #(3, 5, 7)
```

The variable `anotherArray` contains a growable array of default initial size with three values in it.

init

`init` *self* [`initialSize:`*integer*] [`growable:`*boolean*] ⇨ *self*

self	`Array` object
`initialSize:`	`Integer` object
`growable:`	`Boolean` object

Initializes the `Array` object *self*, applying the arguments as follows: The value supplied with `initialSize` is the amount of space to reserve for the initial set of items; it must be 1 or greater. If `growable` is set to `false`, the array cannot be expanded beyond the size set by `initialSize`. If `growable` is set to `true`, it grows in chunks.

If the array is not growable, then the `maxSize` instance variable contains the value that is passed in as the value of `initialSize`. Otherwise, `initialSize` is used as a basis for allocating additional chunks of memory. Do not call init directly on an instance—it is automatically called by the new method.

If you omit an optional keyword, the following defaults are used:

initialSize:20
growable:true

When `growable` is set to `true` on instantiation, ScriptX may create an `Array` object larger (but never smaller) than you specify with `initialSize`, because of its interaction with the memory manager at initialization time. However, when you set `growable` to `false`, the value supplied with `initialSize` is actually used and the instance variable `maxSize` is set to that same value.

Class Methods

Inherited from `Collection`:

`pipe`

Instance Variables

Inherited from Collection:

bounded	maxSize	size
iteratorClass	minSize	uniformity
keyEqualComparator	mutable	uniformityClass
keyUniformity	mutableCopyClass	valueEqualComparator
keyUniformityClass	proprietored	

Instance Methods

Inherited from Collection:

add	forEach	iterate
addMany	forEachBinding	localEqual
addToContents	getAll	map
chooseAll	getAny	merge
chooseOne	getKeyAll	pipe
chooseOneBinding	getKeyOne	prin
deleteAll	getMany	removeAll
deleteBindingAll	getOne	removeOne
deleteBindingOne	hasBinding	setAll
deleteKeyAll	hasKey	setOne
deleteKeyOne	intersects	size
deleteOne	isEmpty	
emptyOut	isMember	

Inherited from LinearCollection:

chooseOneBackwards	findRange	getNthKey
chooseOrdOne	forEachBackwards	getOrdOne
deleteFirst	getFirst	getRange
deleteLast	getLast	localEqual
deleteNth	getMiddle	localLT
deleteRange	getNth	pop

Inherited from Sequence:

addFifth	moveBackward	setFourth
addFirst	moveForward	setLast
addFourth	moveToBack	setNth
addNth	moveToFront	setSecond
addSecond	prepend	setThird
addThird	prependNew	sort
append	setFifth	
appendNew	setFirst	

The following instance methods are defined by Array:

chooseOne (Collection)

chooseOne *self func arg* ⇨ *(none)*

self	Collection object
func	An instance of a subclass of AbstractFunction
arg	Any object

Calls chooseOne exactly as it is defined by Collection, except that Array specializes chooseOne to allow for insertion and deletion of items, but not for rearrangement of items, during processing of the collection. If you set the global function warnings to true, the compiler will print a warning message to the debug stream when the contents of the array change during iteration.

chooseOneBackwards (LinearCollection)

`chooseOneBackwards` *self func arg* ⇨ *(none)*

self	`Collection` object
func	An instance of a subclass of `AbstractFunction`
arg	Any object

Calls `chooseOneBackwards` exactly as it is defined by `LinearCollection`, except that `Array` specializes `chooseOneBackwards` to allow for insertion and deletion of items, but not for rearrangement of items, during processing of the collection. If you set the global function `warnings` to `true`, the compiler will print a warning message to the `debug` stream when the contents of the array change during iteration.

forEach (Collection)

`forEach` *self func arg* ⇨ *(none)*

self	`Collection` object
func	An instance of a subclass of `AbstractFunction`
arg	Any object

Calls `forEach` exactly as it is defined by `Collection`, except that `Array` specializes `forEach` to allow for insertion and deletion of items, but not for rearrangement of items, during processing of the collection. If you set the global function `warnings` to `true`, the compiler will print a warning message to the `debug` stream when the contents of the array change during iteration.

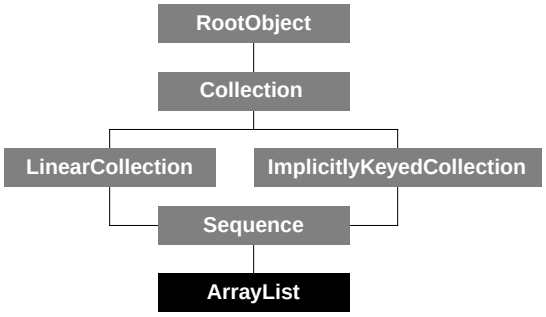
forEachBackwards (LinearCollection)

`forEachBackwards` *self func arg* ⇨ *(none)*

self	`Collection` object
func	An instance of a subclass of `AbstractFunction`
arg	Any object

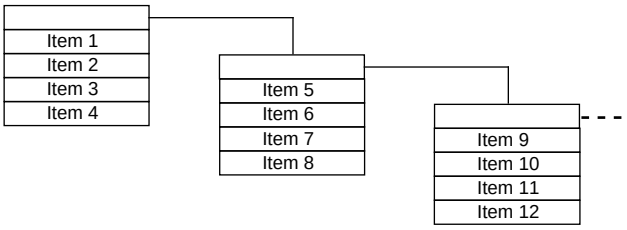
Calls `forEachBackwards` exactly as it is defined by `LinearCollection`, except that `Array` specializes `forEachBackwards` to allow for insertion and deletion of items, but not for rearrangement of items, during processing of the collection. If you set the global function `warnings` to `true`, the compiler will print a warning message to the `debug` stream when the contents of the array change during iteration.

ArrayList



Class type: Core class (concrete)
Resides in: ScriptX and KMP executables
Inherits from: Sequence
Component: Collections

An `ArrayList` object is a list of values, stored as a series of linked memory blocks. Compared to an `Array`, `ArrayList` may be faster for a collection that may grow rapidly, because the entire collection does not have to be copied when it grows beyond its original boundaries. Retrieving an object will be slightly slower for an `ArrayList`, however. As shown in the figure below, each block begins with a header that points to the next block. Rather than each item being separately linked to the next item, items are arranged consecutively, with each block linked to the next block.



When you create an `ArrayList` object (with `new`), you define the size of the first block with `initialSize`. Note, however, that the `size` instance variable will return 0 for a newly created array list, since it does not contain any objects. As you add items, the first memory block becomes full; once it is full and you add another item, it allocates a new block big enough to contain the number of elements specified by `growSize`, and places the new item in the new block. When this block becomes full, additional blocks of the same size are allocated to extend the array list.

Creating and Initializing a New Instance

The following script is an example of how to create a new instance of the `ArrayList` class:

```
al := new ArrayList \
       initialSize:100 \
       growSize:50
```

The variable `al` contains the initialized array. The instance is set to have space for 100 items in the initial set of items and to grow beyond that in increments of 50 items. Use the keyword arguments defined by the `init` method, shown below.

init

`init` *self* [`initialSize:`*integer*] [`growSize:`*integer*] ⇨ *(none)*

self	`ArrayList` object
`initialSize:`	`Integer` object
`growSize:`	`Integer` object

Initializes the `ArrayList` object *self*, applying the arguments as follows: The `initialSize` value is the amount of space to reserve for the initial set of items; it must be 1 or larger. The value supplied with `growSize` is the number of elements to grow by each time the allocated space for the array is full; `growSize` must be a positive number. If either argument is passed in as a value smaller than 1, it is set to 1. Thus an `ArrayList` object is never bounded and can always grow by at least 1. Do not call `init` directly on an instance—it is automatically called by the `new` method.

If you omit a keyword argument, the following defaults are used:

```
initialSize:20
growSize:20
```

ScriptX may create a larger `ArrayList` object (but never smaller) than you specify with `initialSize` or `growsize`, due to its interaction with memory management at initialization time.

Class Methods

Inherited from `Collection`:

`pipe`

Instance Variables

Inherited from `Collection`:

`bounded`	`maxSize`	`size`
`iteratorClass`	`minSize`	`uniformity`
`keyEqualComparator`	`mutable`	`uniformityClass`
`keyUniformity`	`mutableCopyClass`	`valueEqualComparator`
`keyUniformityClass`	`proprietored`	

Instance Methods

Inherited from `Collection`:

`add`	`forEach`	`iterate`
`addMany`	`forEachBinding`	`localEqual`
`addToContents`	`getAll`	`map`
`chooseAll`	`getAny`	`merge`
`chooseOne`	`getKeyAll`	`pipe`
`chooseOneBinding`	`getKeyOne`	`prin`
`deleteAll`	`getMany`	`removeAll`
`deleteBindingAll`	`getOne`	`removeOne`
`deleteBindingOne`	`hasBinding`	`setAll`
`deleteKeyAll`	`hasKey`	`setOne`
`deleteKeyOne`	`intersects`	`size`
`deleteOne`	`isEmpty`	
`emptyOut`	`isMember`	

Inherited from `LinearCollection`:

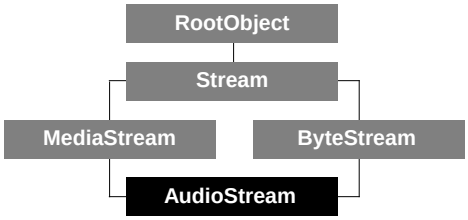
`chooseOneBackwards`	`findRange`	`getNthKey`
`chooseOrdOne`	`forEachBackwards`	`getOrdOne`

```
deleteFirst          getFirst          getRange
deleteLast           getLast           localEqual
deleteNth            getMiddle         localLT
deleteRange          getNth            pop
```

Inherited from Sequence:

```
addFifth             moveBackward      setFourth
addFirst             moveForward       setLast
addFourth            moveToBack        setNth
addNth               moveToFront       setSecond
addSecond            prepend           setThird
addThird             prependNew        sort
append               setFifth
appendNew            setFirst
```

AudioStream



Class type: Core class (concrete)
Resides in: ScriptX and KMP executables
Inherits from: `MediaStream` and `ByteStream`
Component: Media Players

The `AudioStream` class provides services for reading streams of digital sound samples.

The methods `isReadable` and `isSeekable` return `true` for an `AudioStream` instance, and `isWritable` returns false.

Creating and Initializing a New Instance

To create an `AudioStream` instance, import a file containing digitized audio data. The importing process automatically creates either an `AudioStream` instance or a `DigitalAudioPlayer` instance to play the audio stream.

The following script shows how to create an instance of `AudioStream` by importing an AIFF file containing digitized sound. The file "`growl`" would reside on the ScriptX startup directory. To play the digitized sound, you would need to create a `DigitalAudioPlayer` object and set its `mediaStream` instance variable to the audio stream.

```
stream1:= getstream theStartDir "growl" @readable
growlStream:= importMedia theImportExportEngine stream1\
          @sound @AIFF @stream
```

This script shows an example of how to import an AIFF file as an `AudioStream`. You can also import SND and WAVE files in the same manner, in which case you would need to change the `@AIFF` argument to `@SND` or `@WAVE` as appropriate. For more details of the arguments to the method `importMedia` on the global instance `theImportExportEngine`, please see either the "Media Stream Players" chapter in the *ScriptX Components Guide* or the chapter about Importers in the *ScriptX Developer's Guide*.

The `AudioStream` class does support the `new` method, which takes a keyword argument of `inputStream`, which needs to be a byte stream. You would not normally call the `new` method on `AudioStream` to create a new instance. Instead you would import an audio stream as described above. However, for the sake of completeness, the following script illustrates how to create a new instance of the `AudioStream` class by calling the `new` method. For this example, you would need to have previously created the byte stream `myStream`, which should contain digitized audio data.

```
myStream := new AudioStream \
            inputStream:myStream
```

The variable `myStream` points to the newly created audio stream, which has the `ByteStream` instance `myStream` as its input stream.

The `new` method uses the keywords defined in `init`.

init

`init` *self* [`inputStream:`*byteStream*] ⇨ *(none)*

self	`AudioStream` object
`inputStream:`	`ByteStream` object representing the digitized sound source

Initializes the `AudioStream` object *self*, setting `inputStream` as the data source. Do not call `init` directly on an instance—it is automatically called by the `new` method.

If you omit an optional keyword, its default value is used. The defaults are:

`inputStream:`	`undefined`

Instance Variables

Inherited from `MediaStream`:

`dataRate`	`rate`	`variableFrameSize`
`inputStream`	`sampleType`	
`markerList`	`scale`	

The following instance variables are defined in `AudioStream`:

numChannels

self`.numChannels` (read-only) `Integer`

Specifies the number of channels per frame of audio in the stream *self* —for example, 2 for stereo and 4 for quad.

pitch

self`.pitch` (read-only) `Integer`

Specifies the pitch of the sound in the stream *self*. The value returned is a MIDI note number. This variable is generally meaningful on short, highly controlled sound samples.

sampleType

self`.sampleType` (read-only) `NameClass`

Specifies how the audio data in the stream *self* is to be presented. The audio stream *self* coerces the audio data to be of the correct sample type. Typically, when an audio stream is used in conjunction with a digital audio player, this instance variable is be set by a negotiation process between the audio stream and the sound driver that is transmitting the sound. The values defined by ScriptX are `@binaryOffset` and `@twosComplement`.

sampleWidth

self`.sampleWidth` (read-only) `Integer`

Specifies the width of each sample in bits. Typical values are 8 and 16.

Instance Methods

Inherited from `Stream`:

`cursor`	`next`	`seekFromStart`
`flush`	`plug`	`setStreamLength`
`isAtFront`	`previous`	`streamLength`
`isPastEnd`	`read`	`write`

isReadable	readReady	writeReady
isSeekable	seekFromCursor	
isWritable	seekFromEnd	

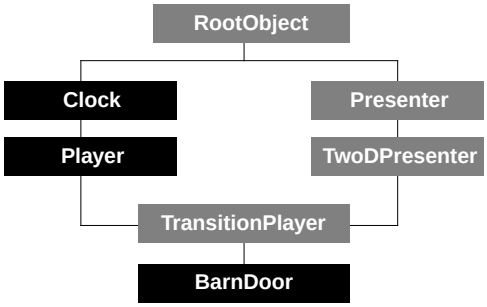
Inherited from ByteStream:

fileIn	readByte	writeString
pipe	readReady	
pipePartial	writeByte	

Inherited from MediaStream:

addMarker	isSeekable	isWritable
isReadable		

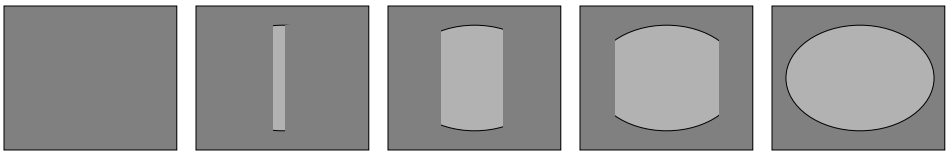
BarnDoor



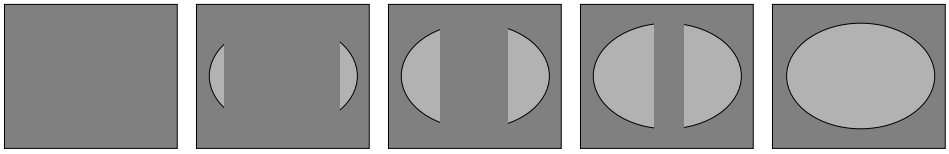
Class type: Core class (concrete)
Resides in: ScriptX and KMP executables
Inherits from: `TransitionPlayer`
Component: Transitions

The `BarnDoor` transition player provides a visual effect similar to two doors opening from the middle, as shown below. A barn door can be set to open two different ways by setting the `direction` instance variable defined by `TransitionPlayer` to either of two values: `@open` or `@close`.

`@open`



`@close`



Creating and Initializing a New Instance

The following script is an example of how to create a new instance of the `BarnDoor` class:

```
myTransition := new BarnDoor \
                    duration:60 \
                    direction:@open \
                    target:myShape
```

The variable `myTransition` contains the initialized transition. The transition reveals the image `myShape` from the center first, with a barn door effect, and has a duration of 60 ticks.

You determine which space the transition will take effect in by adding this instance into that space. Then, when you play the transition player, `myShape` is transitioned into that space.

The `new` method uses the keywords defined in `init`.

init

`init` *self* [`duration:`*integer*] [`direction:`*name*]
[`movingTarget:`*boolean*] [`useOffscreen:`*boolean*] [`target:`*twoDPresenter*]
[`boundary:`*stencil*] [`masterClock:`*clock*] [`scale:` *integer*] ⇨ *(none)*

This method is inherited from `TransitionPlayer` with no change in keywords—refer to that class for details. Do not call `init` directly on an instance—it is automatically called by the `new` method.

Instance Variables

Inherited from `Presenter`:

`presentedBy`	`subPresenters`	`target`

Inherited from `TwoDPresenter`:

`bBox`	`height`	`transform`
`boundary`	`IsImplicitlyDirect`	`width`
`clock`	`isTransparent`	`window`
`compositor`	`isVisible`	`x`
`direct`	`needsTickle`	`y`
`eventInterests`	`position`	`z`
`globalBoundary`	`stationary`	
`globalTransform`	`target`	

Inherited from `Clock`:

`callbacks`	`rate`	`ticks`
`effectiveRate`	`resolution`	`time`
`masterClock`	`scale`	`title`
`offset`	`slaveClocks`	

Inherited from `Player`:

`audioMuted`	`globalContrast`	`globalVolumeOffset`
`dataRate`	`globalHue`	`markerList`
`duration`	`globalPanOffset`	`status`
`globalBrightness`	`globalSaturation`	`videoBlanked`

Inherited from `TransitionPlayer`:

`autoSplice`	`direction`	`movingTarget`
`backgroundBrush`	`duration`	`target`
`cachedTarget`	`frame`	`useOffscreen`

The following instance variables are defined in `Slide`:

direction (TransitionPlayer)

self`.direction` (read-write) `NameClass`

Specifies the direction in which the barn door transition *self* should be applied. Possible values are `@open` and `@close`.

Instance Methods

Inherited from `TwoDPresenter`:

`adjustClockMaster`	`inside`	`show`
`createInterestList`	`localToSurface`	`surfaceToLocal`
`draw`	`notifyChanged`	`tickle`
`getBoundaryInParent`	`recalcRegion`	
`hide`	`refresh`	

Inherited from `Clock`:

`addPeriodicCallback`	`clockAdded`	`pause`
`addRateCallback`	`clockRemoved`	`resume`
`addScaleCallback`	`effectiveRateChanged`	`timeJumped`
`addTimeCallback`	`forEachSlave`	`waitTime`
`addTimeJumpCallback`	`isAppropriateClock`	`waitUntil`

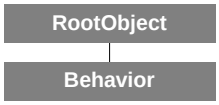
Inherited from `Player`:

`addMarker`	`goToBegin`	`playPrepare`
`eject`	`goToEnd`	`playUnprepare`
`fastForward`	`goToMarkerFinish`	`playUntil`
`getMarker`	`goToMarkerStart`	`resume`
`getNextMarker`	`pause`	`rewind`
`getPreviousMarker`	`play`	`stop`

Inherited from `TransitionPlayer`:

`playPrepare`

Behavior



Class type: Core class (abstract)
Resides in: ScriptX and KMP executables
Inherits from: `RootObject`
Component: Object System Kernel

The `Behavior` class provides the operations common to all classes and metaclasses. `Behavior` is the superclass of `RootClass` and `MetaClass`.

The following "instance" methods of `Behavior` are actually the default class methods for ScriptX classes and metaclasses. All class methods are metaclass instance methods. All metaclasses inherit from `RootClass`, and `RootClass` then inherits from `Behavior`. The distinguished class `MetaClass` also inherits directly from `Behavior`. The following methods are factored and pushed up into `Behavior` so that `MetaClass` can also inherit these standard class methods, making them available for metaclasses as well.

For example, `Behavior` defines the `new` generic function for creating new instances of metaclasses in ScriptX. It is considered an instance method of the `Behavior` class, while it is a class method for ScriptX classes.

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 ScriptX 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.

Instance Methods

allInstances

`allInstances` *self* ⇨ Sequence

Returns a sequence of all instances of the class *self*. Instances of a subclass are considered to be instances of the given class. See also `allDirectInstances`.

canClassDo

`canClassDo` *self generic* ⇨ `Boolean`

self	Any class
generic	Any generic function

Returns `true` if the class *self* implements the given generic function directly. If the class inherits its implementation from another class, `canClassDo` returns `false`. Contrast `canClassDo` with `canObjectDo` (a method defined on `RootObject`), which also returns `true` if the class inherits its implementation of the generic function.

getDirectSubs

`getDirectSubs` *self* ⇨ Sequence

Returns an array with the direct subclasses of the class *self*.

getDirectSupers

`getDirectSupers` *self* ⇨ Sequence

Returns an array with the direct superclasses of the class *self*. This list is returned in precedence order. For an explanation of precedence order, see "Multiple Inheritance" in *ScriptX Language Guide*.

getSubs

`getSubs` *self* ⇨ Sequence

Returns a collection of all direct and indirect subclasses of the class *self*. Note that certain classes in the core classes may not be loaded into memory at startup. These "loadable" classes do not appear on the list that `getSubs` returns until they have been referenced explicitly, and hence loaded into memory.

getSupers

`getSupers` *self* ⇨ Sequence

Returns a collection of all direct and indirect superclasses of the class *self*. See `getSubs` for note about classes that are not loaded into memory at startup.

inflateInstance

`inflateInstance` *self stream* ⇨ *(object)*

self	Class of the object to inflate
stream	A storage stream containing object information

Initializes and returns instances of the class *self* upon retrieval from a storage container. You may override `inflateInstance` as a class method, rather than `inflate` as an instance method, to perform actions specific to a class upon retrieval and inflation of instances from the object store. For example, you may want only one instance of a particular class to be loaded and initialized; this method can check to see if that instance already exists and return it rather than initializing a new instance. You may also partially initialize an object within this method, then perform further initialization in the `inflate` instance method. See the `inflate` instance method for more on how to perform customized initialization upon retrieval from storage.

Note – If you override `inflateInstance` by defining a class method in a particular class, and then create a subclass of that class, you must take certain precautions. Specifically, you must either expect not to have persistent instance variables in the subclass, or you must specialize `inflateInstance` in the subclass to handle any instance variables defined by the new subclass. See the chapter "Title Management" in the *ScriptX Components Guide* for more information on how objects are added to and stored in a container.

isDirectSub

`isDirectSub` *class1 class2* ⇨ `Boolean`

class1	Any class
class2	Any class

Returns `true` if *class1* is a direct subclass of *class2*, with no intervening classes.

isMemberOf

`isMemberOf` *theClass myObject* ⇨ `Boolean`

theClass	Any class
myObject	Any object

Returns `true` if *myObject* is an instance of class *theClass*.

isSub

`isSub` *self super* ⇨ `Boolean`

self	Any class
super	Any class

Returns `true` if *self* inherits (directly or indirectly) from *super*.

methodBinding

`methodBinding` *self generic* ⇨ *(method)*

self	Any class
generic	Any generic function

Returns the method that implements *generic* on the class *self*.

new

`new` *self arg1 arg2 . . . key1*`:`*value1 key2* `:`*value2 . . .* ⇨ *(object)*

self	Any class
arg1 arg2 . . .	Any objects
key	Keyword defined in the `init` or `afterInit` methods
value	An object appropriate to the specified *key*

Allocates memory for a new instance of the class *self*, and then calls `init` and `afterInit` on the new instance, supplying all arguments to both of these methods. Returns the initialized object, regardless of what `init` or `afterInit` returns. (Note that few core classes have an implementation for the `afterInit` method.)

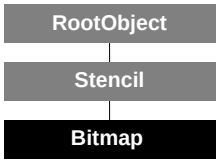
In this document, the description for each concrete class includes sample code for creating a new instance of that class.

When you call the `new` method, specify the class name as the first parameter, followed by the keyword arguments from the `init` and `afterInit` methods for that class. You can enter keyword arguments in any order. Optional keywords are indicated in this manual by square brackets:

```
init self [ data:byteString ] [ colormap:colormap ] bBox:rect.
```

All classes inherit the implementation of `new` defined in `Behavior`. To specialize the initialization of a class, rather than specialize the `new` method, you specialize the `init` and `afterInit` methods. For a general description of how `init` and `afterInit` work, see “The Creation and Initialization Syntax” on page 19 in Chapter 1, “Information Common to All Classes.” The *ScriptX Language Guide* provides a general discussion of how to define or specialize the `init` method.

Bitmap



Class type: Core class (concrete)
Resides in: ScriptX and KMP executables
Inherits from: `Stencil`
Component: 2D Graphics

The `Bitmap` class provides images made up of pixels, each representing a particular color value in a color space or color map. The area of a bitmap may also be surrounded by matte pixels and contain invisible pixels, which are by definition not part of the image produced when the bitmap is rendered. That is, when a bitmap is stroked, filled, or transferred to a surface, its shape is determined by all pixels except those that have the matte color or invisible color. When a bitmap is rendered, any rendering previously performed on the same surface will appear wherever the invisible pixels occur.

Note that `Bitmap` objects by themselves are not presenters—to display a bitmap, create an instance of `TwoDShape` using a `Bitmap` object as the boundary argument.

When using bitmaps in animation, it's useful to set their `x1` and `y1` values for registration points. To see how this is done, refer to the section "Registration Points" in the "Animation" chapter of the *ScriptX Components Guide*.

Note – `Bitmap` instances aren't resizeable and thus aren't suitable for use as the boundary of `TwoDShape` objects that must be resized.

Creating and Initializing a New Instance

You would normally create `Bitmap` objects by importing bitmap files. For details, see the *ScriptX Tools Guide*.

Although you should rarely need to create `Bitmap` objects by calling new on the `Bitmap` class, the following information is provided for the sake of completeness.

The following script is an example of how to create a new instance of the `Bitmap` class:

```
myBitmap := new Bitmap \
            data:myString \
            colormap:theDefault8Colormap \
            bBox:(new Rect x2:100 y2:100)
```

The variable `myBitmap` contains the initialized bitmap. The bitmap's pixel values are specified by the data in `myString`, its color map is set to system default 8-bit color map, and its bounding box is 100 pixels by 100 pixels.

init

`init` *self* [`data:`*byteString*] [`colormap:`*colormap*] [`bitsPerPixel:`*integer*]
`bBox:`*rect* ⇨ *(none)*

self	`Bitmap` object
`data:`	`ByteString` object
`colormap:`	`Colormap` object
`bitsPerPixel:`	`Integer` object

The superclass `Stencil` uses the following keyword:

`bBox:` `Rect` object

Initializes the `Bitmap` object *self*, applying the values supplied with the keywords to the instance variables of the same names, as follows: `data` sets the source of its data, `colormap` sets the color map instance variable used by the bitmap, `bitsPerPixel` sets the pixel depth, and `bBox` sets the area of the bitmap. Do not call `init` directly on an instance—it is automatically called by the `new` method.

If you omit an optional keyword, its default value is used. The defaults are:

`data:`Storage allocated based on `bBox` size and `colormap`
`colormap:theDefault8Colormap`
`bitsPerPixel:8` bits

The global constant `defaultColormap` is determined by the pixel depth of the display system on the underlying hardware platform.

Instance Variables

Inherited from `Stencil`:

`bBox`

The following instance variables are defined in `Bitmap`:

bitsPerPixel

self`.bitsPerPixel` (read-only) `Integer`

Represents the pixel depth of the bitmap *self*. This value is set to correspond to the pixel depth of the color map applied to the bitmap.

colormap

self`.colormap` (read-write) `Colormap`

Contains an instance of `Colormap` to describe the bit-depth and color encoding of pixels in the bitmap *self*. Pixel encodings in the data of a bitmap are assumed to represent keys into this color map, whose values are instances of `Color`. By default, this value is set to `theDefault8Colormap`.

You can only change the value of this instance variable to another `Colormap` of the same pixel depth.

For more on color maps and bitmaps, see the chapter “2D Graphics” in the *ScriptX Components Guide*.

For best performance, you should set a bitmap’s colormap to match that of the window in which it is being displayed. You may do so by resetting either the window or bitmap `colormap` instance variable to match the value of its counterpart.

data

self`.data` (read-write) `ByteString`

Specifies an instance of `ByteString` containing data for the bitmap *self*. The color encoding of pixels within this string is defined by the contents of the `colormap` instance variable. The length of this data is defined by the `size` instance variable, and the length of an individual row within the map is defined by the `rowBytes` instance variable.

Note that you can set the values for individual pixels in a bitmap by setting values in this byte string. Values used must represent valid indexes into the bitmap’s `colormap`. You should not replace the byte string or make other changes to it that affect the dimensions of the bitmap data.

invisibleColor

self.`invisibleColor` (read-write) `Color` or `Integer`

Specifies the `Color` object or the `Colormap` index that should be transparent both around and within the image area when the bitmap *self* is transferred or filled. When the bitmap is rendered, any rendering previously performed on the same surface will appear wherever invisible pixels occur. This variable provides a means of encoding a mask into bitmap data.

When setting this value as an integer, be aware that the indexes to the bitmap's color map are in the range between 0 and the size of the color map - 1.

When setting this instance variable, if the color isn't in the bitmap's color map, an exception is reported. However, if the bitmap's color map has a pixel depth greater than 8, ScriptX assumes that all colors are represented, and no error is reported even if the color map doesn't actually include the color.

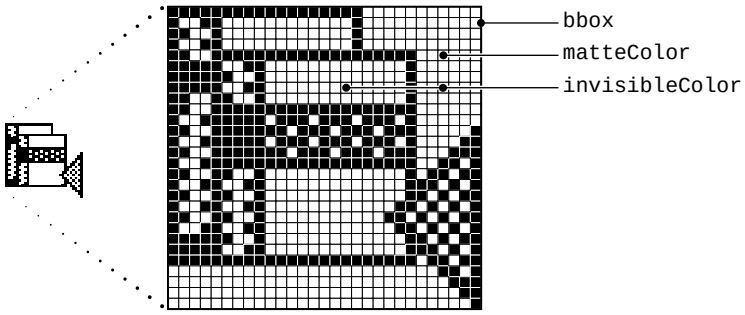


Figure 1: Bitmap's invisibleColor and matteColor

matteColor

self.`matteColor` (read-write) `Color` or `Integer`

Specifies the `Color` object or the `Colormap` index that should be transparent around the image of the bitmap *self* when it is transferred or filled. The `matteColor` is transparent only between the boundary rectangle and the image of the bitmap, not within the image itself. Thus, the boundary of the image represented by the bitmap is defined by pixels of colors other than the `matteColor` value. When the bitmap is rendered, any rendering previously performed on the same surface will appear wherever matte color pixels occur.

When setting this instance variable, if the color isn't in the bitmap's color map, an exception is reported. However, if the bitmap's color map has a pixel depth greater than 8, ScriptX assumes that all colors are represented, and no error is reported even if the color map doesn't actually include the color.

pagingMethod

self.`pagingMethod` (read-write) (object)

The ScriptX importers, discussed in*ScriptX Tools Guide*, can import compressed bitmaps and retain the compression. When you use a bitmap that has compressed data, the data is uncompressed automatically as determined by the value in the bitmap's `pagingMethod` instance variable.

The possible values for this instance variable are `@onload`, `@firstUse`, `@eachUseFromStorage`, and `@eachUseFromMemory`. The meanings of each of these values is discussed in the *ScriptX Tools Guide*.

remapOnDraw

self.remapOnDraw (read-write) Boolean

Determines how pixels are copied from the bitmap *self* to a surface during the rendering operations `transfer` or `fill`. If `remapOnDraw` is `true`, the rendering operation remaps the colors represented in the bitmap to the color map of the surface performing the rendering. If this variable is `false`, the rendering operation simply interprets pixel values in the bitmap as indexes into the surface's color map. By default, the value is `true`.

remapOnSet

self.remapOnSet (read-write) Boolean

Determines how the bitmap *self* responds to a change in its `colormap` instance variable. If `remapOnSet` is `true`, then changing the color map will result in the pixel values in the bitmap being reindexed to the appropriate colors in the new color map. If it is `false`, the bitmap simply interprets existing pixel values as indexes into the new color map. By default, this value is `false`.

For best performance, you should set a bitmap's colormap to match that of the window in which it is being displayed. You may do so by resetting either the window or bitmap `colormap` instance variable. If you set the bitmap's colormap, you should probably set this variable true before doing so.

rowBytes

self.rowBytes (read-only) Integer

Specifies the number of bytes per row within the data of the bitmap *self*.

size

self.size (read-only) Integer

Specifies the size of the bitmap's data string in bytes. This value is always an integer multiple of `rowBytes`.

Instance Methods

Inherited from `Stencil`:

`inside`	`onBoundary`	`transform`
`intersect`	`subtract`	`union`

The following methods are defined in `Bitmap`:

dropData

`dropData` *self* ⇨ (*object*)

self The `Bitmap` object.

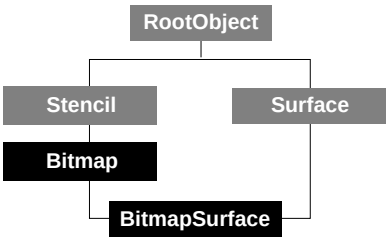
If a bitmap has been loaded from a container such as a title container or library container, you can use the `dropData` method to force the bitmap to drop its data immediately.

When you make a bitmap purgeable, the garbage collector cleans up the memory that the bitmap uses. However, this does not happen instantly. It happens as soon as the garbage collector gets round to it. Before making a bitmap purgeable, you can call

`dropData` on it, to immediately free up the memory used by the bitmap. When the garbage collector gets round to cleaning up the object, it completes the cleanup by freeing the memory occupied by the actual `Bitmap` object.

If you call `dropData` on a bitmap that has not been loaded from a container, (that is, you imported it in the same ScriptX session), you will get an exception.

BitmapSurface



Class type: Core class (concrete)
Resides in: ScriptX and KMP executables
Inherits from: `Bitmap` and `Surface`
Component: 2D Graphics

A `BitmapSurface` object provides an area of memory that can be used to both represent pixel images and perform rendering operations. The pixel image represented by a bitmap surface can thus be altered using the standard rendering operations defined by the surface `Surface`.

To alter a bitmap, use the `Surface` method `transfer` to render the `Bitmap` instance on a `BitmapSurface` object. You then render on the bitmap using other rendering operations and objects; for example, you can stroke a text stencil onto the surface. To use the resulting image, you transfer the `BitmapSurface` object to a `Surface` object using the `transfer` method.

`BitmapSurface` instances can be used as drawing caches. For example, see the "2D Graphics" chapter in the *ScriptX Components Guide*.

Creating and Initializing a New Instance

The following script is an example of how to create a new instance of the `BitmapSurface` class:

```
myDrawingCache := new BitmapSurface \
                  colorMap:myColors \
                  bBox:(new Rect x2:100 y2:100) \
                  data:myString
```

The variable `myDrawingCache` contains the initialized bitmap surface. The surface's bounding box is 100 pixels by 100 pixels, its color map is set to system default 8-bit color map, and its pixel values are specified by the data in `myString`.

init

`init` *self* `bBox:`*rect* [`colorMap:`*colorMap*] [`data:`*byteString*] ⇨ *(none)*

self	`BitmapSurface` object

Superclasses of `BitmapSurface` use the following keywords:

`bBox:`	`Rect` object
`colorMap:`	`ColorMap` object
`data:`	`ByteString` object

Initializes the `BitmapSurface` object *self*. The keyword values are described in the `Bitmap` class discussion. Note that because `BitmapSurface` inherits from both `Bitmap` and `Surface`, the `bBox` and `boundary` instance variables are both set to the *rect* value supplied with the `bBox` keyword. Do not call `init` directly on an instance—it is automatically called by the `new` method.

If you omit an optional keyword, its default value is used. The defaults are:

`colorMap:theDefault8ColorMap`
`data:` Storage allocated based on size of `bBox` and `colorMap`

Instance Variables

Inherited from `Stencil`:

`bBox`

Inherited from `Bitmap`:

`bitsPerPixel`	`invisibleColor`	`rowBytes`
`colorMap`	`matteColor`	`size`
`compressionInfo`	`remapOnDraw`	
`data`	`remapOnSet`	

Inherited from `Surface`:

`boundary`

The following instance variable, inherited from `Bitmap`, is redefined in `BitmapSurface`:

data (BitMap)

self`.data` (read-only) ⇨ *(none)*

Represents that data being rendered by the bitmap surface *self*. Note that this data may not necessarily represent the actual pixel data currently being rendered. To assure correct contents of this instance variable—for example, to perform a screen grab on a particular display surface—you must first coerce the `BitmapSurface` *self* to a `Bitmap`.

Instance Methods

Inherited from `Stencil`:

`inside`	`onBoundary`	`transform`
`intersect`	`subtract`	`union`

Inherited from `Bitmap`:

`(none)`

Inherited from `Surface`:

`fill`	`stroke`	`transfer`

The following instance methods are defined in `BitmapSurface`:

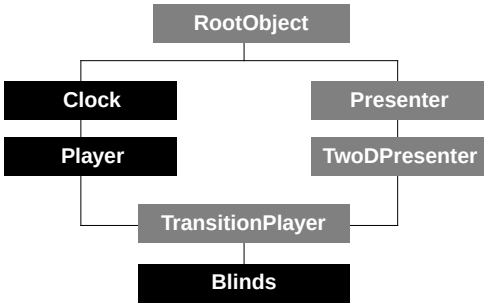
erase

`erase` *self brush* ⇨ *(none)*

self	`BitmapSurface` object
brush	`Brush` object

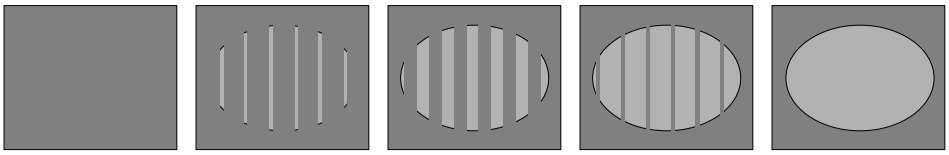
Fills the `boundary` of the bitmap surface *self* with *brush*, effectively erasing any previous rendering on the surface.

Blinds



Class type: Loadable class (concrete)
Resides in: `ltrans.lib`. Works with ScriptX and KMP executables.
Inherits from: `TransitionPlayer`
Component: Transitions

The `Blinds` transition player provides a visual effect that causes the target to gradually appear using vertical (`@vertical`) or horizontal (`@venetian`) bands as shown below. The target appears when the transition is played forward, and disappears when played backward (transition's rate set to -1).



Directions: `@vertical`, `@venetian`

Rate: Can play forward or backward.

For a side-by-side illustrations of all transitions, see the Transitions chapter in the *ScriptX Components Guide*.

Creating and Initializing a New Instance

The following script is an example of how to create a new instance of the `Blinds` class:

```
myTransition := new Blinds \
                    duration:60 \
                    direction:@venetian \
                    target:myShape
```

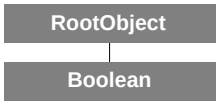
The variable `myTransition` contains the initialized transition. The transition reveals the image `myShape` from the center first, with a barn door effect, and has a duration of 60 ticks.

You determine which space the transition will take effect in by adding this instance into that space. Then, when you play the transition player, `myShape` is transitioned into that space.

The `new` method uses the keywords defined in `init`.

NOTE – For the instance variables and methods, see the `BarnDoor` class.

Boolean



Class type: Core class (concrete, sealed)
Resides in: ScriptX and KMP executables
Inherits from: `RootObject`
Component: Numerics

The `Boolean` class represents the objects `true` and `false`. Note that `Boolean` is not a subclass of `Number`.

When evaluating an expression to test if it is true or false (such as in an `if` statement), ScriptX recognizes only the `false` object as being false. Anything else is true. Therefore, `1` is true, `0` is true, `empty` is true, `undefined` is true, the `true` object is true, and so forth. Therefore, to test if something is true, the surest way is to test that it is not false.

```
if (x <> false) do <expression>
```

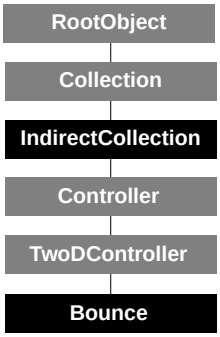
Test against `true` only if you are certain that there is no other way the expression can evaluate to non-false.

ScriptX defines two instances of the `Boolean` class, `true` and `false`, which exist as global constants. See Chapter 3, "Global Constants and Variables" of this volume.

Since `true` and `false` already exist as global constants, there is no reason to create an instance of the `Boolean` class.

Note – The `and`, `or`, and `not` Boolean operators are part of the ScriptX Language; they are not generic functions defined by the `Boolean` class, nor are they implemented directly as global functions. Refer to the *ScriptX Language Guide* for more information. The `xor` Boolean operation is defined as a global function (page 54).

Bounce



Class type: Core class (concrete)
Resides in: ScriptX and KMP executables
Inherits from: `TwoDController`
Component: Controllers

The `Bounce` class is a controller that, when used with the `Movement` controller, causes one or more `Projectile` objects to bounce off the edges of the container. For each projectile in the bounce controller's space, the `tickle` method looks at whether the projectile intersects with the edge of the space. If it does, it changes its `velocity` instance variable. Note that the bounce controller doesn't actually move the projectile—it only changes the value of its `velocity` instance variable. The `Movement` controller moves the projectile according to the current value of `velocity`.

You can specify which projectiles are affected by bounce and their initial velocities. See the `Projectile` class for more details. (`Bounce` is more an example class than a crucial part of the author's toolbox.)

A `Bounce` object is a collection of the `Projectile` objects it controls. These projectiles must also be in the space that the controller is controlling. Projectiles are either automatically or manually added to the bounce controller, according to the `wholeSpace` instance variable. If `wholeSpace` is `false`, you can use the methods defined by `Collection` to add and remove objects from the bounce controller. To ensure that only projectiles are added to a bounce controller, the `protocols` instance variable is set to the `Projectile` class. See the `Controller` class for descriptions of `wholeSpace`, `protocols`, and other general properties of controllers.

Bounce defines the `tickle` method to check all bounce targets at every tick of the presenter's clock.

Creating and Initializing a New Instance

The following script is an example of how to create a new instance of the `Bounce` class, after creating the space it controls:

```
mySpace := new TwoDSpace boundary:(new Rect x2:200 y2:200)
myBounce := new Bounce space:mySpace
```

The variable `myBounce` contains the initialized bounce controller. This instance changes the direction of projectiles in `mySpace` when they hit the edge of the space. The `new` method uses keywords defined in `init`.

init

`init` *self* [`space:`*space*] [`wholeSpace:`*boolean*] [`enabled:`*boolean*]
[`targetCollection:`*sequence*] ⇨ *(none)*

self `Bounce` object

The superclass `Controller` uses the following keywords:

`space:`	`Space` object that holds projectiles
`wholeSpace:`	`Boolean` object
`enabled:`	`Boolean` object

The superclass `TwoDController` uses the following keyword:

`targetCollection:`	`Sequence` object (use carefully)

Initializes the `Bounce` object *self*, applying the values supplied with the keywords to instance variables of the same name. Use discretion in changing the target collection; for more information, see the definition of the `TwoDController` class. Do not call `init` directly on an instance—it is automatically called by the `new` method.

If you omit an optional keyword, its default value is used. The defaults are:

```
space:undefined
wholeSpace:false
enabled:true
targetCollection:(new Array initialSize:4 growable:true)
```

Class Methods

Inherited from Collection:

`pipe`

Instance Variables

Inherited from `Collection`:

`bounded`	`maxSize`	`size`
`iteratorClass`	`minSize`	`uniformity`
`keyEqualComparator`	`mutable`	`uniformityClass`
`keyUniformity`	`mutableCopyClass`	`valueEqualComparator`
`keyUniformityClass`	`proprietored`	

Inherited from `IndirectCollection`:

`targetCollection`

Inherited from `Controller`:

`enabled`	`space`	`wholeSpace`
`protocols`		

protocols (Controller)

self`.protocols` (read-write) `Array`

This instance variable initially contains the class `Projectile` for the bounce controller *self*. This means that any object added to a `Bounce` controller must have `Projectile` as one of its superclasses. See the `Controller` class for further description about this instance variable.

Instance Methods

Inherited from `Collection`:

`add`	`forEach`	`iterate`
`addMany`	`forEachBinding`	`localEqual`
`addToContents`	`getAll`	`map`
`chooseAll`	`getAny`	`merge`
`chooseOne`	`getKeyAll`	`pipe`
`chooseOneBinding`	`getKeyOne`	`prin`
`deleteAll`	`getMany`	`removeAll`
`deleteBindingAll`	`getOne`	`removeOne`
`deleteBindingOne`	`hasBinding`	`setAll`
`deleteKeyAll`	`hasKey`	`setOne`
`deleteKeyOne`	`intersects`	`size`
`deleteOne`	`isEmpty`	
`emptyOut`	`isMember`	

Inherited from `IndirectCollection`:

`isAppropriateObject`	`objectAdded`	`objectRemoved`

Inherited from `Controller`:

`isAppropriateObject`	`tickle`

Since a `Bounce` controller is an indirect collection, you can also use any methods defined in the class specified by `targetCollection`. The target collection is typically an instance of `Array`, which inherits from `Sequence`, so the following instance methods are redirected to this controller.

Accessible from `LinearCollection`:

`chooseOneBackwards`	`findRange`	`getNthKey`
`chooseOrdOne`	`forEachBackwards`	`getOrdOne`
`deleteFirst`	`getFirst`	`getRange`
`deleteLast`	`getLast`	`localEqual`
`deleteNth`	`getMiddle`	`localLT`
`deleteRange`	`getNth`	`pop`

Accessible from `Sequence`:

`addFifth`	`moveBackward`	`setFourth`
`addFirst`	`moveForward`	`setLast`
`addFourth`	`moveToBack`	`setNth`
`addNth`	`moveToFront`	`setSecond`
`addSecond`	`prepend`	`setThird`
`addThird`	`prependNew`	`sort`
`append`	`setFifth`	
`appendNew`	`setFirst`	

The following instance method is defined in `Bounce`:

tickle (Controller)

`tickle` *self clock* ⇨ *self*

self	`Bounce` object
clock	`Clock` object of the space being controlled

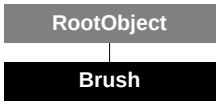
For any moving projectile that intersects the edge of the controlled space, this method sets a new value for `velocity` degraded by its value of `elasticity` (both instance variables of `Projectile`).

When used with the `Movement` class, the projectile then changes directions, bouncing off the edge of the space. Note that this method doesn't actually move the projectile—this method only changes the value of its `velocity` instance variable. The `Movement` controller moves the projectile according to its current value of velocity.

A callback calls this method on the `Bounce` object *self*, supplying the space's clock as the value for *clock*. The callback calls this method once every tick of the space's clock.

For further details, refer to the section "The Ticklish Protocol" in the chapter "Controllers" in the *ScriptX Components Guide*.

Brush



Class type: Core class (concrete)
Resides in: ScriptX and KMP executables
Inherits from: `RootObject`
Component: 2D Graphics

The `Brush` class defines properties of the "paint" that is applied when a rendering operation creates an image on a surface. Some `Brush` properties apply to both the `fill` and `stroke` methods of `Surface`; others apply only to `stroke`:

- Properties applicable to both `fill` and `stroke` are `color`, `pattern`, and `inkMode`.
- Properties exclusive to `stroke` are `lineWidth`, `lineJoin`, `miterLimit`, and `pattern`.

ScriptX provides two global instances of `Brush`: `blackBrush` and `whiteBrush`. A synonym for `whiteBrush` is `defaultBrush`. These global instances are defined in the chapter "Global Constants and Variables."

Creating and Initializing a New Instance

The following script is an example of how to create a new instance of the `Brush` class:

```
lightMagentaBrush := new Brush \
                     color:magentaColor \
                     pattern:grayPattern
```

The variable `lightMagentaBrush` contains the initialized brush. The instance has a color of `magentaColor`. Its pattern is `grayPattern`, which produces a slightly pastel version of the specified color.

init

`init` *self* [`color:`*color*] [`pattern:`*pattern*] ⇨ *(none)*

self	`Brush` object
`color:`	`Color` object
`pattern:`	`Bitmap` object or named ScriptX pattern

Initializes the `Brush` object *self*, applying the arguments to the instance variables of the same name, as follows: `color` is set as the `Brush` color and the bitmap or named `pattern` is set as the pattern. Do not call `init` directly on an instance—it is called automatically by the `new` method.

If you omit an optional keyword, ScriptX returns the system global `whiteBrush` with the following values set:

```
color:whiteColor
pattern:@blackPattern
```

Instance Variables

color (Brush)

self`.color` (read-write) `Color`

Specifies the color to use when filling or stroking with the brush, *self*.

inkMode

self.inkMode (read-write) Integer

Specifies the mixing characteristic used to apply a new paint from the brush *self* over a previous image on the destination surface. Note that some inkMode settings are effective only with black or white values for color; colors other than black or white act like black with these settings. While the effects produced by inkMode can be explained, they may be easier to interpret visually.

The values of this instance variable may be the following integer constants:

srcCopy – Replace all pixels in the destination surface with the color or pattern of the brush (0).

srcOr – If the brush color is white, leaves colored pixels in the destination as they were; if the brush color is not white, applies the source color over all destination pixels (1).

srcXor – If the brush color is white, leaves colored pixels in the destination as they were. If the brush color is not white, inverts the color of destination pixels, turning white to black and non-white to white (2).

srcBic – If the brush color is white, leaves any colored pixels in the destination as they were. If the brush color is another color, applies white over any destination pixels (3).

notSrcCopy – Replaces all pixels in the destination with the inverse of the color of the source. If the source color is white, black pixels are applied; if the source is non-white, white pixels are applied (4).

notSrcOr – If the brush color is non-white, leaves any colored pixels in the destination as they were. If the brush color is white, inverts the color of destination pixels, turning white to black and non-white to white (5).

notSrcXor – If the brush color is non-white, leaves any colored pixels in the destination as they were. If the brush color is white, inverts the color of destination pixels, turning white to black and non-white to white (6).

notSrcBic – If the brush color is non-white, leaves any colored pixels in the destination as they were. If the brush color is white, applies white over any destination pixels (7).

The effects produced by these settings are shown in the following diagram:

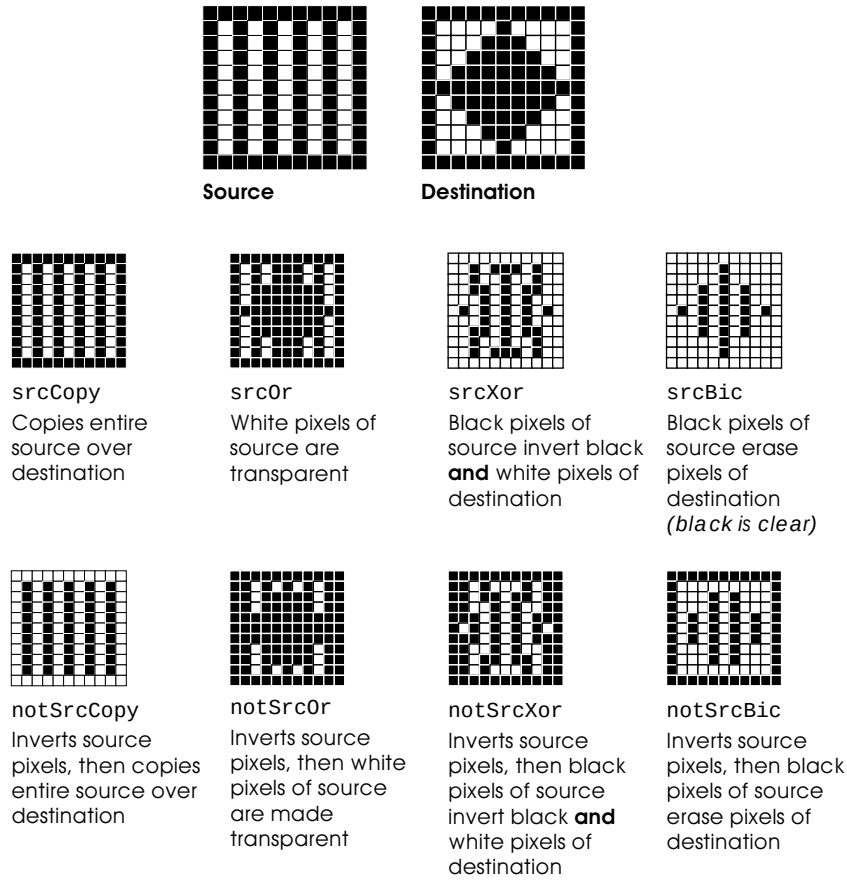


Figure 2: Ink modes and their effects

lineWidth

self.`lineWidth` (read-write) `Number`

Specifies the width of paths rendered by stroke operations. Setting this value to `0` will produce the smallest line width possible for a particular device.

pattern

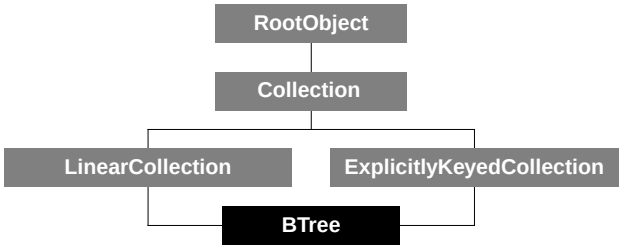
self.`pattern` (read-write) `NameClass` or `Bitmap`

Defines a `Bitmap` instance used to stroke or fill a stencil. If specified, this bitmap is rendered as a repeating tiled pattern within the fill area or along the stroke path. The bitmap supplied must have a square bounding box, and the length of its sides must be a power of 2 (such as 2, 4, 8, and so on). The maximum size of a pattern is 128 pixels wide by 128 pixels high, and the maximum value of width by height by bitdepth is 64K. When a bitmap is used as the pattern of a brush, its `matteColor` and `invisibleColor` instance variables are ignored.

The named patterns defined in ScriptX are `@blackPattern`, `@dkGrayPattern`, `@grayPattern`, `@ltGraypattern`, and `@whitePattern`.

Note – In Microsoft Windows, the maximum size of a bitmap that can be used as a pattern for stroking is 8 pixels by 8 pixels.

BTree

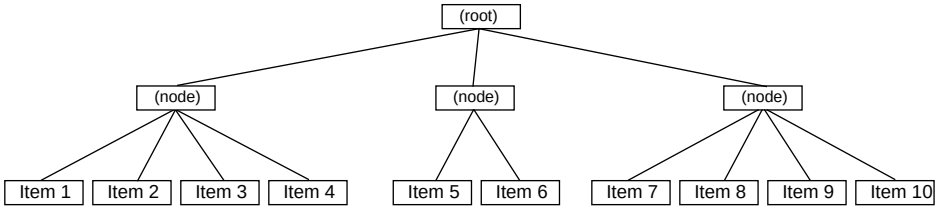


Class type: Core class (concrete)
Resides in: ScriptX and KMP executables
Inherits from: `LinearCollection` and `ExplicitlyKeyedCollection`
Component: Collections

The `BTree` class implements a collection of key-value bindings, sorted by key and stored in a B-tree structure, as shown in the figure below.

B-trees use a comparison function such as `ucmp` for sorting. A B-tree can use any comparison function that compares two objects and returns `@before`, `@same`, and `@after` to yield a consistent ordering on keys. Keys that are added to a B-tree must implement the generic function `localLT` to allow for comparison with other objects. See the discussions of comparison in both the "Collections" and the "Object System Kernel" chapters of the *ScriptX Components Guide*.

B-trees have a specifiable branching factor or "node size" that is greater than or equal to 3 (whereas binary trees have a node size of 2). Comparatively large branching factors aid in searching large disk-based indexes because they minimize the disk hits required to find an entry for a given key.



For additional information on the `BTree` class, see the "Collections" chapter of *ScriptX Components Guide*.

Creating and Initializing a New Instance

The following script is an example of how to create a new instance of the `BTree` class:

```
myTOak := new BTree \
          brFactor:12
          cmpFunction:ucmp
```

The variable `myTOak` contains an initialized instance of a `BTree`. This B-tree has at most 12 branches at each node. The collection sorts key-value pairs by key using the universal comparison function `ucmp`, a function defined on page 52 of Chapter 2, "Global Functions."

init

`init` *self* [`brFactor:`*integer*] [`cmpFunction:`*function*] ⇨ *(none)*

self	`BTree` object
`brFactor:`	`Integer` object
`cmpFunction:`	`AbstractFunction` object

Initializes the `BTree` object *self*, applying the arguments as follows: `brFactor` specifies the branching factor of each B-tree node, or the maximum number of items stored at each node; `cmpFunction` specifies the function that is used as a comparator for keys. The value of `brFactor` must be greater than 2. If you want to specify a branching factor, you must do so when you first create a `BTree` object; there is no setter function for changing the value of `brFactor`.Do not call `init` directly on an instance—it is automatically called by the `new` method.

If you omit an optional keyword, its default value is used:

```
brFactor:32
cmpFunction:ucmp
```

Instance Variables

brFactor

self`.brFactor` (read-only) `Integer`

Returns the branching factor with which the B-tree *self* was created. The branching factor is the maximum number of items at each node. The valve of `brfactor` must be greater than 2.

Instance Methods

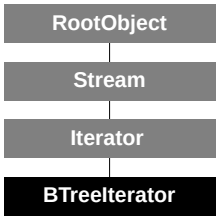
Inherited from `Collection`:

`add`	`forEach`	`iterate`
`addMany`	`forEachBinding`	`localEqual`
`addToContents`	`getAll`	`map`
`chooseAll`	`getAny`	`merge`
`chooseOne`	`getKeyAll`	`pipe`
`chooseOneBinding`	`getKeyOne`	`prin`
`deleteAll`	`getMany`	`removeAll`
`deleteBindingAll`	`getOne`	`removeOne`
`deleteBindingOne`	`hasBinding`	`setAll`
`deleteKeyAll`	`hasKey`	`setOne`
`deleteKeyOne`	`intersects`	`size`
`deleteOne`	`isEmpty`	
`emptyOut`	`isMember`	

Inherited from `LinearCollection`:

`chooseOneBackwards`	`findRange`	`getNthKey`
`chooseOrdOne`	`forEachBackwards`	`getOrdOne`
`deleteFirst`	`getFirst`	`getRange`
`deleteLast`	`getLast`	`localEqual`
`deleteNth`	`getMiddle`	`localLT`
`deleteRange`	`getNth`	`pop`

BTreeIterator



Class type: Core class (concrete)
Resides in: ScriptX and KMP executables
Inherits from: `Iterator`
Component: Collections

The `BTreeIterator` class is used to iterate over any `BTree` object.

Creating and Initializing a New Instance

A new instance of a B-tree iterator is generally created by calling `iterate` on an instance of `BTree`.

Instance Variables

Inherited from `Iterator`:

`key`	`source`	`value`

Instance Methods

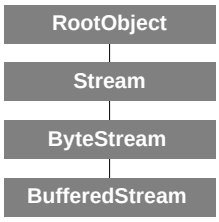
Inherited from `Stream`:

`cursor`	`next`	`seekFromStart`
`flush`	`plug`	`setStreamLength`
`isAtFront`	`previous`	`streamLength`
`isPastEnd`	`read`	`write`
`isReadable`	`readReady`	`writeReady`
`isSeekable`	`seekFromCursor`	
`isWritable`	`seekFromEnd`	

Inherited from `Iterator`:

`exise`	`seekKey`	`seekValue`
`remainder`		

BufferedStream



Class type: Core class (abstract)
Resides in: ScriptX and KMP executables
Inherits from: `ByteStream`
Component: Streams

The `BufferedStream` class defines a general buffered byte stream, allowing ScriptX to handle chunks of data of various types in memory buffers.

Instance Methods

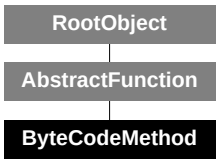
Inherited from `Stream`:

`cursor`	`next`	`seekFromStart`
`flush`	`plug`	`setStreamLength`
`isAtFront`	`previous`	`streamLength`
`isPastEnd`	`read`	`write`
`isReadable`	`readReady`	`writeReady`
`isSeekable`	`seekFromCursor`	
`isWritable`	`seekFromEnd`	

Inherited from `ByteStream`:

`fileIn`	`readByte`	`writeString`
`pipe`	`readReady`	
`pipePartial`	`writeByte`	

ByteCodeMethod



Class type: Core class (concrete, sealed)
Resides in: ScriptX and KMP executables
Inherits from: `AbstractFunction`
Component: Object System Kernel

Every function or method that is defined in the scripter is compiled to create an instance of `ByteCodeMethod`. By contrast, global functions that are defined in the substrate are implemented as `Primitive` objects, and methods that implement generic functions, if defined in the substrate, are implemented as `PrimitiveMethod` objects.

For more information on ScriptX function dispatch, see the "Object System Kernel" chapter of the *ScriptX Components Guide*.

Creating and Initializing a New Instance

You do not define an instance of `ByteCodeMethod` directly. An instance of `ByteCodeMethod` is create automatically when a function or method definition expression is compiled in the scripter.

Instance Variables

debugInfo

self`.debugInfo` (read-only) `DebugInfo`

Returns the `DebugInfo` object that stores debugging information for the `ByteCodeMethod` object *self*. The value of `debugInfo` can be `undefined`.

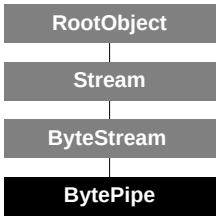
Instance Methods

removeDebugInfo

`removeDebugInfo` *self* ⇨ *(none)*

Sets the value of the instance variable `debugInfo`, defined by the `ByteCodeMethod` object *self*, to `undefined`

BytePipe



Class type: Core class (concrete)
Resides in: ScriptX and KMP executables
Inherits from: `ByteStream`
Component: Threads

`BytePipe` objects are a convenient way for threads to pass chunks of data to each other. An active thread will suspend if it tries to read from a `BytePipe` object that is empty, or write to one that is full. (The value of `status` for the suspended thread is `@waiting`.) `BytePipe` objects may be unbounded in size, in which case they will never cause a thread to suspend on writing.

`BytePipe` objects may be "broken." Breaking a pipe means that writes are no longer allowed; an attempt to write to a broken pipe will report the `brokenPipe` exception. Attempting to read from an empty and broken pipe also reports the `brokenPipe` exception.

`BytePipe` objects implement the ByteStream and Stream protocols. Since they are non-seekable `ByteStream` objects, ScriptX reports the `cantSeek` exception when a script attempts to invoke a method that is implemented only for seekable streams.

Creating and Initializing a New Instance

The following script is an example of how to create a new instance of the `BytePipe` class:

```
byp := new BytePipe \
          maxSize:bufferSize \
          label:@bypLabel
```

The variable `byp` contains the initialized `BytePipe` object, `maxSize` is set to the value of `bufferSize`, and the initialized object is given a label set to `@bypLabel`. The `new` method uses keywords defined in `init`.

init

`init` *self* [`maxSize:`*integer*] [`label:`*object*] ⇨ *(none)*

self	`BytePipe` object
`maxSize:`	`Integer` object representing the size of the pipe
`label:`	Any object, used as a label

Do not call `init` directly on an instance—it is automatically called by the `new` method.

If you omit an optional keyword, its default value is used. The defaults are:

```
maxSize:2000
label:undefined
```

Instance Variables

broken

self.broken (read-only) Boolean

Has a value of true if the byte pipe *self* has been explicitly broken; otherwise, it is false.

label

self.label (read-write) *(object)*

Specifies a label for the byte pipe *self*. The label can be any object; it is displayed when you print the condition, which is useful for debugging.

maxSize

self.maxSize (read-only) Integer

Specifies the maximum allowed size of the byte pipe *self*, in bytes.

size

self.size (read-only) Integer

Specifies the current size of the byte pipe *self*, in bytes.

Instance Methods

Inherited from Stream:

cursor	next	seekFromStart
flush	plug	setStreamLength
isAtFront	previous	streamLength
isPastEnd	read	write
isReadable	readReady	writeReady
isSeekable	seekFromCursor	
isWritable	seekFromEnd	

Inherited from ByteStream:

fileIn	readByte	writeString
pipe	readReady	
pipePartial	writeByte	

The following instance methods are defined in BytePipe:

breakPipe

breakPipe *self* ⇨ *(none)*

Breaks the given byte pipe *self*.

isReadable (Stream)

isReadable *self* ⇨ true

Returns true, since a BytePipe object is always a readable stream. Though it is always readable, there is not necessarily any information in the pipe to read. Call the method readReady to determine whether or not there is currently information in the pipe to be read.

isSeekable (Stream)

`isSeekable` *self* ⇨ `false`

Returns `false`, since a `BytePipe` object is always an unseekable stream. For a discussion of seekable streams, see the definition of the `Stream` class .

isWritable (Stream)

`isWritable` *self* ⇨ `true`

Returns `true`, since a `BytePipe` object is always a writable stream. Although it is always writable, it may already be full, or it may be broken. Call the method `writeReady` to determine whether or not an object can be written to the pipe in its current state.

isPastEnd (Stream)

`isPastEnd` *self* ⇨ `Boolean`

Returns `true` only when the byte pipe *self* is broken and empty.

readReady (Stream)

`readReady` *self* ⇨ `Integer`

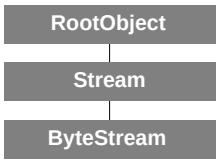
Returns the number of bytes in the byte pipe *self* that are ready to be read out by a reading thread. When all bytes have been read, this value is 0.

writeReady (Stream)

`writeReady` *self* ⇨ `Integer`

Returns the number of bytes in the byte pipe *self* that are available to be written into from a writing thread. If the byte pipe is empty, this returns the `maxSize` (which defaults to 2000).

ByteStream



Class type: Core class (abstract)
Resides in: ScriptX and KMP executables
Inherits from: `Stream`
Component: Streams

The `ByteStream` class provides an abstract implementation of objects that provide byte-wise access to data for reading and writing. Useful subclasses of `ByteStream` include `MemoryStream`, `RamStream`, `ResStream`, `BufferedStream`, `String`, and `BytePipe`.

`NullStreamClass` is a subclass of `ByteStream`, representing a stream containing no data. It has a single global instance, `nullStream`, which is defined in the "Global Constants and Variables" chapter.

Instance Methods

Inherited from `Stream`:

`cursor`	`next`	`seekFromStart`
`flush`	`plug`	`setStreamLength`
`isAtFront`	`previous`	`streamLength`
`isPastEnd`	`read`	`write`
`isReadable`	`readReady`	`writeReady`
`isSeekable`	`seekFromCursor`	
`isWritable`	`seekFromEnd`	

The following instance methods are defined in `ByteStream`:

fileIn

`fileIn` *self* [`debugInfo:`*boolean*] [`module:`*module*] [`quiet:`*boolean*] ⇨ *self*

self	`ByteStream` object containing file to be compiled
`debugInfo:`	`Boolean` object specifying whether to keep the source code. If `true`, the source code is put in the `source` instance variable of the `debugInfo` object in the `ByteCodeMethod` object's `debugInfo` instance variable. (Each method or function in the file is turned into a separate `ByteCodeMethod` object.)
`module:`	`ModuleClass` object
`quiet:`	`Boolean` object

Compiles the ASCII-format ScriptX source code contained in the byte stream *self*, into ScriptX bytecode format and executes the results.

To compile a text file, see the `fileIn` method defined in the `DirRep` class.

If you omit an optional keyword, its default value is used. The defaults are:

```
debugInfo:true
module:scratch
quiet:true
```

Note – The `fileIn` method is available only within ScriptX. The Kaleida Media Player doesn't include the ScriptX bytecode compiler required to execute this function.

pipe

`pipe` *self theObject* ⇨ *(self)*

self	`ByteStream` object
stream	`ByteStream` to be piped

Pipes the contents of *stream* to *self* by writing its entire contents.

pipePartial

`pipePartial` *self stream size* ⇨ *(self)*

self	`ByteStream` object
stream	`ByteStream` to be piped
size	`Integer` number of bytes in *stream* to pipe

Pipes part of the contents of *stream* to *self* by writing its first *size* bytes.

read

`read` *self* ⇨ `Integer`

Reads a byte from the given stream *self*. If there is not enough data to satisfy the request, the request will block until there is enough data. It generates a `cantRead` error if the stream is not readable.

readByte

`readByte` *self* ⇨ `Integer`

Reads a byte from the given stream *self*.

readReady (Stream)

`readReady` *self* ⇨ `Integer`

Returns the number of bytes in the stream *self* that can be read without blocking. Note that for filesystem streams, no guarantee can be made about blocking, so this method usually returns zero.

write

`write` *self value* ⇨ *(none)*

self	`Stream` object
value	`Integer` object

Writes the integer *value* as an 8-bit byte to the stream *self*. The integer should be a value 0 to 255. If the stream is not ready to write the requested size, the request will block until there is enough space or will report a `noSpace` error. A `cantWrite` error is generated if the stream is not writable. Use the method `isWritable` to ensure that the stream is writable.

writeByte

`writeByte` *self value* ⇨ *(none)*

self	`Stream` object
value	`Integer` object

Writes the integer *value* as an 8-bit byte to the stream *self*.

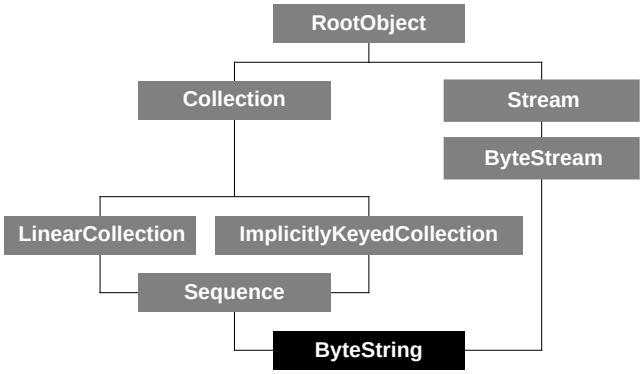
writeString

`writeString` *self string* ⇨ *(none)*

self	`Stream` object
string	`String` object

Writes the given *string* to the stream *self*.

ByteString



Class type: Core class (concrete)
Resides in: ScriptX and KMP executables
Inherits from: `Sequence` and `ByteStream`
Component: Collections

The `ByteString` class provides an object representation for an uninterpreted sequence of bytes. `ByteString` also inherits from `Stream`, and supports write operations. A `ByteString` is not a readable stream, however.

Creating and Initializing a New Instance

The following script is an example of how to create a new instance of the `ByteString` class:

```
obj := new ByteString
```

The variable `obj` is initialized to an empty collection of bytes. You can extend the collection by using the `add`, `append`, etc. `Collection` methods, or you can extend it by writing to *obj* using the `Stream` methods.

init

`init` *self* [`initialSize:`*integer*] [`growable:`*boolean*] ⇨ *(none)*

self	`ByteString` object
`initialSize:`	`Integer` object
`growable:`	`Boolean` object

Initializes the `ByteString` object *self*, applying the arguments as follows: The value supplied with `initialSize` specifies the number of bytes initially allocated for the object and must be a positive integer. Values of 0 or less will be replaced with the default value. If `growable` is `false`, the collection cannot be expanded beyond the size supplied with `initialSize`. If `growable` is `true`, it grows in chunks. Do not call `init` directly on an instance—it is automatically called by the `new` method.

If you omit an an optional keyword, its default value is used. The defaults are:

```
initialSize:20
growable:true
```

ScriptX may create a `ByteString` object that is larger than you specify with `initialSize`, because of its interaction with memory management at initialization time. However, when you set `growable` to `false`, the `initialSize` is actually used and `maxSize` is set to that same value.

Class Methods

Inherited from Collection:

pipe

Instance Variables

Inherited from Collection:

bounded	maxSize	size
iteratorClass	minSize	uniformity
keyEqualComparator	mutable	uniformityClass
keyUniformity	mutableCopyClass	valueEqualComparator
keyUniformityClass	proprietored	

Instance Methods

Inherited from Stream:

cursor	next	seekFromStart
flush	plug	setStreamLength
isAtFront	previous	streamLength
isPastEnd	read	write
isReadable	readReady	writeReady
isSeekable	seekFromCursor	
isWritable	seekFromEnd	

Inherited from ByteStream:

fileIn	readByte	writeString
pipe	readReady	
pipePartial	writeByte	

Inherited from Collection:

add	forEach	iterate
addMany	forEachBinding	localEqual
addToContents	getAll	map
chooseAll	getAny	merge
chooseOne	getKeyAll	pipe
chooseOneBinding	getKeyOne	prin
deleteAll	getMany	removeAll
deleteBindingAll	getOne	removeOne
deleteBindingOne	hasBinding	setAll
deleteKeyAll	hasKey	setOne
deleteKeyOne	intersects	size
deleteOne	isEmpty	
emptyOut	isMember	

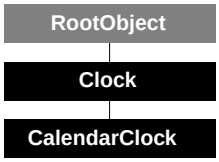
Inherited from LinearCollection:

chooseOneBackwards	findRange	getNthKey
chooseOrdOne	forEachBackwards	getOrdOne
deleteFirst	getFirst	getRange
deleteLast	getLast	localEqual
deleteNth	getMiddle	localLT
deleteRange	getNth	pop

Inherited from Sequence:

addFifth	moveBackward	setFourth
addFirst	moveForward	setLast
addFourth	moveToBack	setNth
addNth	moveToFront	setSecond
addSecond	prepend	setThird
addThird	prependNew	sort
append	setFifth	
appendNew	setFirst	

CalendarClock



Class type: Core class (concrete, sealed)
Resides in: ScriptX and KMP executables
Inherits from: `Clock`
Component: Clocks

The `CalendarClock` class provides an object that tracks current date and time. The instance variable `date` holds a `Date` object that can be set or queried using instance variables of the `Date` class. Every time you query the date object, you get the updated date and time.

The one instance of `CalendarClock` is represented by the global constant `theCalendarClock`. This global is automatically created by the ScriptX runtime environment at startup, with the `rate` and `scale` instance variables both set to 1. The instance `theCalendarClock` uses the current date and time from the underlying operating system for its starting value. If you call the `new` method on `CalendarClock`, to request a new instance, the global instance `theCalendarClock` is returned. For a definition of `theCalendarClock`, see the chapter "Global Constants and Variables."

Instance Variables

Inherited from `Clock`:

`callbacks`	`rate`	`ticks`
`effectiveRate`	`resolution`	`time`
`masterClock`	`scale`	`title`
`offset`	`slaveClocks`	

The following instance variables are defined in `CalendarClock`:

date

self`.date` (read-only) `Date`

Keeps the current date and time as a `Date` instance.

Don't expect the `ticks` instance variable (inherited from `Clock`) to reliably represent an offset from a particular time. Instead, use the value of `date` to represent the current date and time.

rate (Clock)

self.rate (read-only) Number

Inherited from Clock, the rate instance variable is read-only in the CalendarClock class.

scale (Clock)

self.scale (read-only) Number

Inherited from Clock, the scale instance variable is read-only in the CalendarClock class.

time (Clock)

self.time (read-only) Time

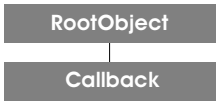
Inherited from Clock, the time instance variable is read-only in the CalendarClock class.

Instance Methods

Inherited from Clock:

addPeriodicCallback	clockAdded	pause
addRateCallback	clockRemoved	resume
addScaleCallback	effectiveRateChanged	timeJumped
addTimeCallback	forEachSlave	waitTime
addTimeJumpCallback	isAppropriateClock	waitUntil

Callback



Class type: Core class (abstract)
Resides in: ScriptX and KMP executables
Inherits from: `RootObject`
Component: Clocks

The `Callback` class provides an abstract mechanism to handle requests for actions related to specific events in the life of a clock. There are several concrete subclasses of `Callback` corresponding to different types of actions:

- `PeriodicCallback` – performs actions at specific intervals
- `RateCallback` – performs actions when the clock's rate changes
- `ScaleCallback` – performs actions when the clock's scale changes
- `TimeCallback` – performs actions when the clock reaches a certain time
- `TimeJumpCallback` – performs actions when the clock's time is reset

You won't usually create an instance of one of these callbacks directly. Instead, you add a callback to a clock using methods defined by the `Clock` class. Once you have the callback, you can set its order and the conditions under which the action will be performed. Callbacks that are specified to be "once only" are performed once, then discarded; others are recalled repeatedly until explicitly discarded. Use the `cancel` method to unschedule a callback. After cancelling a callback, you can schedule it again by setting the value of its `time` instance variable.

Instance Variables

authorData

self`.authorData` (read-write) `Array`

Represents the variable argument list for the function in the `script` instance variable of the callback *self*.

condition

self`.condition` (read-write) `NameClass`

Determines the condition under which the callback *self* will be activated. Valid values for `ScaleCallback`, `RateCallback`, and `TimeJumpCallback` objects are `@lessThan`, `@greaterThan`, `@equal`, `@notEqual`, `@lessThanOrEqual`, `@greaterThanOrEqual`, and `@change` (default).

The default value, `@change`, causes the callback *self* to perform its action with any appropriate change. Valid values for `TimeCallback` and `PeriodicCallback` objects are: `@forward`, `@backward`, and `@either` (default).

label

self`.label` (read-write) *(object)*

Defines a label for the callback *self*, which can be any object useful for identifying the callback. By default, the value is `undefined`.

onceOnly

self.onceOnly (read-write) Boolean

Determines whether the function associated with the callback *self* will be called just once (true) or will be called repeatedly throughout the life of the clock (false).

order

self.order (read-write) Integer

Determines the order of the callback *self* compared with other callbacks of the same priority. When callbacks are scheduled at the same time and with the same priority, those with lower order values will have their functions invoked first. The default value is 0.

priority

self.priority (read-write) NameClass

Determines the priority of the callback *self*. The value may be one of the predefined names @system or @user. By default, the value of priority is set to @user. In most cases, this value should suffice and the priority instance variable should be considered read-only.

script

self.script (read-write) *(function)*

Represents the function used to invoke the action specified for the callback *self*.

Although any global function, anonymous function, or method can be assigned to script, there are differences in how different classes of functions are dispatched. For information on functions and function dispatch, see the "Object System Kernel" chapter of the *ScriptX Components Guide*.

target

self.target (read-write) *(object)*

Represents the first argument for the function in the script instance variable of the callback *self*.

time

self.time (read-write) Time

Indicates the time or time interval at which the callback will fire. To change the time or time interval at which a callback fires, change the value of its time instance variable.

After cancelling a callback, you can make it active again by setting the value of its time instance variable. The act of setting the time instance variable is what schedules the callback.

Instance Methods

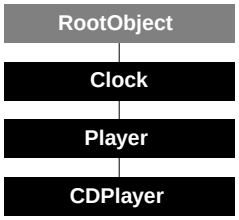
cancel

cancel *self* ⇨ *self*

Cancels the callback *self*, obtained when setting an action for a clock. If the method is currently performing the function associated with the callback, the function will run to completion.

After cancelling a callback, you can make it active again by setting the value of its `time` instance variable. The act of setting the `time` instance variable is what schedules the callback.

CDPlayer



Class type: Loadable class (concrete)
Resides in: `cdcmd.lib`. Works with ScriptX and KMP executables.
Inherits from: `Player`
Component Media Player

The `CDPlayer` class is a loadable extension class provided by Kaleida. The class is a combination of ScriptX code and C code extensions. The `CDPlayer` class lets you play audio CD in a CD-ROM drive while running ScriptX. You can control the audio CD in two ways: by using graphical buttons that appear on the screen, or by calling methods on an instance of `CDPlayer`.

`CDPlayer` is a subclass of `Player`. It supports most of the instance variables and methods that are common to all subclasses of `Player`, such as the instance variables `duration` and `volume`, and the methods `play`, `stop`, `pause` and `fastforward`.

Currently, the `CDPlayer` class lets you interact with one CD-ROM drive. Each time you create a new instance of `CDPlayer`, it establishes a connection to the same CD-ROM drive.

How to Load and Use the CDPlayer Class

`CDPlayer` is a scripted class that can be loaded dynamically. The script files required to load the `CDPlayer` class and create the graphical button interface for it must reside in a directory called `cdplayer` in a directory called `loadable`. The `loadable` directory must be in the same directory as the ScriptX application.

To load the `CDPlayer` class and the graphical button interface, open the `loadme.sx` file in the `loadable/cdplayer` directory using the **Open Title** command in the ScriptX **File** menu. This file in turn loads all the other files needed.

Using the Graphical Interface To Play CD Audio

When the loading is finished, you should see the graphical button interface with five active buttons as shown in Figure 4-3. The buttons let you start, stop, pause, rewind and fastforward the CD in the CD-ROM drive. If you can't see the graphical button interface, look behind the ScriptX Listener window in case it's there.

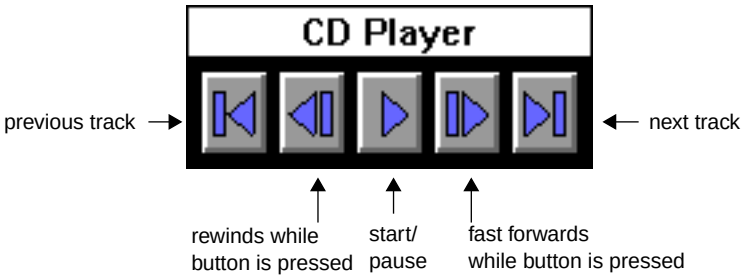


Figure 4-3: The graphical button interface for controlling CD audio

Using the CDPlayer Programmatically

To load the CDPlayer class without loading the graphical button interface, load the file aloadcda.sx from the loadable/cdplayer directory, then load cdplayer.sx.

```
global tmpDir
tmpDir := spawn theStartDir "loadable/cdplayer"
fileIn tmpDir name: "aloadcda.sx"
fileIn tmpDir name: "cdplayer.sx"
```

Whether the CDPlayer class was loaded with or without the graphical button interface, you can play and control CD audio by using methods on an instance of CDPlayer.

To programmatically play the CD in the drive, do the following:

1. Create a new instance of CDPlayer:

   ```
   global mycd:= new CDPlayer
   ```

2. Call the play method to start it playing:

   ```
   play mycd
   ```

3. To change the volume, set the value of the volume instance variable. The value must be an integer between 0 and 255 inclusive, where 0 is the quietest and 255 is the loudest. (Notice that the meaning of the volume instance variable for CDPlayer is different than it is for other audio players.)

   ```
   mycd.volume := 100
   ```

4. Use the methods stop, pause, goToBegin and eject to stop, pause, set the CD player back to the first track, and eject the disc from the drive, respectively.

Setting the volume and audioMuted instance variables on the Windows platform requires a specific wiring of the hardware. The audio connector from the CD-ROM drive needs to be connected to the audio connector on the sound card, as is usually the case for all internal CD drives. If you are using the line in on your SCSI interface to connect your CD drive, setting the volume and audioMuted instance variables has no effect.

Instance Variables

Inherited from Clock:

callbacks	rate	ticks
effectiveRate	resolution	time
masterClock	scale	title
offset	slaveClocks	

Inherited from Player:

audioMuted	globalContrast	globalVolumeOffset
dataRate	globalHue	markerList
duration	globalPanOffset	status
globalBrightness	globalSaturation	videoBlanked

The following instance variables are defined in CDPlayer:

track

self.track (read-write) Fixed

Specifies the current track of the CD Player *self*.

volume

self.volume (read-write) Fixed

Specifies the volume of the CD Player *self*. The range is 0 - 255. The default value is 60.

Instance Methods

Inherited from Clock:

addPeriodicCallback	clockAdded	pause
addRateCallback	clockRemoved	resume
addScaleCallback	effectiveRateChanged	timeJumped
addTimeCallback	forEachSlave	waitTime
addTimeJumpCallback	isAppropriateClock	waitUntil

Inherited from Player:

addMarker	goToBegin	playPrepare
eject	goToEnd	playUnprepare
fastForward	goToMarkerFinish	playUntil
getMarker	goToMarkerStart	rewind
getNextMarker	pause	stop
getPreviousMarker	play	

The following instance methods are defined in CDPlayer:

eject

eject *self* ⇨ Boolean

Physically ejects the CD from the CD-ROM hardware associated with the CD player *self*. It does not delete the CDPlayer instance, therefore when you put another disc in the player, you can continue use the existing CDPlayer instance to play the new CD. However, you need to call the goToBegin method on the CDPlayer instance to set it back to the beginning of the CD. The eject method sets the status instance variable of the CD player to @ejected.

fastForward

fastForward *self* ⇨ Boolean

The CD player *self*advances by ten seconds and continues playing if it was previously playing. This method is inherited from the class Player, but behaves slightly differently for the class CDPlayer than for other Player classes. (For other classes, it continues fast forwarding until you tell it to stop.)

goToNextTrack

goToNextTrack *self* ⇨ Boolean

Sets the seek on the CD to the start of the next track and continues playing if it was playing.

goToPrevTrack

goToPrevTrack *self* ⇨ Boolean

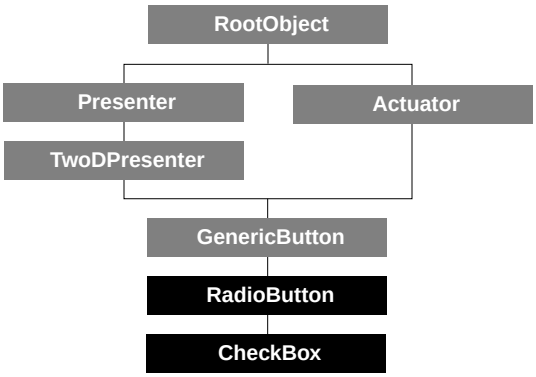
Sets the seek on the CD to the beginning of the previous track and continues playing if it was playing.

rewind

`rewind` *self* ⇨ `Boolean`

The CD player *self* rewinds by ten seconds and continues playing if it was previously playing. This method is inherited from the class `Player`, but behaves slightly differently for the class `CDPlayer` than for other `Player` classes. (For other classes, it continues rewinding until you tell it to stop.)

CheckBox



Class type: Scripted class (abstract)
Resides in: widgets.sxl. Works with ScriptX and KMP executables
Inherits from: RadioButton
Component: User Interface

CheckBox is a user interface Widget Kit class that provides a framed button whose appearance is defined by a stencil object that changes appearance when the button is selected, a text object that is displayed to the right of the button stencil, and a frame that encloses both the button stencil and the text. Clicking the mouse anywhere on or within the check box's frame selects or deselects the button.

A check box is exactly the same as a radio button except for the appearance of the button stencil when the check box is selected or not selected.

Three bitmaps give the button stencil different appearances for different mouse events. These bitmaps are stored in the media directory. The check.bmp bitmap provides the appearance for the button stencil when the check box is not selected; this is square with a black outline and gray fill. The checkC.bmp bitmap provides the appearance for the button stencil when the radio button is selected; this is the same as the unselected appearance but with a black checkmark in the center of the button stencil. The checkDP.bmp bitmap provides the appearance for the button stencil when the mouse button has been clicked on the check box but has not yet been released; this is the same as the unselected appearance but with a black outline of a square inside the edge of the button stencil.

The clipping boundary of a CheckBox object is calculated automatically as a rectangle that encompasses the text stencil and the button stencil.

Creating and Initializing a New Instance

The following script is an example of how to create a new instance of the CheckBox class:

```
myCheckBox := new CheckBox \
        text:"Check Me" \
        frame:(new Frame)
```

The variable myCheckBox contains an initialized CheckBox object. The new method uses the keywords defined in init. The initialization of a check box is exactly the same as the initialization of a radio button.

init

`init` *self* [`fill:`*brush*] [`font:`*font*] [`text:`*string*]
[`boundary:`*stencil*] [`frame:`*frame*] ⇨ *(none)*

self	`CheckBox` object
`fill:`	`Brush` object
`font:`	`Font` object
`text:`	`String` object
`boundary:`	`Stencil` object
`frame:`	`Frame` object

Initializes the `CheckBox` object *self*, applying the values supplied with the keywords to the instance variables of the same name. Creates a new `TextStencil` object to display the specified text in the specified font, and calculates a boundary that encompasses the text stencil and the button stencil. Do not call `init` directly on an instance — it is automatically called by the `new` method.

If you omit one of the keyword arguments, the following defaults are used:

```
fill:whiteBrush
font:theSystemFont
text:"Hello"
boundary:unsupplied
frame:undefined
```

You should not provide a value for `boundary`. If you do not provide a value for `boundary`, then the boundary of the new `CheckBox` object is calculated automatically as a rectangle that encompasses the text stencil and the button stencil.

Instance Variables

Inherited from `Actuator`:

`enabled`	`pressed`	`toggledOn`
`menu`		

Inherited from `Presenter`:

`presentedBy`	`subPresenters`	`target`

Inherited from `TwoDPresenter`:

`bBox`	`height`	`transform`
`boundary`	`IsImplicitlyDirect`	`width`
`clock`	`isTransparent`	`window`
`compositor`	`isVisible`	`x`
`direct`	`needsTickle`	`y`
`eventInterests`	`position`	`z`
`globalBoundary`	`stationary`	
`globalTransform`	`target`	

Inherited from `GenericButton`:

`activateAction`	`pressAction`	`releaseAction`
`authorData`		

Inherited from `RadioButton`:

`fill`	`frame`	`text`
`font`		

Instance Methods

Inherited from Actuator:

activate	press	toggleOff
multiActivate	release	toggleOn

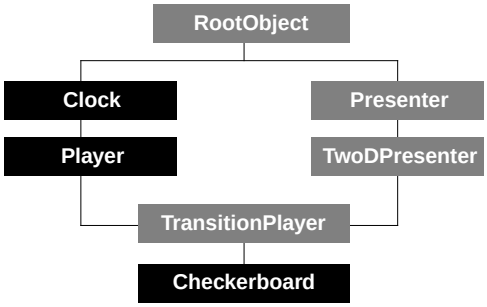
Inherited from TwoDPresenter:

adjustClockMaster	inside	show
createInterestList	localToSurface	surfaceToLocal
draw	notifyChanged	tickle
getBoundaryInParent	recalcRegion	
hide	refresh	

Inherited from GenericButton:

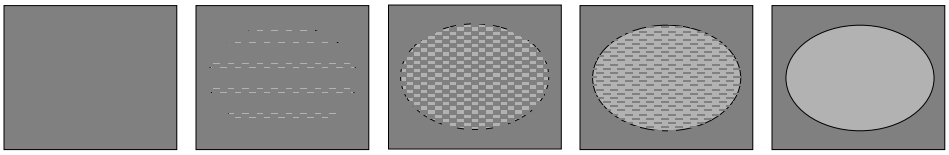
activate	press	release
multiActivate		

CheckerBoard



Class type: Loadable class (concrete)
Resides in: `ltrans.lib`. Works with ScriptX and KMP executables.
Inherits from: `TransitionPlayer`
Component: Transitions

The `CheckerBoard` transition player provides a visual effect that causes the target to gradually appear using a checkerboard pattern as shown below. The target appears when the transition is played forward, and disappears when played backward (transition's rate set to -1).



Directions: *(none)*

Rate: Can play forward or backward.

For a side-by-side illustrations of all transitions, see the Transitions chapter in the *ScriptX Components Guide*.

Creating and Initializing a New Instance

The following script is an example of how to create a new instance of the `Checkerboard` class:

```
myTransition := new Checkerboard \
                    duration:60 \
                    target:myShape
```

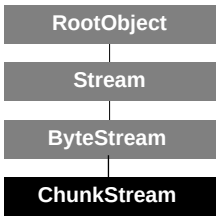
The variable `myTransition` contains the initialized transition. The transition reveals the image `myShape` in a checkerboard pattern and has a duration of 60 ticks.

You determine which space the transition will take effect in by adding this instance into that space. Then, when you play the transition player, `myShape` is transitioned into that space.

The `new` method uses the keywords defined in `init`.

NOTE – For the instance variables and methods, see the `BarnDoor` class.

ChunkStream



Class type: Core class (concrete)
Resides in: ScriptX and KMP executables
Inherits from: `ByteStream`
Component: Media Player

The `ChunkStream` class provides a mechanism for treating disjoint chunks of data as a continuous stream of data. This class was created mainly for use by the class `InterleavedMoviePlayer`.

An `InterleavedMoviePlayer` has an associated byte stream containing interleaved audio and video data for a movie.When the movie plays, the sound data needs to be treated as a stream of continuous data and the video data also needs to be treated as a stream of continuous data. In this case, chunk streams are used to present the audio and video data as continuous, separate streams. See the description of the `InterleavedMoviePlayer` class for more details.

Usually a chunk stream acts as an intermediary stream or a buffer between two other streams, one of which is reading data from the chunk stream and the other that is writing data into the chunk stream. In the case of an `InterleavedMoviePlayer`, the stream doing the reading is an `AudioStream` or a `VideoStream` used as the media stream for a media player, while the stream doing the writing is a `ByteStream` containing interleaved movie data. The chunk stream's read pointer keeps track of the place in the stream that is being read, while the write pointer keeps track of the place in the stream that is being written to.

When the chunk stream is full, no more data can be written into it until some data has been read, thus freeing up space. You can change how much data a chunk stream can hold by setting the value of the `chunkCapacity` instance variable.

Chunk streams are created automatically as needed, usually when a movie is imported as an `InterleavedMoviePlayer`. Users would not usually need to interact directly with chunk streams.

Instance Variables

The following instance variables are defined in `ChunkStream`:

cacheLength

self`.cacheLength` (read-write) `Integer`

Specifies the number of chunks in the chunk stream *self* that are kept after they are used, in case they are needed again. For example, if the player using the chunk stream plays backwards after playing forwards, it will reuse data it has just used. The value of this instance variable can safely be set to zero.

chunkCapacity

self.chunkCapacity (read-write) Integer

Specifies the number of chunks that the chunk stream *self* can hold before it is full. If this value is too low, playback quality may suffer. If the value is too high, memory consumption may be too high.

isFull

self.isFull (read-write) Boolean

Returns `true` if the chunk stream *self* is full, or `false` if it is not full. It is full when the number of queued chunks is equal to the chunk capacity. (That is, the value of the `numChunks` instance variable is the same as the `chunkCapacity` instance variable.)

isSupressed

self.isSuppressed (read-write) Boolean

Specifies whether or not the data in the chunk stream is suppressed. If the value is `true`, the data is suppressed and will not be read, usually because the player using the chunk stream is muted or blanked. If the value is `false`, the data is not suppressed and will be read.

numChunks

self.numChunks (read-write) Integer

Specifies the number of chunks currently in the chunk stream *self*. This value cannot exceed the value of the `chunkCapacity` instance variable.

Instance Methods

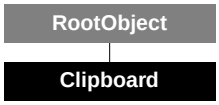
Inherited from `Stream`:

cursor	next	seekFromStart
flush	plug	setStreamLength
isAtFront	previous	streamLength
isPastEnd	read	write
isReadable	readReady	writeReady
isSeekable	seekFromCursor	
isWritable	seekFromEnd	

Inherited from `ByteStream`:

fileIn	readByte	writeString
pipe	readReady	
pipePartial	writeByte	

Clipboard



Class type: Core class (concrete, sealed)
Resides in: ScriptX and KMP executables
Inherits from: `RootObject`
Component: Title Management

The `Clipboard` class represents an area of memory where text can be copied to and pasted from. The one instance of `Clipboard` is represented by the global constant `theClipboard`. This instance is automatically created by the ScriptX runtime environment at startup.

The clipboard allows the exchange of text between any two places—within a ScriptX title, between ScriptX titles, and between a ScriptX title and another application. The current version of the clipboard supports the exchange of only text, and not other kinds of ScriptX objects.

Operations commonly associated with the clipboard are cut, copy, and paste. These operations are methods defined on `TitleContainer` and `Window`. These operations have no specific default implementation; you must implement them for each particular window. The implementation must include the means by which the user can select objects for copying or cutting. For more information, refer to the `cutSelection`, `copySelection` and `pasteToSelection` methods in the `TitleContainer` and `Window` classes.

The operating system has a system clipboard, and every application also has its own clipboard. When using `cutSelection`, `copySelection`, and `pasteToSelection` within an application, such as ScriptX runtime environment, the local clipboard is used. However, when you switch applications, the system clipboard is used for transferring media between applications.

The clipboard will eventually be able to simultaneously hold objects of different media types, such as text, picture and sound.

When ScriptX is switched out, the ScriptX clipboard tries to coerce its contents to text and place the text on the system clipboard. Likewise, when ScriptX switches in, the ScriptX clipboard tries to coerce the contents of the operating system clipboard to text and place it on the ScriptX clipboard.

Because an instance of `Clipboard` is created at startup, there is no need to create another instance.

To understand how the clipboard works, see the “Clipboard” section of the Title Management chapter in the *ScriptX Components Guide*.

Instance Variables

typeList

self`.typeList` (read-only) `Array`

Returns a list of the different media types currently on the clipboard which you can paste into your title. This list is empty if nothing is on the clipboard. It is updated every time an object is cut or copied to the clipboard. This way you can find out what kind of media is available before doing a paste.

The clipboard can hold media that was cut or copied from either ScriptX or the system clipboard, whichever clipboard was used most recently. If the media on the clipboard was copied from within ScriptX, `typeList` contains the value `@native`. If the media

was copied from some other application, its value can only contain `@text` (or is empty if nothing has been copied to the clipboard). These two values have the following meanings:

`@native`	Look on the ScriptX clipboard for any kind of ScriptX object
`@text`	Look on the system clipboard for text

These values are mutually exclusive; the list `typeList` can contain either `@native` or `@text`, but not both. These media types determine whether `getClipboard` should look for data on the system or ScriptX clipboard, and what kind of media to look for. Clipboard media can come from one of two places: the system clipboard, or the ScriptX clipboard. If it comes from the ScriptX clipboard, the media is native to ScriptX. If it comes from the system clipboard, the media is in a system format, not native ScriptX media.

The system clipboard and ScriptX clipboard work as follows: It is the responsibility of any application that uses the system clipboard to make its data available to other applications by converting its media to system-standard types. This must happen whenever the user switches out of that application. The system clipboard can simultaneously hold different types of media. When the user switches from one application to another, whatever media is on the first application's is converted to system-standard types (for text, graphics, sound, video) and placed on the system clipboard. The instance variable `typeList` lists the media types for the object currently on the system clipboard that ScriptX recognizes.

For example, if the system clipboard currently contains text, picture and sound, `typeList` would return `@text`, because that's the only media type currently on the system clipboard that it recognizes. (As later versions of ScriptX allow more media types, such as `@picture` and `@sound`, the list `typeList` could contain any combination of `@text`, `@picture`, and `@sound`, as well as `@native`.)

ScriptX does not do any coercion for `@native`. It only coerces ScriptX objects when it switches out of ScriptX to another application.

Instance Methods

getClipboard

`getClipboard` *self type* ⇨ *(object)*

self	`Clipboard` object
type	`Name` object: `@native` or `@text`

Returns the object currently on the clipboard *self*. If this method finds *type* in `typeList`, then it returns the object on the clipboard of that type as follows:

- If *type* is `@text`, a `String` object is returned from the system clipboard.
- If *type* is `@native`, then the native ScriptX object is returned from the ScriptX clipboard.

If this method does not find *type* in `typeList`, then it returns `undefined`.

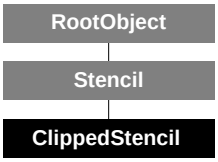
The only clipboard coercing ScriptX can do is to coerce text media to a string object when switching from ScriptX to another application. Note that text that is cut or copied from the Listener window (in the ScriptX authoring environment) is given the *type* `@text`—it is not considered data native to ScriptX.

setClipboard

`setClipboard` *self newObject* ⇨ *newObject*

Clears the contents of the clipboard and then puts *newObject* on the clipboard.

ClippedStencil



Class type: Core class (concrete)
Resides in: ScriptX and KMP executables
Inherits from: `Stencil`
Component: 2D Graphics

The `ClippedStencil` class provides a mechanism for clipping the image in one stencil with the image in another. This is useful when you want to use a clipped stencil with a `TwoDShape` instance, and in other cases where you expect to reuse a particular clipped image repeatedly.

To use `ClippedStencil`, you place an image source stencil into the `outline` instance variable and the clipping stencil into the `clip` instance variable. You can then render the `ClippedStencil` object anytime you want to use the resulting clipped image.

Creating and Initializing a New Instance

The following script is an example of how to create a new instance of the `ClippedStencil` class:

```
myStencil := new ClippedStencil \
             outline:imageStencil \
             clip:clipStencil
```

The variable `myStencil` contains a newly initialized instance of `ClippedStencil`. Its outline stencil is `imageStencil` and its clipping stencil is `clipStencil`.

init

`init` *self* `outline:`*stencil* `clip:`*stencil* ⇨ *(none)*

self	`ClippedStencil` object
`outline:`	`Stencil` object representing the image to be presented
`clip:`	`Stencil` object representing the stencil to be clipped

Initializes the `ClippedStencil` object *self*, applying the arguments as follows: `outline` sets the image to be presented, `clip` sets the stencil used to clip the image. Do not call `init` directly on an instance—it is automatically called by the `new` method.

Instance Variables

Inherited from `Stencil`:

`bBox`

The following instance variables are defined in `ClippedStencil`:

clip

self.clip (read-write) Stencil

Specifies the stencil used in rendering operations to clip the image specified by the outline instance variable. The stencil used to initialize this value may be copied and converted to a Region instance for more efficient clipping. If the Stencil instance supplied to this instance variable been transformed (so that the x1 and y1 values of its bBox instance variable are non-zero), that transformation is ignored.

outline

self.outline (read-write) Stencil

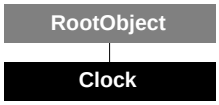
Specifies the stencil used as the image source for rendering.

Instance Methods

Inherited from Stencil:

inside	onBoundary	transform
intersect	subtract	union

Clock



Class type: Core class (concrete)
Resides in: ScriptX and KMP executables
Inherits from: `RootObject`
Component: Clocks

The `Clock` class defines objects that keep time in both ticks and in seconds, minutes, and hours. The time increment marked by a clock's ticks is determined by its scale and its effective rate. The "Clocks" chapter in the *ScriptX Components Guide* describes this time keeping in greater detail.

In addition to time keeping, `Clock` objects provide services related to timing. For example, they can be organized in hierarchies and perform actions at key points in their timing cycles.

Every clock created is associated with a particular `TitleContainer` instance. This enables a whole title to be started, paused, restarted, and stopped in synchronization. If you don't supply a specific title for a clock, it is assigned to `theScratchTitle` global object.

One global instance of `Clock` is created by ScriptX at system startup and is stored in the global constant `theEventTimeStampClock`. This instance is used to generate time-stamps for events; it can be used by titles to generate time values for other purposes as well. Unless reset, its `ticks` value represents milliseconds since the ScriptX runtime environment started. A script can create other instances of `Clock` at any time. Each instance starts with time and rate set to 0. The scale of a clock should be set explicitly when you create a new clock.

To start a clock, you set its `rate` to a value other than 0. To stop a clock, set its `rate` to 0. To reset a clock, set its `time` or `ticks` to 0; setting either of these instance variables resets the other.

Note – When a clock hierarchy is loaded from a storage container, the top clock's rate is always set to zero—even if the clock was running when it was saved.

Creating and Initializing a New Instance

The following script is an example of how to create a new instance of the `Clock` class:

```
myClock := new Clock \
           scale:2 \
           masterClock:myMasterClock
```

The variable `myClock` contains the initialized `Clock` instance. It has 2 ticks on the face of the clock, and its `masterClock` is set to be `myMasterClock`. The new clock's `title` is automatically set to that of the `masterClock`. By default, the clock is stopped; to start it, set `myClock.rate` to a non-zero value —the value 1 causes this clock to run forward at one tick per second relative to its master clock's rate).

init

`init` *self* [`scale:`*integer*] [`masterClock:`*clock*] [`title:`*titleContainer*] ⇨ *(none)*

self	`Clock` object
`scale:`	`Integer` object to use as this clock's scale
`masterClock:`	`Clock` object to use as master for this clock
`title:`	`TitleContainer` object that this clock belongs to

Initializes the `Clock` object *self*, applying the values supplied with the keywords to the instance variables of the same name. Do not call `init` directly on an instance—it is automatically called by the `new` method.

If you omit an optional keyword, its default value is used. The defaults are:

`scale:1`
`masterClock:undefined` (the clock is a top clock)
`title`: `theScratchTitle` global instance (or the `title` of the `masterClock`)

Instance Variables

callbacks

self`.callbacks` (read-only) `Array`

An array containing all callbacks associated with the clock *self*. You don't add callbacks to this array directly; instead, use the appropriate methods (such as `addRateCallback` or `addTimeCallback`) to associate a callback with a clock.

effectiveRate

self`.effectiveRate` (read-only) `Number`

Specifies the effective rate of the clock *self*, which defines its speed in actual seconds. The value of `effectiveRate` is the product of the values of the `rate` instance variable and the master clock's `effectiveRate` instance variable. This is the same as the product of the value of `rate` for *self* and `rate` for the values of all clocks above *self*. If a clock has no master clock, `effectiveRate` is the same as `rate`.

masterClock

self`.masterClock` (read-write) `Clock`

Specifies the clock directly above the clock *self* in the timing hierarchy. If the value is `undefined`, this clock is a top clock.

offset

self`.offset` (read-write) `Integer`

Specifies the ticks between the master clock's zero time and the clock *self*'s time, measured in the master's scale.

rate

self`.rate` (read-write) `Fixed`

Specifies the speed of the clock *self* expressed in terms of its master clock's rate. If you think of the clock as a wall clock with a face, the `rate` is the speed at which the hand sweeps the face of that clock. If the clock is a top clock, the `rate` is the number of ticks per second. To stop a clock, set its rate to zero; this also stops any slaves belonging to

the clock. To start a clock, set its rate to some value other than zero; a positive value runs forward and a negative value runs backward. When you start a clock by setting its rate, any slaves will also begin running at their specified rates.

resolution

self.`resolution` (read-only) `Integer`

Specifies the resolution of the clock *self*, the highest number of real ticks per second that the clock can achieve. This value is based on the capabilities of the underlying hardware clock.

scale

self.`scale` (read-write) `Integer`

Specifies the ticks (which is the number of increments of `time`) in one revolution of the clock *self*. If you think of the clock as being a wall clock with a face, the `scale` is the number of tick marks on the face of that clock. Attempting to set the scale to zero causes ScriptX to report an exception.

slaveClocks

self.`slaveClocks` (read-only) `Array`

An array containing all slave clocks associated with the clock *self*. You never add clocks to this array directly; instead, a clock is added to this list when the clock *self* is set as the slave clock's `masterClock` instance variable.

ticks

self.`ticks` (read-write) `Integer`

Specifies the current time in the clock *self*'s ticks. Setting this value resets the time of the clock and all clocks below it in a timing hierarchy.

time

self.`time` (read-write) `Time`

Specifies the current time in the hours, minutes, seconds, and ticks of the clock's *self*. Setting this value resets the time of the clock and all clocks below it in the timing hierarchy.

title

self.`title` (read-write) `TitleContainer`

Specifies the `TitleContainer` instance that the clock *self* belongs to. You can explicitly set the value of this instance variable only for top clocks. Setting this variable adds the clock *self* to the title container's `topClocks` array. This enables a whole title to be started, paused, restarted, and stopped in synchronization. For slave clocks, this value is set automatically to the `title` of the top clock in its hierarchy. Attempting to set this value explicitly for a slave clock raises an exception. If `title` isn't set explicitly for a top clock, the value defaults to `theScratchTitle` global object.

Instance Methods

addPeriodicCallback

`addPeriodicCallback` *self script target argArray timePeriod* ⇨ `PeriodicCallback`

self	`Clock` object for which the action is performed
script	`Function` object that performs the actual action
target	First argument to the *script* function
argArray	`Array` object with the arguments to the *script* function
timePeriod	`Integer` object representing the number of ticks between each action

Creates and returns a callback object for the clock *self* which will call the function *script* periodically at tick intervals represented by *timePeriod*. The *target* argument specifies the first argument to the *script* function; in the case where *script* is a generic function, *target* is the object whose method will be invoked. The *argArray* argument is an `Array` object whose members represent the list of arguments to the *script* function. This array takes the form #(*arg2*, *arg3*, ...).

The `PeriodicCallback` instance variable `skipIfLate` lets you specify what should happen if the callback falls behind schedule. Set the value for `skipIfLate` to `true` if you want the *script* function called at exactly the correct time interval; it will be skipped should the callback fall behind. The default value for `skipIfLate` is `false`, which causes the *script* function to be called regardless of whether it's behind schedule.

addRateCallback

`addRateCallback` *self script target argArray onceOnly* ⇨ `RateCallback`

self	`Clock` object for which the action is performed
script	`Function` object that performs the actual action
target	First argument to the *script* function
argArray	`Array` object with the arguments to the *script* function
onceOnly	`Boolean` object representing the number of times to perform the action

Creates and returns a callback object for the clock *self* that will call the *script* any time the rate of the clock *self* changes. The *target* argument specifies the first argument to the *script* function; in the case where *script* is a generic function, *target* is the object whose method will be invoked. The *argArray* argument is an `Array` object whose members represent the list of arguments to the *script* function. This array takes the form #(*arg2*, *arg3*, ...).

The *script* function is called not only when the actual rate of the clock changes, but also when its effective rate changes (that is, when a clock above it in the timing hierarchy changes rate). If *onceOnly* is `true`, the clock performs the script the first time the clock's rate changes and then discards it. If it is `false`, the clock performs the script every time the rate changes.

addScaleCallback

`addScaleCallback` *self script target argArray onceOnly* ⇨ `ScaleCallback`

self	`Clock` object for which the action is performed
script	`Function` object that performs the actual action
target	First argument to the *script* function
argArray	`Array` object with the arguments to the *script* function
onceOnly	`Boolean` object representing whether to perform the action once or every time the scale changes

Creates and returns a callback object for the clock *self* that will call the *script* any time the scale of the clock *self* changes. The *target* argument specifies the first argument to the *script* function; in the case where *script* is a generic function, *target* is the object whose method will be invoked. The *argArray* argument is an `Array` object whose members represent the list of arguments to the *script* function. This array takes the form `#`(*arg2*, *arg3*, ...). If *onceOnly* is `true`, the clock performs the action the first time the clock's scale changes and then discards it. If it is `false`, the clock performs the action every time the scale changes.

addTimeCallback

`addTimeCallback` *self script target argArray timeValue onceOnly*

⇨ `TimeCallback`

self	`Clock` object for which the action is performed
script	`Function` object that performs the actual action
target	First argument to the *script* function
argArray	`Array` object with the arguments to the *script* function
timeValue	`Integer` object representing the time of the action
onceOnly	`Boolean` object representing the number of times to perform the action

Creates and returns a callback object for the clock *self* that will call the *script* function when the clock *self* reaches the time given in *timeValue*. The *target* argument specifies the first argument to the *script* function; in the case where *script* is a generic function, *target* is the object whose method will be invoked. The *argArray* argument is an `Array` object whose members represent the list of arguments to the *script* function. This array takes the form `#`(*arg2*, *arg3*, ...). If *onceOnly* is `true`, the clock performs the action the first time it reaches the time specified in *timeValue*, and then discards it. If it is `false`, the clock performs the action whenever it reaches *timeValue* time (as when, for example, the clock's time is reset to zero and it again runs up to the *timeValue* time).

addTimeJumpCallback

`addTimeJumpCallback` *self script target argArray onceOnly*

⇨ `TimeJumpCallback`

self	`Clock` object for which the action is performed
script	`Function` object that performs the actual action
target	First argument to the *script* function
argArray	`Array` object with the arguments to the *script* function
onceOnly	`Boolean` object representing the number of times to perform the action

Creates and returns a callback object for the clock *self* that will call the *script* any time the `time` instance variable is set explicitly (not as the regular passage of time). The *target* argument specifies the first argument to the *script* function; in the case where *script* is a generic function, *target* is the object whose method will be invoked. The *argArray* argument is an `Array` object whose members represent the list of arguments to the *script* function. This array takes the form `#`(*arg2*, *arg3*, ...). If *onceOnly* is `true`, the *script* function performs its action the first time the clock's time is explicitly set. If it is `false`, the *script* function performs the action whenever the clock's time is changed explicitly.

clockAdded

`clockAdded` *self master slave* ⇨ *(none)*

self	`Clock` being made the new master.
master	`Clock` object that is the master clock
slave	`Clock` object invoking this method

Invoked by a clock *(slave)* on its new master *(master)* when the slave changes its `masterClock` instance variable. When first called, the *self* argument is the new master clock. This method is then called recursively up the timing hierarchy until reaching the top clock. As implemented in `Clock`, it does no other work.

This method can be overridden to perform class-specific actions related to the new clock being added to the timing hierarchy. When overriding this method, a `Clock` subclass may want to traverse the timing hierarchy below *slave* to see that all members are of the appropriate class.

clockRemoved

`clockRemoved` *self master slave* ⇨ *(none)*

self	`Clock` object being removed as the master
master	`Clock` object that is the former master of *slave*
slave	`Clock` object invoking this method

Invoked by a clock on its old master *self* when it changes its `masterClock` instance variable. This method is called recursively on master clocks up the timing hierarchy. As implemented in `Clock`, it does no other work and returns `undefined`.

This method can be overridden to enable masters to "clean up" any details when *slave* is removed from *master*.

effectiveRateChanged

`effectiveRateChanged` *self* ⇨ *(none)*

Invoked on the clock *self* when its effective rate changes—that is, when the clock's own rate changes or when the rate of any clock above it in its timing hierarchy changes. Subclasses of `Clock` override this method to perform specific actions whenever a clock's effective rate changes; the more general alternative is to use the `addRateCallback` method to assign a specific action to rate changes.

forEachSlave

`forEachSlave` *self script arg* ⇨ *self*

self	`Clock` object
script	`Function` object
arg	Any object as an argument to the *script*.

Calls the `Function` *script* on each slave of the `Clock` *self*. The function *script* has two arguments: the slave clock and the *arg* object.

isAppropriateClock

`isAppropriateClock` *self slave* ⇨ `Boolean`

self	`Clock` object to be added to a hierarchy
slave	`Clock` object to be made a slave of *self*

Tests whether the `Clock` object *self* is an appropriate clock to serve as master of the clock *slave*. This method returns `true` when *slave* is an instance of the same class as *self* (or a subclass thereof).

pause

`pause` *self* ⇨ `Boolean`

Stops the clock *self*, saving its current `rate`. After calling `pause` on a clock, call its `resume` method to restart it. The `pause` method is invoked automatically on each clock belonging to a `TitleContainer` object when that object's `pause` method is invoked.

If you call `pause` repeatedly on a clock, you must call `resume` the same number of times to restart it.

resume

`resume` *self* ⇨ `Boolean`

Restarts the clock *self* at its previous rate after it has been paused through its `pause` method. Note that if you call `pause` on a clock more than once, you need to call `resume` the same number of times for the clock to actually restart. This method returns `false` if the clock remains paused after it is called, `true` if the clock is running after it is called.

timeJumped

`timeJumped` *self arg2 arg3* ⇨ *(none)*

self	`Clock` object
arg2	An object
arg3	An object

Called on the clock *self* whenever its time is set explicitly. Override this method—instead of calling `addTimeJumpedCallback`—when creating subclasses of `Clock` that should perform specific behavior when their time is set explicitly.

waitTime

`waitTime` *self timeValue* ⇨ *(none)*

self	`Clock` object controlling wait-time
timeValue	`Time` object representing the duration to wait

Blocks the current thread object waiting on the clock *self* for the length of time specified in *timeValue*. Note that multiple threads may wait on the same clock and that this method may be called on system clocks such as `theCalendarClock` and `theEventTimeStampClock` in code that needs to wait for a particular duration.

Note – Don't call `waitTime` on a stopped clock—a clock whose rate or effective rate is 0—or your title may hang indefinitely.

waitUntil

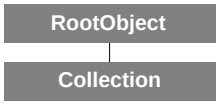
`waitUntil` *self timeValue* ⇨ *(none)*

self	`Clock` object to wait
timeValue	`Time` object specifying time to wait until

Blocks the current thread object until the `time` of *self* reaches *timeValue*. If that time has already passed, this method has no effect. Note that multiple threads may wait on the same clock and that this method may be called on system clocks such as `theCalendarClock` and `theEventTimeStampClock` in code that needs to wait until a particular time.

Note – Don't call `waitUntil` on a stopped clock—a clock whose rate or effective rate is 0—or your title may hang indefinitely.

Collection



Class type: Core class (abstract)
Resides in: ScriptX and KMP executables
Inherits from: `RootObject`
Component: Collections

`Collection` is the root abstract class for groups of elements. The elements of a collection can be sorted or unsorted, and implicitly or explicitly keyed. Some collections, such as strings, store elements that are not objects. (Although characters in a string are not objects, the `String` class implements the Collection protocol for elements of a string as if they were.) Some collections can hold objects of any class such as numbers, strings, bitmaps, or sounds, while others are restrictive. Collections allow you to supply either a list of single values or a list of key-value pairs. Collections can have a fixed, variable, or unbounded number of items.

A collection can store its elements in any data structure, such as a linked list, an array, a hash table, or a B-tree. Each collection class implements its own version of methods that are required for managing data. See page 176 for a list of methods that a collection subclass must implement.

Collections can contain almost any object, including other collections. Each subclass of `Collection` responds to the same set of protocols with a similar behavior, but implements that behavior in a different way. For example, `Array`, `LinkedList`, `HashTable`, and other collection classes all implement the `addMany` and `deleteOne` methods. Each collection has advantages and disadvantages. For example, an array offers outstanding linear access, but performance in adding and removing elements decreases as the size gets large. For a comparison of collection classes on various performance and memory issues, see the "Collections" chapter of the *ScriptX Components Guide*.

The `empty` object is a global constant that can be supplied as a key or returned as a value. For more details, see the description of `empty` on the `EmptyClass`.

Note – Never insert the `empty` object into a collection as a value. A number of collection methods return `empty` to signify that no matching elements were found. Placing the `empty` object in the collection would cause these methods to behave erratically, and possibly fail altogether. Some collections report the `badValue` exception if you try to insert `empty` as a value, or `badKey` if you try to insert is as a key.

The class `Collection` provides a default implementation for most of the `Collection` methods. This default implementation is based on using an `Iterator` object to get access to each member of the collection. You can create a subclass of `Collection`, and by implementing a corresponding iterator, your class will inherit a nearly complete implementation of the collection protocol. Later on, you can specialize some of these inherited methods using more optimal algorithms.

Generic functions in the Collection protocol that use an iterator depend on the integrity of that iterator. Any collection that modifies itself can potentially harm the integrity of its iterator. In particular, note the potential for conflict if a collection is modified by another thread while it is executing a method such as `forEach` or `forEachBinding`. You can protect the integrity of a collection by creating a `Lock` object and applying the method `acquire` to that lock before calling `forEach` and then applying `relinquish` to that lock after the completion of the call to `forEach`. For more information, see the discussion of collections and threads in the "Collections" chapter of the *ScriptX Components Guide*.

Generic functions names in the Collection protocol often have similar component phrases or words, for example, `deleteBindingOne`, `deleteKeyOne` and `deleteOne`. The naming of such generic functions follows this convention: the word "key" signifies that it matches or returns keys, and the word "binding" signifies that it matches or returns bindings (key-value pairs). If both "key" and "binding" are missing, the method matches or returns values.

In the subsequent method descriptions, whenever we say that a value is deleted, it means that a key-value pair is deleted.

Methods defined in `Collection` can be put into the following categories:

Table 4-3:

ADDING	DELETING	ACCESSING
add	`deleteAll`	`getAll`
addMany	`deleteBindingAll`	`getAny`
merge	`deleteBindingOne`	`getKeyAll`
`pipe`	`deleteKeyAll`	`getKeyOne`
	`deleteKeyOne`	`getMany`
	`deleteOne`	`getOne`
SETTING	`emptyOut`	
`setAll`	`removeAll`	GENERAL
`setOne`	`removeOne`	`forEach`
		`forEachBinding`
SEARCHING	MATCHING	`isEmpty`
`chooseAll`	`hasBinding`	`isMember`
`chooseOne`	`hasKey`	`iterate`
`chooseOneBinding`	`intersects`	`map`

These methods are generic functions which can be specialized for subclasses of `Collection`. Which generic function to use depends on what you want to do and what return value you want. For example, there are various ways to add something to a collection. Use `add` to add one item to a collection, and use `addMany` to add a collection to another collection. These two generic functions have no return value and are generally used to populate a collection. Use `merge` if you want to add one collection to another and have a new collection containing both of them returned. (This does not alter either of the source collections.) Use `pipe` if you want to modify a collection by adding another one to it and then have that modified collection returned.

The classes `LinearCollection` and `Sequence` provide additional functionality that depends on the order and positioning of elements, such as getting and setting elements at particular positions, rearranging elements, and traversing a collection in reverse.

See *ScriptX Language Guide* for a discussion of generic functions such as `forEach`, `forEachBinding`, `forEachBackwards`, `map`, `removeOne`, `removeAll`, `chooseOne`, `chooseAll`, `chooseOneBinding`, and `chooseOneBackwards`, which take a function as an argument and apply that function to each element in a collection.

Creating and Initializing a New Instance

Because `Collection` is an abstract class, you cannot create an instance of it. However, you can instantiate any concrete subclass of `Collection` by calling `new` on it; `new` calls `init` and then `afterInit`. The following script is an example of how to create an instance of a subclass of `Collection`, specifically an instance of the `SortedKeyedArray` class:

```
myArray := new SortedKeyedArray \
           initialSize:100 \
               keys:#("bird", "mammal", "reptile") \
               values:#("robin", "dog", "snake")
```

The variable `myArray` contains an initialized instance of `SortedKeyedArray` with the three key-value pairs: (`"bird"`, `"robin"`) (`"mammal"`, `"dog"`) (`"reptile"`, `"snake"`). The `new` method uses the keywords defined in the `init` method of the particular subclass of `Collection` and the `afterInit` method of `Collection`.

afterInit

`afterInit` *self* [`keys:`*collection*] [`values:`*collection*] ⇨ *self*

self	`Array` object
`keys:`	`Collection` object
`values:`	`Collection` object

Applies the arguments as the initial keys and values in the collection. That is, it adds all the given key-value pairs as if by calling `add` *collection key value* in the order they are given.

The keywords `keys` and `values` have no defaults.

Note – The `keys` and `values` keywords are available to all subclasses of `Collection`, but are omitted from the documentation for brevity. For example, the `Array` class has the four keywords: `initialSize`, `growable`, `keys`, and `values`, but only the first two are documented with that class.

Class Methods

pipe

`pipe` *self other* ⇨ `Collection`

self	`Class` object
other	`Collection` object

Pipe is equivalent to the following:

`pipe (new` *self*`)` *other*

This method is a shorthand for the `pipe` instance method on a new instance of a given class.

Instance Variables

bounded

self.bounded (read-only) Boolean

Specifies whether there is an upper or lower limit (or both) on the size of the collection *self*.

The instance variable bounded is descriptive only. Specializing the boundedGetter method to return true does not modify any actual behavior. It is up to a class or object that is bounded to redefine its own behavior.

iteratorClass

self.iteratorClass (read-only) *(class)*

Specifies the class to use for an iterator on the collection *self*.

keyEqualComparator

self.keyEqualComparator (read-only) *(function)*

Specifies the function used to test whether two keys in the collection *self* are the same. See "Comparison Functions" in the Collections chapter of the *ScriptX Components Guide* for a list of built-in functions you can use.

Although any global function, anonymous function, or method can be assigned to keyEqualComparator, there are differences in how different classes of functions are dispatched. For information on functions and function dispatch, see the "Object System Kernel" chapter of the *ScriptX Components Guide*.

keyUniformity

self.keyUniformity (read-only) NameClass

Specifies the uniformity of the keys held in the collection *self*. Possible values for keyUniformity are @commonSuperclass and @sameClass. See the uniformity instance variable for more information.

The instance variable keyUniformity is descriptive only. Creating a new collection class that specializes the keyUniformityGetter method does not modify any actual behavior. It is up to a class or object that restricts its keys to redefine its own behavior.

keyUniformityClass

self.keyUniformityClass (read-only) *(class)*

Specifies the class referred to by keyUniformity for the collection *self*. See the uniformityClass instance variable for more information.

The instance variable keyUniformityClass is descriptive only. Creating a new collection class that specializes the keyUniformityClassGetter method does not modify any actual behavior. It is up to a class or object that restricts its keys to redefine its own behavior.

maxSize

self.maxSize (read-only) Integer

Specifies the maximum size of the collection *self*. For most collections, this value is posInf. Notable exceptions are the bounded arrays: Single, Pair, Triple, and Quad. If you create an instance of a collection, such as Array, that is not growable, the value of maxSize is set to the value you supply with the keyword initialSize.

The instance variable `maxSize` is descriptive only. Creating a new collection class that specializes the `maxSizeGetter` method does not modify any actual behavior. It is up to a class or object that specializes `maxSize` to redefine its own behavior.

minSize

self`.minSize` (read-only) `Integer`

Specifies the minimum size of the collection *self*, usually `0`. Notable exceptions are bounded arrays: `Single`, `Pair`, `Triple`, and `Quad`. For these collections, the value of `minSize` is set to 1, 2, 3, or 4, respectively.

The instance variable `minSize` is descriptive only. Creating a new collection class that specializes the `minSizeGetter` method does not modify any actual behavior. It is up to a class or object that specializes `minSize` to redefine its own behavior.

mutable

self`.mutable` (read-only) `Boolean`

Specifies whether or not items in the collection *self* can be changed using `Collection` methods. For most collections, the value of `mutable` is `true`. For discrete range classes, the value of `mutable` is `false`, even though you can change their bounds or increments. This is because you cannot alter them using `Collection` methods. `StringConstant` is another `Collection` class that is not mutable. A `StringConstant` object must be coerced to a `String` object to be mutable.

The instance variable `mutable` is descriptive only. Creating a new collection class that specializes the `mutableGetter` method does not modify any actual behavior. It is up to the new class to redefine its own behavior.

mutableCopyClass

self`.mutableCopyClass` (read-only) *(class)*

Specifies the class to instantiate when making a mutable, unbounded copy of the collection *self*. In most subclasses of `Collection`, the mutable copy class is the same class as the collection *self*. This class is used to create a copy that is mutable and unbounded, in case the original collection isn't. The `mutableCopyClass` instance variable is required by several other methods defined by `Collection`, including `chooseAll`, `getAll`, `getMany`, `map`, and `merge`.

proprietored

self`.proprietored` (read-only) `Boolean`

When `true`, the collection *self* protects its keys and values by preventing them from being indirectly altered; it ensures that its keys and values can be modified only by using "set" methods directly on the collection itself. When `false`, items can be changed indirectly by anyone that has access to that item in the collection. All collections built-in to ScriptX have `proprietored` set to `false` except the concrete `Range` classes, which are set to `true`.

For example, given a collection with `proprietored` set to `false`, if it contains the string "I am here" as its first value, someone can use `getFirst` to get a string object that is a pointer to the original text. Then any change they make on that string object (not on the collection) modifies the string that's in the collection. With `proprietored` set to `true`, `getFirst` would have given out a *copy* of the full string, with no connection to the original; changing that string would not change the original. The only way then to change the original string would be to use a "set" method on the collection, such as `setFirst`. (Some "set" methods are `setOne` and `setAll` in `Collection`, and `setNth`, `setFirst`, `setLast` in `Sequence`.)

If `proprietored` is true, then a collection keeps its contents protected by handing out only copies of its keys or values when they are requested. Likewise, when someone sets a key or value in the collection, it makes a copy and puts this copy in the collection.

The instance variable `proprietored` is descriptive only. Creating a new collection class that specializes the `proprietoredGetter` method to return `true` does not modify any actual behavior. It is up to a new class that is proprietored to redefine its own access methods.

size

self`.size` (read-only) `Integer`

Specifies the size of the collection *self*, which may in fact be `posInf`, the number representing positive infinity (for an open-ended range, for example). The generic function `sizeGetter` acts as an interface to the generic `size`, also implemented by collections.

uniformity

self`.uniformity` (read-only) `NameClass`

Specifies whether all possible values in the collection *self* are of the same class or instead have a common superclass. In the least uniform collection, all possible values would have at least one common superclass, that being `RootObject`. The `uniformity` instance variable can take only one of two values: `@commonSuperclass` and `@sameClass`.

- If the value of `uniformity` is `@commonSuperclass`, all values in the collection are instances of the class given by `uniformityClass`.
- If the value of `uniformity` is `@sameClass`, then all values in the collection are members of the class given by `uniformityClass`.

If a script attempts to add an object of the wrong class to a collection, the `add` method should report the `badKey` exception.

For most collections, `uniformity` is set to `@commonSuperclass`. Note that the value of `uniformity` is determined as part of the definition of a class or object, not by what is stored in the collection. For example, if you create an `Array` object and store only `Point` objects in it, the `uniformity` instance variable is not set to reflect this. A collection has no way of determining what objects you store in it, unless you define specialized behavior to do so.

The instance variable `uniformity` is descriptive only. Creating a new collection class that specializes the `uniformityGetter` method does not modify any actual behavior. A new class or object that restricts the values it can contain must redefine its own behavior.

uniformityClass

self`.uniformityClass` (read-only) *(class)*

Specifies the class referred to by `uniformity` (see the definition of `uniformity`). For most collection classes, `uniformityClass` is set to `RootObject`, meaning all values in the collection must be instances of `RootObject`. This means any object is allowed. (In other words, the collection has no uniformity.) Notable exceptions are the concrete `Range` classes: for `IntegerRange`, the value of `uniformityClass` is `Integer`, and for `NumberRange`, it is `Number`.

The value of `uniformityClass` is determined as part of the definition of a class or object, and is fixed thereafter. It does not change based on what objects are currently in the collection.

The instance variable `uniformityClass` is descriptive only. Creating a new collection class that specializes the `uniformityClassGetter` method does not modify any actual behavior. A new class or object that restricts the values it can contain must redefine its own behavior.

valueEqualComparator

self`.valueEqualComparator` (read-only) *(function)*

Specifies the function to use to test whether two values in the collection *self* are the same. See "Comparison Functions" in the Collections chapter of the *ScriptX Components Guide* for a list of the built-in functions you can use.

Although any global function, anonymous function, or method can be assigned to `valueEqualComparator`, there are differences in how different kinds of functions are dispatched. For information on functions and function dispatch, see the "Object System Kernel" chapter of the *ScriptX Components Guide*.

Instance Methods

The following instance methods are defined in `Collection`:

add

`add` *self key value* ⇨ *(object)*

self	`Collection` object
key	Any object
value	Any object

Adds the binding specified by *key* and *value* to the collection *self*, inserting the pair in its proper position. The return value is the implicit or explicit key. For implicitly keyed collections, `add` can cause subsequent keys to change. For sequences, it inserts before the specified key.

The `empty` object is always a valid key and has meaning as follows. For implicitly keyed collections, an appropriate key is chosen. For sequences, `add` with an `empty` key has append semantics. For explicitly keyed collections, adding with an empty key is the same as `add` *self value value*.

If a collection determines that either the *key* or *value* is inappropriate, it reports the corresponding exception, either `badKey` or `badValue`. If a bounded collection is already full, it reports the `bounded` exception. Bounded collections include `Single`, `Pair`, `Triple`, and `Quad`, as well as any array object that is initialized with the `growable` keyword set to `false`.

addMany

`addMany` *self another* ⇨ *(none)*

self	`Collection` object
another	`Collection` object

Adds all the items in *another* collection as if by calling `add` on the collection *self*. For unkeyed and implicitly keyed collections, `addMany` uses `empty` as the key. For explicitly keyed collections, `addMany` performs the add with all the key-value pairs. If a bounded collection would grow beyond its maximum size, it reports the `bounded` exception. The state of the collection itself after reporting the `bounded` exception is not specified.

addToContents

`addToContents` *self element* ⇨ `Collection`

self	`Collection` object
element	Any object

Do not call this method directly from the scripter. It is defined to allow the `object` expression in the ScriptX language to work properly. `addToContents` is called automatically for each *element* in the `contents` section of the `object` expression, and it returns the collection *self*. For more information about the `object` expression, see *ScriptX Language Guide*.

chooseAll

`chooseAll` *self func arg* ⇨ `Collection`

self	`Collection` object
func	An instance of a subclass of `AbstractFunction`
arg	Any object

Iterates over the collection *self*, calling the function *func* once for each item in the collection with the given argument *arg*.

func value arg

The `chooseAll` method iterates through each value in the collection *self*. Its return value is another collection containing the list of values for which the function returned `true`. The return collection's class is specified by the value of `mutableCopyClass`. If the collection inherits from `LinearCollection`, then the items are guaranteed to be processed in their natural order and the result is a linear collection in which items are ordered based on their order in the source collection. See the discussion on page 161 concerning generic functions that depend on the integrity of an iterator.

chooseOne

`chooseOne` *self func arg* ⇨ *(object)*

self	`Collection` object
func	An instance of a subclass of `AbstractFunction`
arg	Any object

Iterates over the collection *self*, calling the function *func* with a value from the collection and the given argument *arg*:

func value arg

The `chooseOne` method calls this function once for each item in the collection until the function returns `true`. The return value of `chooseOne` is the first item for which this function returns `true`, or `empty` if no call to this function returns `true`. If the collection has a natural order, that is, if it inherits from `LinearCollection`, then items are guaranteed to be processed in that order. See the discussion on page 161 concerning generic functions that depend on the integrity of an iterator. Note that `Array` specializes `chooseOne` to allow for insertion and deletion of items, although not for wholesale rearrangement of the collection. See the definitions of `chooseOne` and other iterative methods under the `Array` class.

chooseOneBinding

`chooseOneBinding` *self func arg* ⇨ (*object)*

self	`Collection` object
func	An instance of a subclass of `AbstractFunction`
arg	Any object

Iterates over the collection *self*, calling the function *func* with a key-value pair from the collection and the given argument *arg*:

func key value arg

The `chooseOneBinding` method calls this function once for each item in the collection until the function returns `true`. Its return value is the first matching item, or `empty` if no call to this function returns `true`. If the collection has a natural order, that is, if it inherits from `LinearCollection`, then items are guaranteed to be processed in that order. See the discussion on page 161 concerning generic functions that depend on the integrity of an iterator.

deleteAll

`deleteAll` *self value* ⇨ `Integer`

self	`Collection` object
value	Any object

Deletes all bindings that match the given *value* from the collection *self*. Values are compared using the value comparator function specified in `valueEqualComparator`, an instance variable defined by `Collection`. Returns the number of values deleted. This method does not remove objects from memory, unless normal garbage collection applies. It just removes them from the collection.

The `deleteAll` method reports the `bounded` exception (and leaves the collection in an undefined state) if the collection would shrink below the size specified by `minSize`.

deleteBindingAll

`deleteBindingAll` *self key value* ⇨ `Integer`

self	`Collection` object
key	Any object
value	Any object

Deletes from the collection *self* all bindings (key-value pairs) that match the given *key* and *value*. Returns the number of items deleted. This method does not remove either the *key* or *value* objects from memory, unless normal garbage collection applies. It just removes them from the collection. It reports the `bounded` exception (and leaves the collection in an undefined state) if the collection would shrink below the size specified by `minSize`.

deleteBindingOne

`deleteBindingOne` *self key value* ⇨ `Boolean`

self	`Collection` object
key	Any object
value	Any object

Deletes from the collection *self* at most one item whose binding (key and value) matches the given *key* and *value*. Returns `true` if an item is deleted; `false` if it is not deleted. If multiple items match the given *key* and *value*, any one of those items (not necessarily the first one) can be deleted. This method does not remove either the key or value objects from memory, unless normal garbage collection applies. It just removes them from the collection. It reports the `bounded` exception (and leaves the collection in an undefined state) if the collection would shrink below the size specified by `minSize`.

deleteKeyAll

`deleteKeyAll` *self key* ⇨ `Integer`

self	`Collection` object
key	Any object

Deletes from the collection *self* all items whose keys match the given *key*. Returns the number of items deleted. This method does not remove either the key or value objects from memory, unless normal garbage collection applies. It just removes them from the collection. It reports the `bounded` exception (and leaves the collection in an undefined state) if the collection would shrink below the size specified by `minSize`.

deleteKeyOne

`deleteKeyOne` *self key* ⇨ `Boolean`

self	`Collection` object
key	Any object

Deletes from the collection *self* at most one item whose key matches the given *key*. Returns `true` if an item is deleted; `false` if it is not deleted. If multiple items match the given *key*, any one of those items (not necessarily the first one) can be deleted. This method does not remove either the key or value objects from memory, unless normal garbage collection applies. It just removes them from the collection. It reports the `bounded` exception (and leaves the collection in an undefined state) if the collection would shrink below the size specified by `minSize`.

If multiple items have a key that matches the given key, any one of those items, not necessarily the first one, can be deleted. Only if a collection is also a linear collection is it guaranteed to be processed in its natural order.

deleteOne

`deleteOne` *self value* ⇨ `Boolean`

self	`Collection` object
value	Any object

Deletes from the collection *self* at most one item whose value matches the given *value*. The values are compared using the value comparator function specified by `valueEqualComparator`. The method `deleteOne` returns `true` if a value was deleted; otherwise, it returns `false`. This method does not remove either the key or value objects from memory, unless normal garbage collection applies. It just removes them from the collection. It reports the `bounded` exception (and leaves the collection in an undefined state) if the collection would shrink below the size specified by `minSize`.

If multiple items have a value that matches the given value, any one of those items, not necessarily the first one, can be deleted. Only if a collection is also a linear collection is it guaranteed to be processed in its natural order.

emptyOut

`emptyOut` *self* ⇨ *self*

Removes all the items in the collection, leaving the collection empty. This method does not remove objects from memory, unless normal garbage collection applies. It just removes them from the collection. The empty collection *self* is returned. This method can be called on any mutable collection. It is equivalent to calling `removeAll` *self* `true` `undefined` on the collection *self*.

forEach

`forEach` *self func arg* ⇨ *(none)*

self	`Collection` object
func	An instance of a subclass of `AbstractFunction`
arg	Any object

Iterates over the collection *self*, calling the function *func* with a value from the collection and the given argument *arg*.

func value arg

The `forEach` method calls this function once for each value in the collection. If the collection has a natural order, that is, if the collection also inherits from `LinearCollection`, then this method processes items in that order.

This method does not return a value, and any return value from *func* cannot be used, so it must work by side effect. The `forEach` method has no effect on the collection *self* unless *func* is written to do so.

This method is similar to `map`, except that `map` returns a collection. See the description on page 161 concerning generic functions that depend on the integrity of an iterator. Note that `Array` specializes `forEach` to allow for insertion and deletion of items, although not for wholesale rearrangement of the collection. See the definitions of `forEach` and other iterative methods under the `Array` class.

For a method that calls a function on both the key and the value, see `forEachBinding`, also defined by `Collection`. `LinearCollection` defines a related method, `forEachBackwards`, that can be used to iterate through elements of a linear collection in reverse.

forEachBinding

`forEachBinding` *self func arg* ⇨ *(none)*

self	`Collection` object
func	An instance of a subclass of `AbstractFunction`
arg	Any object

Iterates over the collection *self*, calling the function *func* with a binding (key-value pair) from the collection and the given argument *arg*.

func key value arg

The `forEachBinding` method calls this function once for each item in the collection. For implicitly keyed collections, it uses the implicit keys for *key*. If the collection has a natural order, that is, if the collection also inherits from `LinearCollection`, then this method processes items in that order. See the discussion on page 161 concerning generic functions that depend on the integrity of an iterator.

getAll

`getAll` *self key* ⇨ `Collection`

self	`Collection` object
key	Any object

Returns a subset of the collection *self*, where the subset's keys match the given *key*. The `getAll` method compares keys using the key comparator function specified by `keyEqualComparator`. The return collection's class is determined by the value of `mutableCopyClass`.

getAny

`getAny` *self* ⇨ (*object*)

Returns an arbitrary item from the collection *self*. The default implementation for `Collection` itself is simply to get the first element the iterator picks, which for some collections, such as `HashTable`, may well be the same item every time. For collections that also inherit from `LinearCollection`, `getAny` uses a hidden `RandomState` object to generate the item to pick. In such cases, calling `getAny` on the same collection multiple times returns an arbitrary item each time.

When called on a collection whose size is `posInf` (positive infinity), `getAny` reports the `badKey` exception.

getKeyAll

`getKeyAll` *self value* ⇨ `LinkedList`

self	`Collection` object
value	Any object

Returns a collection of all the keys in the collection *self* whose values match the given *value*. The `getKeyAll` method compares values using the value comparator function specified by `valueEqualComparator`.

getKeyOne

`getKeyOne` *self value* ⇨ (*object*)

self	`Collection` object
value	Any object

Returns one key in the collection *self* whose value matches the given *value*, or `empty` if no match is found. The `getKeyOne` method compares values using the value comparator function specified by `valueEqualComparator`. If multiple items match the given value, one of those keys is returned. If the collection has a natural order, that is, if the collection also inherits from `LinearCollection`, then this method processes items in that order.

getMany

`getMany` *self keyCollection* ⇨ `Collection`

self	`Collection` object
keyCollection	`Collection` object

Returns a collection containing every value from the collection *self* whose key matches any key in *keyCollection*, a collection of keys. Returns `empty` if no match is found. The returned collection's class is specified by the value of `mutableCopyClass`, an instance variable defined by `Collection`. If the collection has a natural order, that is, if the collection also inherits from `LinearCollection`, then this method processes items in that order.

getOne

`getOne` *self key* ⇨ (*object*)

self	`Collection` object
key	Any object

Returns one value from the collection *self* whose key matches the key specified by *key*, or `empty` if no match is found. The `getOne` method compares keys using the key comparator function specified by `keyEqualComparator`. If multiple items match the

given key, one of those items is returned. If the collection has a natural order, if the collection also inherits from `LinearCollection`, then this method processes items in that order.

hasBinding

`hasBinding` *self key value* ⇨ `Boolean`

self	`Collection` object
key	Any object
value	Any object

Returns `true` if the collection *self* has a binding (key-value pair) that matches the given *key* and *value*. This method can be used for both implicitly and explicitly keyed collections. It matches keys and values using the functions that are supplied as the values of `keyEqualComparator` and `valueEqualComparator`, instance variables defined by `Collection`.

hasKey

`hasKey` *self key* ⇨ `Boolean`

self	`Collection` object
key	Any object

Returns `true` if the collection *self* has a key that matches the given *key*. This method can be used for both implicitly and explicitly keyed collections. Keys are compared using the key comparator function specified by `keyEqualComparator`.

intersects

`intersects` *self collection* ⇨ `Boolean`

self	`Collection` object
collection	`Collection` object

Returns `true` if any element is common to both the collections *self* and *collection*. Returns `false` if there are no common elements.

isEmpty

`isEmpty` *self* ⇨ `Boolean`

Returns `true` if the collection *self* has no items; otherwise, it returns `false`.

isMember

`isMember` *self value* ⇨ `Boolean`

self	`Collection` object
value	Any object

Returns `true` if the collection *self* contains the value specified by *value*. The `isMember` method compares values using the value comparator function specified by `valueEqualComparator`.

iterate

`iterate` *self* ⇨ `Iterator`

Creates an iterator that can iterate over the collection *self*. This iterator is an instance of the class specified by `iteratorClass`, an instance variable defined by `Collection`. See the discussion on page 161 concerning the integrity of an iterator.

localEqual (RootObject)

`localEqual` *self other* ⇨ `Boolean`

self	`Collection` object
other	`Collection` object

Returns `true` if the collection *self* is of the same class, and has the same size and elements as the collection *other*. Note that `LinearCollection` defines another version of `localEqual`, which all collections that are linear collections inherit.

map

`map` *self func arg* ⇨ `Collection`

self	`Collection` object
func	An instance of a subclass of `AbstractFunction`
arg	Any object

Iterates over the collection *self*, calling the function *func* with a value from the collection and the given argument *arg:*

func value arg

The `map` method calls this function once for each value in the collection and returns a collection containing the results from each of the function invocations, in the order they were called. If the collection has a natural order, that is, if the collection also inherits from `LinearCollection`, then this method processes items in that order. The return collection's class is specified by the value of `mutableCopyClass`.

This method is very similar to `forEach`, except that `map` returns a collection whereas `forEach` does not.

See the discussion on page 161 concerning generic functions that depend on the integrity of an iterator.

merge

`merge` *self other* ⇨ `Collection`

self	`Collection` object
other	`Collection` object

Merges the collection *self* and the collection *other*, copying their values to a new collection, which is then returned. Unlike `addMany`, `merge` does not modify *self*. (With `addMany`, the contents of *other* are added to the original collection *self*.) The return collection's class is specified by the value of `mutableCopyClass`, an instance variable defined by `Collection`.

pipe

`pipe` *self other* ⇨ `Collection`

self	`Collection` object
other	`Collection` object

Means the same as `addMany` *self other*, but returns the collection *self*. The compiler calls `pipe` when it encounters the pipe operator (`|`).

prin (RootObject)

`prin` *self format stream* ⇨ *(none)*

self	`Collection` object
format	`NameClass` object
stream	`Stream` object

Prints the names of the items in the collection *self* with the format given in *format*. `Collection` specializes `prin` to allow for printing only a limited or truncated version of the collection. The value of the *format* argument can be `@unadorned`, `@normal`, `@complete`, or `@debug`. Using `@normal` truncates printing after the tenth item and then shows an ellipsis (. . .). If you want to print the complete contents of the collection, use one of the other standard printing formats (`@complete`, `@debug`, or `@unadorned`).

removeAll

`removeAll` *self func arg* ⇨ `Integer`

self	`Collection` object
func	An instance of a subclass of `AbstractFunction`
arg	Any object

Iterates over the collection *self*, calling the function *func* with a value from the collection and the given argument *arg*:

func item arg

It does this once for each item in the collection; each time the function returns `true`, the current item, the item that caused the function to return `true`, is deleted from the collection. `removeAll` returns the number of values deleted. This method does not remove objects from memory, unless normal garbage collection applies. It just removes them from the collection.

If the collection has a natural order, that is, if it inherits from `LinearCollection`, then items are guaranteed to be processed in that order. This method is only applicable to mutable collections. It reports the `bounded` exception if the collection would shrink below the size specified in `minSize`. See the discussion on page 161 concerning generic functions that depend on the integrity of an iterator.

removeOne

`removeOne` *self func arg* ⇨ `Boolean`

self	`Collection` object
func	An instance of a subclass of `AbstractFunction`
arg	Any object

Iterates over the collection *self*, calling the function *func* with a value from the collection and the given argument *arg*:

func item arg

It does this once for each value in the collection until the function returns `true`, then it stops. When execution stops, the current item, the item that caused the function to return `true`, is deleted from the collection. The `removeOne` method returns `true` if a value was deleted and `false` if one was not. This method does not remove objects from memory, unless normal garbage collection applies. It just removes them from the collection.

If the collection has a natural order, that is, if it inherits from `LinearCollection`, then items are guaranteed to be processed in that order. This method is only applicable to mutable collections. It reports the `bounded` exception if the collection would shrink below the size specified in `minSize`. See the discussion on page 161 concerning generic functions that depend on the integrity of an iterator.

setAll

`setAll` *self key value* ⇨ `Integer`

self	`Collection` object
key	Any object
value	Any object

For every item in the collection *self* whose key matches *key*, `setAll` sets that item's value to *value*. If the key is passed as `empty` it means the same as `setAll` *self value value*. The `setAll` method returns the number of items changed.

If the key or value is inappropriate, `setAll` reports the `badKey` or `badValue` exception. If there are no matches, the `Collection` object remains unchanged.

setOne

`setOne` *self key value* ⇨ *value*

self	`Collection` object
key	Any object
value	Any object

Sets the value of one item in the collection *self* whose key matches *key* to the value given by *value*. If the key is passed as `empty`, it means the same as `setOne` *self value value*.

If there is already a key-value pair that matches *key*, `setOne` replaces it. If not, it adds the pair. If the collection has a natural order, that is, if the collection also inherits from `LinearCollection`, then this method processes items in that order. The `setOne` method returns the value passed in.

If the key or value is inappropriate, `setOne` reports the `badKey` or `badValue` exception. In the case of sequences, the only nonexistent key that is appropriate is the first one beyond the end of the sequence. If a bounded collection is at the size specified by `maxSize`, then `setOne` reports the `bounded` exception.

size

`size` *self* ⇨ `Integer`

Returns the size of the collection *self*. Use the instance variable `size` (that is, the `sizeGetter` method) as the interface for getting the size of a collection. Define a `size` method to specialize the implementation used to determine the size of the collection.

Subclasses Must Implement

Subclasses of `Collection` must unconditionally implement the following two methods; if these methods are not implemented, an exception will be generated when they are called:

```
add
iteratorClassGetter
```

For greater efficiency, subclasses of `Collection` should implement the following methods:

```
equal
isMember
removeAll
removeOne
size
```

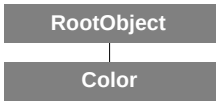
For more efficient keyed collections, subclasses of `Collection` should implement the following methods:

```
deleteBindingOne
deleteKeyOne
getOne
hasKey
hasBinding
setOne
```

Subclasses of `Collection` that modify default behavior should specialize any descriptive methods that are affected so that the new subclass interacts properly with other classes and scripts. The following getter methods in the Collection protocol are descriptive.

```
boundedGetter
keyUniformityGetter
keyUniformityClassGetter
maxSizeGetter
minSizeGetter
mutableGetter
proprietoredGetter
uniformityGetter
uniformityClassGetter
```

Color

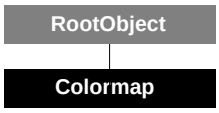


Class type: Core class (abstract)
Resides in: ScriptX and KMP executables
Inherits from: `RootObject`
Component: 2D Graphics

`Color` is an abstract superclass for classes that define how to interpret the color values of pixels within some bitmap, an area of memory, or a display surface. The subclasses of `Color` represent particular color spaces, providing a way to interpret pixel values and extract the encoded color information.

The `RGBColor` subclass of `Color` represents a device-independent representation of RGB and grayscale color spaces. The `ColorMap` class provides a way to create tables or palettes of colors by implementing an array that contains `Color` instances.

Colormap



Class type: Core class (concrete)
Resides in: ScriptX and KMP executables
Inherits from: `RootObject`
Component: 2D Graphics

The `Colormap` class provides an array-like object used to map pixel values in a bitmap to instances of a particular color space. A `Colormap` object also defines the pixel encoding and depth of the values in its sequence. Each value in a `ColorMap` is an `RGBColor` object.

Not every possible pixel value for a particular pixel depth will necessarily serve as an index. The actual color map may have fewer entries than the indices available at that depth. For example, you could have a color map containing just six colors, while the pixel depth is eight bits, potentially allowing for 256 entries.

ScriptX provides several default instances of `Colormap`, see Chapter 3, "Global Constants and Variables", for more information on displaying images on monitors with standard pixel depths.

```
theDefault1Colormap -- 2 colors
theDefault2Colormap -- 4 colors
theDefault4Colormap -- 16 colors
theDefault8Colormap -- 256 colors
```

Although a `ColorMap` object is not a collection, you can use the collection syntax *colormap*`[`*n*`]` to find the *n*th pixel value in a bitmap.

For example:

```
myColormap[45].red
```

returns the value of the `red` instance variable of the `RGBColor` that is the 45th value of the colormap `myColormap`.

Creating and Initializing a New Instance

The following script is an example of how to create a new instance of the `Colormap` class:

```
myMap := new Colormap \
           size:128 \
           bitsPerPixel:8 \
           colorSpace:RGBColor
```

The variable `myMap` contains the newly initialized color map instance. The number of entries it can contain is 128, the number of bits per pixel in the colors it represents is 8, and the color space is `RGBColor`.

init

`init` *self* `bitsPerPixel:`*integer* [`size:`*integer*] [`colorClass:`*rgbColor*]

⇨ *(none)*

self	`Colormap` object
`bitsPerPixel:`	`Integer` object representing the pixel depth
`size:`	`Integer` object representing the number of entries
`colorClass:`	`RGBColor` class

Initializes the `Colormap` object *self*, applying the arguments as follows: `size` sets the numbers of entries in the map, `bitsPerPixel` sets the pixel depth of the map, and `colorClass` represents the color space encoded by the map. Do not call `init` directly on an instance—it is automatically called by the `new` method.

If you omit an optional keyword, its default value is used. The defaults are:

`size:`1
`colorClass:RGBColor` class

When you create a new `Colormap` instance, you must also supply a value for the `bitsPerPixel` keyword; this value must be either 1, 2, 4, or 8. If you supply a value for `bitsPerPixel` but not a size, the value of `size` will be the maximum number of colors for the particular bit depth.

Table 4-4: Relationship between bits per pixel and maximum size

Bits Per Pixel	Maximum Size
1	2
2	4
4	16
8	256

By default, all color values in a new colormap are set to black.

Instance Variables

bitsPerPixel

self`.bitsPerPixel` (read-only) `Color`

Represents the bit depth of pixels that map into the color map *self*. This value reflects the size (in bits) of the largest potential index value into the color map. For example, a color map representing 8 bits per pixel could have as many as 2^8 or 256 entries.

colorSpace

self`.colorSpace` (read-only) `Color` subclass

Represents the class whose instances are used as the values in the color map *self*. By default, the value is `RGBColor`.

size

self`.size` (read-write) `Color`

Represents the actual number of entries in the array of the color map *self*.

Instance Methods

containsColor

`containsColor` *self testColor* — `Boolean`

self	`Colormap` object
testColor	`Color` object being tested for

Determines whether the `Colormap` instance *self* contains the `Color` instance *testColor*. Returns `true` if the color value is in the color map, `false` if not.

copy

`copy` *self* — `Colormap`

Creates and returns a new instance of `Colormap` containing the same values as *self*.

getNth

`getNth` *self n* — `Color`

self	`Colormap` object
n	`Integer` object, index of the color value

Returns the `Color` instance whose index value is *n* in the color map *self*.

localEqual (RootObject)

`localEqual` *self anotherMap* — `Boolean`

self	`Colormap` object
anotherMap	`Colormap` object

Compares the color maps *self* and *anotherMap* and returns `true` if they contain exactly the same mapping of pixel values to colors; that is, the same colors are in both maps and occupy the same position in each.

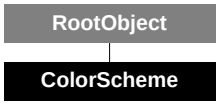
setNth

`setNth` *self n theColor* — `Color`

self	`Colormap` object
n	`Integer` index of the color value
theColor	`Color` object used as value for the index

Sets the color value for index value *n* in the color map *self*.

ColorScheme



Class type: Scripted class (concrete)
Resides in: widgets.sxl. Works with ScriptX and KMP executables
Inherits from: RootObject
Component: User Interface

The ColorScheme class provides a set of standard brushes that help determine the appearance of a widget, a user interface object from the ScriptX widget library. Each of these brushes, stored in the instance variables darkBrush1, darkBrush2, lightBrush1, and lightBrush2, is used to render some element of one of the widgets.

Creating and Initializing a New Instance

The widget library defines several global instances of ColorScheme, which it uses to determine the appearance of other objects it generates. You only need to define a new instance of ColorScheme if you plan to modify or specialize the widget classes to achieve a different appearance.

The following script creates a new instance of ColorScheme:

```
global myCustomBrushes := #(2, 1, 7, 4)
global myCustomColorScheme := new ColorScheme \
       brushIndexArray:myCustomBrushes
```

The global variable myCustomColorScheme contains the initialized ColorScheme object. The ColorScheme class stores a list of pre-set gray levels in its grayLevels class variable. When the object myCustomColorScheme is instantiated, the script applies the gray levels in positions 2, 1, 7, and 4, stored in the class variable grayLevels to set the values of darkBrush1, darkBrush2, lightBrush1, and lightBrush2, four instance variables defined by ColorScheme. For example, if the value of ColorScheme.grayLevels[2] is 102, then the instance variable darkBrush1 for myCustomColorScheme will end up containing a Brush object that is equivalent to the following:

```
new Brush color:(new RGBColor red:102 green:102 blue:102)
```

The new method uses keywords defined in init.

init

init *self* brushIndexArray:*array* ⇨ *(none)*

self	ColorScheme object
brushIndexArray:	Sequence of Integer objects

Initializes the ColorScheme object *self*, applying the keyword brushIndexArray to set the values of the instance variables darkBrush1, darkBrush2, lightBrush1, and lightBrush2. The keyword brushIndexArray is required.

Class Variables

disableBrush

self.disableBrush (read-write) Brush

Specifies a standard brush that is used to determine the disabled appearance of widgets that are based on the ColorScheme object *self*. By default, the value of disableBrush is a Brush object that is equivalent to the following:

```
object Brush
    color:blackColor, pattern:@grayPattern
    settings inkMode:@srcBic
end
```

grayBrushes

self.grayBrushes (read-write) Array

Specifies an Array object of fixed size that contains a set of standard brushes used by instances of the class *self*. The ColorScheme class generates these brushes automatically when the class is instantiated, using the value of each integer in the sequence *self*.grayLevels to determine which RGBColor object to set as the color of the corresponding brush in the sequence *self*.grayBrushes. For example, if the value of *self*.grayLevels[4] is 153, then the value of *self*.grayBrushes[4] is equivalent to the following:

```
new Brush color:(new RGBColor red:153 blue:153 green:153)
```

grayLevels

self.grayLevels (read-write) Array

Specifies a list of arbitrary length that contains Integer objects. These integers, which should be bounded in {0 . . . 255}, are used to set the red, green, and blue levels that are associated with the RGBColor object for a standard set of brushes that is stored in the class variable grayBrushes.

By default, the value of grayLevels is set to #(51,102,136,153,187,204,221). Modify the class definition for ColorScheme to change these default settings, or to add additional gray levels and create additional brush options.

Instance Variables

darkBrush1

self.darkBrush1 (read-write) Brush

Specifies a Brush object, one of four brushes that helps determine the appearance of widgets that use the color scheme *self*.

darkBrush2

self.darkBrush2 (read-write) Brush

Specifies a Brush object, one of four brushes that helps determine the appearance of widgets that use the color scheme *self*.

lightBrush1

self.lightBrush1 (read-write) Brush

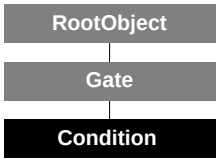
Specifies a Brush object, one of four brushes that helps determine the appearance of widgets that use the color scheme *self*.

lightBrush2

self.`lightBrush2` (read-write) `Brush`

Specifies a `Brush` object, one of four brushes that helps determine the appearance of widgets that use the color scheme *self*.

Condition



Class type: Core class (concrete, sealed)
Resides in: ScriptX and KMP executables
Inherits from: `Gate`
Component: Threads

`Condition` objects represent a gate whose `state` instance variable becomes instantaneously `@open` and then `@closed` again. A thread can wait on a `Condition` object by calling either `gateWait` or `acquire`, which are equivalent, except that `acquire` is a generic function.

When one or more threads are waiting on a `Condition` object and its state is made `@open` (either through the `gateOpen` global function or the `relinquish` method defined by `Gate`), all threads that were waiting on the condition are made active. However, any threads that subsequently try to acquire the condition must wait until it is again explicitly opened.

Creating and Initializing a New Instance

The following script is an example of how to create a new instance of the `Condition` class:

```
myCond := new Condition \
          label:@debug1
```

The variable `myCond` contains the initialized condition. The instance's keyword `label` is set to `@debug1`. The `new` method uses the keywords defined in `init`.

init

`init` *self* [`label:`*object*] ⇨ *(none)*

self	`Condition` object
`label:`	Any object

Initializes the `Condition` object *self*. The value supplied with `label` is applied to the instance variable of the same name. Do not call `init` directly on an instance—it is automatically called by the `new` method.

If you omit an optional keyword, its default value is used. The defaults are:

`label:undefined`

Instance Variables

Inherited from `Gate`:

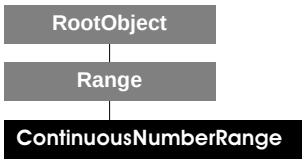
`label` `state`

Instance Methods

Inherited from `Gate`:

`acquire` `relinquish`

ContinuousNumberRange



Class type: Core class (concrete)
Resides in: ScriptX and KMP executables
Inherits from: Range
Component: Collections

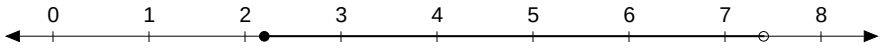
A ContinuousNumberRange object represents a continuous range of numbers. It is not a collection because it does not have a discrete, countable number of elements.

Creating and Initializing a New Instance

The following script is an example of how to create a new instance of the ContinuousNumberRange class:

```
myRange := new ContinuousNumberRange \
            lowerBound:2.2 \
            upperBound:7.4 \
            includesLower:true \
            includesUpper:false
```

The variable myRange contains an instance of ContinuousNumberRange. It incorporates all possible rational and irrational numbers between 2.2 and 7.4, including the lower bound but not the upper bound.



The new method uses the keywords defined in init.

init

init *self* [lowerBound:*number*] [upperBound:*number*]
[includesLower:*boolean*] [includesUpper:*boolean*] ⇨ *(none)*

self	ContinuousNumberRange object
lowerBound:	Number object
upperBound:	Number object
includesLower:	Boolean object
includesUpper:	Boolean object

Initializes the ContinuousNumberRange object *self*, applying the arguments as follows: lowerBound sets the lower numeric bound for the range, and upperBound sets the upper boundary value. These two points are not restricted to integers and can be any kind of real number—integer, fixed, floating, or other.

If includesLower is true, then the value supplied with lowerBound is included in the range; otherwise, all values above but not including lowerBound are in the range. If includesUpper is true, then the value supplied with upperBound is included in the range; otherwise, all values below but not including upperBound are in the range. Do not call init directly on an instance—it is automatically called by the new method.

If you omit an optional keyword, its default value is used. The defaults are:

```
lowerBound:negInf
upperBound:posInf
includesLower:true
includesUpper:true
```

Instance Variables

Inherited from Range:

includesLower	lowerBound	valueClass
includesUpper	size	
increment	upperBound	

ContinuousNumberRange defines these instance variables:

size (Range)

self.size (read-only) Number

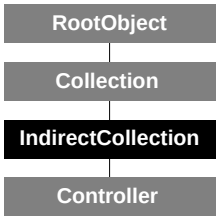
Returns nan, indicating that the number of elements in the continuous number range *self* is not countable. For a definition of nan, see the chapter "Global Constants and Variables."

Instance Methods

Inherited from Range:

withinRange

Controller



Class type: Core class (abstract)
Resides in: ScriptX and KMP executables
Inherits from: `IndirectCollection`
Component: Controllers

The `Controller` class is the root abstract class for all controller objects. A controller object directs the layout or behavior of objects contained in a space, interprets user input events, or performs other control functions. Built-in controller classes include `Interpolator`, `Bounce`, `Gravity`, `Movement` (described in the "Controllers" chapter) and `ActuatorController`, `RowColumnController`, and others (described in the "User Interface" chapter).

Each controller is a collection that holds the objects it controls. (It is a collection by virtue of inheriting from `IndirectCollection`.) These objects must also be in the space that the controller is attached to. Each controller has a `space` instance variable that specifies the particular space it is attached to. The controller controls some or all of the model objects in that space, depending on whether `wholeSpace` is set to `true` or `false`. The controller automatically controls all of the appropriate objects in that space if `wholeSpace` is `true`. The mechanism for this is that the space notifies the controller that an object has been added to the space. The controller uses `isAppropriateObject` to determine if it's an appropriate object for it to control, and if so, adds the object to its collection.

If `wholeSpace` is `false`, the space no longer notifies the controller when it has an object added to it. You must explicitly add the object to the controller if you want it to be controlled

Being collections, controllers provide the `Collection` protocol, but the `Controller` class does not define which `Collection` subclass is used. This allow each concrete subclass to store its target objects in the most appropriate way. It is the instance variable `targetCollection` that allows you to specify what kind of collection you want to use; however, the default value for `targetCollection` is `Array` because that is currently the optimal collection for controllers. Therefore, it is recommended that you not specify another subclass of `Collection` for `targetCollection` unless you have a special and particular need to do so.

Whenever an object is added to a controller, the object is checked to ensure that its protocols match those expected by the controller, in its `protocols` instance variable. If the object does not have the correct protocol, then the object is not added. If the controller's `wholeSpace` is `false`, an exception is thrown; if `true`, no exception is thrown.

Controllers also have the option of performing any function when an object is added, by overriding the `objectAdded` method. For example, a controller could enlarge the 2D space it controls, ensuring it is large enough to hold a certain `TwoDPresenter` object being added to it.

Creating and Initializing a New Instance

Because `Controller` is an abstract class, you cannot create an instance of `Controller`, nor is it useful to call the `new` method on `Controller` directly. However, you should call `init` from subclasses of `Controller` that override `init`, as described below, to properly initialize instances of the subclass.

init

`init` *self* [`space:`*space*] [`wholeSpace:`*boolean*] [`enabled:`*boolean*]
`targetCollection:`*sequence* ⇨ *(none)*

self	`Controller` object
`space:`	`Space` object containing objects to control
`wholeSpace:`	`Boolean` object indicating whether to control all objects in the space
`enabled:`	`Boolean` object to enable and disable the controller

Superclass `IndirectCollection` uses the following keyword:

`targetCollection:`	`Sequence` object

Initializes the `Controller` object *self*, applying the values supplied with the keywords to the instance variables of the same name. Do not call `init` directly on an instance—it is automatically called by the `new` method.

If you omit an optional keyword, its default value is used. The defaults are:

```
space:undefined
wholeSpace:false
enabled:true
```

Class Methods

Inherited from `Collection`:

`pipe`

Instance Variables

Inherited from `Collection`:

`bounded`	`maxSize`	`size`
`iteratorClass`	`minSize`	`uniformity`
`keyEqualComparator`	`mutable`	`uniformityClass`
`keyUniformity`	`mutableCopyClass`	`valueEqualComparator`
`keyUniformityClass`	`proprietored`	

Inherited from `IndirectCollection`:

`targetCollection`

The following instance variables are defined in `Controller`:

enabled

self`.enabled` (read-write) `Boolean`

Specifies whether the controller *self* is currently controlling its target objects or not. To turn it on or off, set `enabled` to `true` or `false`, respectively. The default value for `enabled` for a new controller is `true`.

When `enabled` is set to `false`, controller classes that are being tickled (`Bounce`, `Gravity`, `Interpolator`, and `Movement`) automatically stop being tickled, and when `enabled` is set to `true`, controller classes that are being tickled automatically resume

being tickled. Nothing happens automatically to controller classes where you are posting events. You have to stop posting events if you are disabled. To do this, specialize `enabledSetter` to stop posting events when `enabled` is set to `false` and resume posting events when `enabled` is set to `true`.

protocols

self`.protocols` (read-write) `Array`

Specifies the classes that any object added to the controller *self* must be a kind of. The `isAppropriateObject` method can use this list to tell whether it is appropriate to add an object to the controller's list. If the object has all the listed protocols, it is added.

space

self`.space` (read-write) `Space`

Specifies the space that the controller *self* controls. To determine which objects in that space are controlled, refer to the `wholeSpace` instance variable.

wholeSpace

self`.wholeSpace` (read-write) `Boolean`

If `true`, indicates the controller *self* should control all appropriate model objects in the entire space specified by the `space` instance variable. If `false`, the controller should control only objects added to the controller explicitly. Unless you specify otherwise, the default value for `wholeSpace` for a new controller is `false`.

Instance Methods

Inherited from `Collection`:

`add`	`emptyOut`	`isMember`
`addMany`	`forEach`	`iterate`
`addManyValues`	`forEachBinding`	`map`
`chooseAll`	`getAll`	`merge`
`chooseOne`	`getAny`	`pipe`
`chooseOneBinding`	`getKeyAll`	`prin`
`deleteAll`	`getKeyOne`	`removeAll`
`deleteBindingAll`	`getMany`	`removeOne`
`deleteBindingOne`	`getOne`	`setAll`
`deleteKeyAll`	`hasBinding`	`setOne`
`deleteKeyOne`	`hasKey`	
`deleteOne`	`isEmpty`	

Inherited from `IndirectCollection`:

`isAppropriateObject`	`objectAdded`	`objectRemoved`

Since a controller is an indirect collection, you can also use any methods defined in the class specified by `targetCollection`. The target collection is typically an instance of `Array`, which inherits from `Sequence`, so the following instance methods are redirected to a controller.

Accessible from `LinearCollection`:

`chooseOneBackwards`	`findRange`	`getNthKey`
`chooseOrdOne`	`forEachBackwards`	`getOrdOne`
`deleteFirst`	`getFirst`	`getRange`
`deleteLast`	`getLast`	`localEqual`
`deleteNth`	`getMiddle`	`localLT`
`deleteRange`	`getNth`	`pop`

Accessible from Sequence:

addFifth	moveBackward	setFourth
addFirst	moveForward	setLast
addFourth	moveToBack	setNth
addNth	moveToFront	setSecond
addSecond	prepend	setThird
addThird	prependNew	sort
append	setFifth	
appendNew	setFirst	

The following instance methods are defined in Controller:

isAppropriateObject (IndirectCollection)

`isAppropriateObject` *self addedObject* ⇨ `Boolean`

self	`Controller` object that object is being added to
addedObject	Any object being added to the controller

This method is automatically called when an object is added to the controller *self*. This method uses `isAKindOf` to tell if *addedObject* is an instance of the classes in the `protocols` list; if the object is a kind of all the protocol classes, the object is added to the controller's list, and this method returns `true`.

tickle

`tickle` *self clock* ⇨ *self*

self	`Controller` object
clock	`Clock` object of the space being controlled

Do not call `tickle` directly from the scripter. It is invoked automatically by the space to which the controller *self* is attached, through use of a callback on the given *clock*.

The `Controller` class itself does not actually define a method for `tickle`. Subclasses of `Controller` that perform some repeated action with each tick of the space's clock should implement a method for `tickle`. The `tickle` method is invoked automatically on each controller that implements it, with each tick of the space's clock. For more information, see "The Ticklish Protocol" in the "Controllers" chapter of *ScriptX Components Guide*.

Common Subclass Methods

`Controller` subclasses commonly implement the following methods:

```
enabledGetter
enabledSetter
isAppropriateObject
modelAdded
modelRemoved
objectAdded
objectRemoved
spaceGetter
spaceSetter
tickle
wholeSpaceGetter
wholeSpaceSetter
```

Accepting and Rejecting Events

If the `Controller` subclass can receive events, then it's important to implement the following so it can accept or reject events (accept events when `enabled` is set to `true` and reject events when `enabled` is set to `false`):

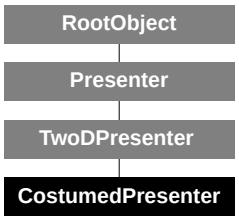
```
enabledGetter
enabledSetter
```

Control Space Dependencies

If the `Controller` subclass has anything dependent on a control space, such as an event interest, it's important to implement the following:

```
spaceGetter
spaceSetter
```

CostumedPresenter



Class type: Core class (concrete)
Resides in: ScriptX and KMP executables
Inherits from: `TwoDPresenter`
Component: Spaces and Presenters

The `CostumedPresenter` class represents objects that can be presented in different ways—objects that "wear" different costumes. A costume can be any other 2D presenter. Use a `CostumedPresenter` object when you want a presenter that can completely change its appearance, say from text to bitmap, while retaining properties such as its position and its connections to other objects such as controllers. A reference to the costume is stored in the `target` instance variable.

Think of a costumed presenter as a picture hanging on a wall—the costumed presenter is the blank place on the wall, and the costume is the picture that is actually presented.

Note that `CostumedPresenter` really implements a kind of delegation. In a sense, `CostumedPresenter` is to presenters what `IndirectCollection` is to collections. When a generic function is called on an instance of `CostumedPresenter`, the instance passes that function on to its costume. Thus, calling `draw` on the instance invokes `draw` on its target presenter. Therefore, the target presenter object is displayed only by way of the `CostumedPresenter` object.

See the class description of `Presenter` for information about the `target` instance variable, which specifies the object that a costumed presenter presents.

Creating and Initializing a New Instance

The following script is an example of how to create a new instance of the `CostumedPresenter` class:

```
cp := new CostumedPresenter \
       target:(new TwoDShape boundary:(new Rect x2:50 y2:50))
```

The variable `cp` contains a new instance of `CostumedPresenter` with a rectangle as its target. The `new` method uses the keyword arguments defined by the `init` method. (Note that `boundary` is omitted because it is ignored.)

init

`init` *self* [`target:`*object*] ⇨ *(none)*

self `CostumedPresenter` object

The superclass `TwoDPresenter` uses the following keyword:

`target:` `TwoDPresenter` object
`stationary:` `Boolean` object

Initializes the `CostumedPresenter` object *self*, setting its appearance by applying the value supplied with the `target` keyword to the `target` instance variable it inherits from `TwoDPresenter`. Do not call `init` directly on an instance—it is automatically called by the `new` method.

If you omit an optional keyword, its default value is used. The defaults are:

```
target:undefined
stationary:false
```

Instance Variables

Inherited from `Presenter`:

`presentedBy`	`subPresenters`	`target`

Inherited from `TwoDPresenter`:

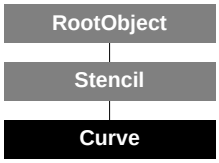
`bBox`	`height`	`transform`
`boundary`	`IsImplicitlyDirect`	`width`
`clock`	`isTransparent`	`window`
`compositor`	`isVisible`	`x`
`direct`	`needsTickle`	`y`
`eventInterests`	`position`	`z`
`globalBoundary`	`stationary`	
`globalTransform`	`target`	

Instance Methods

Inherited from `TwoDPresenter`:

`adjustClockMaster`	`inside`	`show`
`createInterestList`	`localToSurface`	`surfaceToLocal`
`draw`	`notifyChanged`	`tickle`
`getBoundaryInParent`	`recalcRegion`	
`hide`	`refresh`	

Curve



Class type: Core class (concrete)
Resides in: ScriptX and KMP executables
Inherits from: `Stencil`
Component: 2D Graphics

The `Curve` class represents Bezier cubic curves. (Postscript and Adobe Illustrator use cubic Bezier curves.) Users can adjust the control points to make a curve they like.

The geometric shape of a `Curve` object is defined by four points (x1, y1), (x2, y2), (x3, y3), and (x4, y4). The curve starts at (x1, y1) and ends at (x4, y4). The other two points are control points, which determine that curvature of the curve (they do not lie on the curve), as illustrated in Figure 4-4.

The curve starts at (x1, y1). At the starting point, it is tangential to the line from (x1, y1) to (x2, y2). The curve ends at (x4, y4), at which point it is tangential to the line from (x3, y3) to (x4, y4). The curve is always entirely enclosed by the convex quadrilateral defined by the four points.

For more information on the mathematics involved in Bezier curves, please see external mathematical books, such as *Fundamentals of Interactive Computer Graphics*, written by J.D. Foley and A. Van Dam, and published by Addison Wesley, 1982.

You can use the `getPoint` method to find the position of a point on the curve along the distance of the curve. For example, you could find the point half way along the curve, a tenth of the way along the curve, and so on.

You can use the `Curve` class to create editable, single curves. After creating a `Curve` object, you can change the endpoints and the control points. However, you cannot join curves together, or extend them to include more line segments. If you wish to create paths that include curves, or paths that include multiple contours, use the `Path` class. A `Path` object can include any number of line segments, including straight lines, cubic curves, arcs, splines, and so on. However, a `Path` object cannot be modified in the same way as a `Curve` object. Although you can always extend a path to include more line segments, arcs, curves, or splines, you cannot change the points in the path that have already been created. If you wish to modify the existing part of the path, you need to empty out the path and build it again from scratch.

Creating and Initializing a New Instance

The following script is an example of how to create a new instance of the `Curve` class:

```
curve1 := new Curve x1:0 y1:0 \
                    x2:100 y2:250 \
                    x3:200 y3:250 \
                    x4:300 y4:0
```

The variable `curve1` points to a new `Curve` object, that starts at the point (0, 0) and finishes at the point (300, 0). The control points are (100, 250) and (200, 250). Figure 4-4 shows the shape of the curve. The dots indicate the control points.

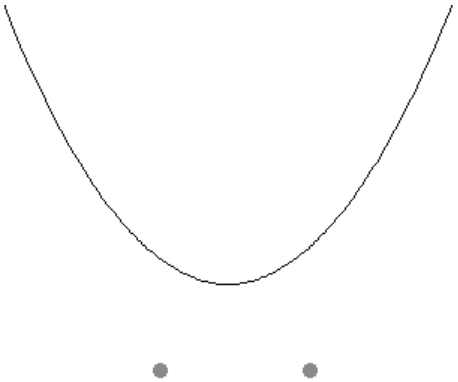


Figure 4-4: Curve1 – The dots indicate the control points

Since a `Curve` object is a `Stencil`, it must be put in a presenter before it can be viewed. You must specify a value for the `stroke` instance variable for the presenter to make the curve visible. (By default, the stroke value is always `undefined`.) The following code shows how to display `curve1` in a window:

```
w := new Window
show w
s := new TwoDShape boundary:curve1 stroke:blackbrush
append w s
```

init

`init` *self* `x1:`*x1* `y1:`*y1* `x2:`*x2* `y2:`*y2* `x3:`*x3* `y3:`*y3* `x4:`*x4* `y4:`*y4*

x1, *y1*	The x and y values of the starting point of the curve
x2, *y2*	The x and y values of the first control point
x3, *y3*	The x and y values of the second control point
x4, *y4*	The x and y values of the end point of the curve

Initializes the `Curve` object.

If you omit any of the optional keywords, ScriptX uses 0 as the default value.

Instance Variables

Inherited from `Stencil`:

`bbox`

The following instance variables are defined in `Curve`:

height

self`.height` (read-write) `Float`

Returns the height of the `Curve` object *self*. When the value of `height` is set, the curve is rescaled automatically and its width is set accordingly, using the value of `realBbox` to determine the object's new dimensions. The object's position, that is, its top left corner, is unchanged in the process.

length

self`.length` (read-only) `Number`

Returns the length of the curve *self*.

realBbox

self.realBbox (read-only) Rect

Returns a Rect object that is the actual bounding rectangle of the curve *self*, including its line width. Compare realBbox with bbox, an instance variable that is inherited from Stencil. The value of bbox reflects the bounding rectangle of the curve's control points, including the line width. Hence, the area of the rectangle returned by bbox is greater than or equal to that of the rectangle returned by realBbox. When the height or width is rescaled, realBbox rather than bbox is used to determine the curve's new dimensions.

width

self.width (read-write) Float

Returns the width of the Curve object *self*. When the value of width is set, the curve is rescaled automatically and its width is set accordingly, using the value of realBbox to determine the object's new dimensions. The object's position, that is, its top left corner, is unchanged in the process.

x1, y1, x2, y2, x3, y3, x4, y4

self.x1 etc. (read-write) Number

x1 and y1 specify the x and y values of the starting point of the curve; x2 and y2 specify the first control point; x3 and y3 specify the second control point; and x4 and y4 specify the end point of the curve.

Instance Methods

Inherited from Stencil:

inside	subtract
intersect	transform
onBoundary	union

The following methods are defined in Curve:

copy

copy *self* ⇨ Curve

Creates and returns a copy of the Curve object *self*.

getAngle

getAngle *self d* ⇨ Number

self	Curve object
d	Number between 0.0 and 1.0 that indicates how far along the curve to get the angle.

Returns the tangent angle (in radians) on the curve at distance *d* along the curve.

getCurvature

getCurvature *self d* ⇨ Number

self	Curve object
d	Number between 0.0 and 1.0 that indicates how far along the curve to get the angle.

Returns the curvature of the curve at distance *d* along the curve. The curvature is positive if the tangent angle increases, that is, the curve at the point is heading in a clockwise direction. The curvature is negative if the curve is heading in a counter-clockwise direction. The curvature is zero if the curve is straight. The inverse of the curvature gives the radius of the tangential circle.

getPoint

`getPoint` *self d* ⇨ `Point`

self	`Curve` object
d	`Number` between 0.0 and 1.0 that indicates how far along the curve to get the point.

Returns the point that lies on the curve at distance *d* along the curve. If *d* is 0, returns the starting point of the curve. If *d* is 1, returns the finishing point. If *d* is 0.25, returns the point that is a quarter of a way along the distance of the curve. If *d* is 0.5, returns the point that is half way along the distance of the curve, and so on.

The following code shows how to create a group space containing a previously-created `Curve` object and also a small round shape. When the group space is displayed in a window, the small round shape will appear at the point half way along the distance of the curve.

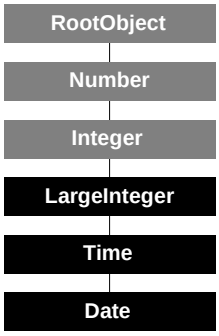
```
g1 := new GroupSpace
s1 := new TwoDShape boundary:curve1 stroke:(new brush color:bluecolor)
s2 := new TwoDShape boundary:(new oval x2:5 y2:5) fill:blackbrush
p1 := getPoint curve1 0.5
s2.x := p1.x
s2.y := p1.y
append g1 s1
append g1 s2
```

moveToZero

`moveToZero` *self* ⇨ `Curve`

Translates the `Curve` object *self* so that the value of `x1` and `y1` are 0. (That is, the starting point of the curve is at (0, 0.)

Date



Class type: Core class (concrete, sealed)
Resides in: ScriptX and KMP executables
Inherits from: `Time`
Component: Numerics

The `Date` class represents a date including the second, minute, hour, day of the week, day of the month, month, and year. When you create an instance of `Date` directly, its value is fixed over time—it does not change with the passing of time. The global constant `theCalendarClock` is an instance of `CalendarClock` that keeps track of the current date and time in its `date` instance variable. `Date` objects are displayed in the form:

Format: *dayOfWeek month day hours*`:`*minutes*`:`*seconds year*

Example: `Thu Nov 3 8:52:5 1994`

The time within a date is expressed in a 24-hour clock—when it reaches 23:59:59, it wraps around to 0:0:0. Internally, instances of `Date` are stored as `LargeInteger` objects with the units of seconds. All arithmetic operations on a date are performed on this integer number of seconds.

To convert a date to its integer number of seconds, coerce the date to a large integer:

```
myDate as LargeInteger -- displays as an integer
```

Both the Macintosh and Microsoft Windows have the concept of a base date. Dates are stored as an offset in seconds from the base date. By default, a new instance of `Date` is set to the current date. Unfortunately, the base date isn't the same on each platform.

The `Date` class inherits the `seconds`, `minutes`, `hours` and `scale` instance variables from the `Time` class. Note, however, that for the `Date` class, `scale` is read-only and its value is always 1.

The `Date` class has the following properties:

- A `Date` object can be coerced to a `String`, such as "`d1 as String`". A `Date` object can also be coerced to any other subclass of `Number`, including `Time`.
- Subtracting two `Date` objects results in a `Time` object representing the time difference between the two dates.
- Adding or subtracting a `Time` object from a `Date` object results in a date. The result takes into account the scale of the `Time` object—that is, the `Time` object in ticks is first divided by its scale to convert it to seconds before being added to the date.
- Adding or subtracting any subclass of `Number` from a `Date` object will add or subtract the corresponding number of seconds from the `Date` class.

Note that the finest resolution for date is seconds, and the finest resolution for time is ticks. Date is displayed as 3 values, while time is displayed as 4 values, as shown in the following table:

	Date	Time
Format	*hours*:*minutes*:*seconds*	*hours*:*minutes*:*seconds*:*ticks*
Example	`8:52:5`	`8:52:5:15`

Creating and Initializing a New Instance

The following script is an example of how to create a new instance of the `Date` class:

```
myDate:= new Date \
       year:1994 \
       month:@november \
       day:3 \
       hours:8 \
       minutes:52 \
       seconds:5
```

The variable `myDate` contains the initialized date with the value `Thu Nov 3 08:52:05 1994`. The `new` method uses the keywords defined in `init`.

init

`init` *self* [`year:`*integer*] [`month:` *integer*] [`day:`*integer*]
[`hours:`*integer*] [`minutes:`*integer*] [`seconds:`*integer*] ⇨ *(none)*

self	`Date` object
`year:`	`Integer` object
`month:`	`Integer` object or `NameClass` object
`day:`	`Integer` object

The superclass `Time` uses the following keywords:

`hours:`	`Integer` object
`minutes:`	`Integer` object
`seconds:`	`Integer` object

Initializes the `Date` object *self*, applying each of the arguments to the instance variable of the same name. Do not call `init` directly on an instance—it is automatically called by the `new` method.

If you omit an optional keyword, its current value is used. If you partially specify a date, the remaining keywords are automatically filled in to reflect their current value.

This example demonstrates default values for keywords on the `Date` class. Suppose that it is April 28, 1995, and the time is 36 seconds after 3:31 pm. If you create a `Date` object without specifying any keyword arguments, the Kaleida Media Player returns a `Date` object with the current date as the value of that object.

```
global currentDate := new Date -- no keywords
⇨  Fri Apr 28 15:31:36 1995 as Date
```

Ten seconds later, you create another instance of `Date`, specifying a different month and year. The Kaleida Media Player returns another `Date` object, whose value reflects the year and month you supply. The values for `day`, `hour`, `minute`, and `second` are still determined by default.

```
global anotherDate := new Date year:1992 month:9 -- two keywords
⇨  Mon Sep 28 15:31:46 1992 as Date
```

Instance Variables

Inherited from `Time`:

`hours`	`scale`	`ticks`
`minutes`	`seconds`	

The following instance variables are defined in `Date`:

day

self`.day` (read-write) `Integer`

Specifies the day of the month as an integer: first day=1; last day=one of 28, 29, 30, or 31.

dayOfWeek

self`.dayOfWeek` (read-write) `Integer`

Specifies the day of the week as an integer: Sunday = 1; Saturday = 7. The day of the week can also be set with a `NameClass` object representing the day: `@sunday`, `@monday`, `@tuesday`, `@wednesday`, `@thursday`, `@friday`, `@saturday`.

month

self`.month` (read-write) `Integer`

Specifies the month of the year as an integer: January = 1; December = 12. The month can also be set with a `NameClass` object representing the month: `@january`, `@february`, `@march`, `@april`, `@may`, `@june`, `@july`, `@august`, `@september`, `@october`, `@november`, `@december`.

scale (Time)

self`.scale` (read-only) `Integer`

Set to `1` for `Date` class and cannot be changed.

year

self`.year` (read-write) `Integer`

Specifies the year in the common era, such as 1994. The valid range of years is from 1970 to 2039.

Instance Methods

Note – Some of these operations are not meaningful for the `Date` class.

Inherited from `Number`:

`abs`	`floor`	`radToDeg`
`acos`	`frac`	`random`
`asin`	`inverse`	`rem`
`atan`	`ln`	`round`
`atan2`	`log`	`sin`
`ceiling`	`max`	`sinh`

`coerce`	`min`	`sqrt`
`cos`	`mod`	`tan`
`cosh`	`morph`	`tanh`
`degTorad`	`negate`	`trunc`
`exp`	`power`	

Inherited from `Integer`:

`length`	`logicalOp`	`lshift`
`logicalAnd`	`logicalOr`	`rshift`
`logicalNot`	`logicalXor`	

Inherited from `Time`:

`addHours`	`addMinutes`	`addSeconds`

The following instance methods are defined in `Date`:

addMonths

`addMonths` *self months* ⇨ *self*

self	`Date` object
months	`Integer` object

Adds the number of months specified by *months* to the date *self*. Note that the result will always be a valid date. For example, adding 1 month to Jan. 30, 1994 00:00:00 will result in the date Feb. 28, 1994 00:00:00 (rounding Feb. 30 to the valid date Feb. 28).

addWeeks

`addWeeks` *self weeks* ⇨ *self*

self	`Date` object
weeks	`Integer` object

Adds the corresponding number of weeks specified by *weeks* to the date *self*. Note that the result will always be a valid date, that is, adding 1 week to Jan. 30, 1994 00:00:00 will result in the date Feb. 6, 1994 00:00:00.

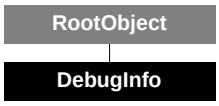
addYears

`addYears` *self years* ⇨ *self*

self	`Date` object
years	`Integer` object

Adds the corresponding number of years specified by *years* to the date *self*. Note that the result will always be a valid date, that is, adding 1 year to Feb. 29, 1992 00:00:00 (a leap year) will result in the date Feb. 28, 1993 00:00:00.

DebugInfo



Class type: Core class (concrete, sealed)
Resides in: ScriptX and KMP executables
Inherits from: `RootObject`
Component: Object System Kernel

The class `DebugInfo` provides a standard template for storing the debugging information that is associated with a `ByteCodeMethod` object. All scripted functions and methods are implemented as bytecode methods.

For more information on ScriptX function dispatch, see the `ByteCodeMethod` class in this volume and the "Object System Kernel" chapter of the *ScriptX Components Guide*.

Creating and Initializing a New Instance

You do not define an instance of `DebugInfo` directly. When the debugger is active, an instance of `DebugInfo` is created automatically for each instance of `ByteCodeMethod`.

Instance Variables

attributes

self.`attributes` (read-write) `KeyedLinkedList`

Stores a `KeyedLinkedList` object, currently empty, for use by the tool which generates the `ByteCodeMethod` object *self*.

positionInfo

self.`positionInfo` (read-write) `SortedKeyedArray`

Used by the debugger to map bytecode offsets to source code for the `ByteCodeMethod` object to which the `DebugInfo` object *self* is attached. See also `symbolInfo`.

source

self.`source` (read-write) `String`

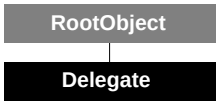
Contains the string that was compiled to create the `ByteCodeMethod` object *self*. The value of `source` can be set to `undefined`.

symbolInfo

self.`symbolInfo` (read-write) `SortedKeyedArray`

Used by the debugger to map bytecode offsets to source code for the `ByteCodeMethod` object to which the `DebugInfo` object *self* is attached. See also `positionInfo`.

Delegate



Class type: Core class (concrete)
Resides in: ScriptX and KMP executables
Inherits from: `RootObject`
Component: Object System Kernel

`Delegate` is a class that can be used to represent other objects. All generic functions called on a `Delegate` object are redirected to the object specified by its `delegate` instance variable.

Creating and Initializing a New Instance

The following script is an example of how to create a new instance of `Delegate`:

```
candidate:= new Delegate \
          delegate:myName
```

The variable `candidate` contains the initialized `Delegate` object, which is set to "delegate" all generic function calls it receives to the object `myName`, which is defined elsewhere.

init

`init` *self* [`delegate:`*object*] ⇨ *(none)*

self	`Delegate` object
`delegate:`	any object

The keyword argument `delegate` is applied to the instance variable of the same name.

Do not call `init` directly on an instance—it is automatically called by the `new` method.

If you omit an optional keyword, its default value is used. The defaults are:

```
delegate:undefined
```

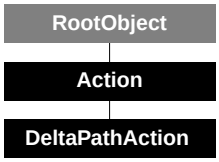
Instance Variables

delegate

self`.delegate` (read-write) *(object)*

Specifies the object to which all method calls on the delegate *self* are redirected.

DeltaPathAction



Class type: Core class (concrete)
Resides in: ScriptX and KMP executables
Inherits from: `Action`
Component: Animation

The `DeltaPathAction` class represents an action that is a relative change in position for a presenter. That is, the presenter will be moved by the given amount (instead of to an absolute position as with a `PathAction` object).

Creating and Initializing a New Instance

The following script is an example of how to create a new instance of the `DeltaPathAction` class:

```
myAction := new DeltaPathAction \
          deltaPosition:(new Point x:50 y:50) \
          targetNum:2 \
          time:100
```

The variable `myAction` holds an initialized instance of `DeltaPathAction`. This instance specifies that the player's second target (`targetNum:2`) start moving smoothly from its position at a time of 10 ticks, to arrive at the point 50,50 at 100 ticks.

init

`init` *self* [`deltaPosition:`*point*] [`targetNum:`*integer*] [`time:`*integer*] ⇨ *(none)*

self	`DeltaPathAction` object
`deltaPosition:`	`Point` object representing the distance the target is to move.

Superclass `Action` uses the following keywords:

`targetNum:`	`Integer` indicating which object in the target list of the player for this `DeltaPathAction` to apply the action to
`time:`	`Integer` object representing the absolute time to trigger the action, in ticks, from the beginning of the action list player

Initializes the `DeltaPathAction` object *self*, applying the values supplied with the keywords to the instance variables of the same name. Do not call `init` directly on an instance—it is automatically called by the `new` method.

If you omit an optional keyword, its default is used:

```
targetNum:0
time:0
deltaPosition:undefined
```

Instance Variables

Inherited from `Action`:

`authorData` `targetNum` `time` `playOnly`

The following instance variables are defined in `DeltaPathAction`:

deltaPosition

self`.deltaPosition` (read-write) `Point`

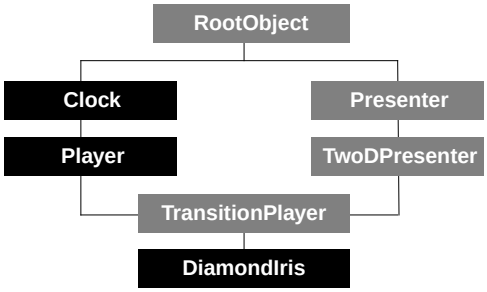
Specifies the amount of change in position in the x and y directions for the presenter when the delta path action *self* is triggered. These values are in the space's unit of measurement and relative to the space's origin.

Instance Methods

Inherited from `Action`:

`trigger`

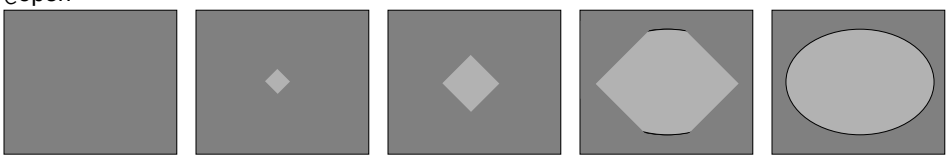
DiamondIris



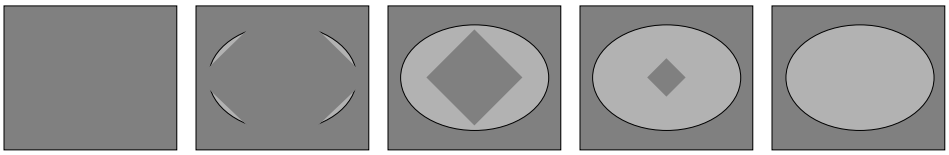
Class type: Loadable class (concrete)
Resides in: `ltrans.lib`. Works with ScriptX and KMP executables.
Inherits from: `TransitionPlayer`
Component: Transitions

The `DiamondIris` transition player provides a visual effect that causes the target to gradually appear from the center (`@open`) or from the edges (`@close`) as shown below. The target appears when the transition is played forward, and disappears when played backward (transition's rate set to -1).

`@open`



`@close`



Directions: `@open`, `@close`

Rate: Can play forward or backward.

For side-by-side illustrations of all transitions, see the Transitions chapter in the *ScriptX Components Guide*.

Creating and Initializing a New Instance

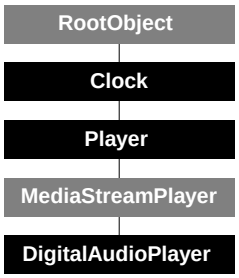
The following is an example of how to create a new instance of the `DiamondIris` class:

```
myTransition := new DiamondIris \
                duration:60 \
                direction:@open \
                target:myShape
```

The variable `myTransition` contains the initialized transition. The transition reveals the image `myShape` from the center first and has a duration of 60 ticks. You determine which space the transition will take effect in by adding this instance into that space. Then, when you play the transition, `myShape` is transitioned into that space. The `new` method uses the keywords defined in `init`.

NOTE – For the instance variables and methods, see the `BarnDoor` class.

DigitalAudioPlayer



Class type: Core class (concrete)
Resides in: ScriptX and KMP executables
Inherits from: `MediaStreamPlayer`
Component: Media Player

The `DigitalAudioPlayer` class provides functionality for playing digital audio. A `DigitalAudioPlayer` instance can be used to play, stop, pause and so on, a digital sound that is held in the player's `mediaStream` instance variable.

The `DigitalAudioPlayer` class provides services for presenting digital sound samples from an `AudioStream` object, using the methods defined on the `Player` class.

The `DigitalAudioPlayer` class has local instance variables that hold information relevant to playing sound, such as `pan`, `pitch` and `volume`.

Creating and Initializing a New Instance

ScriptX provides two ways to create an instance of the `DigitalAudioPlayer` class. One way is to call the `new` method on the class, in which case you must separately import the sound stream to be played by the new player. The other way is to import a sound directly into a `DigitalAudioPlayer` instance.

The following script shows how to create an instance of `DigitalAudioPlayer` by importing an AIFF file containing digitized sound. The object `dittyPlayer` can be used to play the sound imported from the file `ditty` on the ScriptX startup directory.

```
dittyStream := getstream theStartDir "ditty" @readable
dittyPlayer := importMedia theImportExportEngine dittystream \
               @sound @AIFF @player
```

This script shows an example of how to import an AIFF file as a `DigitalAudioPlayer`. You can also import SND and WAVE files in the same manner, in which case you would need to change the `@AIFF` argument to `@SND` or `@WAVE` as appropriate. For more details of the arguments to the method `importMedia` on the global instance `theImportExportEngine`, please see either the "Media Stream Players" chapter in the *ScriptX Components Guide* or the chapter about Importers in the *ScriptX Developer's Guide*.

The following script is an example of how to create a new instance of the `DigitalAudioPlayer` class by calling the `new` method. For this example, you would need previously to have created the player `topPlayer` and have imported the stream

`myStream`:

```
myPlayer := new DigitalAudioPlayer \
            mediaStream:myStream \
            bufferSize:50 \
            masterClock:topPlayer
```

The variable myPlayer points to the newly created audio player, which has the AudioStream instance myAudioStream as its media stream and the Player instance topPlayer as its master clock. It allocates space for 50 percent of the data in the media stream. The object myPlayer can now be used to play the sound in the stream myStream.

The new method uses the keywords defined in init.

init

init *self* [mediaStream:*stream*] [bufferSize:*integer*]
[masterClock: *topPlayer*] [title:*titleContainer*] ⇨ *(none)*

self	DigitalAudioPlayer object
mediaStream:	AudioStream object containing the source data
bufferSize:	Integer object representing the percentage of data for which to allocate space

The Clock superclass uses the following keywords:

masterClock:	Clock object to be used as the master player
scale:	(Ignored by DigitalAudioPlayer)
title:	TitleContainer object to which to add the player

Initializes the DigitalAudioPlayer object *self*, applying the arguments as follows: mediaStream sets the data source for the player, and bufferSize is used to determine what percentage of data in mediaStream to allocate buffer space for. The player is added to the specified titleContainer. Do not call init directly on an instance—it is automatically called by the new method.

If you omit an optional keyword, its default value is used. The defaults are:

```
mediaStream:undefined
bufferSize:50
masterClock:undefined
scale:(Determined by the sample rate of the mediaStream argument)
title: theScratchTitle
```

Instance Variables

Inherited from Clock:

callbacks	rate	ticks
effectiveRate	resolution	time
masterClock	scale	title
offset	slaveClocks	

Inherited from Player:

audioMuted	globalContrast	globalVolumeOffset
dataRate	globalHue	markerList
duration	globalPanOffset	status
globalBrightness	globalSaturation	videoBlanked

Inherited from MediaStreamPlayer:

frameRate	mediaStream

The following instance variables are defined in DigitalAudioPlayer:

pan

self.pan (read-write) Fixed

Specifies the placement of the sound played by the digital audio player *self* in the stereo spectrum. The value should be in the range 0 to 1.0, where 0 is far left and 1.0 is far right. Values greater than 1 will be adjusted to 1.0, values less than 0 will be adjusted to 0.

pitch

self.pitch (read-write) Integer

Specifies the current pitch setting for the sound played by the digital audio player *self*. The value must be zero or a positive number, where 0 is the lowest pitch. The highest acceptable value depends on the sound being played.

If you specify an unacceptable number, an exception is reported.

volume

self.volume (read-write) Fixed

Specifies the local volume value for the sound controlled by the digital audio player *self*. The value is specified in dB (decibels), where a value of 0 dB represents unity gain. Digital audio players are used to play sounds that were imported into ScriptX, and in this case unity gain is the volume of the sound when it was imported.

Values less than 0 make the sound quieter, while values greater than 0 increase the loudness of the sound. The decibel (dB) is a relative measurement; every 6dB drop is equal to halving the volume, modeling human auditory perception. That is, if the player's volume is -6, it plays half as loud as when the volume is 0; if the volume is 12 it plays half as loud as when the volume is -6 and so on.

The actual volume of the sound when it plays is determined by combining the local volume value with the global volume offset, which is passed down from the master player if any.

Not all sound managers can support volume values greater than 0. If you supply a value that is too high, a warning message is printed. If the volume is supplied as a negative value that is too low for you to be able to hear the sound, the sound will still play but it will not be audible.

Instance Methods

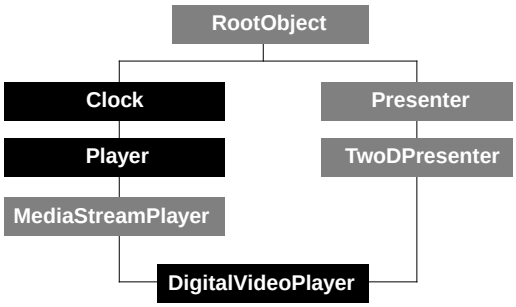
Inherited from Clock:

addPeriodicCallback	clockAdded	pause
addRateCallback	clockRemoved	resume
addScaleCallback	effectiveRateChanged	timeJumped
addTimeCallback	forEachSlave	waitTime
addTimeJumpCallback	isAppropriateClock	waitUntil

Inherited from Player:

addMarker	goToBegin	playPrepare
eject	goToEnd	playUnprepare
fastForward	goToMarkerFinish	playUntil
getMarker	goToMarkerStart	resume
getNextMarker	pause	rewind
getPreviousMarker	play	stop

DigitalVideoPlayer



Class type: Core class (concrete)
Resides in: ScriptX and KMP executables
Inherits from: `MediaStreamPlayer` and `TwoDPresenter`
Component: Media Players

The `DigitalVideoPlayer` class provides video playing capabilities. Because `DigitalVideoPlayer` is a kind of `Presenter`, the video can be displayed on any surface (including, potentially, an offscreen bitmap). There may, however, be limitations on this when hardware accelerators are used that can only send images directly to the screen.

A `DigitalVideoPlayer` instance has an associated video stream, stored in its `MediaStream` instance. When a digital video player plays, it plays its associated video stream.

If the `direct` instance variable of the `DigitalVideoPlayer` is set to `true`, the frames in the video stream are decompressed directly to the screen. If the `direct` instance variable is set to `false`, the video stream maintains an offscreen instance of `BitmapSurface`. The individual video frames are decompressed into the bitmap surface and the contents of the surface are transferred to the screen.

Creating and Initializing a New Instance

Users would not normally create instances of `DigitalVideoPlayer` themselves. If you have a file containing digitized video that you want to import and play, you should import the file into ScriptX as a `MoviePlayer` or `InterleavedStreamPlayer` object. Both of these can then be used to play a movie.

When a file containing a video or movie (the difference being that a movie usually contains video and sound) is imported into ScriptX as either a `MoviePlayer` or `InterleavedStreamPlayer` instance, additional player instances are created. A `DigitalVideoPlayer` is created for each video track, and a `DigitalAudioPlayer` is created for each audio track. These players are created as subplayers of the `MoviePlayer` or `InterleavedStreamPlayer`, which can be used to play the movie. See the discussion of `MoviePlayer` or `InterleavedStreamPlayer` for information on how to create a player to play a movie.

The `DigitalVideoPlayer` class does support the `new` method, one of whose arguments is `mediaStream`, which needs to be a video stream. The only way to actually get a video stream into ScriptX however, is to import a movie, which creates a `DigitalVideoPlayer` anyway. Although there is probably no reason why you would ever want to call the `new` method on the `DigitalVideoPlayer` class directly, an example is shown here for the sake of completeness.

The following script is an example of how to create a new instance of the `DigitalVideoPlayer` class by calling the `new` method:

```
myPlayer := new DigitalVideoPlayer \
          mediaStream:videoStream1 \
          boundary:(new Rect x2:200 y2:200) \
          masterClock:topPlayer
```

The variable `myPlayer` points to the newly created video player, which has the `VideoStream` instance `videoStream1` as its media stream, the `Player` instance `topPlayer` as its master clock, and a rectangular boundary that is to be used as the screen for the video.

The `new` method uses the keywords defined in `init`.

init

`init` *self* [`mediaStream:`*videoStream*] [`boundary:`*stencil*]
[`title:`*titleContainer*] [`masterClock:`*player*] ⇨ *(none)*

self	`DigitalVideoPlayer` object
`mediaStream:`	`VideoStream` object containing the source video data to be played by this `VideoPlayer`

The superclass `TwoDPresenter` uses the following keywords:

`boundary:`	`Stencil` object to use as the boundary for image displayed by the digital video player
`stationary:`	`Boolean` object

The superclass `Clock` uses the following keywords:

`title:`	`TitleContainer` object to add the player to
`masterClock:`	`Player` object

Initializes the `DigitalVideoPlayer` object *self*, applying the arguments as follows: `mediaStream` sets the source of video data, `masterClock` sets the player's master player, and `boundary` sets the size and shape of the player's window. The player is added to the specified `titleContainer`.

If you omit an optional keyword, its default value is used. The defaults are:

```
mediaStream:undefined
boundary:new Rect
stationary:false
title:theScratchTitle
masterClock:undefined (which means the player has no master player)
```

Instance Variables

Inherited from `Clock`:

`callbacks`	`rate`	`ticks`
`effectiveRate`	`resolution`	`time`
`masterClock`	`scale`	`title`
`offset`	`slaveClocks`	

Inherited from `Player`:

`audioMuted`	`globalContrast`	`globalVolumeOffset`
`dataRate`	`globalHue`	`markerList`
`duration`	`globalPanOffset`	`status`
`globalBrightness`	`globalSaturation`	`videoBlanked`

Inherited from `MediaStreamPlayer`:

`frameRate`	`mediaStream`	

Inherited from `Presenter`:

`presentedBy`	`subPresenters`	`target`

Inherited from `TwoDPresenter`:

`bBox`	`height`	`transform`
`boundary`	`IsImplicitlyDirect`	`width`
`clock`	`isTransparent`	`window`
`compositor`	`isVisible`	`x`
`direct`	`needsTickle`	`y`
`eventInterests`	`position`	`z`
`globalBoundary`	`stationary`	
`globalTransform`	`target`	

The following instance variables are defined in `DigitalVideoPlayer`:

frame

self`.frame` (read-write) `Integer`

Specifies the current frame of the video. Use this instance variable to position the player at a specific frame.

frameRate (MediaStreamPlayer)

self`.frameRate` (read-only) `Integer`

Specifies the frame rate of the video for the current frame. This is not the rate of the digital video player *self*, but the rate of the video coming from the stream. The frame rate may vary throughout the video. (This instance variable is inherited from the class `MediaStreamPlayer`.)

inkMode

self`.inkMode` (read-write) `NameClass`

Specifies which transfer, or "ink," to use to apply the video being played by the digital video player *self* to the presenter surface. Valid values are documented in the `Brush` class. The ink mode is not used when the player's `direct` instance variable is set to `true`.

invisibleColor

self`.invisibleColor` (read-write) `Color`

Specifies the invisible color within the image area of the image presented by the video player *self*.

Instance Methods

Inherited from `Clock`:

`addPeriodicCallback`	`clockAdded`	`pause`
`addRateCallback`	`clockRemoved`	`resume`
`addScaleCallback`	`effectiveRateChanged`	`timeJumped`
`addTimeCallback`	`forEachSlave`	`waitTime`
`addTimeJumpCallback`	`isAppropriateClock`	`waitUntil`

Inherited from `Player`:

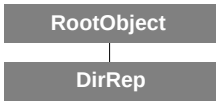
`addMarker`	`goToBegin`	`playPrepare`

eject	goToEnd	playUnprepare
fastForward	goToMarkerFinish	playUntil
getMarker	goToMarkerStart	resume
getNextMarker	pause	rewind
getPreviousMarker	play	stop

Inherited from TwoDPresenter:

adjustClockMaster	inside	show
createInterestList	localToSurface	surfaceToLocal
draw	notifyChanged	tickle
getBoundaryInParent	recalcRegion	
hide	refresh	

DirRep



Class type: Core class (abstract)
Resides in: ScriptX and KMP executables
Inherits from: `RootObject`
Component: Files

The `DirRep` class has methods that allow you to create, delete, test, and open text files, binary files, and directories. Its `fileIn` method also lets you compile scripts written in ScriptX (this method is not available in the Kaleida Media Player).

In addition, it has the `createDir` method to create `DirRep` objects for new directories, and the `spawn` method to create `DirRep` objects for existing directories. For each platform that ScriptX supports, there is a file-system-specific subclass of `DirRep` to represent its directories: the Macintosh has `HFSDirRep`, Windows has `FATDirRep`.

When you create a `DirRep` object, it is always created relative to an existing `DirRep` object. Several important instances of `DirRep` are available for you to use:

- The `directory` instance variable of title, library, and accessory containers,
- The global constants `theRootDir`, `theTempDir`, `theStartDir`, and `theScriptDir` (the latter is available only in the ScriptX development environment after a script has been loaded)

By coercing a `DirRep` into a sequence, you get a collection of strings representing the name of each subdirectory in its path. For example, the following code returns a sequence of strings for the global `theStartDir`:

```
theStartDir as Sequence
⇨  #("d","scriptx","players","kmp")
```

Important – Do not add an instance of `DirRep` to a title container. `DirRep` objects cannot be saved. Calling `update` on an instance will generate an exception.

In ScriptX, `DirRep` objects cannot be persistent because they represent only a temporary and transitory state of the system. Directories can be renamed or moved to other directories or machines with different paths names, aliases, and so forth.

As an alternative to saving an instance of `DirRep`, create a transitory `DirRep` object based on the position of the directory relative to one of the directories described by the global constants `theRootDir`, `theTempDir`, `theStartDir`, and `theScriptDir`. Alternatively, use the `directory` instance variable of a library container. If you require absolute references to directories, you can save strings or sequences representing directory paths, and use them to reconstruct the corresponding `DirRep` instance as needed.

Creating and Initializing a New Instance

You never create an instance of `DirRep` by calling `new`. Instead, you call either `createDir` or `spawn`, depending on whether you want it to represent a new or existing directory.

- To create a *new* directory and return an instance of `DirRep` that represents it, use `createDir`:

```
createDir theStartDir "mysubdir"
```

This creates a directory called `mysubdir` located in the directory where ScriptX resides. If the directory already exists, it creates a different instance of `DirRep` for the same directory.

- To create an instance of `DirRep` for an *existing* directory, use the `spawn` method:

```
spawn theStartDir "mysubdir"
```

This returns a `DirRep` for the directory called `mysubdir` located in the directory where ScriptX resides. If `mysubdir` does not exist, it throws an exception.

Note that the second argument in `createDir` and `spawn` can be specified three ways:

- A string containing just the directory name (as shown above)
- A string containing a series of directory names separated by slashes (`/`):

```
createDir theStartDir "mysubdir/deepdir"
```

- A sequence of strings (each string can have slashes):

```
createDir theStartDir #("mysubdir","deepdir")
```

When creating a directory that is specific to a particular title, normally you want the new directory to be relative to the title container's directory, rather than relative to `theStartDir`. You can do this by using the `titleContainer` instance method, as follows:

```
tc := new TitleContainer dir:theStartDir path:"mytitle.sxt"
createDir tc.directory "mysubdir"
```

Also see the "Files" chapter in the *ScriptX Components Guide* for other examples that create `DirRep` instances and access files.

The `DirRep` class is abstract, meaning you cannot directly create an instance of it. However, the methods `createDir` and `spawn`, as described above, automatically create instances of the appropriate platform-specific subclass of `DirRep`, such as `HFSDirRep` or `FATDirRep`.

Instance Methods

createDir

`createDir` *self path* ⇨ `DirRep`

self	`DirRep` object
path	`Sequence` or `String` object

Creates any new directories in *path* and returns a new `DirRep` instance representing the directory corresponding to *self* plus *path*. If the *path* already exists, it creates a different `DirRep` instance for the same path. If *path* specifies an existing file, an exception is thrown. This method is similar to `spawn`, except that it creates new directories and `spawn` does not.

Note – You can't create a directory directly in `theRootDir`, the global variable for the file system, since this would be equivalent to creating a new volume, hard disk, or device.

createFile

`createFile` *self path type* ⇨`String`

self	`DirRep` object
path	`Sequence` or `String` object
type	`NameClass` object

Creates a text file, binary file or directory (depending on *type*) in a location relative to the directory given by *self* and *path*, where the last string in *path* is the name of the file or directory. If the directories specified in *path* don't exist, this method creates them. The argument *type* is one of these names:

`@text` – a file with data in ASCII format

`@binary` – a file with data in binary format

`@directory` – a directory

If this method succeeds, it returns the filename or directory name as a `String`. If it fails because the specified file already exists, or for any other reason, it raises an exception.

You can create files and directories in any directory by calling `createFile` on the instance of a `DirRep` representing that directory. The following example creates a new text file, called `myfile`, located in the title container's directory:

```
createFile tc.directory "myfile" @text
```

delete

`delete` *self path* ⇨ *(none)*

self	`DirRep` object
path	`Sequence` or `String` object

Deletes the files or subdirectories represented in *path* from the directory represented by *self*. If *path* represents a directory containing files, this method won't delete the directory. If *path* represents a file that is locked or busy, this method won't delete the file.

Note – On Windows systems, this method will delete a file even if it has an open stream on it.

If this method succeeds, it has no return value. If it fails, it reports an exception.

fileIn

`fileIn` *self* `name:`*string* [`debugInfo:`*boolean*]
[`module:`*module*] [`quiet:`*boolean*] ⇨ *self*

self	`DirRep` object containing file to be compiled
`name:`	`Sequence` or `String` object representing the path and source file.
`debugInfo:`	`Boolean` object specifying whether to keep the source code. If `true`, the source code is put in the `source` instance variable of the `debugInfo` object in the

	`ByteCodeMethod` object's `debugInfo` instance variable. (Each method or function in the file is turned into a separate `ByteCodeMethod` object.)
`quiet:`	`Boolean` object

Note – Available only in ScriptX, not the Kaleida Media Player. The KMP doesn't include the ScriptX bytecode compiler required to execute this function.

Compiles the ASCII-format ScriptX source code contained in the file represented by *path* in the directory represented by *self*, into ScriptX bytecode format and executes the results.

To compile a string (or any bytestream not associated with a file), see the `fileIn` method in the `ByteStream` class.

If you omit an optional keyword, its default value is used. The defaults are:

```
debugInfo:true
module:scratch
quiet:true
```

fixNameForOS

`fixNameForOS` *self filename* ⇨ `ByteString`

self	`DirRep` object
filename	`Sequence` or `String` object

Resolves the string *filename* to a platform-specific representation appropriate to the host file system. For example, on a DOS system, this method returns a string that is eight characters maximum, with an extension of three characters maximum.

This method allows a script to test how instances of a `DirRep` subclass will resolve filenames passed as arguments. Methods that use this resolution mechanism to act on specific files include `getFileType` and `getStream`. The string returned by this method is the platform-specific version of *filename*.

getContents

`getContents` *self* ⇨ `ArrayList`

Returns a collection of strings representing the names of files and directories inside the directory represented by *self*.

getFileType

`getFileType` *self path* ⇨ `NameClass`

self	`DirRep` object
path	`Sequence` or `String` object

Returns a name representing the file type of the file-system object indicated by *path*, as listed below. The file and path represented by *path* must already exist.

`@directory` – a directory

`@text` – a file with data in ASCII format

`@binary` – a file with data in binary format

`@title` – a title container file

`@library` – a library container file

`@accessory` – an accessory container file

`@application` – an executable file

`@unknown` – unknown file

getStream

`getStream` *self path mode* ⇨`Stream`

self	`DirRep` object
path	`Sequence` or `String` object
mode	`NameClass` object

Returns a stream to read or write (depending on *mode*) from the file specified in *path* within the directory represented by *self*. If *path* contains subdirectory references, this method resolves them, but doesn't create missing directories. The value of *mode* may be one of three constants: `@readable`, `@writable`, or `@readWrite`. This method doesn't create new files; use the `createFile` method to do so.

isDir

`isDir` *self path* ⇨`Boolean`

self	`DirRep` object
path	`Sequence` or `String` object

Returns `true` if the file-system object represented by the *path* argument is a directory within the directory represented by *self*; otherwise it returns `false`.

isFile

`isFile` *self path* ⇨`Boolean`

self	`DirRep` object
path	`Sequence` or `String` object

Returns `true` if the file-system object represented by the *path* argument is a file; otherwise it returns `false`. The *path* argument is interpreted relative to the directory represented by *self*.

isThere

`isThere` *self path* ⇨`Boolean`

self	`DirRep` object
path	`Sequence` or `String` object

Returns `true` if the file or directory represented by *path* exists relative to the directory *self*; otherwise it returns `false`.

parentDir

`parentDir` *self* ⇨`DirRep`

Returns a directory representing the parent directory *self*. If the parent is inaccessible or does not exist, this method reports an exception.

spawn

`spawn` *self path* ⇨`DirRep`

self	`DirRep` object
path	`Sequence` or `String` object

Creates a new instance of `DirRep` for the existing subdirectory *path* relative to the directory *self*. If all directories in *path* do not already exist, this method throws an exception. The *path* argument can be a string or a sequence of strings. For example:

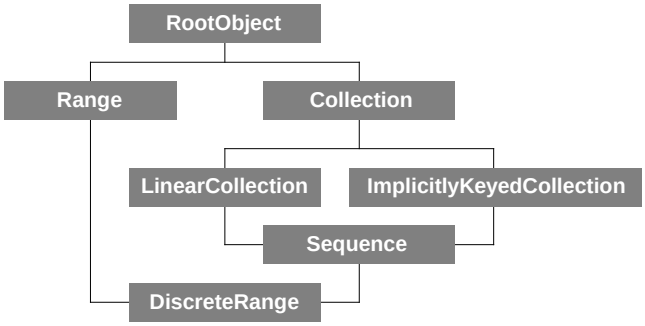
```
spawn theStartDir “tools”
```

Returns the `DirRep` instance for the `Tools` directory inside the ScriptX startup directory.

```
spawn theStartDir “tools/subtools”
```

Returns the `DirRep` instance for the `Subtools` directory inside the `Tools` directory inside the ScriptX startup directory.

DiscreteRange



Class type: Core class (abstract)
Resides in: ScriptX and KMP executables
Inherits from: `Range` and `Sequence`
Component: Collections

The `DiscreteRange` class is an abstract class for objects that display range-like behavior with a noncontinuous increment. `DiscreteRange` mixes in `Range` with `Sequence`, a class that inherits from `Collection`. This allows a discrete range to be used with an iterator—for example, a ScriptX `for` loop can iterate over values in a discrete range.

Among superclasses of `DiscreteRange`, `Range` has precedence in inheritance. Thus, a discrete range implements printing, class coercion, the `sizeGetter` method, and other common behaviors as a range rather than as a collection.

Ranges are immutable as collections. The `Collection` class defines an instance variable, `mutable`, that indicates whether elements can be modified. For instances of `DiscreteRange`, the value of `mutable` is always `false`. Many methods that `DiscreteRange` inherits from `Sequence` and its superclasses are not available to discrete ranges, since they are immutable as collections.

`NumberRange` and `IntegerRange` are also immutable as ranges, since they do not define setter methods for any of their instance variables. However, a subclass of `DiscreteRange` can be a mutable range.

Creating and Initializing a New Instance

Calling `new` on `DiscreteRange` will create an instance of `IntegerRange` or `NumberRange`, depending on the values you supply for the keywords `lowerBound`, `upperBound`, and `increment`. If the values supplied for all keywords are integers, the result will be an instance of `IntegerRange`. If one or more of the values supplied contains a decimal point, the result will be an instance of `NumberRange`. Note that you can also call `new` directly on `IntegerRange` and `NumberRange` to instantiate them.

The following script is an example of how to create a new instance of the `NumberRange` class by calling `new` on `DiscreteRange`.

```
myRange := new DiscreteRange \
       lowerBound:(-2 * pi) \
       upperBound:(2 * pi) \
       increment:(pi / 2)
```

The variable `myRange` contains the initialized number range. The instance contains the 9 numbers from approximately -6.28 to +6.28 at intervals of about 1.57:

-6.283, -4.712, -3.141, -1.570, 0, 1.570, 3.141, 4.712, 6.283

The `new` method uses the keywords defined by the `init` method.

init

`init` *self* `lowerBound:`*number* `upperBound:`*number* [`increment:`*number*] ⇨ *(none)*

self	`NumberRange` object
`lowerBound:`	`Number` object
`upperBound:`	`Number` object
`increment:`	`Number` object

Initializes the `NumberRange` object *self*, applying the values supplied with the keywords to the instance variables of the same name. Do not call `init` directly on an instance—it is automatically called by the `new` method.

If you omit an optional keyword, its default value is used. The default value is:

```
increment:1
```

For negative increments, reverse `lowerBound` and `upperBound`—set the starting value to `lowerBound` and the ending value to `upperBound`, as in the following example:

```
myReverseRange := new DiscreteRange \
              lowerBound:5 \
              upperBound:0 \
              increment:-1.0
```

The variable `myReverseRange` contains an initialized instance of `NumberRange`. This instance incorporates the values 5.000000, 4.000000, 3.000000, 2.000000, 1.000000, 0.000000. In the example above, if the increment given had been 1 instead of 1.0, the result would have been an instance of `IntegerRange` containing the values 5, 4, 3, 2, 1, 0.

Class Methods

Inherited from `Collection`:

`pipe`

Instance Variables

Inherited from `Collection`:

`bounded`	`maxSize`	`size`
`iteratorClass`	`minSize`	`uniformity`
`keyEqualComparator`	`mutable`	`uniformityClass`
`keyUniformity`	`mutableCopyClass`	`valueEqualComparator`
`keyUniformityClass`	`proprietored`	

Inherited from `Range`:

`includesLower`	`lowerBound`	`valueClass`
`includesUpper`	`size`	
`increment`	`upperBound`	

Instance Methods

Inherited from `Collection`:

`add`	`forEach`	`iterate`
`addMany`	`forEachBinding`	`localEqual`
`addToContents`	`getAll`	`map`
`chooseAll`	`getAny`	`merge`
`chooseOne`	`getKeyAll`	`pipe`
`chooseOneBinding`	`getKeyOne`	`prin`

deleteAll	getMany	removeAll
deleteBindingAll	getOne	removeOne
deleteBindingOne	hasBinding	setAll
deleteKeyAll	hasKey	setOne
deleteKeyOne	intersects	size
deleteOne	isEmpty	
emptyOut	isMember	

Inherited from LinearCollection:

chooseOneBackwards	findRange	getNthKey
chooseOrdOne	forEachBackwards	getOrdOne
deleteFirst	getFirst	getRange
deleteLast	getLast	localEqual
deleteNth	getMiddle	localLT
deleteRange	getNth	pop

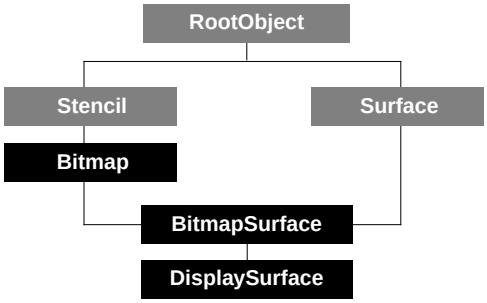
Inherited from Sequence:

addFifth	moveBackward	setFourth
addFirst	moveForward	setLast
addFourth	moveToBack	setNth
addNth	moveToFront	setSecond
addSecond	prepend	setThird
addThird	prependNew	sort
append	setFifth	
appendNew	setFirst	

Inherited from Range:

withinRange

DisplaySurface



Class type: Core class (concrete)
Resides in: ScriptX and KMP executables
Inherits from: `BitmapSurface`
Component: 2D Graphics

The `DisplaySurface` class corresponds to a particular graphic device on the platform. Operations applied to this surface are immediately performed on the device pixel memory. On operating systems that support windowing, an instance of `DisplaySurface` may correspond to an operating system window, and the `bBox` instance variable represents the coordinates of the surface in screen (or display) coordinates. You can create instances of `DisplaySurface` directly using the `new` method. However, in general you get a display surface by creating a new instance of `Window`, then adding `TwoDPresenter` objects to the window. You override the `draw` method defined in `TwoDPresenter` to perform rendering operations on the surface represented by the window.

Note – Although `DisplaySurface` inherits `matteColor` and `invisibleColor` instance variables from the class `Bitmap`, these instance variables are read-only for `DisplaySurface`.

Creating and Initializing a New Instance

The following script is an example of how to create a new instance of the `DisplaySurface` class:

```
myDisplay:= new DisplaySurface \
          bBox:(new Rect x2:100 y2:100)
```

The variable `myDisplay` contains the new `DisplaySurface` object. The surface has a boundary of 100 by 100 and is positioned at the upper-left corner (origin) of the graphic device.

init

`init` *self* [`bBox:`*rect*] [`colorMap:`*colorMap*] [`name:`*string*] ⇨ *(none)*

self	`DisplaySurface` object

Superclasses of `DisplaySurface` use the following keywords:

`bBox:`	`Rect` object
`colorMap:`	`ColorMap` object
`name:`	`String` object

Initializes the `DisplaySurface` object *self*. The other arguments are described with the `Bitmap` class. The `Rect` object argument to the `bBox` keyword specifies the size of the display surface. The `x1` and `y1` values of this `Rect` instance specify the offset of the display surface from the origin of the display device on which it appears. The origin of a display device is at the upper-left of the screen, while the origin of a `DisplaySurface` instance is at the upper-left of the `Rect` instance supplied as the argument to `bBox`. Do not call `init` directly on an instance—it is automatically called by the `new` method.

If you omit an optional keyword, its default is used. The defaults are:

```
colorMap:theDefault8ColorMap
bBox:(new Rect x1:0 y1:40 x2:600 y2:440)
name:"ScriptX Display Surface"
```

Instance Variables

Inherited from `Stencil`:

`bBox`

Inherited from `Bitmap`:

`bitsPerPixel`	`invisibleColor`	`rowBytes`
`colorMap`	`matteColor`	`size`
`compressionInfo`	`remapOnDraw`	
`data`	`remapOnSet`	

Inherited from `Surface`:

`boundary`

Inherited from `BitmapSurface`:

`data`

The following instance variable is defined in `DisplaySurface`:

name

self`.name` (read-write) `Text` or `String`

The name displayed by the window to which the display surface *self* belongs.

Instance Methods

Inherited from `Stencil`:

`inside`	`onBoundary`	`transform`
`intersect`	`subtract`	`union`

Inherited from `Bitmap`:

`getPixelDepth`

Inherited from `Surface`:

`fill`	`stroke`	`transfer`

Inherited from `BitmapSurface`:

`erase`

The following instance methods are defined in `DisplaySurface`:

defaultMatrix

`defaultMatrix` *self units* ⇨ `TwoDMatrix`

self	`DisplaySurface` object
units	`NameClass` object: `@inches`, `@cm`, or `@points`

Returns a matrix that represents the coordinate system of the display surface *self* in the units specified by *units*. The value of the *units* argument can be `@inches`, `@cm`, or `@points`. Currently, the values returned are based on a ratio of 72 pixels per inch.

You can use the matrix returned by this method to transform a point in display coordinates into a point on the surface. In ScriptX coordinate systems, coordinates increase as you move to the right and down.

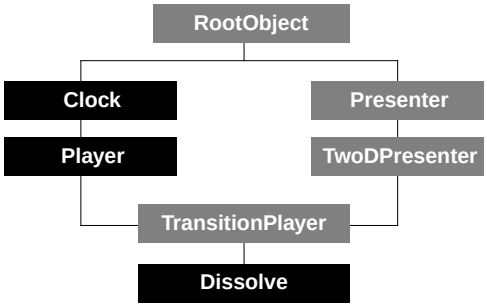
transform

`transform` *self matrix result* ⇨ *self*

self	`DisplaySurface` object
matrix	`TwoDMatrix` object
result	`NameClass` object

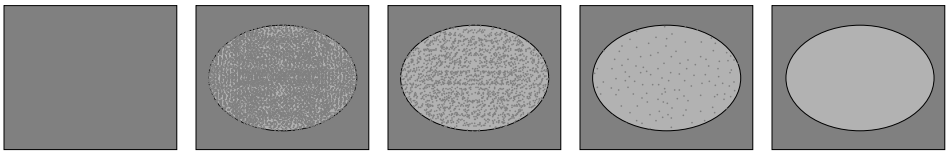
The `transform` method, inherited from `Stencil`, produces no result in the `DisplaySurface` class.

Dissolve



Class type: Loadable class (concrete)
Resides in: `ltrans.lib`. Works with ScriptX and KMP executables.
Inherits from: `TransitionPlayer`
Component: Transitions

The `Dissolve` transition player provides a visual effect that causes the target to gradually appear as small, random dots, as shown below. Unlike most transitions, the transition cannot be played backwards, so the target cannot be gradually hidden using this transition.



Directions: *(none)*

Rate: Must be zero or positive. Cannot play backward.

For a side-by-side illustrations of all transitions, see the Transitions chapter in the *ScriptX Components Guide*.

Creating and Initializing a New Instance

The following script is an example of how to create a new instance of the `Dissolve` class:

```
myTransition := new Dissolve \
                    duration:60 \
                    target:myShape
```

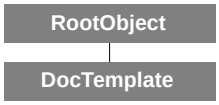
The variable `myTransition` contains the initialized transition. The transition reveals the image `myShape` as randomly distributed pixels and has a duration of 60 ticks.

You determine which space the transition will take effect in by adding this instance into that space. Then, when you play the transition player, `myShape` is transitioned into that space.

The `new` method uses the keywords defined in `init`.

NOTE – For the instance variables and methods, see the `BarnDoor` class.

DocTemplate



Class type: Core class (abstract)
Resides in: ScriptX and KMP executables
Inherits from: `RootObject`
Component: Document Templates

The `DocTemplate` class is a parent class of `Page`, `PageElement`, `PageLayer`, and `PageTemplate`. It provides methods and instance variables needed by objects participating in a document hierarchy.

A "document hierarchy" is a hierarchy of objects in a document. The relationship between the objects in this kind of hierarchy is a "presented by" relationship. In a document, a `Presenter` object is presented by a `PageElement` object, which is usually presented by a `PageLayer` object, which is usually presented by a `PageTemplate`, which is presented by a `Page`, which is presented by a `Document`, which is presented by a `TwoDSpace` such as a `Window`.

Instance Variables

The following instance variables are defined in `DocTemplate`:

data

self`.data` (read-only) *(object)*

Specifies data to be displayed by a page element in a document.

When *self* is a `PageElement` instance, this instance variable returns the actual data to be displayed by the page element.

When *self* is any other object in a document hierarchy, such as a `Document`, `Page`, `PageTemplate`, or `PageLayer`, it specifies data that can be used by a `PageElement`. Only page elements actually display data.

The `data` instance variable is a simulated instance variable, and its value is calculated dynamically each time the instance variable is accessed, which usually occurs when a page opens. This "on the fly" data generation allows the elements of a page in a document to update their data every time a page opens.

The `data` instance variable gets its value by evaluating the expression in the `target` instance variable on *self*. The value in the `target` instance variable may be any constant value, in which case `target` and `data` are one and the same, or it may be a function that is evaluated to produce the data.

When a page opens, the `changePage` method is called on all the `PageElement` objects used on that page. The `changePage` method on a `PageElement` object causes the page element to update its `data` instance variable to determine what to display. The page element gets its data by evaluating the expression in its `target` instance variable (inherited from `Presenter`.). If that expression makes a reference to the `data` instance variable of a parent object of the page element (such as a page or document), for example, then the expression in the `target` instance variable of the appropriate parent is evaluated.

The `data` instance variable is defined at the level of `DocTemplate` to make it available to all classes that inherit from `DocTemplate`, such as `Document`, `Page`, `PageTemplate`, `PageLayer` and `PageElement`.

Note that the `data` instance variable is always dynamically evaluated when it is accessed. Thus you cannot necessarily find the value displayed by a presenter in an open page by retrieving the value of its `data` instance variable.

Suppose that a page has a `TextPresenter` that displays a randomly-generated wise and witting saying. When the page opens, the saying is randomly generated, for example, the text presenter in question says "Look not for cows in Bodfish."

You might think that if you get the value of the `data` instance variable of the text presenter, it will return the same saying as is displayed on the page, but this is not the case. It will randomly generate another saying which could be quite different to the one displayed.

Instance Methods

findNthParent

`findNthParent` *self templateClass nth* ⇨ *(presenter)*

self	`DocTemplate` object
templateClass	A `class` whose instances can participate in a document template hierarchy, such as `PageElement`, `PageLayer`, `PageTemplate`, `Page` or `Document`.
nth	`Integer` object

This method returns an instance of *templateClass* that is presenting the `DocTemplate` instance *self*.

Use this method instead of `findParent` when the document hierarchy contains multiple instances of *templateClass*. This method returns the *n*th instance of *templateClass* above *self* in the document hierarchy.

For example, suppose a page element `e1` is added to a page layer, `layer1`, which in turn is added to a page layer, `layer2`. In this case:

```
findNthParent e1 PageLayer 1
```

returns `layer1`.

```
findNthParent e1 PageLayer 2
```

returns `layer2`.

Since `DocTemplate` is never instantiated, this method is called only by its subclasses.

findParent

`findParent` *self templateClass* ⇨ *(presenter)*

self	`DocTemplate` object
templateClass	A `class` whose instances can participate in a document template hierarchy, such as `PageElement`, `PageLayer`, `PageTemplate`, `Page` or `Document`.

This method returns the instance of *templateClass* that is presenting the `DocTemplate` instance *self*.

For example, if *self* is a page element and *templateClass* is `Document`, `findparent` returns the document containing the page element.

If *self* is a page element and *templateClass* is `Page`, then `findParent` returns the page containing the page element.

If *self* is a page and *templateClass* is `Document`, then `findPresenter` returns the document containing the page, and so on.

Since `DocTemplate` is never instantiated, this method is called only by its subclasses.

getNthParentData

`getNthParentData` *self templateClass nth* ⇨ *(object)*

self	`DocTemplate` object
templateClass	A `class` whose instances can participate in a document template hierarchy, such as `PageElement`, `PageLayer`, `PageTemplate`, `Page` or `Document`.
nth	`Integer` object

Use this method instead of `getParentData` when the document hierarchy contains multiple instances of *templateClass*. This method returns the value of the `data` instance variable of the *n*th such object. The value for the `data` instance variable is obtained automatically by the system by evaluating the expression in the `target` instance variable of the *n*th such object. This method might be useful in the case where `PageLayer` objects contain other `PageLayer` objects thus setting up an extended document hierarchy.

Since `DocTemplate` is never instantiated, this method is called only by its subclasses.

getParentData

`getParentData` *self templateClass* ⇨ *(object)*

self	`DocTemplate` object
templateClass	A `class` whose instances can participate in a document template hierarchy, such as `PageElement`, `PageLayer`, `PageTemplate`, `Page` or `Document`.

Returns the result of evaluating the function in the `target` instance variable on the instance of *templateClass* in the document hierarchy containing the object *self*. The argument *self* is automatically passed into the target function . For an example of `getParentData`:

```
getParentData myPageElement Page
```

returns the resultsof evaluating the `target` instance variable in the `Page` object that contains the page element `myPageElement`. (This is equivalent to calling `myPage.target myPage`.)

As another example:

```
getParentData myPageElement Document
```

returns the result of evaluating the `target` instance variable in the `Document` object that contains the page element `myPageElement`.

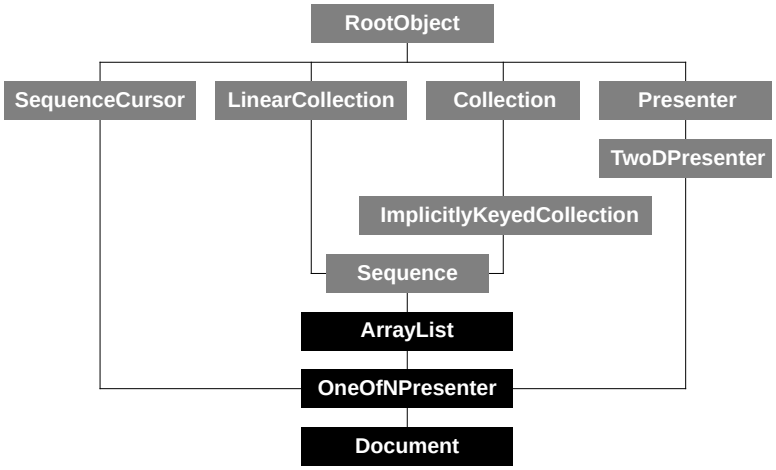
This method takes advantage of the fact that under normal circumstances in a document template hierarchy, a `Document` contains a `Page` which contains a `PageTemplate` which contains multiple `PageLayer` objects, which each contain `PageElement` objects.In this context, the object that presents another object is considered to be its parent. For example, a `Document` is the parent of a `Page`, which is the parent of a `PageTemplate`, which is the parent of a `PageLayer`, which is the parent of a `PageElement`, which is the parent of the presenter object in its `element` instance variable.

The `getParentData` method is a support method that can be called within the expression in the `target` instance variable of an object in a document template hierarchy, such as a `PageElement`. The `getParentData` method finds the "parent" of *self* in *self*'s document template hierarchy and returns the value in the parent's `data` instance variable. The value in the `data` instance variable is dynamically evaluated every time it is accessed, using the expression in the `target` instance variable. The target of an object in a document template hierarchy may be any constant value, or it may be a function which is evaluated to produce the data.

This method is defined at the level of `DocTemplate` so that it is available to all instances of subclasses of `DocTemplate`, such as `Document`, `PageTemplate`, `Page`, `PageLayer` and `PageElement`. Since `DocTemplate` is never instantiated, this method is called only by instances of its subclasses.

For more information, see the "Document Templates" chapter in the *ScriptX Components Guide*.

Document



Class type: Core class (concrete)
Resides in: ScriptX and KMP executables
Inherits from: `OneOfNPresenter`
Component: Document Templates

A `Document` object is a collection of `Page` objects. As a subclass of `OneOfNPresenter`, it displays only one presenter (page) at a time. Because `Document` is a `OneOfNPresenter`, it changes its boundary to whatever presenter it is currently playing.

Creating and Initializing a New Instance

The following script is an example of how to create a new instance of the `Document` class:

```
myDocument := new Document \
    cover:redPlanet \
    name:(new Text string:"The Mars Story")
```

The variable `myDocument` contains an instance of `Document` that can be graphically represented by `redPlanet`, which should be a `Presenter`. The `new` method uses the keywords defined in `init`, on the class `Document`.

To make the document useful, you need to append `Page` objects to it.

init

`init` *self* [`cover:`*presenter*] [`name:`*text*] [`target:`*objectOrFunction*] ⇨ *(none)*

self	`Document` object
`cover:`	`TwoDPresenter` object
`name:`	`Text` object
`target:`	An expression or object that can be used to generate or point to data for the document

Superclasses of `Document` use these keywords:

`boundary:`	(Ignored by `Document`)
`initialSize:`	(Ignored by `Document`)
`growSize:`	(Ignored by `Document`)
`stationary:`	`Boolean` object

Initializes the Document object *self*, applying the arguments as follows: cover is a TwoDPresenter object that is put in the cover instance variable; name is a Text object that is put in the name instance variable; target is an object or expression that evaluates to the data needed by the page elements on a page in the document when the page is displayed. Do not call init directly on an instance—it is automatically called by the new method.

If you omit an optional keyword, its default value is used. The defaults are

```
cover:undefined
name:(Text object with a string of zero length)
target:undefined
stationary:false
```

Class Methods

Inherited from Collection:

```
pipe
```

Instance Variables

Inherited from Presenter:

```
presentedBy          subPresenters        target
```

Inherited from TwoDPresenter:

```
bBox                 height               transform
boundary             IsImplicitlyDirect   width
clock                isTransparent        window
compositor           isVisible            x
direct               needsTickle          y
eventInterests       position             z
globalBoundary       stationary
globalTransform      target
```

Inherited from Collection:

```
bounded              maxSize              size
iteratorClass        minSize              uniformity
keyEqualComparator   mutable              uniformityClass
keyUniformity        mutableCopyClass     valueEqualComparator
keyUniformityClass   proprietored
```

Inherited from SequenceCursor:

```
cursor
```

The following instance variables are defined in Document:

cover

self.cover (read-write) TwoDPresenter

Specifies a single presenter that is used to represent the document *self* should the need arise, much as an icon is associated with a file in a graphical user interface. It is also similar to the poster provided by QuickTime® movies.

data

self.data (read-only) *(object)*

Specifies data that can be used by PageElement objects in the document to determine what to present on a page in the document *self*.

The value of `data` is determined by evaluating the `target` instance variable. The target of a `Document` object may be any constant value, in which case `target` and `data` are one and the same, or it may be a function which is evaluated to produce the data. The data must make sense to the objects contained within the document. Ultimately, `data` can be used by the data reference within `PageElement` objects.

The value of `data` is calculated each time it is needed, for example, each time a page is displayed where the page elements on the page refer to the `data` instance variable of the document.

name

self`.name` (read-write) `Text`

Specifies the name of the document *self*.

target

self`.target` (read-write) *(object or expression)*

This instance variable is inherited from `Presenter`, but is used slightly differently in `Document`. The `target` of a document *self* is an object or function that evaluates to the value for the `data` instance variable of the document. The value in the `target` instance variable of a document is evaluated only if the `PageElement` instances on a `Page` need to access the `data` instance variable on the document to find out what to present.

Basically, the `target` instance variable of a document holds an object or expression that evaluates to data to be used by page elements in the document.

Instance Methods

Inherited from `TwoDPresenter`:

`adjustClockMaster`	`inside`	`show`
`createInterestList`	`localToSurface`	`surfaceToLocal`
`draw`	`notifyChanged`	`tickle`
`getBoundaryInParent`	`recalcRegion`	
`hide`	`refresh`	

Inherited from `Collection`:

`add`	`forEach`	`iterate`
`addMany`	`forEachBinding`	`localEqual`
`addToContents`	`getAll`	`map`
`chooseAll`	`getAny`	`merge`
`chooseOne`	`getKeyAll`	`pipe`
`chooseOneBinding`	`getKeyOne`	`prin`
`deleteAll`	`getMany`	`removeAll`
`deleteBindingAll`	`getOne`	`removeOne`
`deleteBindingOne`	`hasBinding`	`setAll`
`deleteKeyAll`	`hasKey`	`setOne`
`deleteKeyOne`	`intersects`	`size`
`deleteOne`	`isEmpty`	
`emptyOut`	`isMember`	

Inherited from `LinearCollection`:

`chooseOneBackwards`	`findRange`	`getNthKey`
`chooseOrdOne`	`forEachBackwards`	`getOrdOne`
`deleteFirst`	`getFirst`	`getRange`
`deleteLast`	`getLast`	`localEqual`
`deleteNth`	`getMiddle`	`localLT`
`deleteRange`	`getNth`	`pop`

Inherited from Sequence:

addFifth	moveBackward	setFourth
addFirst	moveForward	setLast
addFourth	moveToBack	setNth
addNth	moveToFront	setSecond
addSecond	prepend	setThird
addThird	prependNew	sort
append	setFifth	
appendNew	setFirst	

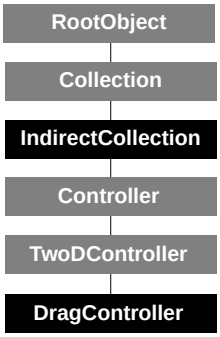
Inherited from SequenceCursor:

backward	forward	goTo

Inherited from OneOfNPresenter:

draw	goTo

DragController



Class type: Core class (concrete)
Resides in: ScriptX and KMP executables
Inherits from: `TwoDController`
Component: User Interface

`DragController` is a controller class that manages `Dragger` objects in the space it controls. A dragger must be associated with a drag controller before it can be dragged. A `DragController` object expects to be attached to model objects that inherit from `TwoDPresenter`, and that mix in the `Dragger` class. Multiple instances of `Dragger` can be attached to a drag controller.

A `DragController` object is a "ticklish" controller. This means that, like `Movement` and `Interpolator`, it implements a `tickle` method that is called with each tick of the space's clock. You control how often the controller can update the location of a dragger by changing the value of `scale` on the space's clock. If the clock's scale is low, dragging may appear jumpy. However, there is no benefit to setting the scale any faster than the compositor's own scale, since it is the compositor's scale that determines how often the screen is updated.

A `DragController` object is a collection of the `Dragger` objects it controls. These draggers must also be in the space that the controller is controlling. Draggers are either automatically or manually added to the drag controller, according to the value of `wholeSpace`, an instance variable defined by `Controller`. If the value of `wholeSpace` is `true`, all `Dragger` objects in the space are controlled by the given `DragController` object. If `wholeSpace` is `false`, you can use the methods defined by `Collection` to add and remove objects individually from the drag controller. To ensure that only draggers are added to a drag controller, the `protocols` instance variable is set to the `Dragger` class. See the `Controller` class for descriptions of `wholeSpace`, `protocols`, and other general properties of controllers. See also the section "Hit Testing" in the "User Interface" chapter of the *ScriptX Components Guide*.

Note – The instance variable `dragInterest` and the method `processMove`, both defined by `DragController` in version 1.0 of ScriptX, have been eliminated in version 1.5. With ScriptX version 1.5, a drag controller polls the mouse device for mouse movements each time `tickle` is called rather than waiting to receive mouse move events.

`DragController` defines the instance variables `dragInterest` and `grabInterest`. These variables store event interests—the `DragController` class is interested only in mouse events. Corresponding with these interests, the class defines the instance methods `processDrop` and `processGrab`. These methods are event receivers, and they are registered automatically as receivers for these event interests. For information on the event system, see the "Events and Input Devices" chapter in the *ScriptX Components Guide*. `DragController` also implements a method for `tickle`, which is called with

each tick of the space's clock. The `tickle` method calls `beforeDrag` and then `afterDrag` on the associated dragger. For information on `tickle`, see "The Ticklish Protocol" in the "Controllers" chapter of *ScriptX Components Guide*.

Table 5: How a DragController and a Dragger work together

DragController (instance variable)	DragController (instance method)	Dragger (instance method)	Dragger (instance variable)
`grabInterest`	`processGrab`	`grab`	`grabAction`
	`tickle`	`beforeDrag`	`beforeDragAction`
	`tickle`	`afterDrag`	`afterDragAction`
`dropInterest`	`processDrop`	`drop`	`dropAction`

Creating and Initializing a New Instance

The following script is an example of how to create a new instance of the `DragController` class, after first creating a space to control:

```
mySpace := new TwoDSpace (new Rect x2:640 y2:480)
mySchlepper := new DragController space:mySpace
```

The variable `mySchlepper` contains the initialized instance. The `new` method uses the keywords defined in `init`.

init

`init` *self* [`space:`*space*] [`wholeSpace:`*boolean*] [`enabled:`*boolean*]
[`targetCollection:`*sequence*] ⇨ *(none)*

self `DragController` object

The superclass `Controller` uses the following keywords:

`space:`	`Space` object
`wholeSpace:`	`Boolean` object
`enabled:`	`Boolean` object

The superclass `TwoDController` uses the following keyword:

`targetCollection:`	`Sequence` object (use with caution)

Initializes the `DragController` object *self*, applying the keyword arguments to instance variables of the same name. Use discretion in changing the target collection; for more information, see the definition of the `TwoDController` class. Do not call `init` directly on an instance—it is automatically called by the `new` method.

If you omit an optional keyword, its default value is used. The defaults are:

```
space:undefined
wholeSpace:false
enabled:true
targetCollection:(new Array initialSize:14 growable:true)
```

Class Methods

Inherited from `Collection`:

`pipe`

Instance Variables

Inherited from `Collection`:

`bounded`	`maxSize`	`size`
`iteratorClass`	`minSize`	`uniformity`
`keyEqualComparator`	`mutable`	`uniformityClass`
`keyUniformity`	`mutableCopyClass`	`valueEqualComparator`
`keyUniformityClass`	`proprietored`	

Inherited from `IndirectCollection`:

`targetCollection`

Inherited from `Controller`:

`enabled`	`space`	`wholeSpace`
`protocols`		

The following instance variables are defined in `DragController`:

currentTarget

self`.currentTarget` (read-write) `Dragger`

Specifies the object that the drag controller *self* is dragging. If the current target changes while a drag is in progress, the new object must have the same value in its `presentedBy` instance variable.

dropInterest

self`.dropInterest` (read-write) `MouseEvent`

This event is registered as an interest so that the drag controller *self* can be notified when an object it is controlling has been dropped. This event interest must be an instance of `MouseEvent`. It defaults to an interest in `MouseUpEvent` objects that is associated with the first mouse button (`@mousebutton1`). Its event receiver is `processDrop`. When a matching event is received, the receiver that registered this interest calls `drop` on the object `currentTarget`.

grabInterest

self`.grabInterest` (read-write) `MouseEvent`

This event is registered as an interest so that the drag controller *self* can be notified when an object it is controlling has been grabbed. This event interest must be an instance of `MouseEvent`. It defaults to an interest in `MouseDownEvent` objects that is associated with the first mouse button (`@mousebutton1`). Its event receiver is `processGrab`. When a matching event is received, the receiver calls `grab` on the object `currentTarget`.

offset

self`.offset` (read-only) `Point`

Specifies the offset from the origin (0, 0) of the target presenter *self* to the position within the target presenter where the presenter was grabbed. This value is expressed in the local coordinates of the target.

protocols (Controller)

self.protocols (read-write) Array

Specifies the required protocols for the drag controller *self*. For instances of DragController, this array contains the classes Dragger and TwoDPresenter. Any object added to a drag controller must inherit from these two classes. See the Controller class for a description of protocols.

Instance Methods

Inherited from Collection:

add	forEach	iterate
addMany	forEachBinding	localEqual
addToContents	getAll	map
chooseAll	getAny	merge
chooseOne	getKeyAll	pipe
chooseOneBinding	getKeyOne	prin
deleteAll	getMany	removeAll
deleteBindingAll	getOne	removeOne
deleteBindingOne	hasBinding	setAll
deleteKeyAll	hasKey	setOne
deleteKeyOne	intersects	size
deleteOne	isEmpty	
emptyOut	isMember	

Inherited from IndirectCollection:

isAppropriateObject	objectAdded	objectRemoved

Inherited from Controller:

isAppropriateObject	tickle

Since a DragController object is an indirect collection, you can also use any methods defined in the class specified by targetCollection. The target collection is typically an instance of Array, which inherits from Sequence, so the following instance methods are redirected to this controller.

Accessible from LinearCollection:

chooseOneBackwards	findRange	getNthKey
chooseOrdOne	forEachBackwards	getOrdOne
deleteFirst	getFirst	getRange
deleteLast	getLast	localEqual
deleteNth	getMiddle	localLT
deleteRange	getNth	pop

Accessible from Sequence:

addFifth	moveBackward	setFourth
addFirst	moveForward	setLast
addFourth	moveToBack	setNth
addNth	moveToFront	setSecond
addSecond	prepend	setThird
addThird	prependNew	sort
append	setFifth	
appendNew	setFirst	

The following instance methods are defined in DragController:

processDrop

`processDrop` *self interest event* ⇨ *(none)*

self	`DragController` object
interest	`MouseEvent` object, the `dropInterest` of *self*
event	`MouseEvent` object that matches *interest*

If the controller *self* is currently dragging an object, it calls `drop` on that object. (No testing is done.)

Do not call `processDrop` from the scripter. It is triggered automatically, and is visible to the scripter so that it can be specialized. The method `processDrop` is the event receiver for `dropInterest`.

processGrab

`processGrab` *self interest event* ⇨ *(none)*

self	`DragController` object
interest	`MouseEvent` object, the `grabInterest` of *self*
event	`MouseEvent` object that matches *interest*

If `wholeSpace` is `true`, `processGrab` looks through the collection of draggers controlled by the drag controller *self* to determine which dragger was hit. If `wholeSpace` is `false`, `processGrab` determines which dragger was hit by examining the `presenter` instance variable of the event.

If a dragger was hit, `processGrab` calls `grab` on that object. This triggers the associated action. A dragger is considered hit if an *event* occurs within its boundary that matches the corresponding *interest*. If no dragger was hit, then `processGrab` rejects the event.

Do not call `processGrab` from the scripter. It is triggered automatically, and is visible to the scripter so that it can be specialized. The method `processGrab` is the event receiver for `grabInterest`.

tickle (Controller)

`tickle` *self clock* ⇨ *(none)*

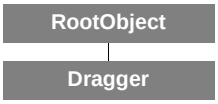
self	`DragController` object
clock	`Clock` object of the space being controlled

The controller's space calls `tickle` on the drag controller *self*, supplying the space's clock as the value for *clock*. It calls `tickle` once with every tick of the clock.

The drag controller polls the mouse device each time `tickle` is called. If the mouse has moved since the last tick of the clock, it calls `beforeDrag` on the associated presenter. It then changes the presenter's coordinates in the space, based on how far the mouse has traveled. (The presenter will appear at the new position after the next compositor cycle.) After it has changed the presenter's coordinates, it calls `afterDrag`. Both `beforeDrag` and `afterDrag` are defined by `Dragger`.

For more information, see the documentation for the `Clock` and `Controller` classes. See also "The Ticklish Protocol" in the "Controllers" chapter of the *ScriptX Components Guide*.

Dragger



Class type: Core class (abstract, mixin)
Resides in: ScriptX and KMP executables
Inherits from: `RootObject`
Component: User Interface

`Dragger` is an abstract, mixin class with a set of fully implemented methods. `Dragger` must be used in conjunction with `DragController`. Once you set up an instance of `DragController` to control a space, you can mix `Dragger` into objects in that space to make them draggable.

In a dragging operation, the user grabs an object, moves it, and finally drops it somewhere within a space. The moving phase is really a continuous series of small moves. `Dragger` has four methods corresponding with the four phases of dragging.

- `grab` for any operation that occurs prior to dragging
- `beforeDrag` for any operation that occurs at the beginning of each move.
- `afterDrag` for any operation that occurs at the end of each move
- `drop` for any operation that occurs when the object is dropped

These methods are not normally called directly from the scripter. They are visible to the scripter so they can be specialized. Note that `grab` and `drop` are each called only once, at the beginning and end of a complete operation. During a dragging operation, the associated drag controller polls the mouse device. Each time the mouse moves, this controller calls `beforeDrag`, then moves the associated presenter, and finally calls `afterDrag`. This allows a script to define an action that occurs both before and after a model object is moved, and to control how the object moves.

A developer can only change the position of an object within the presentation hierarchy at the beginning or end of a drag operation. During a drag operation, `DragController` is limited to changing the location of a model object within that object's current parent presenter.

A developer could specialize `grab` (class-level specialization) or define a function in `grabAction` (instance-level specialization) that moves the object higher in the presentation hierarchy to get a greater range of movement. To change the space that defines the range of movement for a `Dragger` object, add that object to the new space.

See the `DragController` class for a table that shows how instance variables and methods defined by `DragController` correspond with instance variables and methods defined by `Dragger`.

Creating and Initializing a New Instance

You do not create an instance of `Dragger`—it is an abstract class. However, you can create instances of concrete subclasses of `Dragger`. `Dragger` defines an `init` method for initializing these instances.

init

`init` *self* ⇨ *(none)*

self `Dragger` object

Initializes the `Dragger` object *self*. Do not call `init` directly on an instance—it is called automatically by the `new` method.

Instance Variables

afterDragAction

self.`afterDragAction` (read-write) *(function)*

Specifies the function or method that is called after a drag operation on the dragger *self*. This action is invoked only if the mouse has moved since the last time `afterDrag` was called. This function has three arguments:

```
funcName authorData dragger point
```

authorData	data in the `authorData` instance variable.
dragger	the dragger *self* that the action is attached to.
point	the amount that the dragger *self* has moved since the last time `afterDrag` was called.

Although any global function, anonymous function, or method can be assigned to `afterDragAction`, there are differences in how different classes of functions are dispatched. For information on functions and function dispatch, see the "Object System Kernel" chapter of the *ScriptX Components Guide*.

authorData

self.`authorData` (read-write) *(object)*

Supplied as the first argument when the functions defined by `grabAction`, `beforeDragAction`, `afterDragAction`, and `dropAction` are called. Can be any object.

beforeDragAction

self.`beforeDragAction` (read-write) *(function)*

Specifies the function or method that is called before a drag operation on the dragger *self*. This action is invoked only if the mouse has moved since the last time `beforeDrag` was called. This function has the same form and the same three arguments as the one shown above in `afterDragAction`. The *point* argument determines the amount by which the dragger *self* will be moved since the last time `beforeDrag` was called. Note that you can constrain movement by modifying the *x* and *y* in *point*.

Although any global function, anonymous function, or method can be assigned to `beforeDragAction`, there are differences in how different classes of functions are dispatched. For information on functions and function dispatch, see the "Object System Kernel" chapter of the *ScriptX Components Guide*.

dragged

self.`dragged` (read-write) `Boolean`

Indicates whether or not the dragger *self* is currently being dragged. A value of `true` indicates that self is being dragged; `false` indicates that it is not.

dropAction

self.`dropAction` (read-write) *(function)*

Specifies the function or method that is called when a drag operation has been completed on the dragger *self*. This function has the same form and the same three arguments as the one shown above in `afterDragAction`, except that the third argument specifies the location of *self* after the drop occurs.

Although any global function, anonymous function, or method can be assigned to `dropAction`, there are differences in how different classes of functions are dispatched. For information on functions and function dispatch, see the "Object System Kernel" chapter of the *ScriptX Components Guide*.

grabAction

self`.grabAction` (read-write) *(function)*

Specifies the function or method that is called when a drag operation has just begun on the dragger *self*. This function has the same form and the same three arguments as the one shown above in `afterDragAction`, except that the third argument specifies the offset between the object's current location and its location when the pointer grabbed it. It is at this point in a dragging operation, before the model object has actually moved, that a `Dragger` object can promote itself to a different presenter in the presentation hierarchy for a wider range of movement.

Although any global function, anonymous function, or method can be assigned to `grabAction`, there are differences in how different classes of functions are dispatched. For information on functions and function dispatch, see the "Object System Kernel" chapter of the *ScriptX Components Guide*.

Instance Methods

afterDrag

`afterDrag` *self delta* ⇨ *self*

self	`Dragger` object
delta	`Point` object representing the distance moved

Calls the function or method specified in `afterDragAction`. The value of *delta* is the amount that the `Dragger` object *self* has moved since the last time `afterDrag` was called. The `afterDrag` method is automatically called during a drag operation if the mouse has moved since the last time `afterDrag` was called. To specify an operation you want to perform after each drag, you can either override `afterDrag` (class-level specialization) or specify a function in `afterDragAction` (instance-level specialization).

beforeDrag

`beforeDrag` *self delta* ⇨ *self*

self	`Dragger` object
delta	`Point` object representing the distance moved

Calls the function or method specified in `beforeDragAction`. The value of *delta* is the amount that the dragger *self* can move since the last time `beforeDrag` was called. Since the drag has not yet occurred, it is possible to constrain movement. The `beforeDrag` method is automatically called during a drag operation if the mouse has moved since the last time `beforeDrag` was called. To specify an operation you want to perform before each drag, you can either override `beforeDrag` (class-level specialization) or specify a function in `beforeDragAction` (instance-level specialization).

drop

`drop` *self point* ⇨ *self*

self	`Dragger` object
point	`Point` object, representing the final location

Calls the function or method specified in `dropAction`. The value of *point* is the current location of the dragger *self* (after the object is dropped). The `drop` method is automatically called when a drag operation has been completed on *self*. To specify an operation you want to perform with each drop, you can either override `drop` (class-level specialization) or specify a function in `dropAction` (instance-level specialization).

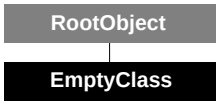
grab

`grab` *self offset* ⇨ *self*

self	`Dragger` object
offset	`Point` *object*

Calls the function or method specified in `grabAction`. The `grab` method is called when a drag operation has just begun on the dragger *self*. The value of *offset* is a `Point` object that indicates the offset between the object's current location (its origin, in local coordinates) and the location where the pointer grabbed it. To specify an operation you want to perform each time an object is grabbed, you can either override `grab` (class-level specialization) or specify a function in `grabAction` (instance-level specialization). If an object should be draggable outside its current presentation space, then this method can be specialized to move that object higher in the presentation hierarchy.

EmptyClass



Class type: Core class (concrete, sealed)
Resides in: ScriptX and KMP executables
Inherits from: `RootObject`
Component: Collections

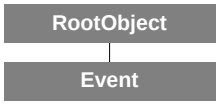
`EmptyClass` is a class that represents an empty key or value. It has a unique instance called `empty`, which is a global constant used with collections.

Note – Never insert the `empty` object into a collection. A number of collection methods return `empty` as the value that signals "no matching elements" in the collection. Placing the `empty` object in the collection would cause these methods to behave erratically, and possibly fail altogether. Some collections report the `badValue` exception if you try to put `empty` in them.

Creating and Initializing a New Instance

There is no reason to explicitly create an instance of `EmptyClass`, since the global instance `empty` is automatically created when the ScriptX runtime environment starts up. See the chapter "Global Constants and Variables" for a definition of the global instance `empty`.

Event



Class type: Core class (abstract)
Resides in: ScriptX and KMP executables
Inherits from: `RootObject`
Component: Events

`Event` is the root abstract class for all types of events. Each `Event` class serves two functions in ScriptX: it indicates actual instances of an event, and it is used as a template to register an interest in a particular event.

Events are used to describe both system and user-defined events. Instances of an `Event` subclass are always created and signaled by software, but the event can originate with either a software or a hardware condition. When an event is generated (for example, when a mouse button is pressed), an instance of the event is created and the event is delivered, through the event system, to a particular event receiver.

Events also express event interests. Every subclass of `Event` maintains a list of interests that have been registered by event receivers. If an event is used to express an interest, its `eventReceiver` instance variable indicates the function, method, or queue that wants to receive the actual instance of the event. The interest is then registered with the event class using the `addEventInterest` method.

Creating and Initializing a New Instance

Although `Event` is an abstract class, it can be instantiated and it implements an `init` method that is called by its subclasses.

init

`init` *self* ⇨ *(none)*

self `Event` object

Initializes the `Event` object *self*. Do not call `init` directly on an instance—it is automatically called by the `new` method.

Class Variables

interests

self`.interests` (read-write) `Collection`

Specifies a collection of interests posted in objects of the `Event` class *self*. You cannot get or set a value until the class has been instantiated. Normally, a script should not directly add or remove elements from this collection. Direct modification of this collection on any event class during event delivery can cause a system crash. Use `addEventInterest` and `removeEventInterest` to modify the interests collection.

Each class of `Event` determines which `Collection` subclass is used to store event interests and what techniques are used for searching this collection. Scripted subclasses are free to override `interests` to use any collection. A subclass can also override the associated methods that search this collection, to arrange event interests in any appropriate order. By default, interests are stored in an array arranged from minimum to maximum according to the `priority` instance variable. Event interests of the same

priority are inserted so that the most recently registered interests at a given priority level come first. Among the core classes, `MouseEvent` and its subclasses specialize this mechanism.

numInterests

self`.numInterests` (read-write) `Integer`

Keeps track of the number of interests currently registered for the `Event` class *self*. A script cannot get or set the value of `numInterests` for an event class until the class has been instantiated.

Normally, a script should not set the value of `numInterests`. It is available to the scripter so that developers who create new `Event` subclasses can modify the data structure used for storing event interests. If you modify the `interests` collection directly, without using `addEventInterest` and `removeEventInterest`, your changes are not registered in `numInterests`.

Note that for the subclasses `MouseUpEvent` and `MouseDownEvent`, the value of `numInterests` is always at least 1, even though no interests may be registered. This is true for internal reasons, and has no logical effect on program execution.

Note – If you add (or remove) an event interest that has previously been added to (or removed from) an interests collection, it is not added (removed) again, but the value of `numInterests` is changed and is no longer accurate.

Class Methods

broadcastDispatch

`broadcastDispatch` *self event* ⇨ `Boolean`

self	`Event` class
event	`Event` object

Do not call `broadcastDispatch` directly from a script. This method contains the rules for broadcast delivery of *event* for a given `Event` subclass. A script calls `broadcast`, an instance method on `Event`, and the `broadcast` method in turn calls `broadcastDispatch`. To override `broadcastDispatch` is to modify the mechanism for broadcast delivery of the entire subclass of `Event`.

signalDispatch

`signalDispatch` *self event rejectable* ⇨ `Boolean`

self	`Event` class
event	`Event` object
rejectable	`Boolean` object

Do not call `signalDispatch` directly from a script. This method contains the rules for signal delivery of *event* for the given `Event` subclass *self*. The *rejectable* argument indicates whether the event is being signaled synchronously (`true`) or asynchronously (`false`). A script calls `signal`, an instance method on `Event`, and the `signal` method in turn calls `signalDispatch`.

Override `signalDispatch` to modify the mechanism for signal delivery of a subclass of `Event`. For example, the class `MouseEvent` overrides `signalDispatch` so that it can traverse the hierarchy of presenters and examine interests that are stored on presenters in seeking an event interest that matches the event.

Instance Variables

Note – The following two phrases are used below to refer to the type of event:
- "Interest-only" means the instance *self* is used as an event interest
- "Event-only" means the instance *self* holds an actual event

Some instance variables in the event system are overloaded; they have a different interpretation, depending on whether an `Event` object represents an event interest or an actual event.

advertised

self`.advertised` (read-only) `Boolean`

Interest-only instance variable. Indicates whether an the event interest *self* is currently in the `interests` collection maintained by its class or in a presenter's `eventInterests` collection. A call to `addEventInterest` sets `advertised` to `true`; `removeEventInterest` sets `advertised` to `false`.

authorData

self`.authorData` (read-write) *(object)*

Interest-only instance variable. Specifies any data you want to associate with the event interest *self*. This value is sent as the first argument of a function when a function is used as an event receiver. The instance variable `authorData` can take any object as its value.

device

self`.device` (read-write) `InputDevice`

Events and event interests. For an event, `device` specifies the input device, such as a mouse or keyboard, that sent the event *self*. When an `Event` object represents an event, the value of device is set automatically by the input device that sends the event.

For an event interest, `device` can be used to specify which input device must send the event in order for the event interest *self* to be satisfied. In subclasses of `MouseEvent` and `KeyboardEvent`, the value of `device` is set by default to the main input device, a mouse or keyboard device, as appropriate. It is not necessary to explicitly set `device`, unless some other behavior is desired.

eventReceiver

self`.eventReceiver` (read-write) *(function)* or `EventQueue`

Interest-only instance variable. Specifies the event receiver, either an `EventQueue` or `Function` object, associated with the event interest *self*. Setting `eventReceiver` attaches an interest to a particular receiver. The receiver cannot receive events, however, until the interest is posted with its event class or presenter by calling `addEventInterest`.

If the event receiver is a function, it must take three arguments. The first argument will automatically receive the contents of `authorData` on the associated interest. The second and third arguments, respectively, receive the event interest and the event itself:

func authorData theInterest theEvent

You should implement the function to return `false` or `@reject` to pass the event on to the next event interest, or `true` or `@accept` to swallow the event.

Several ScriptX core classes receive and process events automatically, including `TextEdit`, `ScrollBar`, and `ActuatorController`. For an example of a method that is an event receiver, see `processEvent`, which is defined by `ScrollBar`. For a discussion of both functions and event queues as event receivers, and for sample scripts, see the "Events and Input Devices" chapter of the *ScriptX Components Guide*.

Although any global function, anonymous function, or method can be assigned to `eventReceiver`, there are differences in how different classes of functions are dispatched. For information on functions and function dispatch, see the "Object System Kernel" chapter of the *ScriptX Components Guide*.

matchedInterest

self`.matchedInterest` (read-write) `Event` or `Array`

Event-only instance variable. Specifies the event interest or interests that matched the event *self*. The return value becomes meaningful once delivery is complete. If the event is signaled, final delivery is to a single matching interest. If the event is broadcast, it is delivered to all event interests associated with the class, so `matchedInterest` returns an array of interests that received the event.

Note that `matchedInterest` is specialized by the `MouseUpEvent` class. Its setter method is meant for use with interests in `MouseUpEvent`.

priority

self`.priority` (read-write) `Integer`

Interest-only instance variable. Specifies the priority of the event interest *self*. Priority is used in storing an event interest on a class's interest list. To change the priority of an event interest, a script must explicitly remove it from the class's interest list, reset its priority, and then add it back on. Possible values range from 1 to 15, where 1 represents the highest priority and 15 the lowest.

timeStamp

self`.timeStamp` (read-write) `LongInteger`

Specifies the time at which the event *self* was sent. The time is taken from the clock stored in the global variable `theEventTimeStampClock`. (Note that this clock is created by the event system upon initialization.)

Instance Methods

accept

`accept` *self* ⇨ `NameClass`

Accepts the event *self* by sending a reply to the waiting event queue with an accept message. This method is used only if the event receiver is an event queue. Returns `@accept` if the event is accepted and `@reject` if it is rejected.

acquireRejectQueue

`acquireRejectQueue` *self* ⇨ `EventQueue`

Returns a reject queue set up to receive accept/reject replies from another thread on the event *self*. It gets the queue from the class's reject queue pool. Every call to `acquireRejectQueue` must be balanced by a call to `relinquishRejectQueue`. This method is normally not called from the scripter. A reject queue is a queue with a single interest that is created automatically when an event is signaled synchronously.

addEventInterest

`addEventInterest` *self* ⇨ `Event` object

Adds the event *self* to the `interests` class variable for the event class. Reports the exception `noEventReceiver` if the `eventReceiver` instance variable has not been set. See also `numInterests`, a class variable defined by `Event`, which is discussed on page 247.

broadcast

`broadcast` *self* ⇨ `Boolean`

Delivers the event *self* to all interested parties. Returns `true` if any party is interested and `false` if no party is interested. Mouse and keyboard events cannot be broadcast.

isSatisfiedBy

`isSatisfiedBy` *self event* ⇨ `Boolean`

self	`Event` object that represents an event interest
event	`Event` object that represents an actual event

Tests whether an event interest is satisfied by an event. Do not call `isSatisfiedBy` directly from a script. It is called by the event system as a result of calling either `signal` or `broadcast`.

By default, `isSatisfiedBy` returns `true`. The method exists to be overriden on subclasses of `Event`, so that an author can define further comparisons and conditions before an event is delivered to an actual receiver. Several subclasses of `Event` in the core classes, including `MouseEvent` and `KeyboardEvent`, specialize `isSatisfiedBy` to make such comparisons.

reject

`reject` *self* ⇨ `Boolean`

Rejects the event *self*. This will send a reply to the waiting event queue with a reject message. This method is used only if the event receiver is an event queue.

relinquishRejectQueue

`relinquishRejectQueue` *self* ⇨ `EventQueue`

Returns a reject queue that was acquired by the method `acquireRejectQueue` to the class's reject queue pool. This method is normally not called from the scripter. See the definition of `acquireRejectQueue` on page 249.

removeEventInterest

`removeEventInterest` *self* ⇨ `Event` subclass

Removes the event *self* from the `interests` class variable for the `Event` subclass. See also `numInterests`, a class variable defined by `Event`, which is discussed on page 247.

sendToQueue

`sendToQueue` *self queue* ⇨ `Boolean`

self	`Event` object
queue	`EventQueue` object

Places the event *self* in the specified *queue*. Returns `true` if the event is added to the queue successfully and `false` if *queue* is not accessible. If called from a regular thread, `sendToQueue` always returns `true`, unless the queue is inaccessible. If the queue is full, the thread will block until it can deliver the event.

Note that `sendToQueue` does not check event interests. It places the event in the queue automatically, and the receiving queue cannot reject it. `sendToQueue` only returns `false` if it is unable to write to the queue. This can happen when the queue is not writable. For example, a process that is running in another thread may have acquired a lock on the queue.

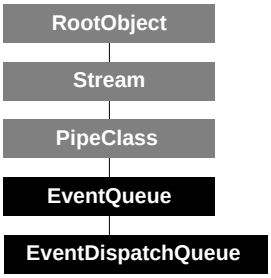
signal

`signal` *self rejectable* ⇨ `Boolean`

self	`Event` object
rejectable	`Boolean`

Delivers the event *self* to the interested party with the highest priority. If *rejectable* is `true`, this method waits until the interested party accepts or rejects the event, and the event is handled synchronously. If the event is rejected, `signal` delivers it to the next interested party. If *rejectable* is `false`, then it does not allow the original party to reject the event, and the event is handled asynchronously. Returns `true` if the event is delivered to one or more parties and `false` if there are no interested parties or if all parties refuse to handle the event.

EventDispatchQueue



Class type Concrete
Resides in: ScriptX and KMP executables
Inherits from: `EventQueue`
Component: Events

Represents a queue of events that will be dispatched sequentially. When queued events are initially dispatched, they are placed on an event dispatch queue. When the event reaches the end of such a queue, the event is delivered to any interested parties.

Events are added to and removed from the queue using methods that are defined on `Stream`. Note that since the `EventDispatchQueue` class represents a non-seekable stream, certain methods that are defined on `Stream` report an exception if called on `EventDispatchQueue`.

ScriptX creates a global instance of `EventDispatchQueue`, stored in the global variable `theUIEventDispatchQueue`, known informally as the user interface event dispatch queue. If you create a scripted subclass of `QueuedEvent`, you can choose to process events through this queue, or you can create another instance of `EventDispatchQueue` and process them separately.

Creating and Initializing a New Instance

The following script is an example of how to create a new instance of the `EventDispatchQueue` class:

```
myEDQueue := new EventDispatchQueue
```

The variable `myEDQueue` contains a new instance of `EventDispatchQueue`. The `new` method applies the `init` method defined on `EventDispatchQueue`, which takes no keyword arguments.

init

`init` *self* ⇨ *(none)*

self `EventDispatchQueue` object

Initializes the `EventDispatchQueue` object *self*. Do not call `init` directly on an instance—it is automatically called by the `new` method.

Instance Variables

Inherited from `PipeClass`:

`broken` `maxSize` `thread`
`label` `size`

The following instance variables are defined in `EventDispatchQueue`:

dispatchThread

self`.dispatchThread` (read-write) `Thread`

Specifies the thread that reads from the `EventDispatchQueue` object *self*.

Instance Methods

Inherited from `Stream`:

`cursor`	`next`	`seekFromStart`
`flush`	`plug`	`setStreamLength`
`isAtFront`	`previous`	`streamLength`
`isPastEnd`	`read`	`write`
`isReadable`	`readReady`	`writeReady`
`isSeekable`	`seekFromCursor`	
`isWritable`	`seekFromEnd`	

Inherited from `PipeClass`:

`acquireQueue`	`isWritable`	`readReady`
`breakPipe`	`pipeSize`	`relinquishQueue`
`isPastEnd`	`pipeSizeOrFail`	`write`
`isReadable`	`read`	`writeNowOrFail`
`isSeekable`	`readNowOrFail`	`writeReady`

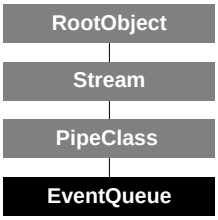
The following instance methods are defined by `EventDispatchQueue`.

processEventQueue

`processEventQueue` *self* ⇨ `Boolean`

Do not call this method from the scripter. This method reads events off the event dispatch queue and delivers them according to the rules of the `QueuedEvent` class. The event dispatch queue is the primary queue, through which queued events must pass to ensure orderly processing by the event system. In general, this method is not overridden.

EventQueue



Class type: Core class (concrete)
Resides in: ScriptX and KMP executables
Inherits from: `PipeClass`
Component: Events

The `EventQueue` class enables the use of events as a communications mechanism between threads in a multithreaded program. An `EventQueue` object represents a stream of events that match a particular interest or group of interests. It is legal for the function that reads from the queue to examine all the events in the queue and process them in any order.

Events are added to and removed from the queue using methods that are defined by `Stream`. Note that since the `EventQueue` class represents a non-seekable stream, certain methods that are defined by `Stream` report an exception if called on `EventQueue`.

Each event class holds a collection of event interests, that is used to identify events that should be placed with a particular receiver. An `EventQueue` object can be the event receiver when an event interest is added to the event class's interest list. An event receiver is specified by the `eventReceiver` instance variable defined by `Event`.

Creating and Initializing a New Instance

The following script is an example of how to create a new instance of the `EventQueue` class, attaching a thread that has been defined previously:

```
myEventQueue := new EventQueue thread:myThread
```

The variable `myEventQueue` contains an initialized instance of `EventQueue`. The thread `myThread` is the thread that will be activated when an event is placed in the queue—presumably, its function reads events that are placed in this queue. The `new` method applies the `init` method on `EventQueue`.

init

`init` *self* [`thread:`*thread*] [`undefined:`*undefined*] ⇨ *(none)*

self `EventQueue` object

Superclasses of `EventQueue` use the following keywords:

`thread:`	`Thread` object associated with the `EventQueue` object
`undefined:`	reserved for future use

Initializes the `EventQueue` object *self*, applying the arguments as follows: `thread:`is a `Thread` object associated with the `EventQueue` instance self. This thread is activated when objects are placed in the queue. Do not call `init` directly on an instance—it is automatically called by the `new` method.

If you omit an optional keyword, its default value is used. The defaults are:

```
thread:undefined
undefined:undefined
```

Instance Variables

Inherited from `PipeClass`:

`broken`	`maxSize`	`thread`
`label`	`size`	

Instance Methods

Inherited from `Stream`:

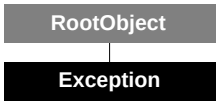
`cursor`	`next`	`seekFromStart`
`flush`	`plug`	`setStreamLength`
`isAtFront`	`previous`	`streamLength`
`isPastEnd`	`read`	`write`
`isReadable`	`readReady`	`writeReady`
`isSeekable`	`seekFromCursor`	
`isWritable`	`seekFromEnd`	

Inherited from `PipeClass`:

`acquireQueue`	`isWritable`	`readReady`
`breakPipe`	`pipeSize`	`relinquishQueue`
`isPastEnd`	`pipeSizeOrFail`	`write`
`isReadable`	`read`	`writeNowOrFail`
`isSeekable`	`readNowOrFail`	`writeReady`

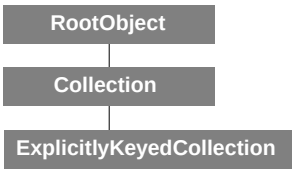
Note that many inherited methods, defined by `Stream`, are specialized by `PipeClass`. The `EventQueue` class applies the method that it inherits from `PipeClass`.

Exception



For all the `Exception` classes, see Appendix B, “Exceptions”.

ExplicitlyKeyedCollection



Class type: Core class (abstract)
Resides in: ScriptX and KMP executables
Inherits from: `Collection`
Component: Collections

`ExplicitlyKeyedCollection` is an abstract class that implements certain methods from the Collection protocol in ways that are unique to collections with explicit keys.

Class Methods

Inherited from `Collection`:

`pipe`

Instance Variables

Inherited from `Collection`:

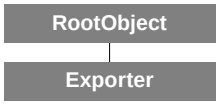
`bounded`	`maxSize`	`size`
`iteratorClass`	`minSize`	`uniformity`
`keyEqualComparator`	`mutable`	`uniformityClass`
`keyUniformity`	`mutableCopyClass`	`valueEqualComparator`
`keyUniformityClass`	`proprietored`	

Instance Methods

Inherited from `Collection`:

`add`	`forEach`	`iterate`
`addMany`	`forEachBinding`	`localEqual`
`addToContents`	`getAll`	`map`
`chooseAll`	`getAny`	`merge`
`chooseOne`	`getKeyAll`	`pipe`
`chooseOneBinding`	`getKeyOne`	`prin`
`deleteAll`	`getMany`	`removeAll`
`deleteBindingAll`	`getOne`	`removeOne`
`deleteBindingOne`	`hasBinding`	`setAll`
`deleteKeyAll`	`hasKey`	`setOne`
`deleteKeyOne`	`intersects`	`size`
`deleteOne`	`isEmpty`	
`emptyOut`	`isMember`	

Exporter



Class type: Core class (abstract)
Resides in: ScriptX and KMP executables
Inherits from: `RootObject`
Component: Import and Export

The `Exporter` class is an abstract class and is not instantiable. `ImportExportEngine` objects use subclasses of `Exporter` to export data. Each subclass of `Exporter` is an exporter conversion class that converts one specific type of internal data object to a specific external data format—for example, a bitmap `TwoDShape` object to the `@pict` format.

An exporter conversion class, or export filter, is a subclass of `Exporter` that implements an `exportToStream` method.

The current release of ScriptX supports only one exporter, the text exporter.

Creating and Initializing a New Instance

Note – When exporting data, do not create an instance of an `Exporter` subclass directly. Create an instance of `ImportExportEngine`, then call the `exportMedia` method on that instance. `ImportExportEngine` creates an instance of the proper subclass of `Exporter` automatically.

init

`init` *self* `mediaCategory:`*name* `inputMediaType:`*name*
`outputMediaType:`*name* ⇨ *(none)*

self	`Exporter` object
`mediaCategory:`	`NameClass` object
`inputMediaType:`	`NameClass` object
`outputMediaType:`	`NameClass` object

Initializes the `Exporter` object *self*, applying the keyword arguments that specify the media category, and the types of input and output media. Do not call `init` directly on an instance—it is automatically called by the `new` method.

Instance Variables

inputMediaType

self`.inputMediaType` (read-only) `NameClass`

Specifies the type of media that the exporter *self* converts—for example, `@pict`, `@aiff` or `@wave`.

mediaCategory

self`.mediaCategory` (read-only) `NameClass`

Specifies the category of media that the exporter *self* converts and exports—for example, `@sound`.

outputMediaType

self.`outputMediaType` (read-only) `NameClass`

Specifies the type of media that the exporter *self* exports—for example, `@pict`.

Instance Methods

exportToStream

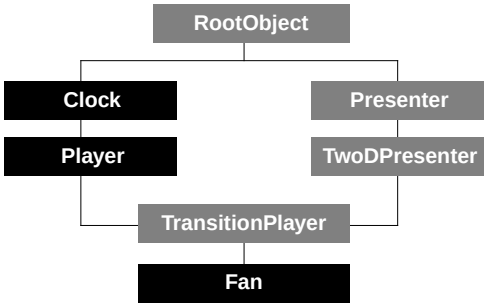
`exportToStream` *self source destination* ⇨ *destination*

self	`Exporter` object
source	Any object
destination	`Stream` object

Returns the *destination* object that `exportToStream` creates. The argument *source* is the object to export using the exporter *self*. The argument *destination* is the stream to which to export the data. If the object in *source* is unknown to the `Exporter`, this method generates the exception `unknownClass`.

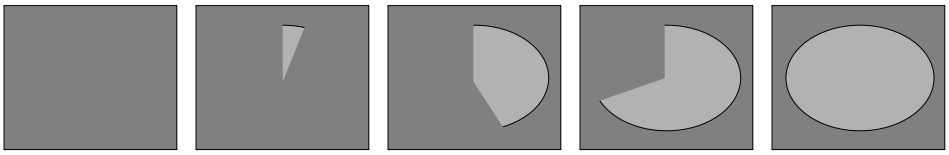
Some exporters allow for extra arguments beyond those listed above. It is then up to the individual exporter to interpret those extra arguments.

Fan



Class type: Loadable class (concrete)
Resides in: `ltrans.lib`. Works with ScriptX and KMP executables.
Inherits from: `TransitionPlayer`
Component: Transitions

The `Fan` transition player provides a visual effect that causes the target to gradually appear swept in a clockwise or counter-clockwise fashion, as shown below. The target appears when the transition is played forward, and disappears when played backward (transition's rate set to -1).



Directions: `@clockwise`, `@anticlockwise` (yes, really)

Rate: Can play forward or backward.

For a side-by-side illustrations of all transitions, see the Transitions chapter in the *ScriptX Components Guide*.

Creating and Initializing a New Instance

The following script is an example of how to create a new instance of the `Fan` class:

```
myTransition := new Fan \
                    duration:60 \
                    direction:@clockwise \
                    target:myShape
```

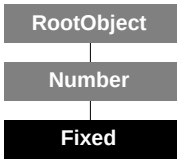
The variable `myTransition` contains the initialized transition. The transition reveals the image `myShape` in a clockwise direction and has a duration of 60 ticks.

You determine which space the transition will take effect in by adding this instance into that space. Then, when you play the transition player, `myShape` is transitioned into that space.

The `new` method uses the keywords defined in `init`.

NOTE – For the instance variables and methods, see the `BarnDoor` class.

Fixed



Class type: Core class (concrete, sealed)
Resides in: ScriptX and KMP executables
Inherits from: `Number`
Component: Numerics

The `Fixed` class represents numbers to $\pm 2^{15}$ (32,768), with 16 bits of fractional information ($1/2^{16}$).

Creating and Initializing a New Instance

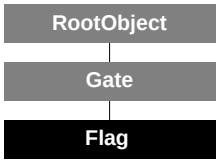
The `Fixed` class has no scripter-level `new` or `init` method. You create a `Fixed` object by coercing an object that belongs to one of the other `Number` classes.

Instance Methods

Inherited from `Number`:

`abs`	`floor`	`radToDeg`
`acos`	`frac`	`random`
`asin`	`inverse`	`rem`
`atan`	`ln`	`round`
`atan2`	`log`	`sin`
`ceiling`	`max`	`sinh`
`coerce`	`min`	`sqrt`
`cos`	`mod`	`tan`
`cosh`	`morph`	`tanh`
`degTorad`	`negate`	`trunc`
`exp`	`power`	

Flag



Class type: Core class (concrete, sealed)
Resides in: ScriptX and KMP executables
Inherits from: `Gate`
Component: Threads

The `Flag` class represents a a gate whose `state` instance variable remains `@open` over a period of time, until it is explicitly closed (its `state` becomes `@closed`). Threads can wait on a flag, and when that flag is set to `@open`, all threads that were waiting on it are made active (runnable).

A thread waits on a `Flag` object with either the `gateWait` global function or the `acquire` method. The two are equivalent, except that `acquire` is a generic function.

Creating and Initializing a New Instance

The following script creates a new instance of the `Flag` class:

```
flg := new Flag \
    label:@flag_21
```

The variable `flg` contains the initialized flag. The `new` method uses keywords defined in `init`.

init

`init` *self* [`label:`*object*] ⇨ *(none)*

self	`Flag` object
`label:`	Any object

Initializes the `Flag` object *self*. The value supplied with the keyword `label` is any object that is displayed when you print, which is useful for debugging. Do not call `init` directly on an instance—it is automatically called by the `new` method.

If you omit an optional keyword, its default value is used. The default is:

`label:undefined`

Instance Variables

Inherited from `Gate`:

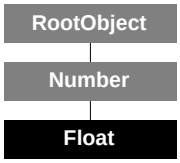
`label` `state`

Instance Methods

Inherited from `Gate`:

`acquire` `relinquish`

Float



Class type: Core class (concrete, sealed)
Resides in: ScriptX and KMP executables
Inherits from: `Number`
Component: Numerics

The `Float` class represents 64-bit IEEE 754 or larger floating-point numbers.

Creating and Initializing a New Instance

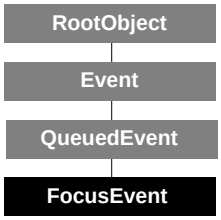
You do not need to explicitly create or initialize an instance of the `Float` class—simply use a number the size of a float in a script. Whenever the compiler encounters a number that cannot be represented as an `ImmediateFloat` object without loss of precisions, it automatically promotes the number to a `Float` object. Any floating point constant represented by seven or more decimal digits is promoted to `Float` (for example, `1.234567`). The `Float` class has no scripter-level `new` or `init` method.

Instance Methods

Inherited from `Number`:

`abs`	`floor`	`radToDeg`
`acos`	`frac`	`random`
`asin`	`inverse`	`rem`
`atan`	`ln`	`round`
`atan2`	`log`	`sin`
`ceiling`	`max`	`sinh`
`coerce`	`min`	`sqrt`
`cos`	`mod`	`tan`
`cosh`	`morph`	`tanh`
`degTorad`	`negate`	`trunc`
`exp`	`power`	

FocusEvent



Class type: Core class (concrete)
Resides in: ScriptX and KMP executables
Inherits from: `QueuedEvent`
Component: Events

The `FocusEvent` class is used to direct input from a particular input device to a given presenter. For example, it directs characters typed on a keyboard to an instance of `TextEdit`. The target of a focus event is a `TwoDPresenter` object that is designated by the `presenter` instance variable. In this example the target is the `TextEdit` instance. (Although focus events are most commonly used with `TextEdit` presenters, the concept of focus applies to any 2D presenter.)

The current version of the Kaleida Media Player supports two input devices: a keyboard and a mouse. At present, focus events direct input only from keyboard devices. The `FocusEvent` class provides a flexible mechanism for focusing input that could be extended to support other input devices.

Presenters store interests in their `eventInterests` instance variable. When the user uses the mouse to click on such a presenter that is interested in focus, the presenter informs the focus manager. The focus manager is a class that is associated with an input device, but it is hidden at the scripter level.

The focus manager maintains a private record of which presenter currently has focus. It sends a `FocusEvent` object whose `focusType` instance variable is set to `@loseFocus` to the presenter that is losing focus. It sends another `FocusEvent` object to the new presenter, with `focusType` set to `@gainFocus`.

Among the core classes, the `TextEdit` class creates and registers interests in receiving focus events that relate to keyboard devices. A `TextEdit` object is a specialized presenter that receives keyboard input. When the focus of a keyboard device changes from one presenter to another, each `TextEdit` object that is affected receives a focus event. In this way, the `FocusEvent` class is used as a notification device, telling `TextEdit` objects when to add or remove their interests in keyboard events.

Creating and Initializing a New Instance

The following script creates a new instance of the `FocusEvent` class:

```
presbyopicFocus := new FocusEvent
```

The variable `presbyopicFocus` contains an initialized focus event. The `new` method calls the `init` method, defined by `FocusEvent` and its superclass, `Event`. Although the `init` method applies no keyword arguments, it is necessary to set the `device`, `eventReceiver`, and `presenter` instance variables before registering a `FocusEvent` object as an event interest.

init

`init` *self* ⇨ *(none)*

self `FocusEvent` object

Initializes the FocusEvent object *self*. Do not call init directly on an instance—it is automatically called by the new method.

Class Variables

Inherited from Event:

interests	numInterests

Class Methods

Inherited from Event:

acquireQueueFromPool	relinquishQueueToPool
broadcastDispatch	signalDispatch

Instance Variables

Inherited from Event:

advertised	eventReceiver	timeStamp
authorData	matchedInterest	
device	priority	

The following instance variables are defined in FocusEvent:

Note – The following two phrases are used below to refer to the type of event:
- "Interest-only" means the instance *self* is used to express an event interest.
- "Event-only" means the instance *self* holds an actual user event.

focusType

self.focusType (read-only) NameClass

Event-only instance variable. Indicates the nature of the change in focus for the presenter that receives the focus event *self*. Can take on one of three possible values: @loseFocus, @gainFocus, and @restoreFocus. If this value is @loseFocus, a presenter that formerly had focus has lost its focus and should remove its interests in events that are generated by the device that sent the event. If this value is @gainFocus or @restoreFocus, then the presenter that receives the focus event should add or restore interests in events generated by that device. The value @gainFocus indicates that the presenter itself has gained focus. The value @restoreFocus indicates that another object or script, not the presenter itself, caused the presenter to gain focus.

presenter

self.presenter (read-write) TwoDPresenter

Interest-only instance variable. Specifies the presenter that will be the target of a change in focus by the focus event interest *self*.

Instance Methods

Inherited from Event:

accept	isSatisfiedBy	sendToQueue
acquireRejectQueue	reject	signal
addEventInterest	relinquishRejectQueue	
broadcast	removeEventInterest	

The following instance methods are defined in `FocusEvent`:

broadcast (Event)

`broadcast` *self* ⇨ `Exception`

Reports the `cantBroadcast` exception if called on the focus event *self*.

isSatisfiedBy (Event)

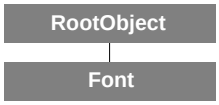
`isSatisfiedBy` *self focusEvent* ⇨ `Boolean`

self	`Event` object that represents an event interest
focusEvent	`Event` object that represents an actual event

Tests whether the event interest *self* is satisfied by the focus event. Do not call `isSatisfiedBy` directly from a script. It is called by the event system as a result of calling either `signal` or `broadcast`.

The class `FocusEvent` specializes `isSatisfiedBy` to test whether the event *focusEvent* was sent to the same presenter as the event interest *self*.

Font



Class type: Core class (abstract)
Resides in: ScriptX and KMP executables
Inherits from: `RootObject`
Component: Text and Fonts

The abstract `Font` class and its concrete subclass `PlatformFont` provide a way for other ScriptX objects (such as text presenters) to locate font data. To render fonts, ScriptX uses the native font technology of the platform on which it is run (Macintosh or Windows). ScriptX can only use outlines or bitmaps of fonts installed and available on the system.

The `Font` and `PlatformFont` classes provide access to a typeface (for example, Helvetica or New Times Roman). Variations on that typeface (Bold, Italic, Condensed) are accessed through the use of attributes on the `Text` or `TextPresenter` classes.

Creating and Initializing a New Instance

`Font` is an abstract class and cannot be generally instantiated. To create general font objects, use the `PlatformFont` class instead. However, the following expression can be used to create an instance of the default system font (Helvetica on the Macintosh, Arial on Windows):

```
myFont := new Font
```

The variable `myFont` contains a new instance of `PlatformFont`.

Class Variables

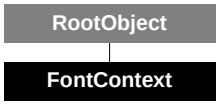
The following class variable is defined on the `Font` class:

default

self`.default` (read-only) `PlatformFont`

Specifies the default system font (an instance of `PlatformFont`).

FontContext



Class type: Scripted class (concrete)
Resides in: widgets.sxl. Works with ScriptX and KMP executables
Inherits from: RootObject
Component: User Interface

The FontContext class provides a template that can be used to determine the text style of text label in an object or set of objects in the ScriptX widget library. A FontContext object incorporates the name of the font, its size, and its leading and descent.

Creating and Initializing a New Instance

The following script creates a new instance of FontContext:

```
global myFC := new FontContext \
       fontName:"Palatino" \
       fontSize:12 \
       leading:15 \
       descent:4
```

The global variable myFC contains the initialized FontContext object. The values supplied for the keywords fontName, fontSize, leading, and descent determine the appearance of widgets whose labels are rendered using myFC as a font context.

The new method uses keywords defined in init.

init

init *self* [fontName:*string*] [fontSize:*number*]
[leading:*number*] [descent:*number*] ⇨ *(none)*

self	FontContext object
fontName:	String object
fontSize:	Number object
leading:	Number object
descent:	Number object

Initializes the FontContext object *self*, applying the keyword arguments to set the initial values of instance variables of the same name.

If you omit an optional keyword, its default value is used. The defaults are:

fontName:(applies the system default font)
fontSize:(varies by platform)
leading:(varies by platform)
descent:(varies by platform)

Class Variables

defaultDescent

self.defaultDescent (read-write) Number

Specifies the default descent for instances of FontContext. When the FontContext class is initialized, it queries the system to determine the system font, and sets the default descent accordingly.

defaultFont

self.defaultFont (read-write) StringConstant

Specifies the default font name for instances of FontContext. When the FontContext class is initialized, it queries the system to determine the system font, and sets the default font accordingly.

defaultLeading

self.defaultLeading (read-write) Number

Specifies the default leading for instances of FontContext. When the FontContext class is initialized, it queries the system to determine the system font, and sets the default leading accordingly.

defaultSize

self.defaultSize (read-write) Number

Specifies the default font size for instances of FontContext. When the FontContext class is initialized, it queries the system to determine the system font, and sets the default size accordingly.

Instance Variables

descent

self.descent (read-write) Number

Specifies the descent that is used in rendering text in labels that use the FontContext object *self*.

font

self.font (read-write) PlatformFont

Specifies the PlatformFont object that is used to render text in labels that use the FontContext object *self*.

fontName

self.fontName (read-write) StringConstant

Specifies the name of the font that is used to render text in labels that use the FontContext object *self*.

fontSize

self.fontSize (read-write) Number

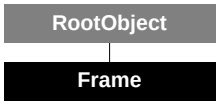
Specifies the size of the font that is used to render text in labels that use the FontContext object *self*.

leading

self.leading (read-write) Number

Specifies the leading that is used in rendering text in labels that use the FontContext object *self*.

Frame



Class type: Scripted class (concrete)
Resides in: `widgets.sxl`. Works with ScriptX and KMP executables
Inherits from: `RootObject`
Component: User Interface

The `Frame` class provides a template that can be used to determine the appearance of an object or set of objects in the ScriptX widget library. A `Frame` object incorporates information about how a widget should be drawn to a surface—what its boundary is, and what brushes are used to render it to a surface. Aspects of this appearance that are determined by the widget's frame include the following:

- The size of the widget, which is set via the `boundary` keyword at initialization.
- The color of its text label, boundary, drop shadow, and fill, and its appearance when it is disabled, which are set via the `scheme` instance variable.

Creating and Initializing a New Instance

The following script creates a new instance of `Frame`:

```
global myFrame := new Frame \
       scheme:(new ColorScheme brushIndexArray:#(1,4,7,5)) \
       boundary:(new Rect x2:128 y2:32)
```

The global variable `myFrame` contains the initialized `Frame` object. The `ColorScheme` object that is specified determines the colors of the text label, fill, boundary, and drop shadow when this `Frame` object is use to draw a widget. The `Rect` object that is supplied with the `boundary` keyword is supplied as an argument to `setBoundary`, an instance method defined by `Frame`. It determines the size and layout of an associated widget's box.

The `new` method uses keywords defined in `init`.

init

`init` *self* `scheme:`*colorScheme* `boundary:`*rect* ⇨ *(none)*

self	`Frame` object
`scheme:`	`ColorScheme` object
`boundary:`	`Rect` object

Initializes the `Frame` object *self*, applying the keyword `scheme` to set the value of the instance variable `scheme`, and applying the keyword `boundary` to call `setBoundary`, which determines the values of the instance variables `topLeftPath` and `botRightPath`. All keywords are required.

Instance Variables

botRightPath

self.botRightPath (read-write) Path

Specifies the Path object that is used in drawing the bottom and right boundaries of a widget object to a surface. The Frame object *self* is incorporated into widgets as a property, and determines the appearance of the widget's box.

scheme

self.scheme (read-write) ColorScheme

Specifies a ColorScheme object that determines which brushes are used in drawing a widget object to a surface. The Frame object *self* is incorporated into widgets as a property, and determines the appearance of the widget's box.

topLeftPath

self.topLeftPath (read-write) Path

Specifies the Path object that is used in drawing the top and left boundaries of a widget object to a surface. The Frame object *self* is incorporated into widgets as a property, and determines the appearance of the widget's box.

Instance Methods

drawLoweredFrame

drawLoweredFrame *self surface clip transform* ⇨ Surface

self	Frame object
surface	Surface object, an instance of BitmapSurface or PrinterSurface
clip	Stencil object
transform	TwoDMatrix object

Draws the paths that create the drop-shadow of a widget object that is drawn using the Frame object *self*. This path is drawn to the given *surface*, using the given *clip* as a clipping stencil, with a position and orientation determined by the *transform* matrix.

drawRaisedFrame

drawRaisedFrame *self surface clip transform* ⇨ Surface

self	Frame object
surface	Surface object
clip	Stencil object
transform	TwoDMatrix object

Draws the paths that create the raised surface of a widget object that is drawn using the Frame object *self*. This path is drawn to the given *surface*, using the given *clip* as a clipping stencil, with a position and orientation determined by the *transform* matrix.

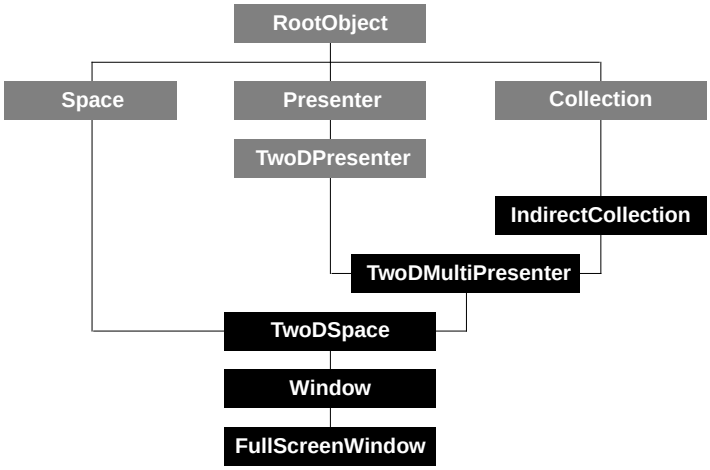
setBoundary

setBoundary *self boundary* ⇨ Path

self	Frame object
boundary	Rect object

Sets the values of the instance variables topLeftPath and botRightPath for the widget frame *self*, incorporating a drop shadow.

FullScreenWindow



Class type: Core class (concrete)
Resides in: ScriptX and KMP executables
Inherits from: `Window`
Component: Spaces and Presenters

The `FullScreenWindow` class represents a non-modal visible region that fills the screen, from edge to edge. Technically speaking, an instance of `FullScreenWindow` is actually a full-screen display surface which has an active region (window) that can be full-screen or smaller. The active region is the window, which is a clipping region—objects outside this active region are not visible. The region outside of the window. to the edge of the screen, is the border. The size of the active region is determined by the `boundary` instance variable. The active region is refreshed every tick of the clock; the border region is drawn once with the brush specified by `borderFill` and, for efficiency, is not refreshed.

Any kind of window can appear in front of a full-screen window. See the `Window` class for further details about windows in general.

For a side-by-side description of all window classes, see the "Spaces and Presenters" chapter in the *ScriptX Components Guide*.

Creating and Initializing a New Instance

The following script is an example of how to create a new instance of the `FullScreenWindow` class:

```
myFullScreen := new FullScreenWindow \
                title:myTitle
```

The variable `myFullScreen` contains an initialized full-screen window. The window belongs to the title container `myTitle`, and fills the screen from edge to edge, with no title bar showing. The `new` method uses keywords defined by the `init` method.

For a completely full-screen window, you can hide the menu bar by calling:

```
hide theTitleContainer.systemMenuBar
```

init

`init` *self* [`type:`*name*] [`title:`*titleContainer*] [`centered:`*boolean*] [`name:`*string*]
[`compositor:`*twoDCompositor*] [`boundary:`*stencil*] [`fill:`*brush*]
[`borderFill:`*brush*] [`scale:`*integer*] [`stroke:`*brush*] [`target:`*object*]
[`targetCollection:`*sequence*] ⇨ *(none)*

self	`FullScreenWindow` object
`borderfill:`	`Brush` object

Superclass `Window` defines the following keywords:

`type:`	Ignored by `FullScreenWindow`
`title:`	`TitleContainer` object
`centered:`	`Boolean` object
`name:`	Ignored by `FullScreenWindow`
`compositor:`	`TwoDCompositor` object

Superclasses of `Window` use the following keywords:

`boundary:`	`Rect` object representing the window perimeter
`fill:`	`Brush` object to fill the background
`scale:`	`Integer` object
`stroke:`	Ignored by `FullScreenWindow`
`target:`	Ignored by `FullScreenWindow`
`targetCollection:`	`Sequence` object
`stationary:`	`Boolean` object

This method is inherited from `Window` with only a few changes to keyword arguments: `borderFill` (not present in `Window`) is applied to the instance variable of the same name, and `name` has no visible effect, since a full-screen window has no title bar. The `centered` keyword makes a difference only if the boundary is smaller than the screen—it centers the boundary on the screen. This is noticeable if you set `borderFill` to `blackBrush`, for example. The boundary keyword has a different default behavior—it is set to the dimensions of the screen.

If you omit an optional keyword, its default value is used. The defaults are:

```
borderFill:undefined
boundary:(set to the dimensions of the screen)
stationary:false
```

Refer to the `Window` class for more details about other keywords.

Class Methods

Inherited from `Collection`:

`pipe`

Instance Variables

Inherited from `Collection`:

`bounded`	`maxSize`	`size`
`iteratorClass`	`minSize`	`uniformity`
`keyEqualComparator`	`mutable`	`uniformityClass`
`keyUniformity`	`mutableCopyClass`	`valueEqualComparator`
`keyUniformityClass`	`proprietored`	

Inherited from `IndirectCollection`:

`targetCollection`

Inherited from `Presenter`:

`presentedBy`	`subPresenters`	`target`

Inherited from TwoDPresenter:

bBox	height	transform
boundary	IsImplicitlyDirect	width
clock	isTransparent	window
compositor	isVisible	x
direct	needsTickle	y
eventInterests	position	z
globalBoundary	stationary	
globalTransform	target	

Inherited from TwoDMultiPresenter:

clock	fill	stroke

Inherited from Space:

clock	protocols	tickleList
controllers		

Inherited from TwoDSpace:

protocols

Inherited from Window:

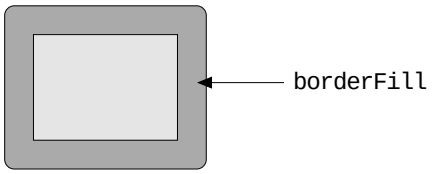
colormap	hasUserFocus	title
displaySurface	name	
fill	systemMenuBar	

The following instance variables are defined in FullScreenWindow:

borderFill

self.borderFill (read-write) Brush

If you specify the boundary of the full-screen window *self* to be smaller than the full screen, only that smaller area is active—the borderFill instance variable specifies the brush to fill the region outside this active area, as shown below. The borderFill brush is drawn directly to the border, bypassing the compositor. This region extends to the edge of the desktop. If borderFill is undefined, the border of the window is not redrawn.



displaySurface (Window)

self.displaySurface (read-only) DisplaySurface

Specifies the display surface onto which the full screen window *self* is displayed. This display surface is created automatically when the window is instantiated. For the more general Window class, the dimensions of the window's own boundary rect coincide with those of its display surface. In contrast with other window classes, a FullScreenWindow object is associated with a display surface whose boundary is a Rect object that has the same dimensions as the screen itself.

type (Window)

self.type (read-only) NameClass

Returns @normal, always.

Instance Methods

Inherited from Collection:

add	forEach	iterate
addMany	forEachBinding	localEqual
addToContents	getAll	map
chooseAll	getAny	merge
chooseOne	getKeyAll	pipe
chooseOneBinding	getKeyOne	prin
deleteAll	getMany	removeAll
deleteBindingAll	getOne	removeOne
deleteBindingOne	hasBinding	setAll
deleteKeyAll	hasKey	setOne
deleteKeyOne	intersects	size
deleteOne	isEmpty	
emptyOut	isMember	

Inherited from IndirectCollection:

isAppropriateObject	objectAdded	objectRemoved

Inherited from TwoDPresenter:

adjustClockMaster	inside	show
createInterestList	localToSurface	surfaceToLocal
draw	notifyChanged	tickle
getBoundaryInParent	recalcRegion	
hide	refresh	

Inherited from TwoDMultiPresenter:

draw	findFirstInStencil	moveToBack
findAllAtPoint	isAppropriateObject	moveToFront
findAllInStencil	moveBackward	objectAdded
findFirstAtPoint	moveForward	objectRemoved

Inherited from Space:

isAppropriateObject	objectAdded	objectRemoved

Inherited from Window:

bringToFront	cut	sendToBack
clear	hide	show
copy	paste	

Since any instance of Window is an indirect collection, you can also use any methods defined in the class specified by targetCollection. The target collection is typically an instance of Array, which inherits from Sequence, so the following instance methods are redirected to a full screen window.

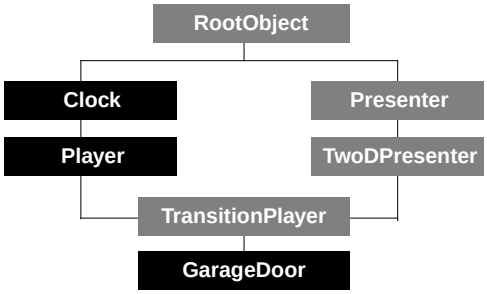
Accessible from LinearCollection:

chooseOneBackwards	findRange	getNthKey
chooseOrdOne	forEachBackwards	getOrdOne
deleteFirst	getFirst	getRange
deleteLast	getLast	localEqual
deleteNth	getMiddle	localLT
deleteRange	getNth	pop

Accessible from Sequence:

addFifth	moveBackward	setFourth
addFirst	moveForward	setLast
addFourth	moveToBack	setNth
addNth	moveToFront	setSecond
addSecond	prepend	setThird
addThird	prependNew	sort
append	setFifth	
appendNew	setFirst	

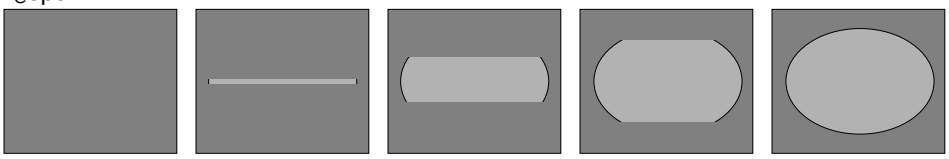
GarageDoor



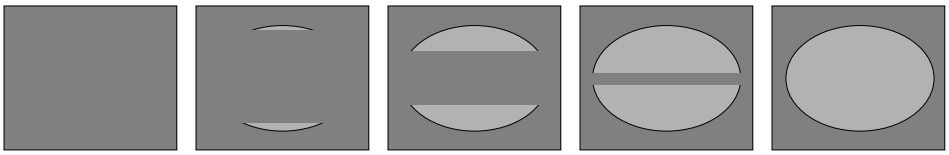
Class type: Loadable class (concrete)
Resides in: `ltrans.lib`. Works with ScriptX and KMP executables.
Inherits from: `TransitionPlayer`
Component: Transitions

The `GarageDoor` transition player provides a visual effect that causes the target to gradually appear horizontally starting either from the center (`@open`) or from the top (`@close`) as shown below. The target appears when the transition is played forward, and disappears when played backward (transition's rate set to -1).

`@open`



`@close`



Directions: `@open`, `@close`

Rate: Can play forward or backward.

For a side-by-side illustrations of all transitions, see the Transitions chapter in the *ScriptX Components Guide*.

Creating and Initializing a New Instance

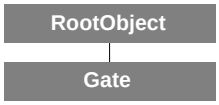
The following is an example of how to create a new instance of the `GarageDoor` class:

```
myTransition := new GarageDoor \
                duration:60 \
                direction:@open \
                target:myShape
```

The variable `myTransition` contains the initialized transition. The transition reveals the image `myShape` horizontal from the center first and has a duration of 60 ticks. You determine which space the transition will take effect in by adding this instance into that space. Then, when you play the transition, `myShape` is transitioned into that space.

NOTE – For the instance variables and methods, see the `BarnDoor` class.

Gate



Class type: Core class (abstract, sealed)
Resides in: ScriptX and KMP executables
Inherits from: `RootObject`
Component: Threads

The `Gate` class represents any obstacle to the execution of a thread. When a thread acquires a gate, the gate's `state` instance variable is set to `@closed` to block the thread from executing. The thread does not continue execution until the gate's `state` instance variable is set to `@open` (the gate is relinquished). The three concrete subclasses of `Gate` are `Condition`, `Flag`, and `Lock`.

The `Gate` class hierarchy is sealed, meaning that gates cannot be subclassed or specialized. However, gates are often attached to other ScriptX objects, allowing those objects to behave much like gates. For example, you could attach read and write locks to any objects in a database implementation, to create a database that is thread safe.

Instance Variables

label

self`.label` (read-write) *(object)*

Specifies a label for the gate *self* that is displayed when you print a representation of the the gate to a stream, such as the console stream. Any object can be used as a label, but the most useful is generally a string constant. Since gates are used internally by other classes, such as pipes and event queues, this label is useful for debugging.

state

self`.state` (read-only) `NameClass`

Indicates the current state of the gate *self*, either `@open` or `@closed`. If it is `@open`, then threads do not have to pause to acquire the gate. If it is `@closed`, then threads will block if they try to acquire the gate.

Instance Methods

acquire

`acquire` *self* ⇨ *(none)*

Acquires the gate *self*. This directly translates into `gateWait` *self*. A thread that calls `acquire` on the gate blocks until the state of the gate is `@open`.

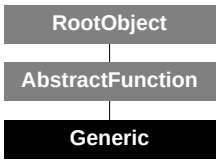
This generic function exists to allow for the definition of classes that behave like `Gate` objects. This can be done by redefining `acquire` to mean `acquire` *x* where *x* is a gate that the class knows about. For an example of a scripted class that adds a gate as an instance method and defines an `acquire` method, see the discussion of collections and threads in the "Collections" chapter of the *ScriptX Components Guide*.

relinquish

`relinquish` *self* ⇨ *(none)*

Relinquishes the gate *self*. This translates directly into the global function `openGate` *self*. (See the discussion of `acquire` for more information.)

Generic



Class type: Core class (concrete, sealed)
Resides in: ScriptX and KMP executables
Inherits from: `AbstractFunction`
Component: Object System Kernel

A `Generic` object represents a generic function. ScriptX methods are invoked indirectly through the use of generic functions, allowing for the separation of interface and implementation. Generic functions allow a single generic name and argument list to refer to multiple implementations of the given generic function that may exist for different classes and objects.

Generic functions are called identically to global functions, with the restriction that a generic must always take the target object as its first positional argument.

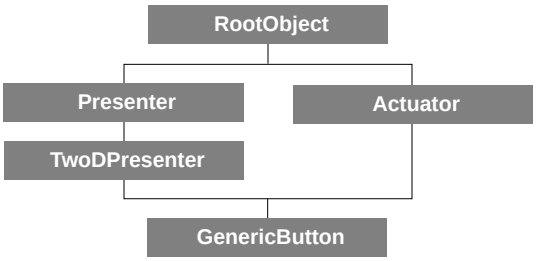
For more information on ScriptX function dispatch, see the "Object System Kernel" chapter of the *ScriptX Components Guide*.

Creating and Initializing a New Instance

You do not define an instance of `Generic` directly. An instance of `Generic` is created automatically when a method definition expression is compiled, and no existing generic has a name binding in the current module for that method.

It is possible to create an instance of `Generic` in C, using the Extending ScriptX API. For information on the extending ScriptX, see the *ScriptX Developer's Guide*.

GenericButton



Class type: Scripted class (abstract)
Resides in: `widgets.sxl`. Works with ScriptX and KMP executables.
Inherits from: `TwoDPresenter` and `Actuator`
Component: User Interface

The `GenericButton` class is a user interface Widget Kit class that provides characteristics that are inherited by all other Widget Kit button classes.

The `GenericButton` class provides the basic functionality for presenting and controlling simple buttons as elements of the user interface. (If you need a button that has different user interaction, see the `PushButton` class.) An actuator controller associated with the button receives and processes mouse events. When generic buttons are pressed, released, or activated, they can invoke an action, a function that is defined by a script.

Creating and Initializing a New Instance

Since `GenericButton` is an abstract class, do not create an instance of `GenericButton`, but rather subclass it. Do not call the `init` method on `GenericButton` directly but use `nextMethod` to call `init` from any subclass of `GenericButton` that overrides `init`, to properly initialize instances of the subclass.

init

`init` *self* [`enabled:true`] [`activateAction:`*function*] ⇨ *(none)*

self	`GenericButton` object
`enabled:`	`Boolean` object
`activateAction:`	function

Initializes the `GenericButton` object *self*, applying the values supplied with the keywords to the instance variables of the same name.

If you omit one of the keyword arguments, the following defaults are used:

```
enabled:true
activateAction:undefined
```

Instance Variables

Inherited from `Actuator`:

`enabled`	`pressed`	`toggledOn`
`menu`		

Inherited from `Presenter`:

`presentedBy`	`subPresenters`	`target`

Inherited from `TwoDPresenter`:

`bBox`	`globalRegion`	`transform`

boundary	globalTransform	width
changed	height	window
clock	imageChanged	x
compositor	isVisible	y
direct	position	z
eventInterests	stationary	
globalBoundary	target	

The following instance variables are defined in GenericButton:

activateAction

self.activateAction (read-write) *(function)*

Specifies the function that is called when the generic button *self* is activated. Initially this instance variable is undefined. You can write a function to perform any action. This function has two arguments:

```
funcName authorData self
```

authorData	data in the authorData instance variable
self	the GenericButton object *self* to which the action is attached

Although any global function, anonymous function, or method can be assigned to activateAction, there are differences in how different classes of functions are dispatched. For information on functions and function dispatch, see the "Object System Kernel" chapter of the *ScriptX Components Guide*.

authorData

self.authorData (read-write) *(object)*

Used as an argument for the function that is specified by activateAction, pressAction, and releaseAction. This instance variable can be any object.

pressAction

self.pressAction (read-write) *(function)*

Specifies the function that is called when the generic button *self* is being pressed. The function has two arguments, as shown above in activateAction.

Although any global function, anonymous function, or method can be assigned to pressAction, there are differences in how different classes of functions are dispatched. For information on functions and function dispatch, see the "Object System Kernel" chapter of the *ScriptX Components Guide*.

releaseAction

self.releaseAction (read-write) *(function)*

Specifies the function that is called when the generic button *self* has been pressed and is released without being activated. Typically, this happens when the user has moved the mouse pointer away from the button before releasing it. The function has two arguments, as shown above in activateAction.

Although any global function, anonymous function, or method can be assigned to releaseAction, there are differences in how different classes of functions are dispatched. For information on functions and function dispatch, see the "Object System Kernel" chapter of the *ScriptX Components Guide*.

Instance Methods

Inherited from `Actuator`:

`activate`	`press`	`toggleOff`
`multiActivate`	`release`	`toggleOn`

Inherited from `TwoDPresenter`:

`adjustClockMaster`	`inside`	`show`
`createInterestList`	`localToSurface`	`surfaceToLocal`
`draw`	`notifyChanged`	`tickle`
`getBoundaryInParent`	`recalcRegion`	
`hide`	`refresh`	

The following instance methods are defined in `GenericButton`:

activate (Actuator)

`activate` *self* ⇨*self*

Tells the generic button *self* that it has been activated, specializing the `activate` method defined by `Actuator`. If the `GenericButton` object *self* is enabled, `activate` calls the function specified by the `activateAction` instance variable. The two arguments to that function are `authorData` and *self*.

multiActivate (Actuator)

`multiActivate` *self numberOfClicks* ⇨ *self*

self	`GenericButton` object
numberOfClicks	`Integer` object indicating number of clicks

If the `GenericButton` object *self* is enabled, `multiActivate` calls `activate` and then calls the `multiActivate` method defined by `Actuator`. The `GenericButton` class does not define separate actions to take if the user clicks the generic button multiple times, but a subclass of `GenericButton` could specialize `multiActivate` to take different actions for different *numberOfClicks* mouse clicks that the button receives.

press (Actuator)

`press` *self* ⇨*self*

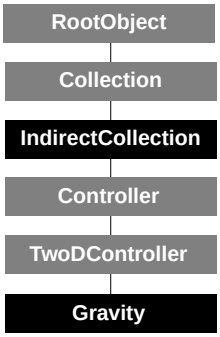
Tells the generic button *self* that it has been pressed, specializing the `press` method defined by `Actuator`. If the `GenericButton` object *self* is enabled, `press` calls the function specified by the `pressAction` instance variable. The two arguments to that function are `authorData` and *self*.

release (Actuator)

`release` *self* ⇨*self*

Tells the generic button *self* that it has been released, specializing the `release` method defined by `Actuator`. If the `GenericButton` object *self* is enabled, `release` calls the function specified by the `releaseAction` instance variable. The two arguments to that function are `authorData` and *self*.

Gravity



Class type: Core class (concrete)
Resides in: ScriptX and KMP executables
Inherits from: `TwoDController`
Component: Controllers

The `Gravity` class is a controller that, when used with the `Movement` controller, causes one or more `Projectile` objects to accelerate in a specified direction. You can specify the acceleration and direction of gravity.

For each projectile in the gravity controller's space, the `tickle` method incrementally modifies the velocity of each projectile. Note that the gravity controller doesn't actually move the projectile—it only changes the value of its `velocity` instance variable. The `Movement` controller moves the projectile according to its current value of velocity. See the `Projectile` class for more details.

A `Gravity` object is a collection of the `Projectile` objects it controls. These projectiles must also be in the space that the controller is controlling. Projectiles are either automatically or manually added to the gravity controller, according to the `wholeSpace` instance variable. If `wholeSpace` is `false`, you can use the methods defined by `Collection` to add and remove objects from the gravity controller. To ensure that only projectiles are added to a gravity controller, the `protocols` instance variable is set to the `Projectile` class. See the `Controller` class for descriptions of `wholeSpace`, `protocols`, and other general properties of controllers.

Creating and Initializing a New Instance

The following script is an example of how to create a new instance of the `Gravity` class, after first creating the space it will control:

```
mySpace := new TwoDSpace boundary:(new Rect x2:200 y2:200)
myGravity := new Gravity space:mySpace wholeSpace:true
```

The variable `myGravity` contains the initialized gravity controller. This controller is set to control all model objects in `mySpace`. The new method uses the keywords defined in `init`.

init

`init` *self* [`space:`*space*] [`wholeSpace:`*boolean*] [`enabled:`*boolean*]
[`targetCollection:`*sequence*] ⇨ *(none)*

self `Gravity` object

The superclass `Controller` uses the following keywords:

`space:`	`Space` object
`wholeSpace:`	`Boolean` object
`enabled:`	`Boolean` object

The superclass `TwoDController` uses the following keyword:

`targetCollection:`	`Sequence` object (use with caution)

Initializes the `Gravity` object *self*, applying the keyword arguments to instance variables of the same name. Use discretion in changing the target collection; for more information, see the definition of the `TwoDController` class. Do not call `init` directly on an instance—it is automatically called by the `new` method.

If you omit an optional keyword, its default value is used. The defaults are:

```
space:undefined
wholeSpace:false
enabled:true
targetCollection:(new Array initialSize:4 growable:true)
```

Class Methods

Inherited from `Collection`:

`pipe`

Instance Variables

Inherited from `Collection`:

`bounded`	`maxSize`	`size`
`iteratorClass`	`minSize`	`uniformity`
`keyEqualComparator`	`mutable`	`uniformityClass`
`keyUniformity`	`mutableCopyClass`	`valueEqualComparator`
`keyUniformityClass`	`proprietored`	

Inherited from `IndirectCollection`:

`targetCollection`

Inherited from `Controller`:

`enabled`	`space`	`wholeSpace`
`protocols`		

The following instance variables are defined in `Gravity`:

acceleration

self`.acceleration` (read-write) `Point`

Specifies the rate at which a projectile would be accelerated by the gravity *self*. The acceleration has both x and y components, specified as a `Point` object.

protocols (Controller)

self`.protocols` (read-write) `Array`

This instance variable initially contains the class `Projectile` for the interpolator *self*. This means that any object added to a `Gravity` controller must have `Projectile` as one of its superclasses. See the `Controller` class for further description about this instance variable.

Instance Methods

Inherited from `Collection`:

`add`	`forEach`	`iterate`
`addMany`	`forEachBinding`	`localEqual`
`addToContents`	`getAll`	`map`
`chooseAll`	`getAny`	`merge`
`chooseOne`	`getKeyAll`	`pipe`
`chooseOneBinding`	`getKeyOne`	`prin`
`deleteAll`	`getMany`	`removeAll`
`deleteBindingAll`	`getOne`	`removeOne`
`deleteBindingOne`	`hasBinding`	`setAll`
`deleteKeyAll`	`hasKey`	`setOne`
`deleteKeyOne`	`intersects`	`size`
`deleteOne`	`isEmpty`	
`emptyOut`	`isMember`	

Inherited from `IndirectCollection`:

`isAppropriateObject`	`objectAdded`	`objectRemoved`

Inherited from `Controller`:

`isAppropriateObject`	`tickle`

Since a `Gravity` controller is an indirect collection, you can also use any methods defined in the class specified by `targetCollection`. The target collection is typically an instance of `Array`, which inherits from `Sequence`, so the following instance methods are redirected to this controller.

Accessible from `LinearCollection`:

`chooseOneBackwards`	`findRange`	`getNthKey`
`chooseOrdOne`	`forEachBackwards`	`getOrdOne`
`deleteFirst`	`getFirst`	`getRange`
`deleteLast`	`getLast`	`localEqual`
`deleteNth`	`getMiddle`	`localLT`
`deleteRange`	`getNth`	`pop`

Accessible from `Sequence`:

`addFifth`	`moveBackward`	`setFourth`
`addFirst`	`moveForward`	`setLast`
`addFourth`	`moveToBack`	`setNth`
`addNth`	`moveToFront`	`setSecond`
`addSecond`	`prepend`	`setThird`
`addThird`	`prependNew`	`sort`
`append`	`setFifth`	
`appendNew`	`setFirst`	

The following instance methods are defined in `Gravity`:

tickle (Controller)

`tickle` *self clock* ⇨*self*

self	`Gravity` class
clock	`Clock` object of the space being controlled

At each tick of the specified *clock*, this method incrementally modifies the velocity of each projectile controlled by the gravity controller *self*. This method modifies the velocity in the following way (where `myProj` is the projectile):

```
myProj.velocity.x :=
          myProj.velocity.x +
```

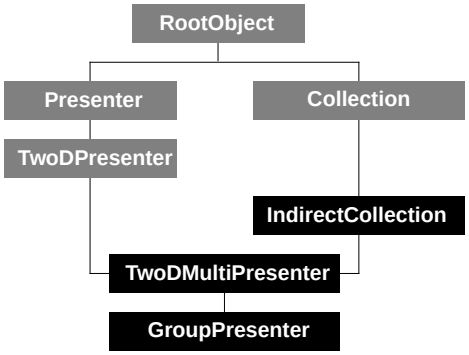
```
            (myProj.acceleration.x * (self.space.clock.scale/100))

myProj.velocity.y :=
            myProj.velocity.y +
            (myProj.acceleration.y * (self.space.clock.scale/100))
```

A callback calls this method on the gravity *self*, supplying the space's clock as the value for *clock*. The callback calls this method once every tick of the space's clock.

For further details, refer to the section "The Ticklish Protocol" in the chapter "Controllers" in the *ScriptX Components Guide*.

GroupPresenter



Class type: Core class (concrete)
Resides in: ScriptX and KMP executables
Inherits from: `TwoDMultiPresenter`
Component: Spaces and Presenters

The `GroupPresenter` class represents a group of presenters. It is the same as `TwoDMultiPresenter` but with two differences:

- The boundary of the group presenter is the union of its subpresenters—the group presenter does not clip its subpresenters. That is, the group presenter's boundary grows to include objects that are added to it. A `TwoDMultiPresenter` instance, however, has a boundary that is fixed and clips its subpresenters.
- The boundary of the group presenter is a region with the irregular shape made up by its subpresenters. The boundary is not necessarily a rectangle, as it is with `TwoDMultiPresenter`.

The stroke of a group presenter outlines the region formed by its subpresenters, as shown in the following illustration. The fill of a group presenter would not be apparent unless any of its subpresenters is invisible. When new members are added to a group presenter, it sorts its subpresenters in their proper z-order, the same as `TwoDMultiPresenter` .

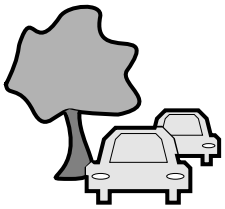


Figure 5: An example of GroupPresenter

In the `new` and `init` methods, the `target` and `boundary` keywords are ignored. The value for `boundary` is discarded. However, the value for `target` is saved in the `target` instance variable and is available for use in any subclass you might create, as a target for a presenter. For example, you might save a series of numbers to the target, and use them to determine the heights of bars in a bar chart.

Note – Besides `init`, a `GroupPresenter` has no additional variables or methods beyond those it inherits. See `TwoDMultiPresenter` for its functional interface.

Creating and Initializing a New Instance

The following script is an example of how to create a new instance of the `GroupPresenter` class:

```
myGroup := new GroupPresenter
```

The variable `myGroup` contains the initialized group presenter. The boundary of this instance enlarges or reduces to the union of the presenters it contains. The `new` method uses the keywords defined in `init`.

For performance reasons, when creating an instance of `GroupPresenter` (actually, any instance of `TwoDMultiPresenter` or its subclasses), you should *not* specify the `targetCollection` keyword, so it can create its default collection. Such presenters require collections that can be traversed quickly when drawing and handing events. Performance could suffer if you change the value of `targetcollection` to something other than the default.

init

`init` *self* [`fill:`*brush*] [`stroke:`*brush*] [`target:`*object*]
[`boundary:`*region*] [`targetCollection:`*sequence*] ⇨ *(none)*

self	`GroupPresenter` object
`fill:`	`Brush` object
`stroke:`	`Brush` object

Superclasses of `GroupPresenter` use the following keywords:

`target:`	Any object (ignored by `GroupPresenter`)
`boundary:`	`Region` object (ignored by `GroupPresenter`)
`targetCollection:`	`Sequence` object (use carefully)
`stationary:`	`Boolean` object

Initializes the `GroupPresenter` object *self*, applying the values supplied with the keywords to instance variables of the same name. The `target` keyword is ignored by `GroupPresenter`, but you are free to use it in a subclass. The `targetCollection` keyword specifies the kind of collection to create. Do not call `init` directly on an instance—it is automatically called by the `new` method.

If you omit an optional keyword, its default is used. The defaults are:

```
target:undefined
fill:undefined
stroke:undefined
boundary:(new Region)
targetCollection:(new Array initialSize:14 growable:true)
stationary:false
```

Class Methods

Inherited from `Collection`:

`pipe`

Instance Variables

Inherited from `Collection`:

`bounded`	`maxSize`	`size`
`iteratorClass`	`minSize`	`uniformity`
`keyEqualComparator`	`mutable`	`uniformityClass`

keyUniformity	mutableCopyClass	valueEqualComparator
keyUniformityClass	proprietored	

Inherited from IndirectCollection:

targetCollection		

Inherited from Presenter:

presentedBy	subPresenters	target

Inherited from TwoDPresenter:

bBox	height	transform
boundary	IsImplicitlyDirect	width
clock	isTransparent	window
compositor	isVisible	x
direct	needsTickle	y
eventInterests	position	z
globalBoundary	stationary	
globalTransform	target	

Inherited from TwoDMultiPresenter:

clock	fill	stroke

Instance Methods

Inherited from Collection:

add	forEach	iterate
addMany	forEachBinding	localEqual
addToContents	getAll	map
chooseAll	getAny	merge
chooseOne	getKeyAll	pipe
chooseOneBinding	getKeyOne	prin
deleteAll	getMany	removeAll
deleteBindingAll	getOne	removeOne
deleteBindingOne	hasBinding	setAll
deleteKeyAll	hasKey	setOne
deleteKeyOne	intersects	size
deleteOne	isEmpty	
emptyOut	isMember	

Inherited from IndirectCollection:

isAppropriateObject	objectAdded	objectRemoved

Inherited from TwoDPresenter:

adjustClockMaster	inside	show
createInterestList	localToSurface	surfaceToLocal
draw	notifyChanged	tickle
getBoundaryInParent	recalcRegion	
hide	refresh	

Inherited from TwoDMultiPresenter:

draw	findFirstInStencil	moveToBack
findAllAtPoint	isAppropriateObject	moveToFront
findAllInStencil	moveBackward	objectAdded
findFirstAtPoint	moveForward	objectRemoved

Since a GroupPresenter object is an indirect collection, you can also use any methods defined in the class specified by targetCollection. The target collection is typically an instance of Array, which inherits from Sequence, so the following instance methods are redirected to a group presenter.

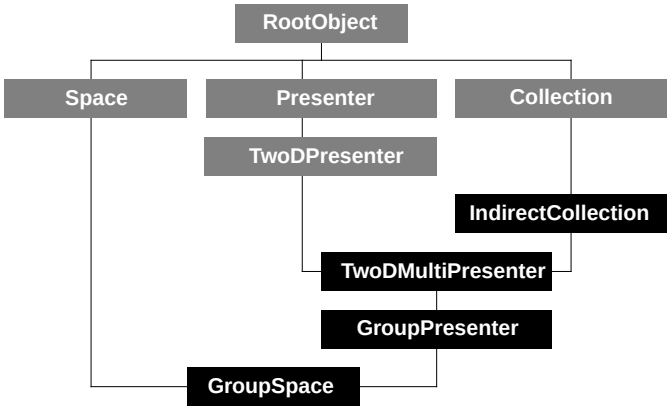
Accessible from `LinearCollection`:

`chooseOneBackwards`	`findRange`	`getNthKey`
`chooseOrdOne`	`forEachBackwards`	`getOrdOne`
`deleteFirst`	`getFirst`	`getRange`
`deleteLast`	`getLast`	`localEqual`
`deleteNth`	`getMiddle`	`localLT`
`deleteRange`	`getNth`	`pop`

Accessible from `Sequence`:

`addFifth`	`moveBackward`	`setFourth`
`addFirst`	`moveForward`	`setLast`
`addFourth`	`moveToBack`	`setNth`
`addNth`	`moveToFront`	`setSecond`
`addSecond`	`prepend`	`setThird`
`addThird`	`prependNew`	`sort`
`append`	`setFifth`	
`appendNew`	`setFirst`	

GroupSpace



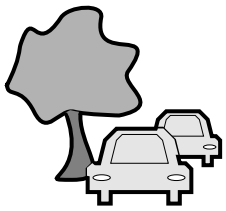
Class type: Core class (concrete)
Resides in: ScriptX and KMP executables
Inherits from: `Space` and `GroupPresenter`
Component: Spaces and Presenters

The `GroupSpace` class provides a way to group presenters together so that they can move, receive events and in other ways be treated as a single object.

The `GroupSpace` class is a combination of the `Space` and `GroupPresenter` classes. It is essentially a space (with a clock, controllers, and protocols), having the additional properties of a `GroupPresenter` object:

- The boundary of the group space is the union of its subpresenters—the group space does not clip its subpresenters. That is, the group space's boundary grows to include objects that are added to it. In contrast, `TwoDSpace` has a boundary that is fixed and clips its subpresenters.
- The boundary of the group space is a region with the irregular shape made up by its subpresenters. The boundary is not necessarily a rectangle, as it is with `TwoDSpace`.

The stroke of a group space outlines the region formed by its subpresenters, as shown in the following illustration. The fill of a group space would not be apparent unless any of its subpresenters is invisible. When new members are added to a group space, it sorts its subpresenters in their proper z-order, the same as `TwoDMultiPresenter`



The `GroupSpace` class is implemented internally as an `Array`.

In the `new` and `init` methods, the `target` and `boundary` keywords are ignored. The value supplied with `boundary` is discarded. However, the value supplied with `target` is saved in the `target` instance variable and is available for use in any subclass you might create, as a target for a presenter. For example, you might save a series of numbers to the target, and use them to determine the heights of bars in a bar chart.

Creating and Initializing a New Instance

The following script is an example of how to create a new instance of the `GroupSpace` class: (Note that `boundary` and `target` are both omitted because they are ignored.)

```
myGroup := new GroupSpace \
       fill:whiteBrush \
       stroke:blackBrush \
       scale:10
```

The variable `myGroup` contains an instance of `GroupSpace`. This instance has a white background, a black border, and its clock is set to 10 ticks per second. You then add objects to this group using the methods available to the `Array` class. The `new` method uses the keyword arguments defined by the `init` method.

For performance reasons, when creating an instance of `GroupSpace` (actually, any instance of `TwoDMultiPresenter` or its subclasses), you should *not* specify the `targetCollection` keyword, so it can create its default collection. Such presenters require collections that can be traversed quickly when drawing and handing events. Performance could suffer if you change the `targetcollection` to something other than the default.

init

`init` *self* [`fill:`*brush*] [`stroke:`*brush*] [`scale:`*integer*]
[`target:`*object*] [`boundary:`*region*] [`targetCollection:`*sequence*] ⇨ *(none)*

self	`GroupSpace` object
`fill:`	`Brush` object
`stroke:`	`Brush` object

Superclasses of `GroupSpace` use the following keywords:

`scale:`	`Integer` object
`target:`	Any object
`boundary:`	`Region` object
`stationary:`	`Boolean` object
`targetCollection:`	`Sequence` object

Initializes the `GroupSpace` object *self*, applying the values supplied with keywords to instance variables of the same name. The `targetCollection` keyword specifies the kind of collection to create. Do not call `init` directly on an instance—it is automatically called by the `new` method.

If you omit an optional keyword, its default value is used. The defaults are:

```
target:undefined
fill:undefined
stroke:undefined
boundary:(new Region)
scale:24
targetCollection:(new Array initialSize:14 growable:true)
stationary:false
```

Class Methods

Inherited from `Collection`:

`pipe`

Instance Variables

Inherited from `Collection`:

`bounded`	`maxSize`	`size`
`iteratorClass`	`minSize`	`uniformity`
`keyEqualComparator`	`mutable`	`uniformityClass`
`keyUniformity`	`mutableCopyClass`	`valueEqualComparator`
`keyUniformityClass`	`proprietored`	

Inherited from `IndirectCollection`:

`targetCollection`

Inherited from `Presenter`:

`presentedBy`	`subPresenters`	`target`

Inherited from `TwoDPresenter`:

`bBox`	`height`	`transform`
`boundary`	`IsImplicitlyDirect`	`width`
`clock`	`isTransparent`	`window`
`compositor`	`isVisible`	`x`
`direct`	`needsTickle`	`y`
`eventInterests`	`position`	`z`
`globalBoundary`	`stationary`	
`globalTransform`	`target`	

Inherited from `TwoDMultiPresenter`:

`clock`	`fill`	`stroke`

Inherited from `Space`:

`clock`	`protocols`	`tickleList`
`controllers`		

The following instance variables are defined in `GroupSpace`:

protocols (Space)

self`.protocols` (read-only) `Array`

The protocols array for instances of `GroupSpace` is initialized to contain the class `TwoDPresenter`. This means that any object added to an instance of `GroupSpace` must have `TwoDPresenter` as one of its superclasses. See the `Space` class for further description about this instance variable.

The attribute “read-only” means that you cannot make the `protocols` instance variable point to a different array—”read-only” does not stop you from adding or removing items from the array.

Instance Methods

Inherited from Collection:

add	forEach	iterate
addMany	forEachBinding	localEqual
addToContents	getAll	map
chooseAll	getAny	merge
chooseOne	getKeyAll	pipe
chooseOneBinding	getKeyOne	prin
deleteAll	getMany	removeAll
deleteBindingAll	getOne	removeOne
deleteBindingOne	hasBinding	setAll
deleteKeyAll	hasKey	setOne
deleteKeyOne	intersects	size
deleteOne	isEmpty	
emptyOut	isMember	

Inherited from IndirectCollection:

isAppropriateObject	objectAdded	objectRemoved

Inherited from TwoDPresenter:

adjustClockMaster	inside	show
createInterestList	localToSurface	surfaceToLocal
draw	notifyChanged	tickle
getBoundaryInParent	recalcRegion	
hide	refresh	

Inherited from TwoDMultiPresenter:

draw	findFirstInStencil	moveToBack
findAllAtPoint	isAppropriateObject	moveToFront
findAllInStencil	moveBackward	objectAdded
findFirstAtPoint	moveForward	objectRemoved

Inherited from Space:

isAppropriateObject	objectAdded	objectRemoved

Since a GroupSpace object is an indirect collection, you can also use any methods defined in the class specified by targetCollection. The target collection is typically an instance of Array, which inherits from Sequence, so the following instance methods are redirected to a group space.

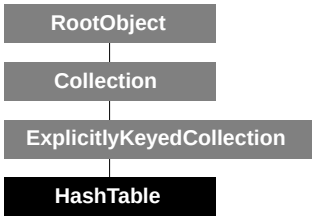
Accessible from LinearCollection:

chooseOneBackwards	findRange	getNthKey
chooseOrdOne	forEachBackwards	getOrdOne
deleteFirst	getFirst	getRange
deleteLast	getLast	localEqual
deleteNth	getMiddle	localLT
deleteRange	getNth	pop

Accessible from Sequence:

addFifth	moveBackward	setFourth
addFirst	moveForward	setLast
addFourth	moveToBack	setNth
addNth	moveToFront	setSecond
addSecond	prepend	setThird
addThird	prependNew	sort
append	setFifth	
appendNew	setFirst	

HashTable



Class type: Core class (concrete)
Resides in: ScriptX and KMP executables
Inherits from: `ExplicitlyKeyedCollection`
Component: Collections

A `HashTable` object is a collection of key-value pairs, implemented using a bucket-based hash table data structure.

A hash table has a fixed number of *buckets*, each of which is conceptually a growable array. Since the buckets are growable, they can become arbitrarily large without overflowing. This minimizes the overhead associated with search and retrieval, however there is overhead associated with adding new items to a hash table, especially as the number of items stored in any individual bucket grows large. Choose the number of buckets judiciously to reflect the trade-off between maintenance time when adding items, search time when retrieving items, and memory requirements.

At initialization, a developer has the option to specify a hash function, using the `hasher` keyword. Each bucket must be searched in a linear fashion to retrieve values. An effective and efficient hash function is one that requires little processing time, while spreading data stored in the hash table evenly over buckets.

If a hashing function is not supplied when an instance of `HashTable` is created, then all keys added to a `HashTable` object must implement the `hashOf` method (see note on page 295).

Creating and Initializing a New Instance

The following script is an example of how to create a new instance of the `HashTable` class:

```
myCornedBeefHash := new HashTable \
       numBuckets:250
```

The variable `myCornedBeefHash` contains the initialized hash table, with 250 buckets. The `new` method uses the keywords defined in `init`.

init

`init` *self* [`hasher:`*function*] [`numBuckets:`*integer*] ⇨ *(none)*

self	`HashTable` object
`hasher:`	`AbstractFunction` object
`numBuckets:`	`Integer` object

Initializes the `HashTable` object *self*, applying the values supplied with the keywords as follows: A table with the number of buckets specified by `numBuckets` is created. Optimal hash distribution depends on the `hasher` function that is used to return a hash value for the key value, but using a prime number for `numBuckets` is often advisable. Do not call `init` directly on an instance—it is automatically called by the `new` method.

If you omit an optional keyword, the following defaults are used:

```
hasher:hashOf
numBuckets:27
```

Note – In the core classes, only `String`, `NameClass`, and `ImmediateInteger` provide an implementation of `hashOf`. Scripted classes, and other classes in the core classes, can be specialized to implement `hashOf`, or a developer can specialize another function to use as a hasher. A hasher function can be defined that calls the generic `hashOf` on keys that implement it, such as names and strings.

Class Methods

Inherited from `Collection`:

`pipe`

Instance Variables

Inherited from `Collection`:

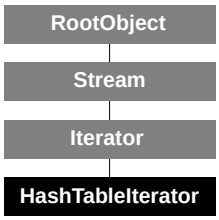
`bounded`	`maxSize`	`size`
`iteratorClass`	`minSize`	`uniformity`
`keyEqualComparator`	`mutable`	`uniformityClass`
`keyUniformity`	`mutableCopyClass`	`valueEqualComparator`
`keyUniformityClass`	`proprietored`	

Instance Methods

Inherited from `Collection`:

`add`	`forEach`	`iterate`
`addMany`	`forEachBinding`	`localEqual`
`addToContents`	`getAll`	`map`
`chooseAll`	`getAny`	`merge`
`chooseOne`	`getKeyAll`	`pipe`
`chooseOneBinding`	`getKeyOne`	`prin`
`deleteAll`	`getMany`	`removeAll`
`deleteBindingAll`	`getOne`	`removeOne`
`deleteBindingOne`	`hasBinding`	`setAll`
`deleteKeyAll`	`hasKey`	`setOne`
`deleteKeyOne`	`intersects`	`size`
`deleteOne`	`isEmpty`	
`emptyOut`	`isMember`	

HashTableIterator



Class type: Core class (concrete)
Resides in: ScriptX and KMP executables
Inherits from: `Iterator`
Component: Collections

The `HashTableIterator` class represents iterators that iterate over any `HashTable` objects.

Creating and Initializing a New Instance

A new instance of a hash table iterator is generally created by calling `iterate` on an instance of `HashTable`.

Instance Variables

Inherited from `Iterator`:

`key`	`source`	`value`

Instance Methods

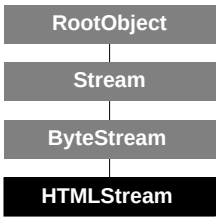
Inherited from `Stream`:

`cursor`	`next`	`seekFromStart`
`flush`	`plug`	`setStreamLength`
`isAtFront`	`previous`	`streamLength`
`isPastEnd`	`read`	`write`
`isReadable`	`readReady`	`writeReady`
`isSeekable`	`seekFromCursor`	
`isWritable`	`seekFromEnd`	

Inherited from `Iterator`:

`exise`	`seekKey`	`seekValue`
`remainder`		

HTMLStream



Class type: Loadable class (concrete)
Resides in: web.lib. Works with ScriptX and KMP executables
Inherits from: ByteStream
Component: Streams

The HTMLStream class implements the parsing of ASCII text data. HTML is a standard for data transmission over the Internet and the World Wide Web.

HTML tags are delimited by angle brackets, which may enclose multiple starting elements or a single ending element. In the following example, the anchor tag contains the HREF starting element. The value of that HREF element is a string, the URL (universal resource locator) for Apple Computer's World Wide Web server. In this example, the text "Apple Computer" is enclosed by the start element and end element. A slash after the initial angle bracket delimiter indicates an ending element or closing tag.

```
<A HREF="http://www.apple.com">Apple Computer</A>
```

The anchor tag thus applies to the text that is enclosed by the start element and the end element. An HTMLStream object is a parser that passes through an ASCII text file and calls one of three functions on each character or element.

When you create an instance of HTMLStream, you supply three parsing functions as keyword arguments: startElement, endElement, and putCharacter. These functions can be methods defined by a subclass of HTMLStream, or they can be primitives (global functions). Parsing functions are good candidates for using the ScriptX external API that is described in the "Extending ScriptX" chapter of the *ScriptX Tools Guide*.

The startElement function has the following form:

```
functionName stream integer [ key:value ]*
```

The first argument is the stream itself. The second argument is an integer, the integer value of the HTML tag. A complete list of HTML tags and their integer values follows. Finally, the startElement function takes a variable-length list of HTML elements. In the example above, the <A> tag (anchor) has the integer value 0, and is supplied with one element, the HREF element. The value of this HREF element is supplied to the startElement function, assigned to the keyword argument HREF, as a string with the value "http://www.apple.com". Thus, if startFunction is a ScriptX function, and myHTMLStream is an HTMLStream object that is instantiated with startFunction as the value supplied for startElement, then startFunction could be called as follows:

```
startFunction myHTMLStream 0 HREF:"http://www.apple.com"
```

Keyword arguments to this function correspond to the possible HTML elements for the given tag. The start function should ignore any elements it does not understand or implement. (In this way, it does not break as new elements are added to the HTML standard.)

The endElement function has the same form as the startElement function, except that it does not take keyword arguments. If endFunction is a ScriptX function that is used to instantiate myHTMLStream, then endFunction could be called as follows:

```
endFunction myHTMLStream 0
```

Finally, the function supplied for putCharacter is called with the stream and the character, represented as an integer, as arguments:

```
functionName stream integer
```

ScriptX has no character data type, so *integer* represents an ASCII character having a value between 0 and 127. If handleCharacter is a ScriptX function, supplied as the value of putCharacter to instantiate myHTMLStream, then handleCharacter could be called as follows to handle the lowercase "a" character, which has an ASCII value of 97:

```
handleCharacter myHTMLStream 97
```

The following table supplies a list of HTML tags and their integer values in the ScriptX HTMLStream class:

A	0	ISINDEX	44
ABBREV	1	KBD	45
ABSTRACT	2	L	46
ACRONYM	3	LI	47
ADDED	4	LINK	48
ADDRESS	5	LISTING	49
ARG	6	LIT	50
B	7	MARGIN	51
BASE	8	MATH	52
BLOCKQUOTE	9	MENU	53
BODY	10	NEXTID	54
BOX	11	NOTE	55
BR	12	OL	56
BYLINE	13	OPTION	57
CAPTION	14	OVER	58
CHANGED	15	P	59
CITE	16	PERSON	60
CMD	17	PLAINTEXT	61
CODE	18	PRE	62
COMMENT	19	Q	63
DD	20	QUOTE	64
DFN	21	RENDER	65
DIR	22	REMOVED	66
DL	23	S	67
DT	24	SAMP	68
EM	25	SELECT	69
FIG	26	STRONG	70
FOOTNOTE	27	SUB	71
FORM	28	SUP	72
H1	29	TAB	73
H2	30	TABLE	74
H3	31	TD	75
H4	32	TEXTAREA	76
H5	33	TH	77
H6	34	TITLE	78
H7	35	TR	79
HEAD	36	TT	80

HR	37	U	81
HTML	38	UL	82
HTMLPLUS	39	VAR	83
I	40	XMP	84
IMAGE	41	XSCRIPTX	85
IMG	42	LAST	86
INPUT	43		

Creating and Initializing a New Instance

The following script creates a new instance of HTMLSTream:

```
global myHTMLParser := new HTMLParser \
       startElement:startFunction \
       endElement:endFunction \
       putCharacter:handleCharacter
```

The functions startFunction, endFunction, and handleCharacter are assumed to be defined elsewhere, either as scripted methods on a subclass of HTMLStream, as global functions, or as primitives or methods defined in C and bound to ScriptX names through the ScriptX external API. The new method uses keyword arguments defined by init.

init

init *self* [startElement:*function*] [endElement:*function*]
[putCharacter:*function*] ⇨ *(none)*

self	HTMLSTream object
startElement:	ByteCodeMethod or Primitive object
endElement:	ByteCodeMethod or Primitive object
putCharacter:	ByteCodeMethod or Primitive object

Initializes the HTMLStream object *self*, where startElement, endElement, and putCharacter can each be used to specify a function that is called automatically. Do not call init directly on an instance—it is called automatically by the new method.

If you omit an optional keyword, its default value is used. The defaults are:

```
startElement:undefined
endElement:undefined
putCharacter:undefined
```

Instance Methods

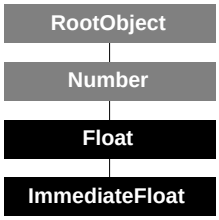
Inherited from Stream:

cursor	next	seekFromStart
flush	plug	setStreamLength
isAtFront	previous	streamLength
isPastEnd	read	write
isReadable	readReady	writeReady
isSeekable	seekFromCursor	
isWritable	seekFromEnd	

Inherited from ByteStream:

fileIn	readByte	writeString
pipe	readReady	
pipePartial	writeByte	

ImmediateFloat



Class type: Core class (concrete, sealed)
Resides in: ScriptX and KMP executables
Inherits from: `Float`
Component: Numerics

The `ImmediateFloat` class represents floating point numbers that do not require the range or precision offered by the `Float` class. Although the `ImmediateFloat` class is based on the IEEE 32-bit standard for floating point numbers, two bits of the mantissa are sacrificed to store an object tag, resulting in some loss of precision.

.

Table 6: Characteristics of the ImmediateFloat class

binary size	30 bits, stored in a 32-bit word, omitting the two low-order bits
storage characteristics	1 bit for the sign, 8 bits for the exponents, and 29 bits for the mantissa
decimal range	$\pm 3.4 \times 10^{38}$
decimal precision	$\pm 1.2 \times 10^{-38}$ (six digits)

Floating point numbers that can be represented by the `ImmediateFloat` class are not technically converted into objects. As with the `ImmediateInteger` class, ScriptX collapses the representation of an immediate object into its own pointer. To the rest of ScriptX, immediate objects appear in every way like real objects. The term “immediate float” indicates there is no indirection involved in access to the floating point value (indirection is required for access to regular objects). The rest of this manual refers to immediate floats as if they were full-fledged objects.

For information on the automatic conversion and coercion of operands in arithmetic calculations, see the *ScriptX Language Guide*.

Creating and Initializing a New Instance

The `ImmediateFloat` class has no scripter-level `new` method. There is no need to explicitly create or initialize an instance of the `ImmediateFloat` class. The compiler stores a floating point value as an `ImmediateFloat` object, unless there would be a loss of range or precision. Floating point constants with less than 6 decimal digits are stored as immediate floats. Floating point constants with 7 or more decimal digits are stored as regular floats.

```
getClass (local g := 1.23456; g)
⇨ ImmediateFloat
getClass (local h := 1.234567; h)
⇨ Float
```

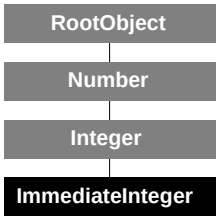
You can force the compiler to store any number as an immediate float, although this may involve truncation or lost precision.

Instance Methods

Inherited from `Number`:

`abs`	`floor`	`radToDeg`
`acos`	`frac`	`random`
`asin`	`inverse`	`rem`
`atan`	`ln`	`round`
`atan2`	`log`	`sin`
`ceiling`	`max`	`sinh`
`coerce`	`min`	`sqrt`
`cos`	`mod`	`tan`
`cosh`	`morph`	`tanh`
`degTorad`	`negate`	`trunc`
`exp`	`power`	

ImmediateInteger



Class type: Core class (concrete, sealed)
Resides in: ScriptX and KMP executables
Inherits from: `Integer`
Component: Numerics

The `ImmediateInteger` class represents signed integers from -2^{29} to $2^{29} - 1$ (that is, –536,870,912 to +536,870,911).

For optimization purposes, integers of this size are never technically made into objects, with all the associated overhead that object require; they remain integers. For this reason, `ImmediateInteger` is preferred over `LargeInteger` whenever possible. To the rest of ScriptX, these integers appear in every way like objects. This also explains the term “immediate integer”, which indicates that there is no indirection involved in accessing the integer (indirection is required when accessing objects). The rest of this manual refers to immediate integers as if they were full-fledged objects.

Creating and Initializing a New Instance

The `ImmediateInteger` class has no scripter-level `new` method. There is no need for you to explicitly create or initialize an instance of the `ImmediateInteger` class—simply use whatever integer you want in a script. Whenever the compiler encounters an integer up to $\pm 2^{29}$, it automatically coerces it to an `ImmediateInteger` object. If the integer is larger, the compiler object converts it to a `LargeInteger` or `Float`, as appropriate. The `ImmediateInteger` class has no scripter-level `new` or `init` method.

Instance Methods

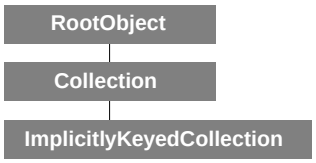
Inherited from `Number`:

`abs`	`floor`	`radToDeg`
`acos`	`frac`	`random`
`asin`	`inverse`	`rem`
`atan`	`ln`	`round`
`atan2`	`log`	`sin`
`ceiling`	`max`	`sinh`
`coerce`	`min`	`sqrt`
`cos`	`mod`	`tan`
`cosh`	`morph`	`tanh`
`degTorad`	`negate`	`trunc`
`exp`	`power`	

Inherited from `Integer`:

`length`	`logicalOp`	`lshift`
`logicalAnd`	`logicalOr`	`rshift`
`logicalNot`	`logicalXor`	

ImplicitlyKeyedCollection



Resides in: ScriptX and KMP executables
Inherits from: `Collection`
Class type: Abstract
Component: Collections

`ImplicitlyKeyedCollection` is the abstract superclass of all collections with implicit keys.

An "implicit" key is a key that is provided by the collection rather than by the user. Thus, collections that inherit from `ImplicitlyKeyedCollection` have their key values assigned by the collection.

The most common kind of implicitly keyed collection is `Sequence`, whose keys are the series of integers 1, 2, 3, and so on. These integers represent ordinal positions of elements within the collection.

Class Methods

Inherited from `Collection`:

`pipe`

Instance Variables

Inherited from `Collection`:

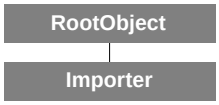
`bounded`	`maxSize`	`size`
`iteratorClass`	`minSize`	`uniformity`
`keyEqualComparator`	`mutable`	`uniformityClass`
`keyUniformity`	`mutableCopyClass`	`valueEqualComparator`
`keyUniformityClass`	`proprietored`	

Instance Methods

Inherited from `Collection`:

`add`	`forEach`	`iterate`
`addMany`	`forEachBinding`	`localEqual`
`addToContents`	`getAll`	`map`
`chooseAll`	`getAny`	`merge`
`chooseOne`	`getKeyAll`	`pipe`
`chooseOneBinding`	`getKeyOne`	`prin`
`deleteAll`	`getMany`	`removeAll`
`deleteBindingAll`	`getOne`	`removeOne`
`deleteBindingOne`	`hasBinding`	`setAll`
`deleteKeyAll`	`hasKey`	`setOne`
`deleteKeyOne`	`intersects`	`size`
`deleteOne`	`isEmpty`	
`emptyOut`	`isMember`	

Importer



Class type: Core class (abstract)
Resides in: ScriptX and KMP executables
Inherits from: `RootObject`
Component: Import and Export

The `Importer` class is an abstract class and is not instantiable. `ImportExportEngine` objects use subclasses of `Importer` to import data. Each subclass of `Importer` is an importer conversion class that converts a stream that has a specific external data format to a specific type of internal data object—for example, a stream in the `@pict` format to a bitmap `TwoDShape` object.

An importer conversion class, or import filter, is a subclass of `Importer` that implements an `importFromStream` method.

For full details of how to import media into ScriptX, see the *ScriptX Tools Guide*.

Creating and Initializing a New Instance

Note – When importing data, do not create an instance of an `Importer` subclass directly. Create an instance of `ImportExportEngine`, then call the `importMedia` method on that instance. `ImportExportEngine` creates an instance of the proper subclass of `Importer` automatically.

init

`init` *self* `mediaCategory:`*name* `inputMediaType:`*name*
`outputMediaType:`*name* ⇨ *(none)*

self	`Importer` object
`mediaCategory:`	`NameClass` object
`inputMediaType:`	`NameClass` object
`outputMediaType:`	`NameClass` object

Initializes the `Importer` object *self*, applying the keyword arguments that specify the media category, and the types of input and output media. Do not call `init` directly on an instance—it is automatically called by the `init` method.

Instance Variables

inputMediaType

self`.inputMediaType` (read-only) `NameClass`

Specifies the type of media that the importer *self* converts—for example, `@pict`, `@aiff` or `@wave`.

mediaCategory

self`.mediaCategory` (read-only) `NameClass`

Specifies the category of media that the importer *self* converts and imports—for example, `@sound`.

outputMediaType

self.`outputMediaType` (read-only) `NameClass`

Specifies the type of media that the importer *self* exports—for example, `@pict`.

Instance Methods

importFromStream

`importFromStream` *self source* ⇨ *(object)*

self	`Importer` object
source	`Stream` object

Returns the imported *object* that `importFromStream` creates. The *argument source* is the stream to import using the importer *self*—for example, `@pict`. If the stream in *source* is unknown to the `Exporter`, this method generates the exception `unknownMedia`.

Some importers allow for extra arguments beyond those listed above. It is then up to the individual importer to interpret those extra arguments. For example, the PICT importer may have an argument *bitDepth* to specify the bit depth to convert the image into.

ImportExportEngine

Class type: Core class (concrete)
Resides in: ScriptX and KMP executables
Inherits from: `RootObject`
Component: Import and Export

The `ImportExportEngine` class provides an interface to Importer and Exporter objects. To import or export data call the `importMedia` or `exportMedia` method on `theImportExport` global instance, which is a pre-defined instance of the `ImportExportEngine` class.

For full details of how to import media into ScriptX and export media from ScriptX, see the *ScriptX Tools Guide*.

Creating and Initializing a New Instance

Note – The global variable `theImportExportEngine` is an instance of `ImportExportEngine`. Thus, a script really has no need to create a new instance of `ImportExportEngine`.

The following sample script shows how you can create a new, nonstandard instance of the `ImportExportEngine` class:

```
myEngine := new ImportExportEngine
```

The variable `myEngine` points to the initalized import-export engine. The `new` method uses the keywords defined in `init`.

init

`init` *self* ⇨ *(none)*

self	`ImportExportEngine` object

Initializes the import export engine *self*. The `init` method in the `ImportExportEngine` class has no keyword arguments. Do not call `init` directly on an instance—it is automatically called by the `new` method.

Instance Methods

exportMedia

`exportMedia` *self source destination mediaCategory inputMediaType outputMediaType* ⇨ `Stream`

self	`ImportExportEngine` object
source	Any internal data object
destination	`Stream` or `Sequence` object, as defined by `Exporter`
mediaCategory	`NameClass` object
inputMediaType	`NameClass` object
outputMediaType	`NameClass` object

Exports the media described by the arguments and returns the *stream* that `exportMedia` creates. The argument *source* is the internal data object to export using the `ImportExportEngine` object *self*. An `Exporter` defines the type of *destination* object that it requires—generally, a `Stream` to which to export the data object. However, some `Exporter` objects require a `Sequence` as their *destination* object. The argument *mediaCategory* is the category of media to which the data object belongs. The argument *inputMediaType* is the media type of the data object being exported. The argument *outputMediaType* is the media type to export to the stream. If `ImportExportEngine` cannot find an `Exporter` object that can export a data object of the required category and type, and convert that data object to the required external data format, the `exportMedia` method generates the exception `importerExporterNotFound`.

ScriptX ships with only one exporter, the text exporter.

Some exporters allow extra arguments beyond those listed above. Thus, any keywords you pass to the `exportMedia` method are passed on to the `exportToStream` method in the exporter. It is then up to the individual exporter to interpret those extra arguments.

importMedia

`importMedia` *self source mediaCategory inputMediaType outputMediaType*

⇨ *(object)*

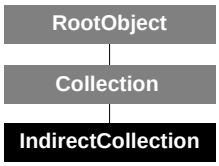
self	`ImportExportEngine` object
source	`Stream` object or `Collection` object, as defined by `Importer`
mediaCategory	`NameClass` object
inputMediaType	`NameClass` object
outputMediaType	`NameClass` object

Imports the media described by the arguments and returns the object that `importMedia` creates, or `undefined` if the method fails to execute. An `Importer` object defines the type of *source* object that it requires—generally, a `Stream` to import using the `ImportExportEngine` object *self*. However, some `Importer` objects require a `Collection` object that constitutes a path to a file that the Importer converts to a stream. The argument *mediaCategory* is the category of media in the stream being imported. The argument *inputMediaType* is the type of media in the stream. The argument *outputMediaType* is the class of internal data object to which this method converts the media in the stream. If `ImportExportEngine` cannot find an `Importer` object that can import a stream that contains data of the required category and type, and convert that data to the required class of internal data object, the `importMedia` method generates the exception `importerExporterNotFound`.

Some importers allow extra arguments beyond those listed above. Thus, any keywords you pass to the `importMedia` method are passed on to the `importFromStream` method in the importer. It is then up to the individual importer to interpret those extra arguments.

See the *ScriptX Tools Guide* for a full list of the arguments to `importMedia` for importing different kinds of media.

IndirectCollection



Class type: Core class (concrete)
Resides in: ScriptX and KMP executables
Inherits from: `Collection`
Component: Collections

`IndirectCollection` is a subclass of `Collection` that does not contain any objects, but instead has an instance variable `targetCollection` that is a standard collection object. If you add or delete objects from an instance of `IndirectCollection`, they are actually placed into or removed from the `targetCollection` object.

Many ScriptX classes inherit from `IndirectCollection`, which is often mixed in with other classes. Indirect collections include classes that are the foundation of ScriptX title and tool development, such as `TitleContainer`, `Window`, `TwoSpace`, `TwoDMultiPresenter`, and `Controller`.

In effect, the use of `IndirectCollection` rather than any specific implementation of `Collection` factors out the collection behavior of the class from its other behavior. In a given title, a developer could create one controller that is an array, another that is a linked list, and yet another that is a sorted keyed array.

Important – The value of `targetCollection` for an `IndirectCollection` object should always be a built-in ScriptX class. If you want to modify the behavior of an indirect collection, subclass `IndirectCollection` and create additional methods. The target collection cannot be a user-defined subclass of `Collection`, nor can it be a specialized instance of a collection class.

`IndirectCollection` implements the union of all the protocols associated with all classes of collections. Any operation performed on an instance of `IndirectCollection` is first handled by its specialization in `IndirectCollection` and then redirected to the `targetCollection` object. (Indirection gives the class its name, the "indirect" collection.)

This indirection supports a notification protocol. An indirect collection implements three generic functions: `isAppropriateObject`, `objectAdded`, and `objectRemoved`. `IndirectCollection` itself supplies only placeholders—these methods are meant to be specialized by subclasses.

Every time you attempt to insert an object into an indirect collection, `isAppropriateObject` is called automatically. If the method returns `true`, the process is allowed to continue. If the method returns `false`, the process is aborted. The other two methods are called whenever an object is added to or removed from the target collection.

Notification requires that operations be redirected at the lowest level—for example, `addMany` is implemented as multiple calls to the `add` method. `IndirectCollection` automatically redirects all generics that are defined by `Collection`, `LinearCollection`, and `Sequence`.

Some collections are not linear collections or sequences. This creates a potential conflict that developers should be aware of. For example, linear collections implement the `pop` method, which is not defined in the Collection protocol. If the target collection is not a

linear collection, an indirect collection still redirects a call to pop to the target collection. For this reason, canObjectDo may fail to test an object properly, as in the following example:

```
object myIndirectCollection (IndirectCollection)
targetCollection:(new HashTable)
contents "google", "zillion", "gazillion"
end
⇨ #<IndirectCollection+ over #(#1="zillion":#1#, #2="gazillion":#2#, \
   #3="google":#3#) as HashTable>
canObjectDo myIndirectCollection pop
⇨ true
```

Although canObjectDo returned true, the operation reports an exception when you attempt to pop an item from the collection. The workaround is to call canObjectDo directly on the target collection.

```
canObjectDo myIndirectCollection.targetCollection pop
⇨ false
```

Creating and Initializing a New Instance

In general, you do not create an instance of IndirectCollection using new, but instead define a subclass of IndirectCollection and create an instance of it. This subclass can map the target collection's protocol onto one of its instance variables, as shown in the example below. You normally override the objectAdded, objectRemoved, or isAppropriateObject methods.

This example defines a class named DefaultList that supplies an instance of LinkedList as its default target collection, if the targetCollection keyword is omitted. Of course, one advantage to using an indirect collection is the ease with which you can substitute one target collection for another.

```
class DefaultList (IndirectCollection)
inst methods
method init self #rest args #key targetCollection: -> (
    if (targetCollection = unsupplied) then (
        apply nextMethod self targetCollection:(new LinkedList) args
    )
    else (
        apply nextMethod self args
-- Create an instance of the above class
myIndirect := new DefaultList
```

init

`init` *self* `targetCollection:`*collection* ⇨ *(none)*

self	IndirectCollection object
targetCollection:	Collection object

Initializes the IndirectCollection object *self*, applying the value of the targetCollection keyword to the instance variable of the same name. All methods defined in the Collection, LinearCollection, and Sequence protocols are redirected to the collection assigned to the targetCollection keyword. The targetCollection keyword is required, however many subclasses of IndirectCollection specify a target collection by default. Do not call init directly on an instance—it is automatically called by the new method.

Class Methods

Inherited from `Collection`:

`pipe`

Instance Variables

Inherited from `Collection`:

`bounded`	`maxSize`	`size`
`iteratorClass`	`minSize`	`uniformity`
`keyEqualComparator`	`mutable`	`uniformityClass`
`keyUniformity`	`mutableCopyClass`	`valueEqualComparator`
`keyUniformityClass`	`proprietored`	

The following instance variables are defined in `IndirectCollection`:

targetCollection

self`.targetCollection` (read-write) `Collection`

Specifies the `Collection` instance that the indirect collection *self* represents.

Instance Methods

Inherited from `Collection`:

`add`	`forEach`	`iterate`
`addMany`	`forEachBinding`	`localEqual`
`addToContents`	`getAll`	`map`
`chooseAll`	`getAny`	`merge`
`chooseOne`	`getKeyAll`	`pipe`
`chooseOneBinding`	`getKeyOne`	`prin`
`deleteAll`	`getMany`	`removeAll`
`deleteBindingAll`	`getOne`	`removeOne`
`deleteBindingOne`	`hasBinding`	`setAll`
`deleteKeyAll`	`hasKey`	`setOne`
`deleteKeyOne`	`intersects`	`size`
`deleteOne`	`isEmpty`	
`emptyOut`	`isMember`	

The following instance methods are defined in `IndirectCollection`:

isAppropriateObject

`isAppropriateObject` *self addedObject* ⇨ `Boolean`

self	`IndirectCollection` object
addedObject	Any object added to the collection

This method is automatically called when the object *addedObject* is about to be added to an indirect collection. It should perform any operations that need to be done before an object is actually added to the collection. In the `IndirectCollection` class this method always returns `true`—you can override this method in any subclass of `IndirectCollection` you define. If this method returns `false` for *addedObject*, an exception is reported and the object is not added to the collection.

This method is called automatically by any generic in the `Collection`, `LinearCollection`, or `Sequence` protocols that adds an object to a collection, such as `add`, `addNth`, `prepend`, or `setOne`.

objectAdded

`objectAdded` *self addedKey addedObject* ⇨ *(none)*

self	`IndirectCollection` object
addedKey	Any object
addedObject	Any object

Performs a particular action after the object *addedObject* is added to the collection *self*. The parameter *addedKey* is the key that can be used to retrieve the *addedObject*. The particular action is determined by the implementation of this method in the collection *self*.

`IndirectCollection` defines only an empty placeholder method for `objectAdded`—you typically subclass `IndirectCollection` and specialize this method in that subclass to perform the action you want to occur every time an object is added. Note that this method is called only if the method `isAppropriateObject` has already tested the object and returned `true`.

Do not call this method directly—it is called automatically (after `isAppropriateObject`) by any method in `IndirectCollection` that adds an object to a collection, such as `add`, `addNth`, `prepend`, and `setOne`.

objectRemoved

`objectRemoved` *self removedObject* ⇨ *(none)*

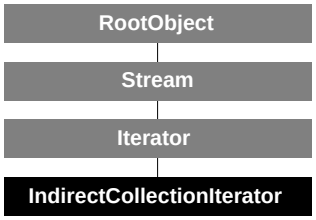
self	`IndirectCollection` object
removedObject	Any object removed from the collection

Performs a particular action after the *removedObject* has been removed from the collection. This method automatically gets called by all methods in `IndirectCollection` that can remove the object *removedObject* from a collection, such as `delete`, `deleteNth`, `deleteFirst`, or `setOne`.

`IndirectCollection` defines only an empty placeholder method for `objectRemoved`—you typically subclass `IndirectCollection` and specialize this method in that subclass to perform the action you want to occur every time an object is removed.

Do not call this method directly—it is called automatically by any method in `IndirectCollection` that removes an object from a collection, such as `deleteOne`, `deleteNth`, `deleteAll`, and `removeOne`.

IndirectCollectionIterator



Class type: Core class (concrete)
Resides in: ScriptX and KMP executables
Inherits from: `Iterator`
Component: Collections

The `IndirectCollectionIterator` class represents an iterator that iterates over any `IndirectCollection` object.

Creating and Initializing a New Instance

A new instance of an indirect collection iterator is generally created by calling `iterate` on an instance of `IndirectCollection`.

Instance Variables

Inherited from `Iterator`:

`key`	`source`	`value`

Instance Methods

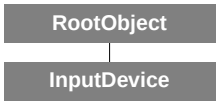
Inherited from `Stream`:

`cursor`	`next`	`seekFromStart`
`flush`	`plug`	`setStreamLength`
`isAtFront`	`previous`	`streamLength`
`isPastEnd`	`read`	`write`
`isReadable`	`readReady`	`writeReady`
`isSeekable`	`seekFromCursor`	
`isWritable`	`seekFromEnd`	

Inherited from `Iterator`:

`exise`	`seekKey`	`seekValue`
`remainder`		

InputDevice



Class type: Core class (abstract)
Resides in: ScriptX and KMP executables
Inherits from: `RootObject`
Component: Input Devices

`InputDevice` is an abstract class that represents instances of user input devices such as a keyboard or mouse. Input devices provide an interface to hardware devices installed in the system. Input devices hide details of device implementation, which are specific to a given platform, providing a standard interface for each kind of device. Each subclass of `InputDevice` keeps its own private list of input devices, with each device having its own unique device ID.

Creating and Initializing a New Instance

`InputDevice` is an abstract class, and cannot be instantiated. It does, however, implement an `init` method. This `init` method is applied whenever a subclass of `InputDevice`, such as `PhysicalKeyboardDevice`, is created.

To change the way instances are created and initialized, override the `init` instance method as described below. Do not override the `new` method.

init

`init` *self* [`enabled:`*boolean*] [`deviceID:`*integer*] ⇨ *(none)*

self	`InputDevice` object
`enabled:`	`Boolean` object
`deviceID:`	`Integer` object

Initializes the `InputDevice` *self*, applying the values supplied with the keywords to the instance variables of the same name.

If `enabled` is true, the device is enabled. The value of `deviceID` is the identification number for the device, allowing the system to distinguish different devices that are connected. It is unique for any particular subclass of `InputDevice`. Do not call `init` directly on an instance—it is called automatically by the `new` method.

If you omit an optional keyword, its default value is used. The defaults are:

`enabled:true`
`deviceID:`first available ID number for the given `InputDevice` subclass

Class Methods

getDeviceFromList

`getDeviceFromList` *self deviceID* ⇨ `InputDevice`

self	`InputDevice` subclass
deviceID	`Number` object

This is a class method that must be implemented by all subclasses. Use `getDeviceFromList` to check if a device has been created. This method returns the previously created instance of the input device *self* that corresponds to the given *deviceID*. It returns `empty` if no instance of the device has been created with the ID specified by *deviceID*.

For example, `getDeviceFromList MouseDevice 1` returns the mouse instance that has a device ID of 1.

The input device is originally created by using the `new` method, and its `deviceID` is usually set by default. (See the appropriate subclass of `InputDevice`.) Thereafter, when `getDeviceFromList` is called, *deviceID* is used as a key to search the private list of devices that the subclass keeps. This private list of devices allows the ScriptX Player and the title to access the same device instance by referring to it by device ID.

Instance Variables

deviceID

self`.deviceID` (read-only) `Integer`

Specifies the device ID for the input device *self*. As a rule, the value of `deviceID` is a default integer value that the system supplies. This instance variable is used to determine which device within a class of input devices generated an event. The ID is guaranteed to be unique only within each subclass of `InputDevice`. Device IDs range from 1 to the number of devices of that type that are available.

enabled

self`.enabled` (read-write) `Boolean`

Is `true` if the device *self* is allowed to send events; otherwise, it is `false`.

focusable

self.`focusable` (read-only) `Boolean`

Indicates whether the input device *self* is focusable and is currently being managed by a focus manager. The focus manager is stored in the `focusManager` instance variable.

focusManager

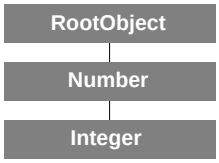
self.`focusManager` (read-only) *(object)*

Indicates the focus manager that is currently managing focus on the input device *self*. In ScriptX version 1.1, `focusManager` is set automatically on instantiation for instances of `KeyboardDevice`, a subclass of `InputDevice`. The focus manager for a keyboard device is an instance of `KeyboardFocusManager`.

Subclasses Must Implement

In ScriptX, you cannot create a scripted subclass of `InputDevice` for a new physical device. Since device interfaces are specific to hardware, such a subclass requires programming in OIC. You can create a scripted subclass of `InputDevice` only for a virtual device that is operated from built-in physical devices and has no connection to any new physical device.

Integer



Class type: Core class (abstract, sealed)
Resides in: ScriptX and KMP executables
Inherits from: `Number`
Component: Numerics

The `Integer` class represents numbers without any fractional part. The `Integer` family of classes includes `ImmediateInteger`, `LargeInteger`, `Time`, `Date` and `PacketTime`.

Instance Methods

Inherited from `Number`:

`abs`	`floor`	`radToDeg`
`acos`	`frac`	`random`
`asin`	`inverse`	`rem`
`atan`	`ln`	`round`
`atan2`	`log`	`sin`
`ceiling`	`max`	`sinh`
`coerce`	`min`	`sqrt`
`cos`	`mod`	`tan`
`cosh`	`morph`	`tanh`
`degTorad`	`negate`	`trunc`
`exp`	`power`	

The following instance methods are defined by all subclasses of `Integer`

length

`length` *self* ⇨ `Integer`

Returns the number of bits needed to represent the integer in two's complement. Note that `length i` is equivalent to the following:

```
ceiling (log (if i < 0 then -i else i + 1) 2)
```

logicalAnd

`logicalAnd` *self integer* ⇨ `Integer`

self	`Integer` object
integer	`Integer` object

Returns the bitwise logical and operation of *self* and *integer*.

logicalNot

`logicalNot` *self* ⇨ `Integer`

Returns the bitwise logical not operation of *self*.

logicalOr

`logicalOr` *self integer* ⇨ `Integer`

self	`Integer` object
integer	`Integer` object

Returns the bitwise logical or operation of *self* and *integer*.

logicalXor

`logicalXor` *self integer* ⇨ `Integer`

self	`Integer` object
integer	`Integer` object

Returns the bitwise logical exclusive or operation of *self* and *integer*.

lshift

`lshift` *self numbits* ⇨ `Integer`

self	`Integer` object
numbits	`Number` object

Shifts the bits in the integer *self* left by the number of bits specified by *numbits*. The argument *numbits* is truncated if it is not an integer. If *numbits* is positive, shift left. If it is negative, shift right.

If an `ImmediateInteger` object is shifted and the result is out of range for the class, the result is not promoted to `LargeInteger`. However, if a `LargeInteger` object is shifted and the result can be stored in an `ImmediateInteger` object, it is automatically demoted. This behavior is depicted in the following example.

```
global r := rshift 1000 1 -- an ImmediateInteger object
⇨  500
lshift r 1 -- now shift it back in the other direction
⇨  1000
-- example where large integer is shifted into ImmediateInteger range
global q := rshift 1000000000 1
⇨  500000000
lshift q 1 -- now shift it back in the other direction
⇨  -73741824
```

Note that bit-shift operation may change the value of the sign bit.

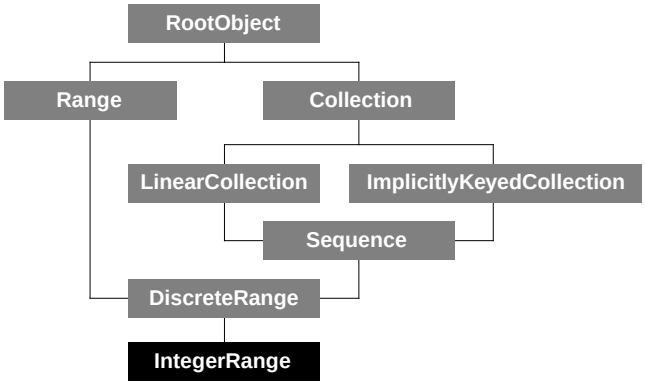
rshift

`rshift` *self numbits* ⇨ `Integer`

self	`Integer` object
numbits	`Number` object

Shifts the bits in the integer *self* right by the number of bits specified by *numbits*. The argument *numbits* is truncated if it is not an integer. If *numbits* is positive, shift right. If it is negative, shift left. The sign of the result is determined by whether a 0 or a 1 gets shifted into the sign bit. See the defintion of `lshift` for additional information.

IntegerRange



Class type: Core class (concrete)
Resides in: ScriptX and KMP executables
Inherits from: `DiscreteRange`
Component: Collections

The `IntegerRange` class represents a discrete range of integers. The default is to include the values at the lower and upper bounds.

Creating and Initializing a New Instance

The following script is an example of how to create a new instance of the `IntegerRange` class:

```
myRange := new IntegerRange \
           lowerBound: 0 \
           upperBound: 100 \
           increment: 5
```

The variable `myRange` contains the initialized integer range. This instance incorporates every fifth integer from 0 to 100, and is equivalent to the following series in set notation:

{0, 5, 10, 15, . . . 100}

The `new` method uses the keywords defined by the `init` method.

init

`init` *self* `lowerBound:`*integer* `upperBound:`*integer* [`increment:`*integer*] ⇨ *(none)*

self	`IntegerRange` object
`lowerBound:`	`Integer` object
`upperBound:`	`Integer` object
`increment:`	`Integer` object

Initializes the `IntegerRange` object *self*, applying the values supplied with the keywords to the instance variables of the same name. Do not call `init` directly on an instance—it is automatically called by the `new` method.

If the lowerbound is greater than the upperbound and the increment is omitted, the new range is empty. For negative increments, reverse the `lowerBound` and `upperBound`—set the starting value to `lowerbound` and the ending value to `upperBound`, as in the following example:

```
init self lowerBound:10 upperBound:0 increment:-1
```

The keyword `increment` is required for ranges with a negative increment; it is optional only for ranges with a positive increment.

If you omit an optional keyword, its default value is used. The only default is:

```
increment:1
```

Class Methods

Inherited from `Collection`:

```
pipe
```

Instance Variables

Inherited from `Collection`:

`bounded`	`maxSize`	`size`
`iteratorClass`	`minSize`	`uniformity`
`keyEqualComparator`	`mutable`	`uniformityClass`
`keyUniformity`	`mutableCopyClass`	`valueEqualComparator`
`keyUniformityClass`	`proprietored`	

Inherited from `Range`:

`includesLower`	`lowerBound`	`valueClass`
`includesUpper`	`size`	
`increment`	`upperBound`	

The following instance variables are redefined in `IntegerRange`:

includesLower (Range)

self`.includesLower` (read-only) `Boolean`

Returns `true`.

includesUpper (Range)

self`.includesUpper` (read-only) `Boolean`

Returns `true`.

valueClass (Range)

self`.valueClass` (read-only) *(class)*

Returns `Integer`.

Instance Methods

Inherited from `Collection`:

`add`	`forEach`	`iterate`
`addMany`	`forEachBinding`	`localEqual`
`addToContents`	`getAll`	`map`
`chooseAll`	`getAny`	`merge`
`chooseOne`	`getKeyAll`	`pipe`
`chooseOneBinding`	`getKeyOne`	`prin`
`deleteAll`	`getMany`	`removeAll`
`deleteBindingAll`	`getOne`	`removeOne`
`deleteBindingOne`	`hasBinding`	`setAll`
`deleteKeyAll`	`hasKey`	`setOne`

deleteKeyOne	intersects	size
deleteOne	isEmpty	
emptyOut	isMember	

Inherited from LinearCollection:

chooseOneBackwards	findRange	getNthKey
chooseOrdOne	forEachBackwards	getOrdOne
deleteFirst	getFirst	getRange
deleteLast	getLast	localEqual
deleteNth	getMiddle	localLT
deleteRange	getNth	pop

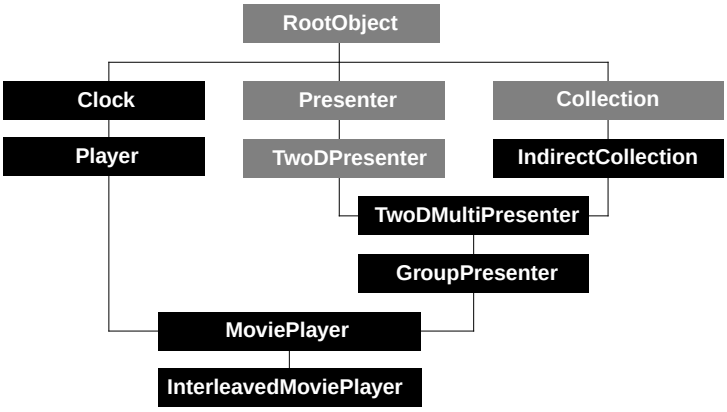
Inherited from Sequence:

addFifth	moveBackward	setFourth
addFirst	moveForward	setLast
addFourth	moveToBack	setNth
addNth	moveToFront	setSecond
addSecond	prepend	setThird
addThird	prependNew	sort
append	setFifth	
appendNew	setFirst	

Inherited from Range:

withinRange

InterleavedMoviePlayer



Class type: Core class (concrete)
Resides in: ScriptX and KMP executables
Inherits from: `MoviePlayer`
Component: Media Players

The `InterleavedMoviePlayer` class provides methods for playing interleaved movies in ScriptX.

When importing a movie into ScriptX, you can choose to import it as an interleaved or non-interleaved movie. If you intend to play an imported movie from a CD, you should import it into ScriptX as an `InterleavedMoviePlayer` to preserve the interleaving. If you intend to play it back from hard disk, then you can import it as a non-interleaved movie (that is, import it as a `MoviePlayer`) although you will usually get better performance if you import it as an `InterleavedMoviePlayer`.

When a frame of a movie plays, data is needed for both the video and the audio tracks. If the data for the tracks are not interleaved, the video data needed for a frame may be arbitrarily distant on the storage medium from the audio data needed for the same frame. When playing back movies with non-interleaved data from a hard disk, the extra search time required to seek to non-sequential positions is relatively small and does not significantly affect the speed of playback. However, the search time becomes significant if the movie is played back from a CD.

When the audio and video streams are interleaved, the video data and audio data required for a frame are located sequentially on the storage medium, thus minimizing the search time between each frame.

When ScriptX imports a movie file as an interleaved movie, it creates an `InterleavedMoviePlayer`. The interleaved movie player's `interleavedStream` instance variable has a `ByteStream` instance containing interleaved sound and video data for the movie. The importing process also creates `DigitalAudioPlayer` instances that are responsible for playing the sound in the movie, and `DigitalVideoPlayer` instances that are responsible for playing the video in the movie. (See the discussions of the classes `DigitalAudioPlayer` and `DigitalVideoPlayer` for more information.)

Just like a `MoviePlayer` instance, an `InterleavedMoviePlayer` instance is a master player for all the slave players needed to play the movie. To play the movie, call the `play` method on the interleaved movie player; to pause the movie call `pause` on the interleaved movie player; to stop the movie call `stop` on the interleaved movie player, and so on. When an interleaved movie player receives a method such as `play`, `pause`, `stop` and so on, it calls that method on each of its slave players and thus the movie plays, pauses, stops and so on.

However, in the case of an `InterleavedMoviePlayer`, the data for the movie starts off as interleaved data in a `ByteStream` instance, which is created when the movie is imported into ScriptX.

To actually be played, sound data must be passed in a stream to a `DigitalAudioPlayer`, and video data must be passed in a stream to a `DigitalVideoPlayer`. How then does the data pass from the byte stream containing interleaved data to separate streams that are used as the media streams for `DigitalAudioPlayer` and `DigitalVideoPlayer` instances?

`DigitalAudioPlayer` and `DigitalVideoPlayer` instances both play the data in the stream in their `mediaStream` instance variable. However, the stream in a `mediaStream` instance variable is an object that itself has an input stream, which is stored in its `inputStream` instance variable. These input streams hold the actual data. Figure 6 shows the relationships through instance variables of an `InterleavedMoviePlayer`, its subplayers (stored in the `slaveClocks` instance variable), their media streams, and the media streams' input streams.

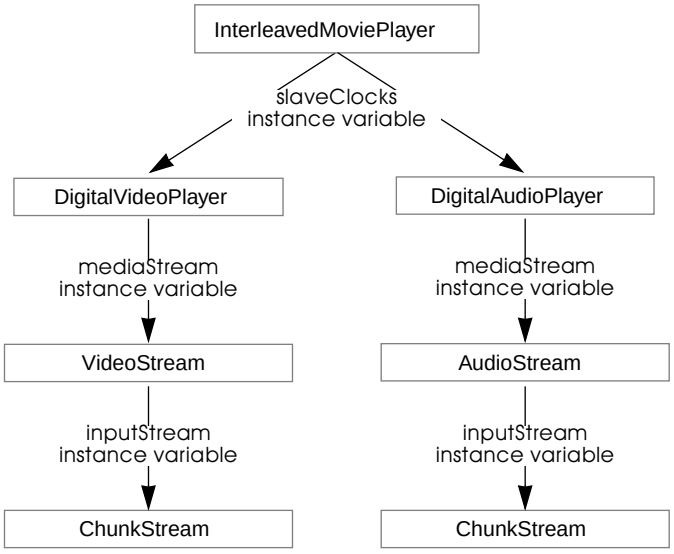


Figure 6: How interleaved movie players work

When a `VideoStream` or `AudioStream` participates in the playing of an interleaved movie, its input stream is a `ChunkStream` instance. The class `ChunkStream` is a specialized class of `Stream` that takes data from disjointed locations in one stream and turns it into separate continuous streams, as illustrated in Figure 7.

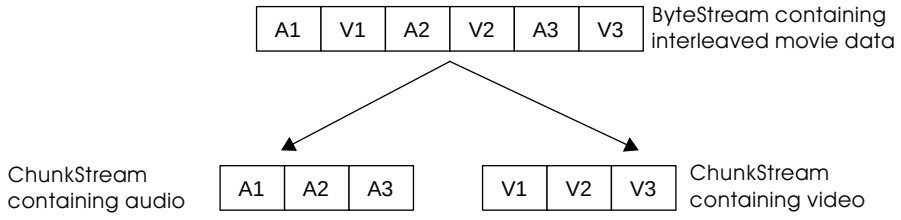


Figure 7: Chunk streams can be used to separate interleaved data

An interleaved movie player has a callback that transfers data from the bytestream containing the interleaved data to chunk streams, sending data for video to one chunk stream and data for sound to another.

As the slave players, such as the `DigitalAudioPlayer` and `DigitalVideoPlayer` instances play, they read the data out of the appropriate chunk streams (going through the intermediary media streams). As space becomes available in the chunk streams, the callback on the interleaved movie player transfers more data into the chunk streams.

Figure 8 combines Figure 6 and Figure 7 to show how interleaved data stored as a byte stream in the `interleavedStream` instance variable of the `InterleavedMoviePlayer` finds its way into separate chunk streams which are the player's media streams' input streams.

The callback on the interleaved movie player transfers data from the interleaved bytestream to the chunk streams ahead of the time that the slave players need to read the data from the chunk streams. When a slave player such as a digital audio player needs to play a frame, it can usually immediately get data from the chunk stream, thus reducing the access time since no seeking is done when the data is needed.

The callback works its way sequentially along the interleaved byte stream, transferring data to the chunk streams. If a chunk stream is still full of data when the callback on the interleaved movie player is ready to write to it, the callback waits for a while, then tries again to write to the chunk stream.

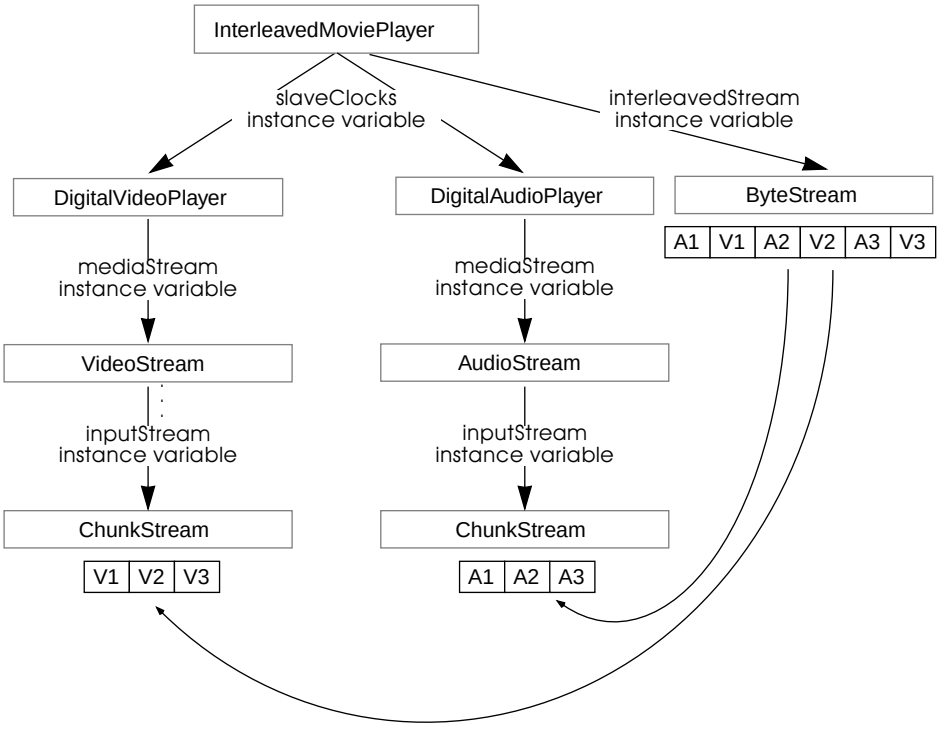


Figure 8: More on how interleaved movie players work

When you want to play an interleaved movie, you don't have to worry about how the data gets transferred from the bytestream to the chunk streams. You don't need to know anything about chunk streams. All you really need to know to play an interleaved movie is that you import the movie file to an `InterleavedMoviePlayer`, and then use the usual player methods to play and control the movie. However, there may be times when you want to fine tune the transference of data from the bytestream to the chunk streams to increase movie playback efficiency. In this case, you can modify instance variables and behavior of the chunk streams. See the discussion of the `ChunkStream` class for more details.

Creating and Initializing a New Instance

To create an `InterleavedMoviePlayer` instance, import a file containing a movie. The importing process automatically creates the `InterleavedMoviePlayer` instance and all the other players and streams needed to play the movie.

The following script shows how to create an instance of `InterleavedMoviePlayer` by importing a Quicktime file containing a digitized movie. The object `waltzMovie` can be used to play the movie imported from the file `waltz` if it was stored on the ScriptX startup directory.

```
waltzStream := getstream theStartDir "waltz" @readable
waltzMovie := importMedia theImportExportEngine waltzStream \
          @movie @quicktime @InterleavedMoviePlayer
```

This script shows an example of how to import a QuickTime file as an `InterleavedMoviePlayer`. When importing a QuickTime file as an interleaved movie player, you can specify optional `container` and `copydata` keyword arguments. The `container` argument indicates a title container in which to store the raw media for the imported movie. The `copydata` argument indicates whether or not to actually copy the imported movie into ScriptX or not. For more details of the arguments to the method `importMedia` on the global instance `theImportExportEngine`, please see either the "Media Stream Players" chapter in the *ScriptX Components Guide* or the chapter about importers in the *ScriptX Tools Guide*.

When calling the `importMedia` method to import a movie file into ScriptX as an interleaved movie, you can specify an optional `copyData` keyword argument to determine whether or not to copy the movie data into ScriptX. If you don't copy it in, you can play the movie directly from the file on the storage medium. In this case, a machine specific `ByteStream` object is created as the interleaved data stream, and if you want to save the interleaved movie player to a title container you must take some extra steps. See the "Importing Media" chapter in the *ScriptX Tools Guide* for more details.

After creating an `InterleavedMoviePlayer` instance by importing a movie, append it to a visible surface such as a `Window` to use as its "screen". Call its `play` method to start it playing.

Instance Variables

Inherited from `Collection`:

`bounded`	`maxSize`	`size`
`iteratorClass`	`minSize`	`uniformity`
`keyEqualComparator`	`mutable`	`uniformityClass`
`keyUniformity`	`mutableCopyClass`	`valueEqualComparator`
`keyUniformityClass`	`proprietored`	

Inherited from `IndirectCollection`:

`targetCollection`

Inherited from `Presenter`:

`presentedBy`	`subPresenters`	`target`

Inherited from `TwoDPresenter`:

`bBox`	`height`	`transform`
`boundary`	`IsImplicitlyDirect`	`width`
`clock`	`isTransparent`	`window`
`compositor`	`isVisible`	`x`
`direct`	`needsTickle`	`y`
`eventInterests`	`position`	`z`
`globalBoundary`	`stationary`	

globalTransform	target	
globalBoundary	position	z
globalRegion	stationary	

Inherited from TwoDMultiPresenter:

fill	stroke

Inherited from Clock:

callbacks	rate	ticks
effectiveRate	resolution	time
masterClock	scale	title
offset	slaveClocks	

Inherited from Player:

audioMuted	globalContrast	globalVolumeOffset
dataRate	globalHue	markerList
duration	globalPanOffset	status
globalBrightness	globalSaturation	videoBlanked

Inherited from MoviePlayer:

frameRate

The following instance variables are defined in InterleavedMoviePlayer:

interleavedStream

self.interleavedStream (read-write) ByteStream

Specifies the bytestream containing the interleaved movie data for the interleaved movie player *self*.

preRollLength

self.preRollLength (read-write) Number

When an InterleavedMoviePlayer is first prepared to play (by calling the playPrepare method), it reads some amount of data (typically a second's worth) off the storage device, deinterleaves it, and places it in the chunk streams. This is known as a "pre-roll" and is used to give the system some amount of leeway or threshold for error.

When you play an InterleavedMoviePlayer, it attempts to pass data to its media streams through their chunk streams at a rate that's equivalent to the amount consumed by the media streams. If everything goes as it should there should always be the pre-roll amount of data available in the chunk stream. If the InterleavedMoviePlayer slows down or is late in passing data, for example, if it has to do a seek on the CD, presentation can still go on temporarily, since there's data in the pipeline. The pre-roll amount is controlled by the preRollLength instance variable. (The default is 1 second).

transferRate

self.transferRate (read-write) Integer

Specifies the rate at which the player *self* expects to read data from the stream stored in its interleavedStream instance variable. The transfer rate is measured in bytes/second and defaults to 300K (that is, 300 * 1024) which is the transfer rate for a double speed CD drive.

Instance Methods

Inherited from Collection:

add	forEach	iterate
addMany	forEachBinding	localEqual
addToContents	getAll	map

chooseAll	getAny	merge
chooseOne	getKeyAll	pipe
chooseOneBinding	getKeyOne	prin
deleteAll	getMany	removeAll
deleteBindingAll	getOne	removeOne
deleteBindingOne	hasBinding	setAll
deleteKeyAll	hasKey	setOne
deleteKeyOne	intersects	size
deleteOne	isEmpty	
emptyOut	isMember	

Inherited from TwoDPresenter:

adjustClockMaster	hide	refresh
createInterestList	inside	show
draw	localToSurface	surfaceToLocal
getBoundaryInParent	notifyChanged	tickle

Inherited from TwoDPresenter:

adjustClockMaster	inside	show
createInterestList	localToSurface	surfaceToLocal
draw	notifyChanged	tickle
getBoundaryInParent	recalcRegion	
hide	refresh	

Inherited from Clock:

addPeriodicCallback	clockAdded	pause
addRateCallback	clockRemoved	resume
addScaleCallback	effectiveRateChanged	timeJumped
addTimeCallback	forEachSlave	waitTime
addTimeJumpCallback	isAppropriateClock	waitUntil

Inherited from Player:

addMarker	goToBegin	playPrepare
eject	goToEnd	playUnprepare
fastForward	goToMarkerFinish	playUntil
getMarker	goToMarkerStart	resume
getNextMarker	pause	rewind
getPreviousMarker	play	stop

Since an interleaved movie player is an indirect collection, you can also use any methods defined in the class specified by targetCollection. The target collection is typically an instance of Array, which inherits from Sequence, so the following instance methods are redirected to this player.

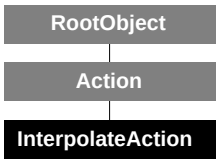
Accessible from LinearCollection:

chooseOneBackwards	findRange	getNthKey
chooseOrdOne	forEachBackwards	getOrdOne
deleteFirst	getFirst	getRange
deleteLast	getLast	localEqual
deleteNth	getMiddle	localLT
deleteRange	getNth	pop

Accessible from Sequence:

addFifth	moveBackward	setFourth
addFirst	moveForward	setLast
addFourth	moveToBack	setNth
addNth	moveToFront	setSecond
addSecond	prepend	setThird
addThird	prependNew	sort
append	setFifth	
appendNew	setFirst	

InterpolateAction

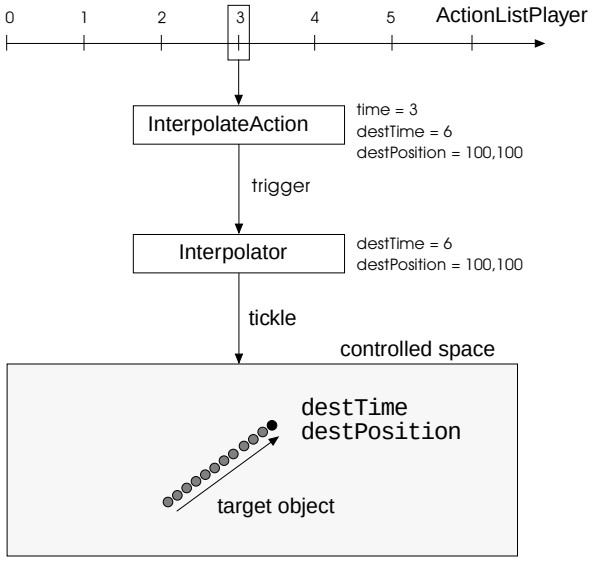


Class type: Core class (concrete)
Resides in: ScriptX and KMP executables
Inherits from: `Action`
Component: Animation

`InterpolateAction` class represents an action that determines the destination point and time for objects controlled by an `Interpolator` object. An interpolate action, along with the interpolator controller causes a smooth change in position. In contrast, `PathAction` causes an abrupt change (and does not require a controller). To be triggered, an instance of `InterpolateAction` needs to be added to the action list of an action list player, then the player needs to be played.

An `Interpolator` controller is required to move the object; this interpolator must be added to the action list player's list of targets. When the interpolate action triggers, all objects contained in the interpolator move toward the destination point. The targets to move must be in a space controlled by an `Interpolator` controller. (See the `Interpolator` class.)

As shown in the following figure, at a specified time in the action list player, the interpolate action triggers. The interpolate action's `trigger` method simply passes the destination point and time to the interpolator at the time specified by the `time` instance variable. The interpolator controller's `tickle` method then causes the target to move. To continue moving the object, another interpolate action could trigger at 6 ticks, updating the same interpolator with a new destination time and position.



Creating and Initializing a New Instance

The following script is an example of how to create a new instance of the `InterpolateAction` class:

```
myAction := new InterpolateAction \
          destPosition:(new Point x:50 y:50) \
          destTime:20 \
          targetNum:2 \
          time:10
```

The variable `myAction` holds an initialized instance of `InterpolateAction`. This instance specifies that, at the player's time of 10 ticks, the player's second target (`targetNum:2`) move smoothly from its position to arrive at the point 50,50 at 20 ticks.

To trigger this action, it first needs to be added to the action list of an action list player; then the player needs to be played. In addition, the target must be in a space that's controlled by an `Interpolator` controller.

init

`init` *self* [`destPosition:`*point*] [`destTime:`*integer*] [`targetNum:`*integer*] [`time:`*time*] ⇨ *(none)*

self	`InterpolateAction` object
`destPosition:`	`Point` object representing the destination position to which to move the target object
`destTime:`	`Integer` object representing the time in ticks at which the target object is to arrive

Superclass `Action` uses the following keywords:

`targetNum:`	`Integer` object indicating the position of the `Interpolator` object in the target list of the player
`time:`	`Integer` representing the time in ticks to trigger the action

Initializes the `InterpolateAction` object *self*, applying the values supplied with the arguments to the instance variables of the same name. At the time specified by `time`, this action will move the target specified by `targetNum` so that it will arrive at the destination `destPosition` at the time specified by `destTime`. Do not call `init` directly on an instance—it is automatically called by the `new` method.

If you omit an optional keyword, its default value is used. The defaults are:

```
destPosition:undefined
destTime:undefined
targetNum:0
time:0
```

Instance Variables

Inherited from `Action`:

`authorData` `targetNum` `time` `playOnly`

The following instance variables are defined in `InterpolateAction`:

destPosition

self`.destPosition` (read-write) `Point`

Specifies the destination point in absolute coordinates toward which the target object should start heading. This destination point is held by the `InterpolateAction` object *self*. The x-y coordinates are in the space's unit of measurement (pixels) and relative to the space's origin.

destTime

self.`destTime` (read-write) `Integer`

Specifies the destination time, in ticks, at which the target object should reach the destination. This destination time is held by the `InterpolateAction` object *self*. The time is measure in the time of the `ActionListPlayer` object.

Instance Methods

Inherited from `Action`:

`trigger`

The following instance method is defined in `InterpolateAction`:

trigger

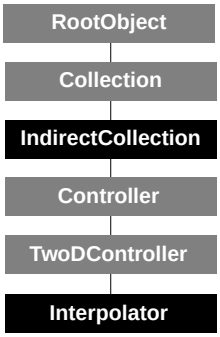
`trigger` *self interpolator player* ⇨ *target*

self	`InterpolateAction` object
interpolator	`Interpolator` object
player	`ActionListPlayer` object

Causes the interpolate action *self* to pass its `destTime` and `destPosition` values to the `Interpolator` controller specified by *interpolator*. This method is called by an `ActionListPlayer` object at the time specified by the `time` instance variable. This action list player is passed in as the third argument *player*.

When the action list player calls trigger, the value for *interpolator* is automatically determined by taking the `targetNum` instance variable and finding the object in the corresponding slot in the player's target list. If the `targetNum` instance variable is out of range (that is, less than 1 or greater than the size of the target list), the value of the *interpolator* is `empty`. Also, if there is no *interpolator* in a certain slot in the target list yet, the value of *interpolator* is `undefined`.

Interpolator



Class type: Core class (concrete)
Resides in: ScriptX and KMP executables
Inherits from: `TwoDController`
Component: Controllers

The `Interpolator` class is a controller that moves one 2D presenter objects in a straight line, smoothly to a specified destination point at a specified destination time. The destination point and time is determined by an instance of the `InterpolateAction` class. The object to be moved is the first presenter in the interpolator's space that has also been added to the interpolator, as described below. For this interpolator to work, the presenter must also be added to the targets of an action list player, and the player must be played.

Note – An interpolator can move only one 2D presenter object.

The interpolator controller's `tickle` method actually causes the presenter to move. The instance of `InterpolateAction` merely specifies where and when the presenters will arrive.

An `Interpolator` object is a collection containing the `TwoDPresenter` objects it can control—the interpolator controls only the first object in its collection. This presenter must also be in the space that the controller is controlling. Presenters are either automatically or manually added to the interpolator controller, according to the `wholeSpace` instance variable. If `wholeSpace` is `false`, you can use the methods defined by `Collection` to add and remove objects from the controller. To ensure that only 2D presenters are added to an interpolator controller, the `protocols` instance variable is set to the `TwoDPresenter` class. See the `Controller` class for descriptions of `wholeSpace`, `protocols`, and other general properties of controllers.

Creating and Initializing a New Instance

The following script is an example of how to create a new instance of the `Interpolator` class, after first creating its control space:

```
mySpace := new TwoDSpace boundary:(new Rect x2:200 y2:200)
       function myFunc self target clock -> \
       (self.target.width := 2 * self.target.width)

myInterpolator := new Interpolator \
              clock:mySpace.clock \
              space:mySpace
```

The variable `myInterpolator` contains the initialized interpolator. This interpolator causes any targets added to the interpolator to move smoothly to their destination positions, arriving at their destination times. The action list player `myALP` controls the rate of movement. The `new` method uses the keywords defined in `init`.

The interpolator uses the clock supplied with the `clock` keyword to find out how close it is to `destTime` of the `InterpolateAction` object.

init

`init` *self* [`clock:`*clock*] [`script:`*function*] [`space:`*space*] [`wholeSpace:`*boolean*] [`enabled:`*boolean*] [`targetCollection:`*sequence*]

⇨ *(none)*

self	`Interpolator` object
`clock:`	`Clock` object
`script:`	A function to run at the destination time

The superclass `Controller` uses the following keywords:

`space:`	`Space` object containing the targets to be moved
`wholeSpace:`	`Boolean` object
`enabled:`	`Boolean` object

The superclass `TwoDController` uses the following keyword:

`targetCollection:`	`Sequence` object (use carefully)

Initializes the `Interpolator` object *self*, applying the keyword arguments to instance variables of the same name, and the clock supplied with the `clock:` keyword is the clock against which the `InterpolateAction` object's destination time is determined. Use discretion in changing the target collection; for more information, see the definition of the `TwoDController` class. Do not call `init` directly on an instance—it is automatically called by the `new` method.

If you omit an optional keyword, its default value is used. The defaults are:

```
clock:undefined  (If space is defined, the space's clock is used)
space:undefined
wholeSpace:false
enabled:true
targetCollection:(new Array initialSize:1 growable:true)
```

Class Methods

Inherited from `Collection`:

`pipe`

Instance Variables

Inherited from `Collection`:

`bounded`	`maxSize`	`size`
`iteratorClass`	`minSize`	`uniformity`
`keyEqualComparator`	`mutable`	`uniformityClass`
`keyUniformity`	`mutableCopyClass`	`valueEqualComparator`
`keyUniformityClass`	`proprietored`	

Inherited from `IndirectCollection`:

`targetCollection`

Inherited from `Controller`:

`enabled`	`space`	`wholeSpace`
`protocols`		

The following instance variables are defined in `Interpolator`:

protocols (Controller)

self`.protocols` (read-write) `Array`

This instance variable initially contains the class `TwoDPresenter` for the interpolator *self*. This means that any object added to an interpolator controller must have `TwoDPresenter` as one of its superclasses. See the `Controller` class for further description of this instance variable.

script

self`.script` (read-write) *(function)*

Specifies the function that runs at the destination time of each `InterpolateAction` object controlled by the interpolator *self*. The interpolator calls this function with the following arguments, where *self* is the interpolator, *target* is the object to be moved, and *clock* is the clock of the interpolator's space (that is, whatever clock was given to the interpolator when it was created).

`myFunc` *self target clock*

Although any global function, anonymous function, or method can be assigned to `script`, there are differences in how different classes of functions are dispatched. For information on functions and function dispatch, see the "Object System Kernel" chapter of the *ScriptX Components Guide*.

Instance Methods

Inherited from `Collection`:

`add`	`forEach`	`iterate`
`addMany`	`forEachBinding`	`localEqual`
`addToContents`	`getAll`	`map`
`chooseAll`	`getAny`	`merge`
`chooseOne`	`getKeyAll`	`pipe`
`chooseOneBinding`	`getKeyOne`	`prin`
`deleteAll`	`getMany`	`removeAll`
`deleteBindingAll`	`getOne`	`removeOne`
`deleteBindingOne`	`hasBinding`	`setAll`
`deleteKeyAll`	`hasKey`	`setOne`
`deleteKeyOne`	`intersects`	`size`
`deleteOne`	`isEmpty`	
`emptyOut`	`isMember`	

Inherited from `IndirectCollection`:

`isAppropriateObject`	`objectAdded`	`objectRemoved`

Inherited from `Controller`:

`isAppropriateObject`	`tickle`

Since an `Interpolator` controller is an indirect collection, you can also use any methods defined in the class specified by `targetCollection`. The target collection is typically an instance of `Array`, which inherits from `Sequence`, so the following instance methods are redirected to this controller.

Accessible from `LinearCollection`:

`chooseOneBackwards`	`findRange`	`getNthKey`
`chooseOrdOne`	`forEachBackwards`	`getOrdOne`

deleteFirst	getFirst	getRange
deleteLast	getLast	localEqual
deleteNth	getMiddle	localLT
deleteRange	getNth	pop

Accessible from Sequence:

addFifth	moveBackward	setFourth
addFirst	moveForward	setLast
addFourth	moveToBack	setNth
addNth	moveToFront	setSecond
addSecond	prepend	setThird
addThird	prependNew	sort
append	setFifth	
appendNew	setFirst	

The following instance methods are defined in Interpolator:

setDestination

`setDestination` *self destPoint destTime* ⇨ `Interpolator`

self	`Interpolator` object
destPoint	`Point` object
destTime	`Integer` object

Sets the destination point *destPoint* and time *destTime* any target controlled by the interpolator *self* will be moved to.

tickle (Controller)

`tickle` *self clock* ⇨ *(none)*

self	`Interpolator` object
clock	`Clock` object of the space being controlled

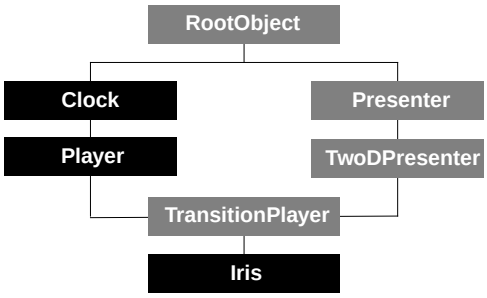
For each instance of `InterpolateAction`, this method causes the target presenter to move smoothly to its destination time and position. This method creates smooth motion by changing the presenter's `x` and `y` instance variables a small increment of distance each clock tick.

A callback calls this method on the interpolator *self*, supplying the space's clock as the value for *clock*. The callback calls this method once every tick of the space's clock.

Notice that the `Interpolator` actually moves the presenter; the instance of `InterpolateAction` merely specifies which presenter to move, and where and when it will arrive.

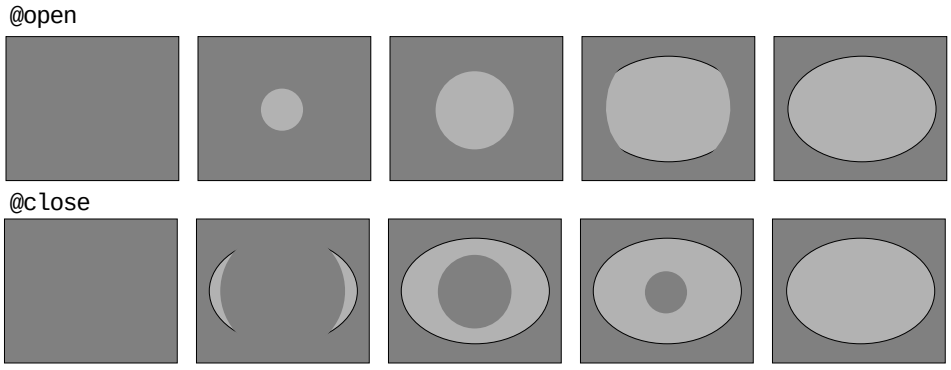
For further details, refer to the section "The Ticklish Protocol" in the "Controllers" chapter in the *ScriptX Components Guide*.

Iris



Class type: Core class (concrete)
Resides in: ScriptX and KMP executables
Inherits from: `TransitionPlayer`
Component: Transitions

The `Iris` class provides a circular visual effect similar to the iris of a camera opening, as shown below. You set the iris to open one of two different ways by setting the value of `direction`, an instance variable defined by `TransitionPlayer`, to either `@open` or `@close`.



Creating and Initializing a New Instance

The following script is an example of how to create a new instance of the `Iris` class:

```
myTransition := new Iris \
                duration:60 \
                direction:@open \
                target:myShape \
                useOffscreen:true
```

The variable `myTransition` contains the initialized transition. When you play `myTransition`, a circular iris opens, causing the presenter `myShape` to appear over a period of 60 ticks. The transition player is set to use an offscreen cache to draw the presenter more efficiently.

You determine in which space the transition will take effect in by adding this instance to that space. Then, when you play the transition player, `myShape` is "transitioned" into that space.

The `new` method uses the keywords defined in `init`.

init

`init` *self* [`duration:`*integer*] [`direction:`*name*]
[`movingTarget:`*boolean*] [`useOffscreen:`*boolean*] [`target:`*twoDPresenter*]
[`boundary:`*stencil*] [`masterClock:`*clock*] [`scale:` *integer*] ⇨ *(none)*

This method is inherited from `TransitionPlayer` with no change in keywords—refer to that class for details. Do not call `init` directly on an instance—it is automatically called by the `new` method.

Instance Variables

Inherited from `Presenter`:

`presentedBy`	`subPresenters`	`target`

Inherited from `TwoDPresenter`:

`bBox`	`height`	`transform`
`boundary`	`IsImplicitlyDirect`	`width`
`clock`	`isTransparent`	`window`
`compositor`	`isVisible`	`x`
`direct`	`needsTickle`	`y`
`eventInterests`	`position`	`z`
`globalBoundary`	`stationary`	
`globalTransform`	`target`	

Inherited from `Clock`:

`callbacks`	`rate`	`ticks`
`effectiveRate`	`resolution`	`time`
`masterClock`	`scale`	`title`
`offset`	`slaveClocks`	

Inherited from `Player`:

`audioMuted`	`globalContrast`	`globalVolumeOffset`
`dataRate`	`globalHue`	`markerList`
`duration`	`globalPanOffset`	`status`
`globalBrightness`	`globalSaturation`	`videoBlanked`

Inherited from `TransitionPlayer`:

`autoSplice`	`direction`	`movingTarget`
`backgroundBrush`	`duration`	`target`
`cachedTarget`	`frame`	`useOffscreen`

The following instance variables are defined in `Iris`:

direction (TransitionPlayer)

self`.direction` (read-write) `NameClass`

Specifies the direction in which the iris transition *self* should be applied. Possible values are `@open` and `@close`.

Instance Methods

Inherited from `TwoDPresenter`:

`adjustClockMaster`	`inside`	`show`
`createInterestList`	`localToSurface`	`surfaceToLocal`
`draw`	`notifyChanged`	`tickle`
`getBoundaryInParent`	`recalcRegion`	
`hide`	`refresh`	

Inherited from `Clock`:

`addPeriodicCallback`	`clockAdded`	`pause`
`addRateCallback`	`clockRemoved`	`resume`
`addScaleCallback`	`effectiveRateChanged`	`timeJumped`
`addTimeCallback`	`forEachSlave`	`waitTime`
`addTimeJumpCallback`	`isAppropriateClock`	`waitUntil`

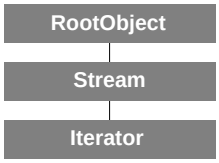
Inherited from `Player`:

`addMarker`	`goToBegin`	`playPrepare`
`eject`	`goToEnd`	`playUnprepare`
`fastForward`	`goToMarkerFinish`	`playUntil`
`getMarker`	`goToMarkerStart`	`resume`
`getNextMarker`	`pause`	`rewind`
`getPreviousMarker`	`play`	`stop`

Inherited from `TransitionPlayer`:

`playPrepare`

Iterator



Class type: Core class (abstract)
Resides in: ScriptX and KMP executables
Inherits from: `Stream`
Component: Collections

`Iterator` is an abstract class that defines the protocol common to all iterators.

Iterators are the mechanism for stepping through every element of a collection. Every collection must respond to the `iterate` method by returning an iterator that, at the least, is capable of retrieving each element of the collection via sequential calls to the `next` method.

An iterator has a `cursor` method that indicates numerically the current position in the collection. The collection that is being processed by the iterator can be retrieved using the iterator's `source` instance variable. A newly created iterator's cursor value is 0, and the iterator is considered to be positioned before the first item in the collection. When you call `next` on the iterator, it takes one step, and we say that it now "points to" an item in the collection. The iterator's `key` and `value` instance variables are used to retrieve the object from the collection. The `value` instance variable can also be used to alter the collection, by replacing the item pointed to with a new object.

As an iterator is a kind of a stream, its methods defined in the `Stream` class (`isAtFront`, `cursor` and `isPastEnd`) are shown in the figure. Like streams, many iterators support the concept of "seeking" or repositioning the cursor.

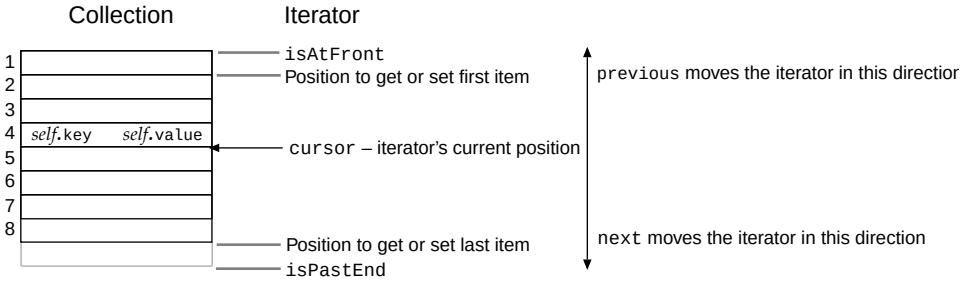


Figure 9: An iterator's cursor points to the current item in its collection.

Just because a collection is mutable does not mean that its iterator will allow you to modify it. For example, If you tried to write the value 6 to the sorted array [1,2,3,4,5] whose iterator was positioned at element 3, you might end up with [1,2,6,4,5], which would invalidate the sorted property of the array.

Once you have created an iterator on a collection, the iterator is valid as long as you call methods on the iterator. All iterator behavior becomes undefined once you modify its collection "behind the back" of the iterator. That is, modifying any item by calling a method (such as `setNth`) directly on the collection invalidates the iterator; to continue iterating, a new iterator must be created (such as with `iterate`).

For more details, see the "Collections" chapter in the *ScriptX Components Guide*.

Note – When you create an iterator (or call `atFront` on it), the cursor is positioned before the first item of the collection. At this position, getting or setting the key or value (for example, using the `key` or `value` instance variables) reports an error. You must call `next` (or otherwise move the cursor into the collection) to get a value. This setup enables the syntax for iteration to gracefully handle an empty collection. You must also call `next` after deleting an item via an iterator (the `excise` method).

Creating and Initializing a New Instance

Because `Iterator` is an abstract class, you cannot create an instance of it. The easiest way to create an iterator is to call `iterate` on a collection. This will create the correct iterator subclass to iterate over the collection. However, you can also instantiate any concrete subclass of `Iterator` by calling `new` on that subclass.

init

`init` *self* `collection:`*collection* ⇨ *(none)*

self	`Iterator` object
`collection:`	`Collection` object

Initializes the `Iterator` object *self* to iterate over the collection supplied with the `collection` keyword and sets its cursor to 0. Do not call `init` directly on an instance—it is automatically called by the `new` method.

Instance Variables

key

self`.key` (read-only) *(object)*

Specifies the key at the current cursor position of the iterator *self* (as shown in Figure 9). This key is updated every time the cursor moves. If the cursor is at the front of the collection, `key` reports `errorBound`.

source

self`.source` (read-only) `Collection`

Specifies the collection object that the iterator *self* iterates over.

value

self`.value` (read-write or read-only) *(object)*

Specifies the value at the current cursor position of the iterator *self* (as shown in Figure 9). This value is updated every time the cursor moves. If the cursor is at the front of the collection, `value` reports `errorBound`.

This instance variable is read-write or read-only depending on the particular collection being iterated over—if the values in the collection can be set, then you can write to the `value` instance variable to change its value.

Instance Methods

Inherited from `Stream`:

`cursor`	`next`	`seekFromStart`
`flush`	`plug`	`setStreamLength`

`isAtFront`	`previous`	`streamLength`
`isPastEnd`	`read`	`write`
`isReadable`	`readReady`	`writeReady`
`isSeekable`	`seekFromCursor`	
`isWritable`	`seekFromEnd`	

The following instance methods are defined in `Iterator`:

excise

`excise` *self* ⇨ *(none)*

Deletes the key-value pair from the source collection pointed to by the cursor of the iterator *self* (as shown in Figure 9). After an excise, the iterator is in an undefined state; you must call a seek method (like `seekFromStart`) or the `next` method to move the cursor into a known position. Note that the `previous` and `seekFromCursor` methods should not be used directly after an excise. The `excise` method does not delete the key object and value object themselves (unless normal garbage collection applies); it just removes them from the collection.

The exception `iteratorBoundary` is reported if `excise` is called when the iterator's `value` and `key` instance variables are not valid (for example, if the cursor is ahead of the first item, or if the cursor is more than one position past the end). The `excise` method reports `immutable` if the source collection is immutable, or `bounded` if the source collection would shrink below `minSize`.

remainder

`remainder` *self* ⇨ `Collection`

Returns a collection of the items from the source collection between the cursor of the iterator *self* and the end of the source collection.

seekKey

`seekKey` *self key* ⇨ `Boolean`

self	`Iterator` object
key	Any object

Moves the cursor of the iterator *self* to the first occurrence of the given *key* in the source collection of iterator *self*. Returns `true` if the key was found, or `false` if it was not found (in which case the iterator is left past the end of the collection). If the iterator is not seekable, `seekKey` reports `notSeekable`. All built-in collections are seekable. You can find out if any iterator is seekable by calling `isSeekable` (from the `Stream` class) on it.

seekValue

`seekValue` *self value* ⇨ `Boolean`

self	`Iterator` object
value	Any object

Moves the cursor of the iterator *self* to the first occurrence of the given *value* in the source collection. Returns `true` if the value was found, or `false` if it was not found (in which case the iterator is left past the end of the collection). If the iterator *self* is not seekable, `seekValue` reports `notSeekable`. All built-in collections are seekable. You can find out if any iterator is seekable by calling `isSeekable` (from the `Stream` class) on it.

Subclasses Must Implement

Subclasses of `Iterator` must implement the following:

`source`	(`Iterator`)
`cursor`	(`Stream`)
`next`	(`Stream`)
`excise`	(`Iterator`)
`isPastEnd`	(`Stream`)
`valueGetter`	(`Iterator`)
`valueSetter`	(`Iterator`) but only if the values are settable

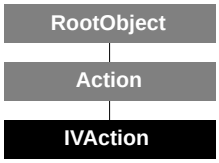
All subclasses for which the source collection is keyed:

`keyGetter`	(`Iterator`)

All subclasses which are seekable:

`previous`	(`Stream`)
`seekFromStart`	(`Stream`)
`seekFromEnd`	(`Stream`)
`seekKey`	(`Iterator`)

IVAction



Class type: Core class (concrete)
Resides in: ScriptX and KMP executables
Inherits from: `Action`
Component: Animation

`IVAction` class represents an action that will change a certain instance variable in the target object to a specified value at a specified time. This is done by specifying the setter function for the instance variable and its new value. For example, you can change the width of a 2D presenter to 300 by specifying `widthSetter` as the setter function and `value` to 300.

Creating and Initializing a New Instance

The following script is an example of how to create a new instance of the `IVAction` class:

```
myAction:= new IVAction \
       setterFunction:widthSetter \
       value:300 \
       targetNum:2 \
       time:10
```

The variable `myAction` holds an initialized instance of `IVAction`. This instance specifies that, at the player's time of 10 ticks, the player's second target (`targetNum:2`) has its instance variable whose setter function is named `widthSetter` change to the value 300. The `new` method uses the keywords defined in `init`.

init

`init` *self* [`setterFunction:`*generic*] [`value:`*object*]
[`targetNum:`*integer*] [`time:`*integer*] ⇨ *(none)*

self	`IVAction` object
`setterFunction:`	`Generic` object for the setter function of the instance variable to change
`value:`	Any object. This is the value to put in the instance variable

Superclass `Action` uses the following keywords:

`targetNum:`	`Integer` indicating which object in the target list of the player to apply the action to
`time:`	`Integer` object representing the time in ticks to trigger the action

Initializes the `IVAction` object *self*, applying the values supplied with the keywords to the instance variables of the same name. At the specified `time`, this action causes the generic specified by `setterFunction` to be called with the given `value` on the `targetNum` target object. Do not call `init` directly on an instance—it is automatically called by the `new` method.

If you omit an optional keyword, its default is used. The defaults are:

```
setterFunction:undefined
value:undefined
targetNum:0
```

```
time:0
```

Instance Variables

Inherited from `Action`:

`authorData` `targetNum` `time`
`playOnly`

The following instance variables are defined in `IVAction`:

setterFunction

self`.setterFunction` (read-write) `Generic`

Specifies the method (actually, the generic) that sets the instance variable you want to change in the target object. For example, the setter function for the width instance variable for a 2D presenter would be set as follows:

```
myAction.setterFunction := widthSetter
```

value

self`.value` (read-write) *(object)*

Specifies the value to change the instance variable to.

Instance Methods

Inherited from `Action`:

`trigger`

The following instance method is defined in `IVAction`:

trigger (Action)

`trigger` *self target player* ⇨ *target*

self	`IVAction` object
target	Any object
player	`ActionListPlayer` object

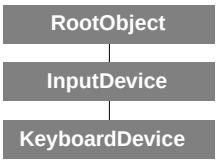
Causes the iv action *self* to call the generic specified by `setterFunction` with the argument given by the `value` instance variable, on the `targetNum` target object, as follows:

self`.setterFunction` *player*`.targets[`*self*`.targetNum]` *self*`.value`

This method is called by an `ActionListPlayer` object at the time specified by the `time` instance variable. This action list player is passed in as the third argument *player*.

When the action list player calls trigger, the value for *target* is automatically determined by taking the `targetNum` instance variable and finding the object in the corresponding slot in the player's target list. If the `targetNum` instance variable is out of range (that is, less than 1 or greater than the size of the target list), the value of the *target* is `empty`. Also, if there is no *target* in a certain slot in the target list yet, the value of *target* is `undefined`.

KeyboardDevice



Class type: Core class (abstract)
Resides in: ScriptX and KMP executables
Inherits from: `InputDevice`
Component: Input Devices

`KeyboardDevice` is an abstract class that acts as an interface between the ScriptX Player and the native operating system. A `KeyboardDevice` instance—that is, an object that belongs to a subclass of `KeyboardDevice`—receives keyboard events from the device driver and sends them to other classes as ScriptX events.

Each key has an associated key string and key code, as shown in the table below. The key name is the same as the key string, but coerced to `NameClass`. When an instance of a `KeyboardDevice` subclass generates a keyboard event, it fills in the `keycode` instance variable, defined by `KeyboardEvent`, to identify the key that was pressed or released.

Key codes are divided into two groups: Unicode and ScriptX.

- Standard Unicode characters are represented by positive key codes (including zero). For characters with key code 32 or greater, the character is the key name. For characters with key code between 0 and 32, the key name is not well defined.
- ScriptX "action keys" and modifier keys are represented by negative key codes. Action keys include functions keys, directional keys, and the keys on a numeric keypad. They have the key strings shown in Table 7.

ScriptX supports Unicode/ISO 10646 characters. Unicode is a standard set of 65,536 characters, presenting a wide range of characters, glyphs and symbols. ISO 10646 is a method variable-length encoding standard for Unicode characters that supports a variety of orderings on the standard Unicode character set. For a description of Unicode and ISO 10646, see the "Text and Fonts" chapter of the *ScriptX Components Guide*.

In the table that follows, there are gaps in the key codes, which are reserved for possible future use. Number and function keys are arranged in ascending order, so that `f2 = f1 + 1` and `numpad2 = numpad + 1`.

Table 7: Key codes in ScriptX

Key String	Key Code	Key String	Key Code	Key String	Key Code
Function Keys		Numeric Keypad		Directional Keypad	
`f1`	`-19`	`numpad0`	`-29`	`up`	`-40`
`f2`	`-18`	`numpad1`	`-28`	`down`	`-41`
`f3`	`-17`	`numpad2`	`-27`	`left`	`-42`
`f4`	`-16`	`numpad3`	`-26`	`right`	`-43`
`f5`	`-15`	`numpad4`	`-25`	`home`	`-44`
`f6`	`-14`	`numpad5`	`-24`	`end`	`-45`
`f7`	`-13`	`numpad6`	`-23`	`pageUp`	`-46`
`f8`	`-12`	`numpad7`	`-22`	`pageDown`	`-47`

Table 7: Key codes in ScriptX *(Continued)*

Key String	Key Code
f9	–11
f10	–10
f11	–9
f12	–8
f13	–7
f14	–6
f15	–5
Modifier Keys	
shift	–80
control	–81
alt	–82
command	–83
capLock	–90
numLock	–91
scrollLock	–92

Key String	Key Code
numpad8	–21
numpad9	–20
multiply	–30
divide	–31
add	–32
subtract	–33
decimal	–34
numEnter	–35
equals	–36

Key String	Key Code
insert	–48
delete	–49
Miscellaneous Keys	
cancel	–60
backspace	–61
tab	–62
enter	–63
escape	–64
pause	–65
snapshot	–66
clear	–67

Creating and Initializing a New Instance

Normally, you cannot invoke `new` on an abstract class. However, in the case of the `KeyboardDevice` class, the system allows you to do so. In the current release of ScriptX, this creates an instance of `PhysicalKeyboard`.

```
myKeyboard := new KeyboardDevice \
    deviceID:1 \
    enabled:true \
    autoRepeat:true
```

The variable `myKeyboard` contains an initialized instance of a subclass of `KeyboardDevice` (`PhysicalKeyboard` in Version 1.0 of the Kaleida Media Player). The `new` method calls the `init` method defined and Version 1.1 by `KeyboardDevice` and its superclasses. These `init` methods apply several keyword arguments.

init

`init` *self* [`autoRepeat:`*boolean*] [`deviceID:`*integer*] [`enabled:`*boolean*] ⇨ *(none)*

self	`KeyboardDevice` object
`autoRepeat:`	`Boolean` object

Superclass `InputDevice` uses the following keywords:

`deviceID:`	`Integer` object
`enabled:`	`Boolean` object

Initializes the `KeyboardDevice` object *self*, applying the arguments as follows: the `autoRepeat` keyword sets an initial value for the instance variable of the same name. The `deviceID` keyword specifies an integer ID number that is unique to a particular subclass of `InputDevice`. It allows a title to request a particular keyboard if more than one keyboard is attached to the system. By default, the Kaleida Media Player normally

assigns the first available device ID. The `enabled` value activates the device if set to `true`. Do not call `init` directly on an instance—it is automatically called by the `new` method.

If you omit an optional keyword, its default value is used. The defaults are:

```
autoRepeat:false
deviceID:1
enabled:true
```

Class Methods

Inherited from `InputDevice`:

`getDeviceFromList`

Instance Variables

Inherited from `InputDevice`:

`deviceID` `focusable` `focusManager`
`enabled`

The following instance variables are defined in `KeyboardDevice`:

autoRepeat

self`.autoRepeat` (read-write) `Boolean`

If `autoRepeat` is `true` and the keyboard *self* has this capability, then the device will generate events automatically when a key is held down. Setting `autoRepeat` to `false` guarantees that key repeats are not generated. Setting this instance variable to `true` does not guarantee that key repeats are generated—key repeats depend on the capabilities of the underlying operating system and hardware.

focusManager

self`.focusManager` (read-only) `KeyboardFocusManager`

Specifies the keyboard focus manager that is managing focus on the device *self*.

keyModifiers

self`.keyModifiers` (read-only) `Array`

Specifies the current state of the modifier keys, returning a list of active keys as an `Array` object. The two kinds of modifier keys are called shift and state keys. A shift key is active if it is being pressed; a state key is active if it is currently toggled on. Possible values are `@shift`, `@control`, `@alt`. `@command`, `@capLock`, `@numLock`, and `@scrollLock`.

```
myKbd.keyModifiers -- with shift and control keys down
⇨  #(@shift, @control).
```

Instance Methods

existKey

`existKey` *self keyName* ⇨ `Boolean`

self	`KeyboardDevice` object
keyName	`NameClass` or `Integer` object

Returns `true` if the keyboard *self* supports the given key name. `KeyName` may be a key name taken from the keycode list for keys that ScriptX defines, or an integer for standard Unicode keys. If `true` is returned, then the keyboard device is capable of generating that key. This method must be implemented by all subclasses of `KeyboardDevice`. To specify a key name for a key, precede the name you find in Table 7 with `@`.

```
existKey kbrd @numLock -- returns true if kbrd supports the NumLock key
```

getKeyName

`getKeyName` *self keyCode* ⇨`NameClass`

self	`KeyboardDevice` object
keyCode	`Integer` object

Returns a `NameClass` object for the given *keyCode* for the keyboard device *self*, where *keyCode* is one of the key codes defined by ScriptX. Negative key codes are defined by ScriptX (see Table 7) and return the key name for that key. Key codes greater than 0 return `undefined`. An invalid or undefined key code also returns `undefined`.

getKeyString

`getKeyString` *self keyCode* ⇨ `String`

self	`KeyboardDevice` object
keyCode	`Integer` object

Returns a string that contains the name for the given *keyCode* for the keyboard device *self*. Unicode characters with key code ≥ 32 will return that character as a string. Key codes between 0 and 31 are not well defined. Negative key codes are defined by ScriptX (see Table 7) and return the key string for that key. An invalid or undefined key code will return `undefined`. Note that a key name is a `NameClass` object, and has a corresponding entry in the system name table. A key string, unlike a key name, is not interned as a `NameClass` object.

isModifierActive

`isModifierActive` *self keyName* ⇨`Boolean`

self	`KeyboardDevice` object
keyName	`NameClass` object representing a modifier key.

Returns `true` or `false`, depending on whether the modifier key *keyName* is active on the `KeyboardDevice` object *self*. See the `keyModifiers` instance variable for a list of possible values.

isModifierInactive

`isModifierInactive` *self keyName* ⇨`Boolean`

self	`KeyboardDevice` object
keyName	`NameClass` object representing a modifier key.

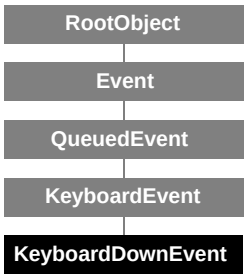
Returns `true` or `false`, depending on whether the modifier key *keyName* is inactive on the `KeyboardDevice` object *self*. See the `keyModifiers` instance variable for a list of possible values.

Subclasses must implement

Subclasses must implement the following methods:

`existKey`

KeyboardDownEvent



Class type: Core class (concrete)
Resides in: ScriptX and KMP executables
Inherits from: `KeyboardEvent`
Component: Events

The `KeyboardDownEvent` class represents the pressing of a key on a keyboard by a user. Each occurrence is represented by a different instance.

Some systems represent the gesture of continually pressing a key as a type of event that is separate and distinct from a single keyboard-down event. In ScriptX, the core classes define only two concrete classes of keyboard events: `KeyboardDownEvent` and `KeyboardUpEvent`. A `KeyboardDevice` object generates a series of `KeyboardDownEvent` instances if a key is pressed continually. (Some hardware keyboards may not support this behavior.) For more information, see the class `KeyboardDevice`.

Creating and Initializing a New Instance

The following script is an example of how to create a new instance of the `KeyboardDownEvent` class:

```
myKeyDown := new KeyboardDownEvent
```

The variable `myKeyDown` contains the initialized instance. To use `myKeyDown` as an interest, you must then specify values for the `device` and `eventReceiver` instance variables. The `new` method takes no arguments. The class `KeyboardDownEvent` has no `init` method—its initialization is inherited from its superclasses.

Class Variables

Inherited from `Event`:

`interests` `numInterests`

Inherited from `QueuedEvent`:

`dispatchQueue`

Class Methods

Inherited from `Event`:

`acquireQueueFromPool` `relinquishQueueToPool`
`broadcastDispatch` `signalDispatch`

Instance Variables

Inherited from Event:

advertised	eventReceiver	timeStamp
authorData	matchedInterest	
device	priority	

Inherited from QueuedEvent:

secondaryDispatchStyle	secondaryRejectable

Inherited from KeyboardEvent:

keyCode	maxKeyCode	minKeyCode
keyModifiers		

Instance Methods

Inherited from Event:

accept	isSatisfiedBy	sendToQueue
acquireRejectQueue	reject	signal
addEventInterest	relinquishRejectQueue	
broadcast	removeEventInterest	

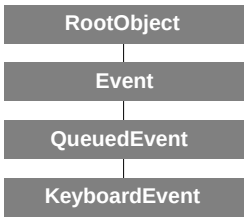
Inherited from QueuedEvent:

broadcast	secondarySignal	signal
secondaryBroadcast		

Inherited from KeyboardEvent:

broadcast	isModifierInactive	isSatisfiedBy
isModifierActive		

KeyboardEvent



Class type: Core class (abstract)
Resides in: ScriptX and KMP executables
Inherits from: `QueuedEvent`
Component: Events

`KeyboardEvent` is a class that represents any kind of keyboard event. Its concrete subclasses, `KeyboardDownEvent` and `KeyboardUpEvent`, represent particular kinds of keyboard events. An instance of one of these subclasses holds the event state:

- Which key was pressed or released
- Which modifier keys were pressed (Shift, Control, Alt, Command, CapLock, NumLock, or ScrollLock) while the key was pressed or released

Refer to `FocusEvent` for information on how presenters can gain focus for keyboard events.

Note – The following two phrases are used below to refer to the type of event:
- "Interest-only" means the instance *self* is used to express an event interest.
- "Event-only" means the instance *self* holds an actual user event.

Creating and Initializing a New Instance

Since `KeyboardEvent` is an abstract class, it is never instantiated, however it has an `init` method that is inherited by its subclasses. Its `init` method has no keyword arguments.

init

`init` *self* ⇨ *(none)*

self `KeyboardEvent` object

Initializes the `KeyboardEvent` object *self*.

Class Variables

Inherited from `Event`:

`interests` `numInterests`

Inherited from `QueuedEvent`:

`dispatchQueue`

Class Methods

Inherited from `Event`:

`acquireQueueFromPool` `relinquishQueueToPool`

broadcastDispatch signalDispatch

Instance Variables

Inherited from Event:

advertised eventReceiver timeStamp
authorData matchedInterest
device priority

Inherited from QueuedEvent:

secondaryDispatchStyle secondaryRejectable

The following instance variables are defined in KeyboardEvent:

keyCode

self.keyCode (read-write) Integer

Event-only instance variable. Specifies the key that was pressed or released. For more information, see the class KeyboardDevice.

keyModifiers

self.keyModifiers (read-write) Array

If a KeyboardEvent instance is used as an interest, keyModifiers indicates the set of key modifiers that must be pressed for the interest to be satisfied. If it is used as an event, it indicates the set of modifiers currently pressed. The two kinds of modifier keys are called shift and state keys. A shift key is active if it is being pressed. A state key is active if it is currently toggled on. Possible values for modifier keys are @shift, @control, @alt, @command, @capLock, @numLock, and @scrollLock. For additional information about keys, see the KeyboardDevice class.

```
myMouseEvent.keyModifiers -- suppose the shift key was pressed
⇨ #(@shift)
myMouseInterest.keyModifiers := #(@control,@shift) -- set modifiers
⇨ #(@control,@shift)
```

maxKeyCode

self.maxKeyCode (read-write) Integer

Interest-only instance variable. Specifies the maximum value for keyCode that can satisfy the event interest *self*. The instance method isSatisfiedBy, defined by Event but specialized by KeyboardEvent, compares keyCode with maxKeyCode. The interest *self* cannot be satisfied if keyCode is greater than maxKeyCode. If maxKeyCode = minKeyCode, then the interest can only be satisfied by a single keyCode value. By default, maxKeyCode is set to an arbitrarily high value, outside the range of keyCode values that is supported by any keyboard device.

For example, when creating a key-down interest in the four arrow keys (up, down, left, right), use these keycode values:

```
myKeyboardDownEvent.maxKeyCode := -40
myKeyboardDownEvent.minKeyCode := -43
```

minKeyCode

self.minKeyCode (read-write) Integer

Interest-only instance variable. Specifies the minimum value for keyCode that can satisfy the event interest *self*. The instance method isSatisfiedBy, defined by Event but specialized by KeyboardEvent, compares keyCode with minKeyCode. The interest *self* cannot be satisfied if keyCode is less than minKeyCode. If minKeyCode = maxKeyCode, then the interest can only be satisfied by a single keyCode value. By default, minKeyCode is set to an arbitrarily low value, outside the range of keyCode values that is supported by any keyboard device.

Instance Methods

Inherited from Event:

accept	isSatisfiedBy	sendToQueue
acquireRejectQueue	reject	signal
addEventInterest	relinquishRejectQueue	
broadcast	removeEventInterest	

Inherited from QueuedEvent:

broadcast	secondarySignal	signal
secondaryBroadcast		

The following instance methods are defined in KeyboardEvent:

broadcast (Event)

broadcast *self* ⇨ Exception

Reports the cantBroadcast exception if called on the keyboard event *self*.

isModifierActive

isModifierActive *self keyName* ⇨Boolean

self	KeyboardEvent object
keyName	NameClass object representing a modifier key.

Returns true or false, depending on whether the modifier key *keyName* is active on the KeyboardEvent object *self*. See the keyModifiers instance variable for a list of possible values.

isModifierInactive

isModifierInactive *self keyName* ⇨Boolean

self	KeyboardEvent object
keyName	NameClass object representing a modifier key.

Returns true or false, depending on whether the modifier key *keyName* is inactive on the keyboard event *self*. See the keyModifiers instance variable for a list of possible values.

isSatisfiedBy (Event)

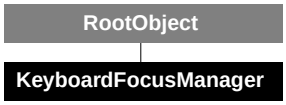
isSatisfiedBy *self event* ⇨ Boolean

self	Event object that represents an event interest
event	Event object that represents an actual event

The method `isSatisfiedBy` tests whether an event interest is satisfied by an event. Do not call `isSatisfiedBy` directly from a script. It is called by the event system as a result of calling either `signal` or `broadcast`.

The `KeyboardEvent` class specializes `isSatisfiedBy` to make several comparisons. It checks whether the device that generated the event is the same device that the event interest has registered an interest in. If the interest specifies one or more modifier keys, then `isSatisfiedBy` checks that those keys were pressed when the event occurred. If the interest sets a particular value or range of values for `keyCode` by setting the instance variables `maxKeyCode` or `minKeyCode`, then `isSatisfiedBy` checks that the value of `keyCode` for the event is in range.

KeyboardFocusManager



Class type: Core class (concrete)
Resides in: ScriptX and KMP executables
Inherits from: `RootObject`
Component: Input Devices

`KeyboardFocusManager` is a concrete class that manages focus on one keyboard device. Since version 1.1 of ScriptX supports only a single, physical keyboard device, there is at most one instance of `KeyboardFocusManager` in the system, accessible through the `focusManager` instance variable on that device.

Creating and Initializing a New Instance

Normally, you do not create a keyboard focus manager by calling `new` on the `KeyboardFocusManager` class. An instance of `KeyboardFocusManager` is created automatically when a `KeyboardDevice` object is instantiated, and it is set to manage focus on that device. `KeyboardFocusManager` defines an `init` method.

init

`init` *self device* ⇨ *(none)*

self	`FocusManager` object
`device:`	`KeyboardDevice` object

Initializes the `KeyboardFocusManager` object *self*, setting the value of the instance variable `device` to the value supplied with the keyword device. Do not call `init` directly on an instance—it is called automatically by the `new` method.

All keywords arguments defined by the `init` method are required.

Instance Variables

device

self.`device` (read-only) `KeyboardDevice`

Specifies the `KeyboardDevice` object that the `KeyboardFocusManager` object *self* is currently managing focus on.

Instance Methods

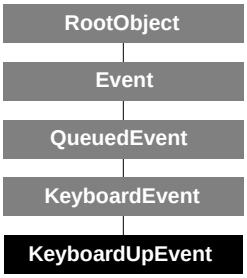
forceFocus

`forceFocus` *self presenter* ⇨ *(none)*

self	`KeyboardFocusManager` object
presenter	`TwoDPresenter` object

Forces the keyboard focus manager *self* to send a focus event (with `focusType` set to `@gainFocus`) to *presenter*, a `TwoDPresenter` object. If another presenter has focus, it also sends a focus event (with `focusType` set to `@loseFocus`) to that presenter, informing it that it is losing focus. If you call `forceFocus` with *presenter* set to `undefined`, the presenter that currently has focus loses focus.

KeyboardUpEvent



Class type: Core class (concrete)
Resides in: ScriptX and KMP executables
Inherits from: `KeyboardEvent`
Component: Events

The `KeyboardUpEvent` class represents the release of a key on a keyboard by a user. Each occurrence is represented by a different instance.

Creating and Initializing a New Instance

The following script is an example of how to create a new instance of the `KeyboardUpEvent` class:

```
myKeyUp := new KeyboardUpEvent
```

The variable `myKeyUp` contains the initialized instance. The `new` method takes no arguments. The class `KeyboardUpEvent` has no `init` method—its initialization is inherited from its superclasses.

Class Variables

Inherited from `Event`:

`interests` `numInterests`

Inherited from `QueuedEvent`:

`dispatchQueue`

Class Methods

Inherited from `Event`:

`acquireQueueFromPool` `relinquishQueueToPool`
`broadcastDispatch` `signalDispatch`

Instance Variables

Inherited from `Event`:

`advertised` `eventReceiver` `timeStamp`
`authorData` `matchedInterest`
`device` `priority`

Inherited from `QueuedEvent`:

`secondaryDispatchStyle` `secondaryRejectable`

Inherited from `KeyboardEvent`:

`keyCode` `maxKeyCode` `minKeyCode`

keyModifiers

Instance Methods

Inherited from Event:

accept	isSatisfiedBy	sendToQueue
acquireRejectQueue	reject	signal
addEventInterest	relinquishRejectQueue	
broadcast	removeEventInterest	

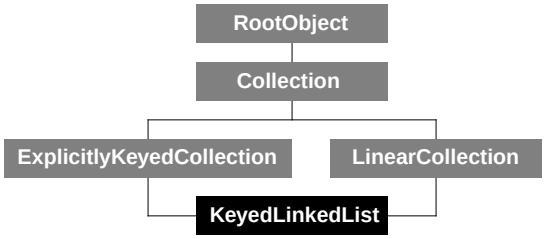
Inherited from QueuedEvent:

broadcast	secondarySignal	signal
secondaryBroadcast		

Inherited from KeyboardEvent:

broadcast	isModifierInactive	isSatisfiedBy
isModifierActive		

KeyedLinkedList



Class type: Core class (concrete)
Resides in: ScriptX and KMP executables
Inherits from: `ExplicitlyKeyedCollection` and `LinearCollection`
Component: Collections

A `KeyedLinkedList` consists of objects with three components, a key, a value, and another `KeyedLinkedList` object. `KeyedLinkedList` objects are recursive data structures, they can be separated into components that are also `KeyedLinkedList` objects using the `head` and `tail` methods. See the discussion in "LinkedList" on page 385 for more information on recursive data structures.

Creating and Initializing a New Instance

The following script is an example of how to create a new instance of the `KeyedLinkedList` class:

```
myList := new KeyedLinkedList
```

The variable `myList` contains the initialized keyed linked list. You can also create a linked list by using the pound sign (`#`) shortcut, separating keys from values with a colon (`:`) :

```
myList := #("a":1, "b":2, "c":3)
```

init

`init` *self* ⇨ *self*

self `KeyedLinkedList` object

Initializes the `KeyedLinkedList` object *self*, to be an empty list. Do not call `init` directly on an instance—it is automatically called by the `new` method.

Note – Every object that is added to an instance of `KeyedLinkedList` creates a new `KeyedLinkedList` as its new sublist; thus the `init` method will be called once for each added object. If you want to create a subclass of `KeyedLinkedList`, you must be aware of this behavior. If you want your subclass to have an `init` method that is only called once when the entire collection is created, use a subclass of `IndirectCollection` setting the `targetCollection` to a `KeyedLinkedList`.

Class Methods

Inherited from `Collection`:

`pipe`

Instance Variables

Inherited from `Collection`:

`bounded`	`maxSize`	`size`
`iteratorClass`	`minSize`	`uniformity`
`keyEqualComparator`	`mutable`	`uniformityClass`
`keyUniformity`	`mutableCopyClass`	`valueEqualComparator`
`keyUniformityClass`	`proprietored`	

Instance Methods

Inherited from `Collection`:

`add`	`forEach`	`iterate`
`addMany`	`forEachBinding`	`localEqual`
`addToContents`	`getAll`	`map`
`chooseAll`	`getAny`	`merge`
`chooseOne`	`getKeyAll`	`pipe`
`chooseOneBinding`	`getKeyOne`	`prin`
`deleteAll`	`getMany`	`removeAll`
`deleteBindingAll`	`getOne`	`removeOne`
`deleteBindingOne`	`hasBinding`	`setAll`
`deleteKeyAll`	`hasKey`	`setOne`
`deleteKeyOne`	`intersects`	`size`
`deleteOne`	`isEmpty`	
`emptyOut`	`isMember`	

Inherited from `LinearCollection`:

`chooseOneBackwards`	`findRange`	`getNthKey`
`chooseOrdOne`	`forEachBackwards`	`getOrdOne`
`deleteFirst`	`getFirst`	`getRange`
`deleteLast`	`getLast`	`localEqual`
`deleteNth`	`getMiddle`	`localLT`
`deleteRange`	`getNth`	`pop`

The following instance methods are defined in `KeyedLinkedList`:

head

`head` *self* ⇨ *(object)*

Returns the "value" portion of the keyed linked list *self*.

getKey

`getKey` *self* ⇨ *(object)*

Returns the key associated with the current "value" or `head` of the keyed linked list *self*.

getList

`getList` *self key* ⇨ `KeyedLinkedList`

self	`KeyedLinkedList` object
key	Any object

Returns the sublist of the keyed linked list *self* beginning with the element whose key matches the *key* value. Returns `empty` if there is no matching key.

getListValue

`getListValue` *self value* ⇨ `KeyedLinkedList`

self	`KeyedLinkedList` object
value	Any object

Returns the sublist of the keyed linked list *self* beginning with the element whose `head` value matches the *value* parameter. Returns `empty` if there is no matching entry.

setHead

`setHead` *self value* ⇨ *self*

self	`KeyedLinkedList` object
value	Any object

Sets the "value" portion of the keyed linked list *self* to *value*. Equivalent to:

```
(
    local i := iterate self
    next i
    i.value := value
)
```

setKey

`setKey` *self newKey* ⇨ *newKey*

self	`KeyedLinkedList` object
newKey	Any object

Sets the key associated with the current "value" or `head` of the keyed linked list *self* to *newKey*.

setTail

`setTail` *self newList* ⇨ *self*

self	`KeyedLinkedList` object
newList	`KeyedLinkedList` object

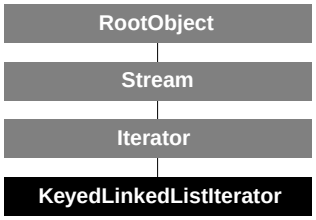
Grafts *newList* onto the keyed linked list *self*. Similar to `addMany` *self newList*, except that the internal structure of *self* is modified to point directly to *newList*.

tail

`tail` *self* ⇨ `KeyedLinkedList`

Returns the immediate sublist of *self*, that is, the `KeyedLinkedList` excluding the current `head` and its key. Returns `empty` if the current node is the last element of the collection.

KeyedLinkedListIterator



Class type: Core class (concrete)
Resides in: ScriptX and KMP executables
Inherits from: `Iterator`
Component: Collections

A `KeyedLinkedListIterator` object iterates over any `KeyedLinkedList` object.

Creating and Initializing a New Instance

A new instance of a keyed linked list iterator is generally created by calling `iterate` on an instance of `KeyedLinkedList`.

Instance Variables

Inherited from `Iterator`:

`key`	`source`	`value`

Instance Methods

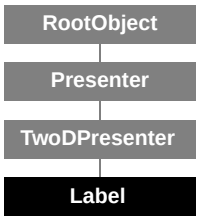
Inherited from `Stream`:

`cursor`	`next`	`seekFromStart`
`flush`	`plug`	`setStreamLength`
`isAtFront`	`previous`	`streamLength`
`isPastEnd`	`read`	`write`
`isReadable`	`readReady`	`writeReady`
`isSeekable`	`seekFromCursor`	
`isWritable`	`seekFromEnd`	

Inherited from `Iterator`:

`exise`	`seekKey`	`seekValue`
`remainder`		

Label



Class type: Scripted class (concrete)
Resides in: `widgets.sxl`. Works with ScriptX and KMP executables
Inherits from: `TwoDPresenter`
Component: User Interface

The `Label` class is a presenter that acts as a template for a text label. The ScriptX widget kit, a set of templates for building user interface objects that have a standard look and feel, uses the `Label` class to display text as part of other widgets.

A label incorporates a text stencil, which it renders to a display surface. The properties of a label are really the properties of this text stencil, together with some additional information for drawing it inside a colored frame, with a drop shadow.

Creating and Initializing a New Instance

The following script creates a new instance of `Label`:

```
global myLabel := new Label \
       text:"KALEIDA LIVES!"
       boundary:(new Rect x2:160 y2:40)
```

The variable `myLabel` contains an initialized instance of `Label`, which presents a text stencil with the string "KALEIDA LIVES!" in a presenter with the given boundary. The string will be rendered with the default system font, since no value is supplied for `font`. The `new` method uses the keywords defined in `init`.

init

`init` *self* [`text:`*string*] [`font:`*font*] [`boundary:`*stencil*]
[`stationary:`*boolean*] ⇨ *(none)*

self	`Label` object
`text:`	`String` object containing the text to be displayed
`font:`	`PlatformFont` object

Superclasses of `Label` use the following keywords:

`boundary:`	`Rect` object, the boundary of the presenter
`target:`	Ignored by `Label`
`stationary:`	`Boolean` object

Initializes the `Label` object *self*, applying the keywords as follows: `text` is applied to set the `string` keyword of a `TextStencil` object, which is stored in the instance variable `textStencil`; `font` is applied to set the platform-specific font of that text stencil; and `boundary` is used to set the instance variable of the same name. The `target` keyword is ignored. Do not call `init` directly on an instance—it is called automatically by the `new` method.

If you omit an optional keyword, its default value is used. The defaults are:

```
text:("untitled")
font:(theSystemFont)
```

```
boundary:(a Rect object that spans the given string)
target:undefined
stationary:true
```

Since the default settings are based on platform-specific fonts, appearance may vary across platforms unless all keyword arguments are fully specified.

Instance Variables

Inherited from Presenter:

presentedBy	subPresenters	target

Inherited from TwoDPresenter:

bBox	height	transform
boundary	IsImplicitlyDirect	width
clock	isTransparent	window
compositor	isVisible	x
direct	needsTickle	y
eventInterests	position	z
globalBoundary	stationary	
globalTransform	target	

The following instance variables are defined in Label:

alignment

self.alignment (read-write) NameClass

Specifies how the string that the label *self* presents should be aligned. Possible values are @flushLeft, @center, and @flushRight. The default value is @center.

autoResize

self.autoResize (read-write) Boolean

Specifies whether the label *self* is resized automatically when the height of the label's font is changed.By default, autoResize is set to true.

font

self.font (read-write) PlatformFont

Specifies the PlatformFont object that is used to render the label's text.

fill

self.fill (read-write) Brush

Specifies the Brush object that is used to fill the label *self*. The default is the global constant whiteBrush.

text

self.text (read-write) String

Specifies the String object that is the target string of the label's text stencil.

textStencil

self.textStencil (read-write) TextStencil

Specifies the text stencil, a 2D graphics object, that the label *self* draws to the display surface. This stencil draws the string that referenced by the label's text property.

textTransform

self.textTransform (read-write) TwoDMatrix

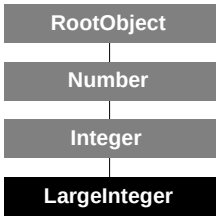
Specifies the TwoDMatrix object that the Label object *self* uses internally to render its text stencil. Although it uses the standard transformation matrix, defined by TwoDPresenter and stored in the instance variable globalTransform, for drawing the label's frame, a label maintains another matrix in textTransform to allow its text to be positioned independently of the frame in which it is drawn.

Instance Methods

Inherited from TwoDPresenter:

adjustClockMaster	inside	show
createInterestList	localToSurface	surfaceToLocal
draw	notifyChanged	tickle
getBoundaryInParent	recalcRegion	
hide	refresh	

LargeInteger



Class type: Core class (concrete, sealed)
Resides in: ScriptX and KMP executables
Inherits from: `Integer`
Component: Numerics

The `LargeInteger` class represents integers to -2^{63} to $2^{63} - 1$.

Note – See the `Integer` and `Number` classes for the inherited methods that can be called on `LargeInteger` objects.

Creating and Initializing a New Instance

There is no need to explicitly create or initialize an instance of `LargeInteger`—simply use whatever integer you want in a script. Whenever the compiler encounters an integer larger in absolute value than an `ImmediateInteger` that will fit in a large integer, it automatically promotes the integer to a `LargeInteger` object. If the integer won't fit in a large integer, the compiler promotes it to a `Float` object. The `LargeInteger` class has no scripter-level `init` method.

Instance Methods

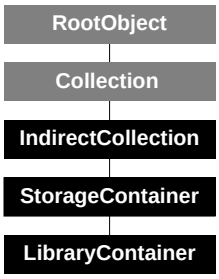
Inherited from `Number`:

`abs`	`floor`	`radToDeg`
`acos`	`frac`	`random`
`asin`	`inverse`	`rem`
`atan`	`ln`	`round`
`atan2`	`log`	`sin`
`ceiling`	`max`	`sinh`
`coerce`	`min`	`sqrt`
`cos`	`mod`	`tan`
`cosh`	`morph`	`tanh`
`degTorad`	`negate`	`trunc`
`exp`	`power`	

Inherited from `Integer`:

`length`	`logicalOp`	`lshift`
`logicalAnd`	`logicalOr`	`rshift`
`logicalNot`	`logicalXor`	

LibraryContainer



Class type: Core class (concrete)
Resides in: ScriptX and ScriptX Player executables
Inherits from: `StorageContainer`
Component: Title Management

The `LibraryContainer` class represents a file on disk that contains a set of loadable classes and objects, has a start-up script, and is identifiable by name, version, and copyright. Use an instance of `LibraryContainer` (for brevity, a library) to organize and store code and data that is used by a title. Since a library can contain references to objects that are in other libraries, it can act as both a physical and a logical container.

`LibraryContainer` has two subclasses, `TitleContainer` and `AccessoryContainer`, that are specialized for building interactive programs. An instance of `TitleContainer` has a menu bar and clipboard, and maintains a list of its clocks, players, and windows so the title as a whole can be paused, resumed, muted, opened, saved, and closed. An instance of `AccessoryContainer` is a library that can implement the `getAccessory` method, a built-in hook for communication with titles.

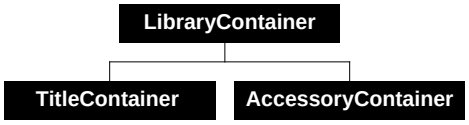
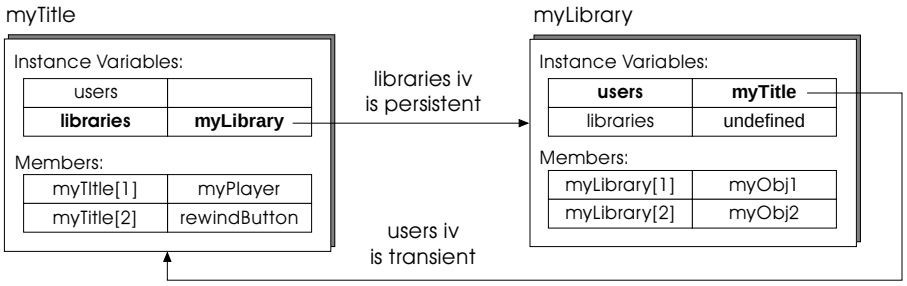


Figure 4-10: The LibraryContainer family of classes

An library is useful for storing classes and objects that can be added dynamically to a title. Examples of libraries include importers, exporters, transitions, preference files, or any other useful set of classes and objects. A library can comprise any kind of classes or objects that the title can use, including user interface items, such as menus, windows, and pushbuttons. A library can also contain a set of data, such as audio streams or bitmap objects. When the library is opened, its classes and objects are available to the title.

The ScriptX Player maintains a collection, accessible via the global variable `theOpenContainers`, of all currently open library, title, and accessory containers in the runtime environment. ScriptX also defines several other system globals that identify open title containers.

A library is not meant to be a stand-alone application. A library is intended to be used by a title or other library, as shown in the following figure. You cannot open a library from the operating system as you can a title. Normally you open a title first, which then loads any libraries and accessories that it needs. It can load them automatically at start-up time if they are listed in its `libraries` instance variable.



When a title opens a library, it calls `open` and specifies itself as a user of the library (using the `user` keyword) to inform the library that the title is using it. Several titles can be users of the same library—the library keeps a list of its users. The library remains open as long as it has at least one user. When its last user closes, the library closes and is automatically saved, unless it was opened in `@readonly` mode.

Library, title, and accessory containers cannot be embedded in one another. For example, you cannot add a library to a title. To make a required library part of a title, you instead specify the title as a user of the library (either with the `user` keyword when calling `new` or `open` on the library, or by calling the `addUser` method).

A library container is a collection. It inherits its collection behavior through `IndirectCollection`, and its default target collection is an array. You might want to specify a different target collection, such as an explicitly keyed collection. Add to a library container any objects that you want saved to the library. You can add objects explicitly, by using methods defined by `Collection`, `LinearCollection`, or `Sequence` such as `add`, `append`, or `addMany`.

Objects that are added directly to the library's target collection are called top-level objects. You can also add objects implicitly, for example, by reference through an instance variable of a top-level object. A top-level object can also be a collection that contains other objects. Objects that are added implicitly are automatically added to the container, and are referred to as subobjects. (If a subobject has already been added to another storage container, then only a reference is added.) Objects that have been added to a library container, both top-level objects and subobjects, are called persistent objects, even if they have not yet been saved to the library container.

When a library is opened, a reference to the objects it contains is brought into memory, so that those objects can then be referenced by the program. The objects themselves are not loaded into memory automatically. The library container's start-up script can be designed to load any objects that need to be preloaded. Other objects in the library are loaded into memory as they are used.

Note that `open` is a class method defined by the `LibraryContainer` class, not an instance method. To open a given instance of `LibraryContainer`, call `open` on the `LibraryContainer` class. Indicate the instance of `LibraryContainer` that is to be opened by referring to the container's file through the `path` and `dir` keywords.

Opening a library container causes ScriptX to run the script in the library's `preStartupAction` instance variable and to load all libraries referenced in the library's own `libraries` instance variable. If a `user` is supplied, `open` then calls `addUser`. Finally, it prepends the new library to `theOpenContainers` and the library's path to `theSearchContainerList` and runs the script in `startupAction`.

Libraries can only contain ScriptX classes and objects. If you want to load non-ScriptX code, such as C code, you must use the Loader component. See the chapter "Extending ScriptX" in the *ScriptX Tools Guide*.

Creating and Initializing a New Instance

The following script creates an instance of `LibraryContainer`:

```
myLib := new LibraryContainer \
          dir:myTitle.directory \
          path:"MyLib.sxl" \
          name:"My Library" \
          user:myTitle \
          targetCollection:(new HashTable)
```

The variable `myLib` contains the initialized library container, which is stored in a new file called `MyLib.sxl` in the directory given by `myTitle.directory`. Its name when it is asked to print is "`My Library`". The `myLib` library container is used by the `myTitle` title container, meaning the library container will remain open as long as that title container is open. The library's target collection is a hash table, and so its objects can be identified using explicit keys. The `new` method uses the keywords defined in `init`.

Note – The convention for naming library container files, valid across all platforms, is to use the `.sxl` extension, as shown above (meaning "ScriptX Library").

---

init

`init` *self* [`dir:`*dirRep*] `path:`*collectionOrString* [`name:`*string*] [`user:`*libraryContainer*] [`targetCollection:`*collection*] ⇨ *(none)*

self	`LibraryContainer` object
`dir:`	`DirRep` object
`path:`	`Collection` or `String` object
`name:`	`String` object
`user:`	`LibraryContainer` object

The superclass `IndirectCollection` uses the following keyword:

`targetCollection:`	`Collection` object

Initializes the `LibraryContainer` object *self*, creating a new file specified by the `dir` and `path` keywords, where `dir` is an optional directory, and `path` is an optional directory and required filename. The `name` keyword specifies a string that is displayed when printing the library.

If *libraryContainer* is supplied with the `user` keyword, `init` then calls `addUser`, where it prepends the given *libraryContainer* to *self*`.users`, then adds *self* to *libraryContainer*`.libraries`.

Then `init` prepends the new library to `theOpenContainers`. Finally, it returns the newly created library container.

The library container's collection is specified by the `targetCollection` keyword. It is useful to make the target collection an explicitly keyed collection, such as an instance of `HashTable`, `SortedKeyedArray`, or `BTree`, so that each item in the library container can be identified by a name constant key. For example, if a library contained a pushbutton, you could access it with `lc[@button]`. As is true with any collection, do not use strings as keys—use names or string constants instead.

Do not call `init` directly on an instance. It is automatically called by the `new` method.

If you omit an optional keyword, its default value is used, with one exception: If `user` is omitted, it is not used, and so has no default. The default values are as follows:

```
dir:theStartDir
name:undefined
targetCollection:(new Array initialSize:14 growable:true)
```

Class Methods

Inherited from `Collection`:

`pipe`

Inherited from `StorageContainer`:

`open`

The following class methods are defined in `LibraryContainer`:

open (StorageContainer)

`open` *self* [`dir:`*dirRep*] `path:`*collection* [`mode:`*name*] [`user:`*libraryContainer*]
⇨ `LibraryContainer`

self	`LibraryContainer` class
`dir:`	`DirRep` object
`path:`	`Collection` or `String` object
`mode:`	`NameClass` object: `@update` or `@readonly`
`user:`	`LibraryContainer` object that uses *self*

Opens the library container *self*, loading it but not its contained objects into memory. If supplied, the previously-opened title, library, or accessory container specified by the `user` keyword is added as a user of *self*.

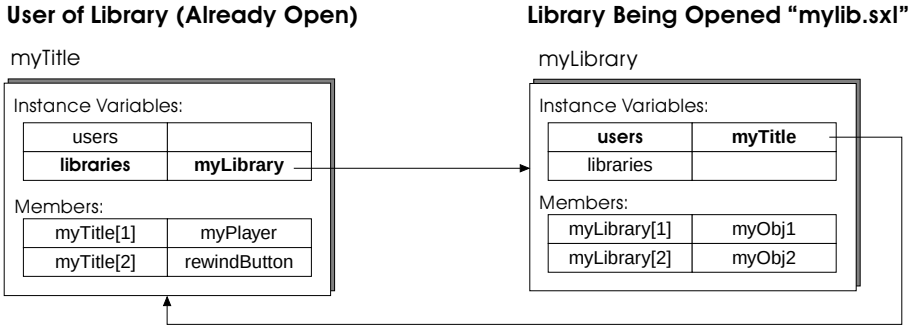
If you omit an optional keyword, its default value is used. The exception is `user`; if `user` is omitted it is not used. The defaults are:

```
dir:theStartDir
mode:@readonly
```

To illustrate, the following diagram shows a library container called `myLibrary` (shown on the right) that you want to open. It is used by a library (title, library, or accessory) container called `myTitle` (on the left). The following shows how to open `myLibrary` with `myTitle` as its user:

```
myLibrary := open LibraryContainer path:"MYLIB.SXL" user:myTitle
```

This example opens a file named `MYLIB.SXL` located in `theStartDir`, and adds `myTitle` to its `users` list. It also adds `myLibrary` to the `myTitle.libraries` list. The `users` list ensures that the library stays open as long as it has users; when you later close `myTitle`, the library `myLibrary` automatically closes only if it has no other users.



The `open` method looks for the file being opened in the location specified by `path` and `dir`. If it does not find the file in that location, it next looks in `theContainerSearchList`. This list automatically contains the directory holding the ScriptX or ScriptX Player executable (added at start-up), and contains the directory holding each open title container (added when the titles are opened). If the file is not found, it reports an exception.

Once it finds the file, the open method opens the file, then calls the function in preStartupAction, which can check that the minimum requirements (such as available memory) are satisfied and can add paths to theContainerSearchList. If this function returns false, then the library is purged, an exception is thrown, and the open method stops. If the function returns true, execution continues.

The open method then finds and loads each library in the libraries instance variable, and prepends the library container *self* to each loaded library's users list. It looks for these libraries in theContainerSearchList.

Next, if *libraryContainer* is supplied with the user keyword, open then calls addUser, which prepends the given *libraryContainer* to *self*.users, and then adds *self* to *libraryContainer*.libraries. Then open prepends the new library to theOpenContainers, calls the function in startupAction, and finally returns the newly opened library container. (See the definition of the addUser instance method for more information.)

If the library is already open, then the open method causes the startupAction to run again, but does not return another instance of the library.

In most cases, calling open on a library container that is already open does not report an exception. However, if you call open on a library container that has been created (with new) but has never been closed, an exception is reported. You must close or update a new library container at least once before you can call open on it.

Instance Variables

Inherited from Collection:

bounded	maxSize	size
iteratorClass	minSize	uniformity
keyEqualComparator	mutable	uniformityClass
keyUniformity	mutableCopyClass	valueEqualComparator
keyUniformityClass	proprietored	

Inherited from IndirectCollection:

targetCollection

The following instance variables are defined in LibraryContainer:

copyright

self.copyright (read-write) String

Specifies a string with copyright information for the library container *self*.

directory

self.directory (read-only) DirRep

Specifies the directory supplied with the dir keyword in the init or open method of the library *self*. This value and the value supplied with the path keyword form the complete path to the library; therefore, this value is the complete directory only if the path keyword contains just the filename. If the path keyword includes a subpath, then that subpath must be concatenated to the value of directory to get the full directory.

This instance variable is transient—the value of directory is not saved when the library is closed.

libraries

self.`libraries` (read-only) `Array`

Specifies a collection of the libraries to be opened by the library *self*. The libraries are opened in the order specified in this list. Although titles and accessories can be on this list, they typically are not.

Do not directly add or remove libraries from this list. This list is automatically maintained through calls to `addUser` and `removeUser`, methods defined in `LibraryContainer`.

Any library added to this list automatically gets opened when the library *self* is opened. This list should include any libraries that are required by the parent library.

This instance variable is persistent—it is saved when the library container is closed. This list keeps track internally of the filename of each library so that when *self* is later opened, the library files can be opened.

name

self.`name` (read-write) `String`

Specifies a string that describes the library *self*. This string is printed when you call `print` on the library.

preStartupAction

self.`preStartupAction` (read-write) *(function)*

Specifies a function that should determine it is okay to load the library *self*. Typically, this function should do two things:

- If necessary, add to `theContainerSearchList` any locations where the libraries listed in the `libraries` instance variable might be found.
- Check that minimum requirements for loading this library are satisfied by the environment—such as color depth and available memory.

A pre-startup function should be quick and small and should not block the thread or interfere with the thread scheduler. (For example, it should not call `waitTime` or `threadCriticalUp`.) Blocking the thread stops event processing, preventing the system from processing other events.

A `preStartupAction` function is called by the library's `afterLoading` method, which is called from `load`, which is called from the `open` class method. A `preStartupAction` function should take one argument and return a `Boolean` value. The function is called automatically with the library container *self* as its only argument:

```
func self
```

A `preStartupAction` function should return `true` if it is okay to continue loading this library; otherwise, it should return `false`. If it returns `false`, the library is purged, an exception is reported, and execution stops.

Although any global function, anonymous function, or method can be assigned to `preStartupAction`, there are differences in how different classes of functions are dispatched. For information on functions and function dispatch, see the "Object System Kernel" chapter of the *ScriptX Components Guide*.

startupAction

self.`startupAction` (read-write) *(function)*

Specifies a function that runs when the `LibraryContainer` instance *self* starts up. A `startupAction` function should take one argument and perform whatever tasks are appropriate when the library is opened. A typical title loads some of its objects from

storage, and creates other objects it needs on the fly. The usual purpose of a startup function is to determine what objects are available when the library is opened. Some of these objects may be loaded from the object store, and others are created or initialized on the fly. The start-up function's return value is unimportant and is ignored. The function is called automatically with the library container *self* as its only argument:

```
func self
```

A startup function should be quick and small and should not block the thread or interfere with the thread scheduler. (For example, it should not call `waitTime` or `threadCriticalUp`.) Blocking the thread stops event processing, preventing the system from processing other events.

When the start-up function loads the libraries in the `libraries` instance variable, it does not check to see if those libraries have previously been loaded. Since this function runs after those libraries have been loaded, it should check to see if an instance of the library *self* has been opened twice, and if so, purge the most recent instance.

A `startupAction` function can be designed to load any objects or classes from the library that the title needs to get started. It can also create any objects that are needed by the library.

For instances of `TitleContainer`, this function often creates a window, clock, or player. Special care needs to be taken if a `startupAction` function creates a window, clock, or player. The `Window`, `Clock`, and `Player` classes define a `title` instance variable, which is set to `theScratchTitle` unless otherwise specified. `TitleContainer`, a subclass of `LibraryContainer`, defines the instance variables `topClocks`, `topPlayers`, and `windows`, which it uses to maintain a list of the windows, clocks, and players that it manages. If these objects are not properly registered with an associated title, then a window does not get closed when the title is closed, and a clock or player is not paused when the title is paused.

Although any global function, anonymous function, or method can be assigned to `startupAction`, there are differences in how different classes of functions are dispatched. For information on functions and function dispatch, see the "Object System Kernel" chapter of the *ScriptX Components Guide*.

See `preStartupAction` for a related startup function. When a library is opened, the function in `startupAction` runs only if the `preStartupAction` function has returned `true`.

terminateAction

self`.terminateAction` (read-write) *(function)*

Specifies the function to be invoked when `close` is called on the library container *self*, and the library has no more users. Do not call this function directly—this function is called automatically from the `terminate` method. Write a `terminateAction` function to accept the library container *self* as its only argument:

```
myTermFunc self
```

Write this function to perform whatever cleanup tasks should occur before closing the library, such as setting global variables to `undefined`. This function cannot include any interaction with the user (that should be done in the library's `close` method).

Although any global function, anonymous function, or method can be assigned to `terminateAction`, there are differences in how different classes of functions are dispatched. For information on functions and function dispatch, see the "Object System Kernel" chapter of the *ScriptX Components Guide*.

type

self.type (read-only) Array

Specifies an array of author-defined names that describe the library *self*. Each member of the array should be an instance of NameClass. This array is a convenience for developers and is not used by ScriptX. A title container could examine this list in its pre-startup action to determine if the library has the attributes it needs. For example, a title could inspect this list and load a library based on the presence of certain keywords or name tokens.

You can add or remove names from this list—its "read-only" status means only that you cannot assign a different array to this instance variable.

users

self.users (read-only) Array

Specifies the array of open library containers that use the library *self*. Do not directly add or remove libraries from this list. Use addUser or removeUser to modify this list.

For example, if a title opens a library, the title is automatically added to the library's users list. (Likewise, the library is added to the title's libraries list.)

Circular references are not allowed (*a* uses *b* which uses *c* which uses *a*). The same user can appear in the array more than once, to simulate a reference count, but, then each occurrence of user must be removed. (This list is not saved in the storage container.)

This list is transient and is automatically maintained by the init, open, and close methods in LibraryContainer—when these method are called with a value for the user keyword, they add or remove a library from this list.

version

self.version (read-write) Number

Specifies the version number of the library *self*. This is a number (rather than a string) so versions can be mathematically compared to see which is later.

Instance Methods

Inherited from Collection:

add	forEach	iterate
addMany	forEachBinding	localEqual
addToContents	getAll	map
chooseAll	getAny	merge
chooseOne	getKeyAll	pipe
chooseOneBinding	getKeyOne	prin
deleteAll	getMany	removeAll
deleteBindingAll	getOne	removeOne
deleteBindingOne	hasBinding	setAll
deleteKeyAll	hasKey	setOne
deleteKeyOne	intersects	size
deleteOne	isEmpty	
emptyOut	isMember	

Inherited from IndirectCollection:

isAppropriateObject	objectAdded	objectRemoved

Inherited from StorageContainer:

close	update
requestPurgeForAllObjects	

Since a LibraryContainer object is an indirect collection, you can also use any methods defined in the class specified by targetCollection. The target collection is by default an instance of Array, which inherits from Sequence; in this case, the following instance methods are accessible:

Accessible from LinearCollection:

chooseOneBackwards	findRange	getNthKey
chooseOrdOne	forEachBackwards	getOrdOne
deleteFirst	getFirst	getRange
deleteLast	getLast	localEqual
deleteNth	getMiddle	localLT
deleteRange	getNth	pop

Accessible from Sequence by redirection:

addFifth	moveBackward	setFourth
addFirst	moveForward	setLast
addFourth	moveToBack	setNth
addNth	moveToFront	setSecond
addSecond	prepend	setThird
addThird	prependNew	sort
append	setFifth	
appendNew	setFirst	

The following instance methods are defined in LibraryContainer:

addUser

addUser *self newUser* ⇨ *(none)*

self	LibraryContainer object to be used by *newUser*
newUser	LibraryContainer object to be using *self*

Makes the library container *newUser* a user of library container *self*. Checks for circular references and if none, this method prepends the given *newUser* to the users instance variable of the library container *self*. Then it prepends the library container *self* to the libraries list of *newUser*. In this way, both the libraries have references to each other. A circular reference exists when a uses b, which uses c, which uses a.

For example, if title myTitle wants to start using library myLibrary, then the title should call:

```
addUser myLibrary myTitle
```

which adds myTitle to myLib.users and adds myLib to myTitle.libraries.

This method automatically gets called by the init and open class methods of LibraryContainer. However, there are a few cases where you would need to call this method directly. For example, if you open a library, then create a title and realize at some later point that the title should use the library—at this point you would add the title as a user (rather than close and re-open the library, specifying the user keyword).

close (StorageContainer)

close *self* [user:*libraryContainer*] ⇨ Boolean

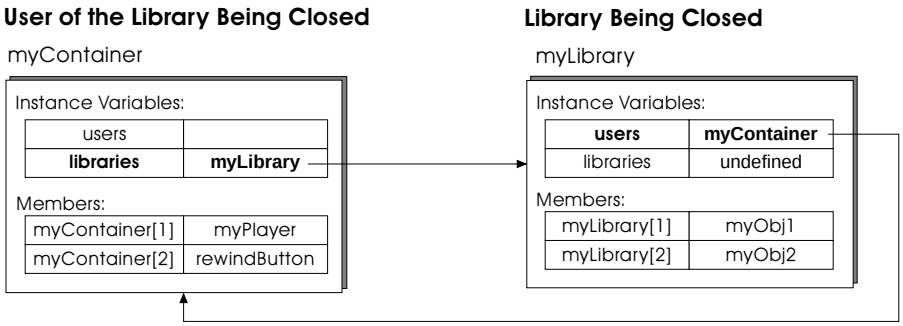
self	LibraryContainer object to close
user:	LibraryContainer object currently using *self*

Conditionally closes the library container *self*, saves it if it was opened using the @update mode, and makes the library and its contained objects purgeable. The library container is actually closed only if it has no users other than the one specified with the user keyword. (Note that the corresponding open method is a class method rather than an instance method.)

To illustrate, the following diagram shows a library container called `myLibrary` (shown on the right) that we want to close. It is being used by a title, accessory, or library container called `myContainer` (on the left). To close `myLibrary`, you would call:

```
close myLibrary user:myContainer
```

This expression removes `myContainer` from `myLibrary.users`. If that `users` list is empty, it then closes and saves `myLibrary`. Note that if `myLibrary` has other users, it is not closed. Also notice that it does not attempt to close `myContainer`.



The details are as follows. First the `close` method calls `removeUser` *self libraryContainer*. If the *self*`.users` list is empty, this method saves *self*`.libraries` instance variable, and calls `close` on each of its libraries. Then it calls `terminate` on *self*, which does several things including saving the objects in the library *self* if it was created with `new` or opened with `@update`, and closing the underlying storage container. Then the `close` method calls `requestPurge` on *self* and all its contained objects to free them for garbage collection. Finally, it returns `true`.

However, if the `users` list is not empty, the `close` method returns `false` and does not close the container. This way, the library container will be closed only if no other libraries are using it.

If you want to add user interaction, such as confirming whether to close the library, then override this method in a subclass. While the `close` method can be called several times and canceled, `terminate` and `terminateAction` can be called only once. See `terminate` for more details.

If you omit the `user` keyword, the keyword is not used (it has no default).

If you open a library container using `@readonly` mode and modify its contents (for example, with `append`), then when you call `close` on it, it will give a write error and will not close properly.

For details of closing a title container, refer to the `close` method in `TitleContainer`.

isAppropriateObject (IndirectCollection)

`isAppropriateObject` *self addedObject* ⇨ `Boolean`

Checks that *addedObject*, the object added to the library container *self*, is not a kind of `LibraryContainer`. Returns `true` if the object is not a library container and otherwise returns `false`. This prevents library container from being embedded inside other library containers.

recurPrin (RootObject)

`recurPrin` *self arg stream state* ⇨ *(none)*

Prints out the name of library *self*, as specified by the `name` instance variable.This method does not print the contents of the library, as you might otherwise expect it to do, because that would cause the entire library to load. See the `recurPrin` method in the `RootObject` class for more details on the arguments.

removeUser

`removeUser` *self currentUser* ⇨ *self*

self	`LibraryContainer` object to stop being used by *currentUser*
currentUser	`LibraryContainer` object to stop using *self*

Makes the library container *currentUser* no longer a user of the library container *self*. It first removes *currentUser* from the `users` instance variable of the library container *self*. Then it removes *self* from the `libraries` list of *currentUser*. In this way, cross-references between the libraries are removed.

terminate

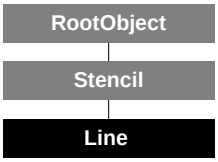
`terminate` *self* ⇨ *self*

Do not call this method directly—this method automatically gets called from the `close` method. You may need to override `terminate` in a subclass to perform cleanup. This method cannot contain any user interaction—do that in the `close` method. While `close` can be called several times and canceled, `terminate` can be called only once. Once `terminate` is called, there is no turning back—the library will close.

You can perform cleanup actions either by overriding this method or by implementing a terminate function in the library's `terminateAction`. As part of cleanup, you should set user-defined global variables to `undefined` so the objects they reference can be removed by the garbage collector.

First, `terminate` calls the function specified by `terminateAction`—see the definition of that instance variable for more details. Next, it removes the library container *self* from the global variable `theOpenContainers` list. Then it closes *self* and its underlying storage container, which saves the library container *self* and all of its contained objects if its mode is `@create` or `@update`. Finally, it calls `close` on each of the libraries in its `libraries` list with *self* as a user. The library *self* and its contents are made purgeable only after `terminate` is completed and control has returned to the `close` method from which it was called.

Line



Class type: Core class (concrete)
Resides in: ScriptX and KMP executables
Inherits from: `Stencil`
Component: 2D Graphics

The `Line` class is a subclass of `Stencil` optimized for rendering lines, defined by two points connected by a single straight line segment. In the ScriptX imaging model, the largest pixel values defining a line are excluded from the resulting image. For example, if a line is defined with the end points (0,0) and (20,20), the pixel at location 20,20 won't be rendered.

Note that `Line` objects are not presenters—to display a line, create an instance of `TwoDShape` using a `Line` object as the `boundary` argument. To set the thickness of a `Line` instance, set `lineWidth` on the instance of `TwoDShape`

Creating and Initializing a New Instance

The following script is an example of how to create a new instance of the `Line` class:

```
myLine := new Line \
       x1:0 \
       y1:0 \
       x2:50 \
       y2:50
```

The variable `myLine` contains the new `Line` instance. The new method uses the keywords defined in `init`.

init

`init` *self* [`x1:`*number*] [`y1:`*number*] [`x2:`*number*] [`y2:`*number*] ⇨ *(none)*

self	`Line` object
`x1:`	`Number` object
`y1:`	`Number` object
`x2:`	`Number` object
`y2:`	`Number` object

Do not call `init` directly on an instance—it is automatically called by the `new` method.

If you omit an optional keyword, its default value is used. The defaults are:

```
x1:0
y1:0
x2:1
y2:1
```

Instance Variables

Inherited from `Stencil`:

`bBox`

The following instance variables are defined in `Line`:

angle

self`.angle` (read-write) `Number`

Specifies the angle that `x2`, `y2` makes as a vector from `x1`, `y1`. This instance variable returns the angle in radians. Positive values indicate clockwise rotation, negative values indicated counterclockwise rotation. A 0 value positions the line horizontally, with `y1` and `y2` set to the same value. Setting this variable moves point `x2`, `y2`, keeping `x1`, `y1` and `length` constant.

length

self`.length` (read-write) `Number`

Specifies the length of the line *self*. Setting this variable moves point `x2`, `y2`, keeping `x1`, `y1` and `angle` constant.

x1

self`.x1` (read-write) `Number`

Specifies the x coordinate of the first point in the line *self*.

x2

self`.x2` (read-write) `Number`

Specifies the x coordinate of the second point in the line *self*.

y1

self`.y1` (read-write) `Number`

Specifies the y coordinate of the first point in the line *self*.

y2

self`.y2` (read-write) `Number`

Specifies the y coordinate of the second point in the line *self*.

Instance Methods

Inherited from `Stencil`:

`inside`	`onBoundary`	`transform`
`intersect`	`subtract`	`union`

The following instance methods are defined in `Line`:

copy

`copy` *self* ⇨ `Line`

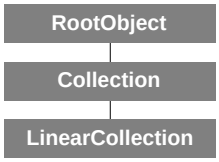
Creates and returns a copy of the line *self*.

moveToZero

`moveToZero` *self* ⇨ `Line`

Repositions the line *self* so that the values of `x1` and `y1` are 0, adjusting the values of `x2` and `y2` to maintain the original angle and length.

LinearCollection



Class type: Core class (abstract)
Resides in: ScriptX and KMP executables
Inherits from: `Collection`
Component: Collections

The `LinearCollection` class is a mix-in abstract class for collections that support an ordering of their items, either explicitly or implicitly.

Every linear collection is also a collection. `LinearCollection` must be mixed in with `Collection` or one of its subclasses. Since `LinearCollection` defines several methods that are also defined by `Collection` to allow for a natural ordering of elements, it must have precedence over `Collection` in the list of superclasses from which a subclass inherits.

Several of the method definitions for this class refer to the *ordinal position* of an item in a collection. For a definition of ordinal position, and of *cursor position*, which corresponds to ordinal position, see the `String` class.

Instance Methods

chooseOneBackwards

`chooseOneBackwards` *self func arg* ⇨ *(object)*

self	`LinearCollection` object
func	An instance of a subclass of `AbstractFunction`
arg	Any object

Calls the function *func* on values in the linear collection *self*, starting at the end and working towards the first item in the collection. This method is analogous to `chooseOne`, defined by `Collection`. This method includes the argument *arg* when calling the function.

func item arg

The `chooseOneBackwards` method calls this function once for each item in the collection until the function returns `true`. Its return value is the first item on which the function returns `true`, or `empty` if no call to this function returns `true`. See the discussion on page 161 in the class definition of “Collections” concerning generic functions that depend on the integrity of an iterator. Note that `Array` specializes `chooseOneBackwards` to allow for insertion and deletion of items, although not for wholesale rearrangement of the collection. See the definitions of this and other iterative methods under the `Array` class.

chooseOrdOne

`chooseOrdOne` *self func arg* ⇨ `Integer`

self	`LinearCollection` object
func	An instance of a subclass of `AbstractFunction`
arg	Any object in the linear collection *self*

Similar to `chooseOne`, an instance method defined by `Collection`. It calls the function *func* on each item in the collection, supplying the argument *arg*.

func item arg

The `chooseOrdOne` method calls this function once for each item in the collection until the function returns `true`. Its return value is the ordinal position of the first item on which the function returns `true`, or `0` if no call to this function returns `true`. See the discussion on page 161 in the class definition of "Collections" concerning generic functions that depend on the integrity of an iterator.

deleteFirst

`deleteFirst` *self* ⇨ *(none)*

`deleteSecond` *self* ⇨ *(none)*

`deleteThird` *self* ⇨ *(none)*

`deleteFourth` *self* ⇨ *(none)*

`deleteFifth` *self* ⇨ *(none)*

These five methods are really macros that expand to `deleteNth` *self* `1`, `deleteNth` *self* `2` and so on, where *self* is a `LinearCollection` object. These methods do not remove objects from memory, unless normal garbage collection applies. They only remove them from the collection.

These methods report the `bounded` exception (leaving the collection in an undefined state) if the collection would shrink below the size specified by `minSize`.

deleteLast

`deleteLast` *self* ⇨ *(none)*

Removes the last key-value pair from the linear collection *self*. This method does not remove objects from memory, unless normal garbage collection applies. It only removes them from the collection.

This method reports the `bounded` exception (leaving the collection in an undefined state) if the collection would shrink below the size specified by `minSize`.

deleteNth

`deleteNth` *self ordinal* ⇨ `Boolean`

self	`LinearCollection` object
ordinal	`Integer` object

Removes the key-value pair in the position specified by *ordinal* from the linear collection *self*. This method does not remove objects from memory, unless normal garbage collection applies. It only removes them from the collection. Returns `true` if a key-value pair was found at that position and removed, or `false` if the specified ordinal is beyond the last item in the collection. The `deleteNth` method is analogous to `deleteOne`, defined by `Collection`.

This method reports the `bounded` exception (leaving the collection in an undefined state) if the collection would shrink below the size specified by `minSize`.

deleteRange

`deleteRange` *self toMatch* ⇨ `Boolean`

self	`LinearCollection` object
toMatch	`LinearCollection` object

Finds the first range of contiguous values inside the linear collection *self* that matches the values in the linear collection *toMatch*, as if by calling `findRange`, another method defined by `LinearCollection`. If there is a match, `deleteRange` removes all those items from the source collection. This method returns `true` if a match was found and removed; otherwise, it returns `false`. This method does not remove objects from memory, unless normal garbage collection applies. It only removes them from the collection.

This method reports the `bounded` exception (leaving the collection in an undefined state) if the collection would shrink below the size specified by `minSize`.

findRange

`findRange` *self toMatch* ⇨ `Integer`

self	`LinearCollection` object
toMatch	`LinearCollection` object

Finds the first range of contiguous values inside the linear collection *self* that matches the values in the linear collection *toMatch*. Returns the ordinal number of the number start of that range, or 0, if no such range was found.

```
global myArray := #(1,3,5,7,8) -- An array with five values
findRange myArray #(3,5)
⇨ 2
findRange myArray #(3,8)
⇨ 0
```

forEachBackwards

`forEachBackwards` *self func arg* ⇨ *(none)*

self	`LinearCollection` object
func	An instance of a subclass of `AbstractFunction`
arg	`Object` object

Calls the function *func* on values in the linear collection *self*, starting at the end and working forward, with the argument *arg*. The `forEachBackwards` method is analogous to `forEach`, defined by `Collection`. For more information, see the definition of `forEach`. See the discussion on page 161 in the class definition of "Collections" concerning generic functions that depend on the integrity of an iterator. Note that `Array` specializes `forEachBackwards` to allow for insertion and deletion of items, although not for wholesale rearrangement of the collection. See the definitions of this and other iterative methods under the `Array` class.

getFirst

`getFirst` *self* ⇨ *(object)*

`getSecond` *self* ⇨ *(object)*

`getThird` *self* ⇨ *(object)*

`getFourth` *self* ⇨ *(object)*

`getFifth` *self* ⇨ *(object)*

These methods are really macros that expand to `getNth` *self* `1`, `getNth` *self* `2`, and so on, where *self* is a `LinearCollection` object.

getLast

`getLast` *self* ⇨ *(object)*

Gets the last value of the linear collection *self*.

getMiddle

`getMiddle` *self* ⇨ *(object)*

Gets the middle value of the linear collection *self*. For an even-sized linear collection, `getMiddle` gets the next value after the middle value.

getNth

`getNth` *self ordinal* ⇨ *(object)*

self	`LinearCollection` object
ordinal	`Integer` object representing an ordinal position

Gets the value from the linear collection *self* at the position specified by *ordinal*.

getNthKey

`getNthKey` *self ordinal* ⇨ *(object)*

self	`LinearCollection` object
ordinal	`Integer` object representing an ordinal position

Gets the key from the linear collection *self* at the position specified by *ordinal*.

getOrdOne

`getOrdOne` *self value* ⇨ `Integer`

self	`LinearCollection` object
value	Any object

Returns the ordinal position for a value of the linear collection *self* that matches the given *value*. The values are matched using the value comparator function specified by `valueEqualComparator`, an instance variable defined by `Collection`. Since `getOrdOne` creates an iterator and searches a linear collection in its natural order, the first item that matches is always chosen.

getRange

`getRange` *self start end* ⇨ `LinearCollection`

self	`LinearCollection` object
start	`Integer` object representing an ordinal position
end	`Integer` object representing an ordinal position

Gets from the linear collection *self* another collection containing the values in the ordinal positions *start* through *end*. This method returns another collection that is an instance of the class specified by the `mutableCopyClass`, an instance variable defined by `Collection`.

localEqual (RootObject)

`localEqual` *self other* ⇨ `Boolean`

self	`Collection` object
other	`Collection` object

Compares individual items in the linear collection *self* element by element with items in the linear collection *other*. Elements are compared universally. Returns true if *self* is locally equal to *other*. Note that, in contrast with the definition of `localEqual` that the `Collection` class provides, elements of a linear collection must be in the same order for `localEqual` to return `true`.

```
global array1 := #("dog","cat")
global array2 := #("cat","dog")
localEqual array1 array2
⇨  false
```

The `localEqual` method is not usually called directly. It is one four primitives (`comparable`, `eq`, `localEqual`, and `localLt`) that are used to define all ScriptX comparison functions. For definitions of ScriptX comparison functions, see Chapter 2, "Global Functions." For more information on comparison of objects, see the discussion in the "Object System Kernel" chapter of the *ScriptX Components Guide*.

localLT (RootObject)

`localLT` *self other* ⇨ `Boolean`

Compares individual items in the linear collection *self* element by element with items in the linear collection *other*. Elements are compared universally. Returns `true` if *self* is less than *other*.

```
global array1 := #(2,5,"dog","cat")
global array2 := #(2,5,"cat","dog")
global array3 := #(5,2,"cat","dog")
global array4 := #("cat","dog",5,2)
localLT array1 array2
⇨  false
localLT array1 array3
⇨  true
localLT array1 array4 -- involves universal comparison of 2 and "cat"
⇨  true
```

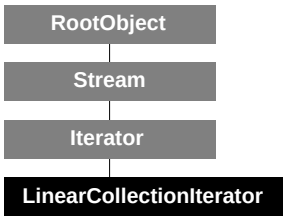
The `localLt` method is not usually called directly. It is one four primitives (`comparable`, `eq`, `localEqual`, and `localLt`) that are used to define all ScriptX comparison functions. For definitions of ScriptX comparison functions, see Chapter 2, "Global Functions." For more information on comparison of objects, see the discussion in the "Object System Kernel" chapter of the *ScriptX Components Guide*.

pop

pop *self* ⇨ *(object)*

Removes the first key-value pair from the linear collection *self*, and returns its value. This method does not remove objects from memory, unless normal garbage collection applies. It only removes them from the collection. To perform the complementary push operation, use the `prepend` method.

LinearCollectionIterator



Class type: Core class (concrete)
Resides in: ScriptX and KMP executables
Inherits from: `Iterator`
Component: Collections

A `LinearCollectionIterator` object iterates over any linear collection.

Creating and Initializing a New Instance

A new instance of a linear collection iterator is generally created by calling `iterate` on an instance of a subclass of `LinearCollection`.

Instance Variables

Inherited from `Iterator`:

`key`	`source`	`value`

Instance Methods

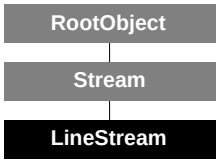
Inherited from `Stream`:

`cursor`	`next`	`seekFromStart`
`flush`	`plug`	`setStreamLength`
`isAtFront`	`previous`	`streamLength`
`isPastEnd`	`read`	`write`
`isReadable`	`readReady`	`writeReady`
`isSeekable`	`seekFromCursor`	
`isWritable`	`seekFromEnd`	

Inherited from `Iterator`:

`exise`	`seekKey`	`seekValue`
`remainder`		

LineStream



Class type: Core class (concrete)
Resides in: ScriptX and KMP executables
Inherits from: `Stream`
Component: Streams

Provides a line-oriented, read-only implementation of the Stream protocol. A line stream acts as a filter, reading properly terminated lines of characters from a source stream and returning them as `String` objects through its `read` method.

Creating and Initializing a New Instance

The following script is an example of how to create a new instance of the `LineStream` class:

```
myLine := new LineStream \
       source:sourceStream
```

The `myLine` variable contains the initialized line stream, which the uses `sourceStream` as its source.

init

`init` *self* `source:`*dataSource* ⇨ *(none)*

self	`LineStream` object
`source:`	`Stream` object

Initializes the `LineStream` object *self*, applying the arguments as follows: `source` is the stream containing the source of data. Do not call `init` directly on an instance—it is automatically called by the `new` method.

Instance Methods

Inherited from `Stream`:

`cursor`	`next`	`seekFromStart`
`flush`	`plug`	`setStreamLength`
`isAtFront`	`previous`	`streamLength`
`isPastEnd`	`read`	`write`
`isReadable`	`readReady`	`writeReady`
`isSeekable`	`seekFromCursor`	
`isWritable`	`seekFromEnd`	

The following instance methods, inherited from the abstract class `Stream`, are overridden in `LineStream` to behave as described:

isReadable (Stream)

`isReadable` *self* ⇨ `Boolean`

Returns `true`.

isSeekable (Stream)

`isSeekable` *self* ⇨ `Boolean`

Returns `false`.

isWritable (Stream)

`isWritable` *self* ⇨ `Boolean`

Returns `false`.

next (Stream)

`next` *self* ⇨ `Boolean`

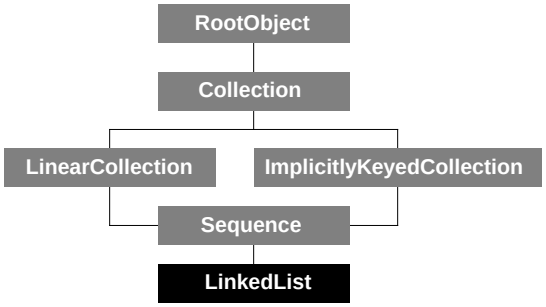
If the cursor is positioned at the last line, returns `false`. Otherwise, positions the cursor at the beginning of the next line and returns `true`.

read (Stream)

`read` *self* ⇨ `String`

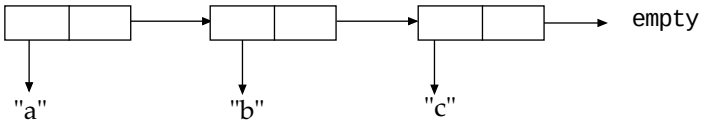
Reads a properly terminated line of characters as a `String` object from the stream *self*. Proper line termination characters include carriage return (CR), linefeed (LF) carriage return and linefeed (CRLF), and null.

LinkedList



Class type: Core class (concrete)
Resides in: ScriptX and KMP executables
Inherits from: Sequence
Component: Collections

The `LinkedList` class is a recursive data structure consisting of two parts: the `head`, which holds a collection value, and the `tail`, which points to another `LinkedList`. This structure is diagrammed below:



The figure shows a `LinkedList` containing three elements: the strings `"a"`, `"b"`, and `"c"`. The `head` of the list in the figure is the value `"a"`, and the `tail` is the sublist `#("b","c")`.

Linked lists are relatively memory efficient, but access can be slow when performing any operations other than prepending a new value or traversing the list in sequential order.

Creating and Initializing a New Instance

The following script is an example of how to create a new instance of the `LinkedList` class:

```
ll := new LinkedList
```

The variable `ll` is set to a list with no elements.

init

`init` *self* ⇨ *(none)*

self `LinkedList` object

Initializes the `LinkedList` object *self*, to be an empty list. Do not call `init` directly on an instance—it is automatically called by the `new` method.

Note – Every object that is added to a `LinkedList` creates a new sublist, and thus the `init` method is called repeatedly. If you want to create a subclass of `LinkedList`, you must be aware of this behavior. If you want your subclass to have an `init` method that is only called once when the entire collection is created, use a subclass of `IndirectCollection` setting the `targetCollection` to a `LinkedList`.

Class Methods

Inherited from Collection:

pipe

Instance Variables

Inherited from Collection:

bounded	maxSize	size
iteratorClass	minSize	uniformity
keyEqualComparator	mutable	uniformityClass
keyUniformity	mutableCopyClass	valueEqualComparator
keyUniformityClass	proprietored	

Instance Methods

Inherited from Collection:

add	forEach	iterate
addMany	forEachBinding	localEqual
addToContents	getAll	map
chooseAll	getAny	merge
chooseOne	getKeyAll	pipe
chooseOneBinding	getKeyOne	prin
deleteAll	getMany	removeAll
deleteBindingAll	getOne	removeOne
deleteBindingOne	hasBinding	setAll
deleteKeyAll	hasKey	setOne
deleteKeyOne	intersects	size
deleteOne	isEmpty	
emptyOut	isMember	

Inherited from LinearCollection:

chooseOneBackwards	findRange	getNthKey
chooseOrdOne	forEachBackwards	getOrdOne
deleteFirst	getFirst	getRange
deleteLast	getLast	localEqual
deleteNth	getMiddle	localLT
deleteRange	getNth	pop

Inherited from Sequence:

addFifth	moveBackward	setFourth
addFirst	moveForward	setLast
addFourth	moveToBack	setNth
addNth	moveToFront	setSecond
addSecond	prepend	setThird
addThird	prependNew	sort
append	setFifth	
appendNew	setFirst	

The following instance methods are defined in LinkedList:

head

head *self* ⇨ *(object)*

self LinkedList object

Returns the "value" portion of linked list *self*.

setHead

`setHead` *self value* ⇨ *self*

self	`LinkedList` object
value	any object

Sets the "value" portion of the linked list *self* to *value*. `setHead` is equivalent to the following:

```
(   local i := iterate self
    next i
    i.value := value
)
```

tail

`tail` *self* ⇨ `LinkedList`

self	`LinkedList` object

Returns the immediate sublist of *self*, that is, the `LinkedList` excluding the current `head`. Returns `empty` if the current node is the last element of the collection.

setTail

`setTail` *self newList* ⇨ *self*

self	`LinkedList` object
newList	`LinkedList` object

Grafts *newList* onto the linked list *self*. Similar to `addMany self newList`, except that the internal structure of *self* is modified to point directly to *newList*.

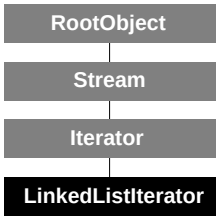
getListValue

`getListValue` *self value* ⇨ *LinkedList*

self	`LinkedList` object
value	any object

Returns the sublist of the linked list *self* beginning with the element whose `head` value matches the *value* parameter. Returns `empty` if there is no matching entry.

LinkedListIterator



Class type: Core class (concrete)
Resides in: ScriptX and KMP executables
Inherits from: `Iterator`
Component: Collections

A `LinkedListIterator` object iterates over `LinkedList` objects.

Creating and Initializing a New Instance

You create a new instance of `LinkedListIterator` by calling `iterate` on a `LinkedList` object.

For any given collection, the Kaleida Media Player uses the value of the virtual instance variable `iteratorClass` to determine which iterator to create. Every subclass of `Collection` must implement the `iteratorClassGetter` method. For the `LinkedList` class, the value of `iteratorClass` is `LinkedListIterator`.

Instance Variables

Inherited from `Iterator`:

`key`	`source`	`value`

Instance Methods

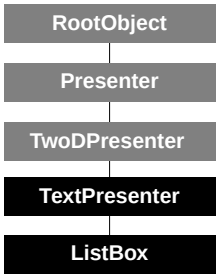
Inherited from `Stream`:

`cursor`	`next`	`seekFromStart`
`flush`	`plug`	`setStreamLength`
`isAtFront`	`previous`	`streamLength`
`isPastEnd`	`read`	`write`
`isReadable`	`readReady`	`writeReady`
`isSeekable`	`seekFromCursor`	
`isWritable`	`seekFromEnd`	

Inherited from `Iterator`:

`excise`	`seekKey`	`seekValue`
`remainder`		

ListBox



Class type: Scripted class (concrete)
Resides in: `widgets.sxl`. Works with ScriptX and KMP executables
Inherits from: `TextPresenter`
Component: User Interface

`ListBox` is a text presenter that presents a list of strings.

Creating and Initializing a New Instance

The following script creates a new instance of `ListBox`:

```
myListBox := new ListBox \
       font:myFontContext \
       width:90 \
       list:("Apple","IBM","Toshiba")
```

The variable `myListBox` contains an initialized instance of `ListBox`. It applies the font context `myFontContext`, which must be defined elsewhere, to create a `ListBox` object of width 90, targeting the given list. The presenter's boundary and target are set automatically, based on these keyword values.

The `new` method uses the keywords defined in `init`.

init

`init` *self* [`font:`*font*] [`width:`*integer*] [`list:`*sequence*]
[`stationary:`*boolean*] [`fill:`*brush*] [`stroke:`*brush*] ⇨ *(none)*

self	`ListBox` object
`font:`	`FontContext` object
`width:`	`Integer` object
`list:`	`Sequence` object

Superclasses of `ListBox` use these keywords.

`boundary:`	`Rect` object, ignored by `ListBox`
`target:`	`String` object, ignored by `ListBox`
`stationary:`	`Boolean` object
`fill:`	`Brush` object
`stroke:`	`Brush` object

Initializes the `ListBox` object *self*, applying the arguments as follows: The `font` keyword takes a `FontContext` object, which it uses to set the attributes of the list box. The `width` keyword, together with the attributes set by `font` and the number of items in the list, is used to determine the list box's boundary. The `list` keyword takes a sequence of strings, or items that can be freely coerced to strings, which it uses to set the target of the list box. The `boundary` and `target` keywords are ignored, since the list box's boundary and target are set at initialization, based on the values supplied for `font`, `width`, and `list`. A script can set the values of `stationary`, `fill`, and `stroke` at

initialization; they are applied to instance variables of the same name, defined by TwoDPresenter.Do not call init directly on an instance—it is called automatically by the new method.

If you omit an optional keyword, its default value is used. The defaults are:

```
font:theSystemFont
width:300
list:(#())
stationary:false
fill:whiteBrush
stroke:blackBrush
```

Instance Variables

Inherited from Presenter:

presentedBy	subPresenters	target

Inherited from TwoDPresenter:

bBox	height	transform
boundary	IsImplicitlyDirect	width
clock	isTransparent	window
compositor	isVisible	x
direct	needsTickle	y
eventInterests	position	z
globalBoundary	stationary	
globalTransform	target	

Inherited from TextPresenter:

attributes	fill	selectionForeground
cursor	inset	stroke
cursorBrush	offset	
enabled	selectionBackground	

The following instance variables are defined in ListBox:

font

self.font (read-write) FontContext

Specifies the FontContext object that is used to determine the text attributes of the list box *self*.

list

self.list (read-write) Sequence

Specifies a sequence of strings, or objects that can be coerced to strings, that the list box *self* presents.

numLines

self.numLines (read-write) Integer

Specifies the number of lines of text in the ListBox self. Do not set the value of numLines. It is determined automatically each time the contents of the list change.

Instance Methods

Inherited from TwoDPresenter:

adjustClockMaster	inside	show

createInterestList	localToSurface	surfaceToLocal
draw	notifyChanged	tickle
getBoundaryInParent	recalcRegion	
hide	refresh	

Inherited from TextPresenter:

calculate	getOffsetForXY	processMouseDown
copySelection	getPointForOffset	

The following instance methods are defined in ListBox:

getListOrdinal

getListOrdinal *self string* ⇨ Integer

self	ListBox object
string	String object

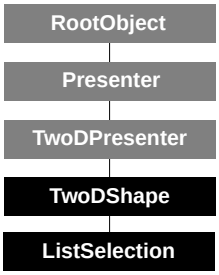
Returns the position of the specified *string*, within the list *self*. If the item is not found, getListOrdinal returns 0.

recalcHeight

recalcHeight *self* ⇨ Integer

Returns the height of the ListBox *self*, based on the value of numLines and the text attributes of the list box, such as its font, leading, and descent. Specialize recalcHeight to perform any action that must occur each time the contents of the list are changed.

ListSelection



Class type: Scripted class (concrete)
Resides in: widgets.sxl. Works with ScriptX and KMP executables
Inherits from: TwoDShape
Component: User Interface

The ListSelection class is used for creating the rectangular selection that highlights text in an instance of ScrollBox. The ink mode is set to @srcXor to enable the highlight to quickly reverse the text, displaying it as white text on a dark background.

Creating and Initializing a New Instance

The ListSelection is automatically created whenever a widget is created that needs it, such as ScrollListBox and MultiListBox, so you would not normally need to create an instance.

The following script is an example of how to create a new instance of the ListSelection class. The parent keyword argument is required, and determines which 2D presenter the instance is to be put into.

```
myListSel := new ListSelection \
             parent:myScrollListBox \
             width:myScrollListBox.width \
             font:myFont
```

The variable myListSel contains the initialized list selection rectangle. The rectangle is put into myScrollListBox, its width is the same as myScrollListBox, and its size makes it fit the specified font. The new method uses the keywords defined in init.

init

`init` *self* `parent:`*twoDPresenter* [`fill:`*brush*] [`stroke:`*brush*] [`width:`*number*] [`font:`*fontContext*] ⇨ *(none)*

self	TwoDShape object
parent:	TwoDPresenter object
fill:	Brush object
stroke:	Brush object
width:	Number object
font:	FontContext object

Initializes the ListSelection object *self*, making *self* a subpresenter of the parent object, and applying the arguments to the instance variables of the same name.

The inherited keywords target and boundary are not shown here because they are not needed—the list selection's size is determined by the width and font size. However, those keywords work the same as with any 2D shape. Do not call init directly on an instance—it is automatically called by the new method.

If you omit an optional keyword, its default is used. The defaults are:

```
fill:(new Brush color:blackColor)
```

```
stroke:undefined
width:10
font:theSystemFont
stationary:false
```

Class Variables

fill

self.`fill` (read-write) `Brush`

Specifies the color that fills all instances of the `ListSelection` rectangle. This is a class variable so that all instances of `ListSelection` will have the same color.

Instance Variables

Inherited from `Presenter`:

`presentedBy`	`subPresenters`	`target`

Inherited from `TwoDPresenter`:

`bBox`	`height`	`transform`
`boundary`	`IsImplicitlyDirect`	`width`
`clock`	`isTransparent`	`window`
`compositor`	`isVisible`	`x`
`direct`	`needsTickle`	`y`
`eventInterests`	`position`	`z`
`globalBoundary`	`stationary`	
`globalTransform`	`target`	

Inherited from `TwoDShape`:

`fill`	`stroke`	`target`

Instance Methods

Inherited from `TwoDPresenter`:

`adjustClockMaster`	`inside`	`show`
`createInterestList`	`localToSurface`	`surfaceToLocal`
`draw`	`notifyChanged`	`tickle`
`getBoundaryInParent`	`recalcRegion`	
`hide`	`refresh`	

The following instance methods are defined in `ListSelection`:

changeFont

`changeFont` *self font lineNum* ⇨ *(none)*

self	`ListSelection` object
font	`FontContext` object
lineNum	`Integer` representing the line number to be highlighted

Changes the height and position of the `ListSelection` rectangle *self* based on the size of the given *font* and *lineNum*.

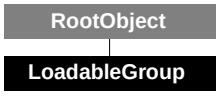
selectLine

`selectLine` *self lineNum doAction* ⇨ `Number`

self	`ListSelection` object
lineNum	`Integer` representing the line number to be highlighted
doAction	Ignored

Sets the y-value of the `ListSelection` rectangle *self*, based on the line number *lineNum*. The *doAction* argument is ignored.

LoadableGroup



Class type: Core class (concrete)
Resides in: ScriptX and KMP executables
Inherits from: `RootObject`
Component: Loader

The `LoadableGroup` class represents groups of loadable units that can be loaded by the Loader component. Each group maintains a load list of units to be loaded whenever the group is processed by the Loader component, and a unit list of all the units in the group.

Creating and Initializing a New Instance

`LoadableGroup` instances are created and saved by instances of the `Loader` class. You do not instantiate a Loadable Group object directly.

Instance Methods

getHandle

`getHandle` *self* ⇨ `String`

Returns the `String` representing the complete path to the directory containing the loadable group. This string can be used in the `spawn` method called on `theRootDir` global to create a `DirRep` representing the directory where the loadable group file resides, as the following code demonstrates:

```
myGroupDir := spawn theRootDir (getHandle myGroup)
```

getLoadableUnit

`getLoadableUnit` *self name* ⇨ `LoadableUnit`

self	`LoadableGroup` object
name	`NameClass` object

Checks the unit list of the loadable group *self* and returns the loadable unit whose name matches the argument *name*.

getLoadList

`getLoadList` *self* ⇨ *(object)*

Returns the load list of the loadable group *self*.

getUnitList

`getUnitList` *self* ⇨ *(object)*

Returns the unit list of the loadable group *self*.

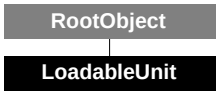
initializeGroup

`initializeGroup` *self filePath loadList unitList* ⇨ `LoadableGroup`

self	`LoadableGroup` object
filePath	`String` object
loadList	`Collection` object
unitList	`Collection` object

Initializes the loadable group *self* from the path represented by *filePath*, the given *loadList*, and the given *unitList*.

LoadableUnit



Class type: Core class (concrete)
Resides in: ScriptX and KMP executables
Inherits from: `RootObject`
Component: Loader

The `LoadableUnit` class defines the atomic units for the Loader component. Units have a type that indicates how the Loader component should handle them during the load process. They also have a system and unit version, to prevent version skew and to track which version is loaded.

Creating and Initializing a New Instance

`LoadableUnit` instances are created and returned by instances of the `Loader` class.

Instance Variables

entry

self`.entry` (read-only) *(function)*

Represents the entry function for the loadable unit *self*.

Although any global function, anonymous function, or method can be assigned to `entry`, there are differences in how different classes of functions are dispatched. For information on functions and function dispatch, see the "Object System Kernel" chapter of the *ScriptX Components Guide*.

exit

self`.exit` (read-only) *(function)*

Represents the exit function for the loadable unit *self*.

Although any global function, anonymous function, or method can be assigned to `exit`, there are differences in how different classes of functions are dispatched. For information on functions and function dispatch, see the "Object System Kernel" chapter of the *ScriptX Components Guide*.

Instance Methods

clrOnLoadList

`clrOnLoadList` *self* ⇨ *(none)*

Clears the loadable unit *self* from the load list.

getEntry

`getEntry` *self* ⇨ `NameClass`

Returns the name of the entry point for the loadable unit *self*.

getFile

`getFile` *self* ⇨ `NameClass`

Returns a `Pair` object representing the `DirRep` subclass instance for the directory and the `NameClass` instance for the object file associated with the loadable unit *self*.

getHandle

`getHandle` *self* ⇨ `NameClass`

Returns the name of the loadable unit *self*.

getSysVers

`getSysVers` *self* ⇨ `Integer`

Returns the system version of the loadable unit *self*.

getUnitType

`getUnitType` *self* ⇨ *(object)*

Returns the type of the loadable unit *self*.

getUnitVers

`getUnitVers` *self* ⇨ `Integer`

Returns the unit version of the loadable unit *self*.

initializeUnit

`initializeUnit` *self sourcePath unitName sourceFile entryPoint sysVersion unitVersion unitType* ⇨ *(object)*

self	`LoadableUnit` object
`source Path`	`DirRep` object
`unitName`	`NameClass` object
`sourceFile`	`NameClass` object
`entryPoint`	`NameClass` object
`sysVersion`	`Integer` object
`unitVersion`	`Integer` object
`unitType`	`NameClass` object

Initializes the loadable unit *self* and assigns it the directory representative *sourcePath* representing its directory, the name *unitName*, the list of source files *sourceFile*, the entry point *entryPoint*, the system version *sysVersion*, the unit version *unitVersion*, and the type *unitType*. The *unitType* argument can be one of `@loaderTypeEphemeral` or `@loaderTypeLinkable`.

isOnLoadList

`isOnLoadList` *self* ⇨ `Boolean`

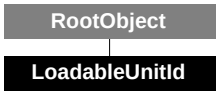
Determines whether the loadable unit *self* is on its group's load list. Returns `true` if so, `false` if not.

setOnLoadList

`setOnLoadList` *self* ⇨ *(none)*

Sets the loadable unit *self* on the load list.

LoadableUnitId



Class type: Core class (concrete)
Resides in: ScriptX and KMP executables
Inherits from: `RootObject`
Component: Loader

The `LoadableUnitId` class provides...

Instance of `LoadableUnitId` are returned by the `Loader` method `loadModule` when it loads a unit. The `Loader` methods `process` and `processGroup` return collections of these objects, one for each unit loaded successfully. The ScriptX runtime environment defines four instances of `LoadableUnitId` as global constants. These globals are returned by `loadModule` to represent loading errors.

Creating and Initializing a New Instance

There is no need to create instances of `LoadableUnitId` since four global instances already exist. The global instances `loadableUnitIdNull`, `loadableUnitIdError`, `loadableUnitIdLoading`, and `loadableUnitIdInitErr` are defined in the chapter "Global Constants and Variables."

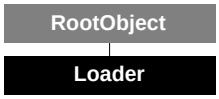
Instance Methods

relinquish

`relinquish` *self* ⇨ *(none)*

Releases the code represented in a loadable unit identified by the loadable unit ID *self*, and makes its memory available for garbage collection.

Loader



Class type: Core class (concrete)
Resides in: ScriptX and KMP executables
Inherits from: `RootObject`
Component: Loader

The `Loader` class is the primary class for the Loader component. It maintains the tables of currently loading and successfully loaded units. An instance of this class drives the load and unload operations. It obtains help from `LoaderHelper` to load units into platform-specific contexts.

Creating and Initializing a New Instance

The following script creates a new instance of the `Loader` class:

```
myLoader := new Loader

The variable myLoader contains a new instance of the Loader class.
```

init

`init` *self* ⇨ *(none)*

self	`Loader` object

Initializes the `Loader` object *self*. Do not call `init` directly on an instance—it is automatically called by the `new` method.

Class Methods

exportNamedObject

`exportNamedObject` *self name value* ⇨ *value*

self	`Loader` class
name	`NameClass` instance representing the symbol
value	Object representing the address of an object to export

Adds the object *value* with the specified name *name* to the Loader component's list of exported symbols. It returns *value*.

loaderError

`loaderError` *self id* ⇨ `Boolean`

self	`Loader` class
id	`LoadableUnitId` object

Checks to see if the loadable unit ID identified by *id* was successfully loaded by loader *self*. Returns `true` if the *id* argument equals `LoadableUnitIdError`; otherwise, it returns `false`.

nameToExport

`nameToExport` *self name* ⇨ *(object)*

self	`Loader` class
name	`NameClass` object

Converts the specified `NameClass` object *name* into an exportable name. Returns the export value matching the name.

releaseLoadableUnit

`releaseLoadableUnit` *self id* ⇨ `LoaderCode`

self	`Loader` class
id	`LoadableUnitId` object

Releases the loadable unit identified by *id*. Returns an instance of `LoaderCode` representing the results of the operation.

unexportNamedObject

`unexportNamedObject` *self name* ⇨ *(object)*

self	`Loader` class
name	`NameClass` object

Removes the object with the specified name *name* from the Loader component's list of exported symbols.

Instance Methods

getGroup

`getGroup` *self filePath* ⇨ `LoadableGroup`

self	`Loader` object
filePath	`String` object for loadable group file

Returns a loadable group in the specified *filePath* from the loader *self*. The string *filePath* is relative to `theStartDir`, so that passing in `"foo/bar"` access the directory specified by `(spawn theStartDir "foo/bar")`.

loaderValue

`loaderValue` *self id* ⇨ *(object)*

self	`Loader` object
id	`LoadableUnitId` object

Returns the value returned from the entry point code in the loader *self*, for the loadable unit represented by *id*. This entry point code is called from within the `Loader` instance method `loadModule` before it returns the *id* for the unit.

loadModule

`loadModule` *self group unit* ⇨ `LoadableUnitId`

self	`Loader` object
group	`LoadableGroup` instance to load from
unit	`LoadableUnit` instance to load

Loads the specified *unit* from the *group*. If successful, returns a `LoadableUnitId` instance representing the loaded unit. If unsuccessful, it returns a global instance of `LoadableUnitId` indicating the error condition.

processGroup

`processGroup` *self group* ⇨ `Collection`

self	`Loader` object
group	`LoadableGroup` object

Processes the *group*. Returns a collection of loadable unit ids representing the units successfully loaded. (The collection doesn't include error instances of `LoadableUnitId` for units that failed to load.)

process

`process` *self filePath* ⇨ `Collection`

self	`Loader` object
filePath	`String` object for the loadable unit

Processes the loadable group in the file represented by *filePath*. Returns a collection of loadable unit ids in the loader *self* representing the units successfully loaded. (The collection doesn't include error instances of `LoadableUnitId` for units that failed to load.) The string *filePath* is relative to `theStartDir`, so that passing in `"foo/bar"` access the directory specified by `(spawn theStartDir "foo/bar")`.

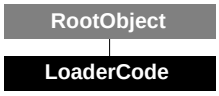
saveGroup

`saveGroup` *self group* ⇨ `Boolean`

self	`Loader` object
group	`LoadableGroup` object

Saves the *group* (and all loadable units within it) in the loader *self*, using the *name* instance variable of the `LoadableGroup` object as the filename.

LoaderCode



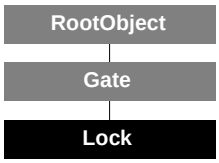
Class type: Core class (concrete)
Resides in: ScriptX and KMP executables
Inherits from: `RootObject`
Component: Loader

Instances of `LoaderCode` are returned by `releaseLoadableUnit` method (defined in `Loader`) to indicate the results of the operation.

Creating and Initializing a New Instance

There is no need to create instances of `LoaderCode` since three global instances already exist. The global instances `loaderCodeBad`, `loaderCodeError`, and `loaderCodeOk` are defined in the chapter "Global Constants and Variables."

Lock



Class type: Core class (concrete, sealed)
Resides in: ScriptX and KMP executables
Inherits from: `Gate`
Component: Threads

The `Lock` class is an object that only one thread may own at any given time. When a thread acquires a lock, the `state` instance variable is set to `@closed` to block the thread from executing. The thread does not continue execution until the `state` instance variable is set to `@open` (the lock is relinquished).

`Thread` objects can wait on a `Lock` object, and, when they become active, they are guaranteed that no other thread that waited on that `Lock` object is currently active. Threads wait on a lock with `gateWait`, `acquire`, or one of the combination operations. (The `gateWait` global function is the same as the `acquire` method on `Gate`, except that `acquire` is a generic function.) Threads allow another waiting thread to acquire the lock with either the `openGate` global function or the `relinquish` method on `Gate`.

A thread is allowed to acquire the same `Lock` object multiple times without releasing it. However, it must relinquish the lock the same number of times it acquired the lock before another thread can acquire it. A thread can only relinquish locks that it has acquired. Attempting to relinquish a lock that has another owner throws the `wrongOwner` exception. Attempting to relinquish a lock that no thread owns reports the `notLocked` exception. If a thread finishes execution or is killed while it owns a lock, the lock is automatically released.

It is often convenient, and visually pleasing, to wrap code that is associated with a lock within a code block, as in the following example. If two threads have access to this block of code, only one will be able to “do some stuff” at a time—the two threads have serial access to this code.

```
acquire myLock
(
    -- do some stuff
)
relinquish myLock
```

Creating and Initializing a New Instance

The following script is an example of how to create a new instance of the `Lock` class:

```
myLock := new Lock \
       label:"lock1"
```

The variable `myLock` contains the initialized lock, with the label "`lock1`". The `new` method uses the keywords defined in `init`.

init

`init` *self* [`label:`*object*] ⇨ *(none)*

self	`Lock` object
`label:`	Any object

Initializes the `Lock` object *self*, where the value supplied with `label` is applied to the `label` instance variable. Do not call `init` directly on an instance—it is automatically called by the `new` method.

If you omit an optional keyword, its default value is used. The defaults are:

```
label:undefined
```

Instance Variables

Inherited from `Gate`:

`label` `state`

The following instance variable is defined by `Lock`:

thread

self`.thread` (read-only) ⇨ `Thread`

Specifies the thread that currently owns the lock *self*, or `undefined` if the lock currently has no owner.

Instance Methods

Inherited from `Gate`:

`acquire` `relinquish`

The following instance methods are defined in `Lock`:

acquire (Gate)

`acquire` *self* ⇨ *(none)*

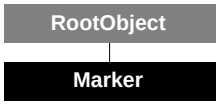
Acquires the lock *self*. This directly translates into `gateWait` *self*. A thread that calls `acquire` on a `Lock` object blocks if another thread already owns the lock; however, if no other thread owns the lock, then the thread that called `acquire` gets the lock and continues to run.

relinquish (Gate)

`relinquish` *self* ⇨ *(none)*

Relinquishes the lock *self*. This translates directly into the global function `gateOpen` *self*. If a thread attempts to relinquish a lock that is owned by another thread, ScriptX reports the `wrongOwner` exception. If a thread attempts to relinquish a lock that is not owned by any thread, ScriptX reports the `notLocked` exception.

Marker



Class type: Core class (concrete)
Resides in: ScriptX and KMP executables
Inherits from: `RootObject`
Component: Players

The `Marker` class provides a means of annotating the start and finish times of time ranges in a media stream. Markers can be attached to either a media stream or a player and have meaning only within that context.

When a media stream is imported, if it has any existing markers, they are imported with it as `Marker` instances. For example, `Marker` objects are created from information in the marker chunk of an AIFF audio file and are added to the `markerList` of the corresponding `AudioStream`. The `markerList` of the `AudioStream` is copied to the `DigitalAudioPlayer` it is assigned to. Markers can also be created explicitly at any other time and added to the `markerList` of a `MediaStream` or `Player` instance by using the `addMarker` method.

Note that time ranges for markers can be nested or can overlap each other. For example, if `MarkerA` has a start time of 10 and a finish time of 100 and `MarkerB` has a start time of 50 and a finish time of 200, these two markers can both be added to a player.

If the start time of the marker is the same as its finish time, then the marker can be used to mark a point in time rather than a range.

Creating and Initializing a New Instance

The following script is an example of how to create a new instance of the `Marker` class:

```
myMarker := new Marker \
            start:1 \
            finish:50 \
            label:"First marker"
```

In this example, the variable `myMarker` points to an instance of `Marker` whose start time is 1 and whose finish time is 50. That is, this marker can be used to mark the range from tick 1 to tick 50 for a media stream or media stream player.

The `new` method uses the keywords defined in `init`.

init

`init` *self* `start:`*number* [`finish:`*number*] `label:`*string* ⇨ *(none)*

self	`Marker` object
`start:`	A `Number` indicating the starting value of the marker in ticks.
`finish:`	A `Number` indicating the finishing value of the marker in ticks.
`label:`	A `String` to be used as the marker's label.

Initializes the `Marker` object *self*, setting the values of the `start`, `finish` and `label` instance variables to the corresponding keyword values. Do not call `init` directly on an instance—it is automatically called by the `new` method.

If you omit an optional keyword, its default value is used. The defaults are:

`finish:` The same as start.

Instance Variables

finish

self.`finish` (read-write) `Integer`

Specifies the finish of the time range marked by the marker *self*.

label

self.`label` (read-write) `Text`

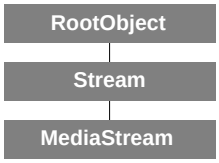
Specifies the name or description of the marker *self*. The author can search for this label in the list of markers on a particular media stream or player.

start

self.`start` (read-write) `Integer`

Specifies the start of the time range marked by the marker *self*.

MediaStream



Class type: Core class (abstract)
Resides in: ScriptX and KMP executables
Inherits from: `ByteStream`
Component: Media Players

The `MediaStream` class defines methods for opening and accessing streams of media data. `MediaStream` encompasses the behavior common to streams of digital media data of all types, while subclasses such as `DigitalAudioStream` and `DigitalVideoStream` provide media-specific implementations.

In `MediaStream` and its subclasses, a frame represents an individual, coherent piece of media data. The precise content of a frame depends on the media-specific implementation provided by subclasses of `MediaStream`. For a video stream, a frame is the data necessary to generate a single complete image. For an audio stream, a frame is a collection of sound samples—one sample per channel of audio (for example, 1 sample for mono, 2 samples for stereo and 4 samples for quad).

Instance Variables

dataRate

self`.dataRate` (read-only) `Integer`

Specifies the number of bytes per second that the stream *self* needs to read in order not to skip presenting some of its media.

inputStream

self`.inputStream` (read-write) `ByteStream`

Specifies the source of the raw media data which the media stream *self* will interpret as media. (A `MediaStreamPlayer` plays its `mediaStream` which in turn has an `inputStream`.)

isChunked

self`.isChunked` (read-only) `Boolean`

Specifies whether or not the input stream for the stream *self* is a ChunkStream object.

isSuppressed

self`.isSuppressed` (read-only) `Boolean`

Specifies whether or not the data for the stream *self* is suppressed or not.

For example, if the muted instance variable of a digital audio player is set to true, the player's audio stream would be suppressed. If the blanked instance variable of a digital video player is set to true, the player's video stream would be suppressed.

markerList

self.`markerList` (read-only) `SortedKeyedArray`

Specifies the sequence of markers associated with the media stream *self*. When a media file is imported, markers in the media stream are retained. Markers on the player supersede markers on the media in marker searches begun from the player.

rate

self.`rate` (read-write) `Integer`

Specifies the portion of the player's rate that the stream *self* should be responsible for emulating (such as by dropping samples of audio or frames of video). Defaults to 1.0 meaning that the media stream should deliver data in the normal fashion.

If this rate is less than 1.0, then the apparent size of the media stream will increase so that more samples can be read. The additional samples can be generated by interpolating between existing samples or by repeating some samples. If this rate is greater than 1.0, then the apparent size will decrease and fewer samples can be read. If this rate is negative, samples will be returned in reverse order.

Users should not set this instance variable directly. It is set by calling either the `playPrepare` or `play` method on the player playing the stream *self*.

sampleType

self.`sampleType` (read-only) `Integer`

Describes the representation used for samples in the stream *self*. See the individual `MediaStream` subclasses for a description of the sample types specific to those streams.

scale

self.`scale` (read-only) `Integer`

Specifies the number of frames that need to be presented per second to properly present the media stream *self*. Players accessing media streams usually adopt this value as their scale so that all player times can be easily mapped to positions within the media stream.

variableFrameSize

self.`variableFrameSize` (read-only) `Boolean`

Specifies whether the frames in the media stream *self* are all the same size or whether they vary in size

If the value is `true`, all the frames are not necessarily the same number of bytes long.

If the value is `false`, every frame of the media (a sound sample, a single picture in an animation or movie, and so on) is the same number of bytes long.

Instance Methods

Inherited from `Stream`:

`cursor`	`next`	`seekFromStart`
`flush`	`plug`	`setStreamLength`
`isAtFront`	`previous`	`streamLength`
`isPastEnd`	`read`	`write`
`isReadable`	`readReady`	`writeReady`
`isSeekable`	`seekFromCursor`	
`isWritable`	`seekFromEnd`	

The following instance methods are defined in `MediaStream`:

addMarker

`addMarker` *self marker* ⇨ `Marker`

self	`MediaStream` object
marker	`Marker` *object*

Adds the `Marker` object *theMarker* to the media stream *self*. The marker is added to the array in the media stream's `markerList` instance variable. Returns *theMarker* if the addition is successful, `undefined` if not. A marker has a start and finish time, and can be used to mark time ranges on the stream *self*.

isReadable (Stream)

`isReadable` *self* ⇨ `true`

Returns `true`, since all `MediaStream` instances are intended to be readable.

isSeekable (Stream)

`isSeekable` *self* ⇨ `true`

Returns `true`, since all `MediaStream` instances are intended to be seekable.

isWritable (Stream)

`isWritable` *self* ⇨ `false`

Returns `false`, since `MediaStream` instances aren't intended to be writable.

prepareStream

`prepareStream` *self param1 param2* ⇨ `Boolean`

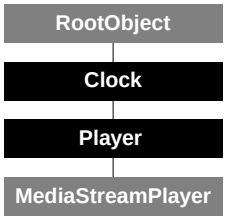
Prepares the media stream *self* to play by initializing any resources used by the stream. In media production terms, this may be thought of as "pre-roll." Subclasses of `MediaStream` may implement `prepareStream` to allocate file streams, acquire hardware devices, locate codecs, and so on. The `MediaStreamPlayer` method `playPrepare` calls this method on the player's `mediaStream`. The arguments *param1* and *param2* are used differently by the different `MediaStream` subclasses. If not documented in a particular subclass, they are unused.

unprepareStream

`unprepareStream` *self* ⇨ `Boolean`

Releases resources initialized for playing the media stream *self* by `prepareStream`. For example, subclasses of `MediaStream` may implement `unprepareStream` to release file streams, hardware devices, or codecs claimed by the media stream when initialized. The `MediaStreamPlayer` method `playUnprepare` calls this method on a player's `mediaStream`.

MediaStreamPlayer



Class type: Core class (abstract)
Resides in: ScriptX and KMP executables
Inherits from: `Player`
Component: Media Players

The `MediaStreamPlayer` class extends the behavior defined in the `Player` class to provide generalized control for playing streams of media. `MediaStreamPlayer` provides methods to deal with the common characteristics of various digital media types; for example, for dealing with rate and time changes in the playback of digital media.

In `MediaStreamPlayer` and its subclasses, a frame represents an individual, coherent piece of media data. The precise content of a frame depends on the media-specific implementation provided by subclasses of `MediaStreamPlayer`. For a video player, a frame is the data necessary to generate a single complete image. For an audio player, a frame is a collection of sound samples that fill a single unit of the sampling rate (for example, the sound samples that occupy 1/44,000 of a second at a 44kHz sampling rate). An audio frame includes one sample per channel of audio; thus, 1 sample for mono, 2 samples for stereo and 4 samples for quad.

Note – The inherited `status` instance variable has two additional possible values for `MediaStreamPlayer`. The value `@present` indicates that media is present but the the `playPrepare` method has not been called on the player so it is not ready to play. The value `@notPresent` indicates that media is not present and, thus, the player cannot play.

Instance Variables

Inherited from `Clock`:

`callbacks`	`rate`	`ticks`
`effectiveRate`	`resolution`	`time`
`masterClock`	`scale`	`title`
`offset`	`slaveClocks`	

Inherited from `Player`:

`audioMuted`	`globalContrast`	`globalVolumeOffset`
`dataRate`	`globalHue`	`markerList`
`duration`	`globalPanOffset`	`status`
`globalBrightness`	`globalSaturation`	`videoBlanked`

The following instance variables are defined in `MediaStreamPlayer`:

frameRate

self`.frameRate` (read-write) `Integer`

Specifies the number of frames per “second” that the player *self* plays the frames in its media stream.

The `frameRate` reflects the true frame rate of the media while `scale` represents the tick rate used by the player to present that media.

Although the `frameRate` instance variable has read-write properties, the only occasion when you should write to this instance variable is when defining a new class of `MediaStreamPlayer`. In this case, the `new` method or a method that sets the value of the `mediaStream` instance variable might need to access the `frameRate` instance variable.

mediaStream

self`.mediaStream` (read-write) `MediaStream`

Specifies the stream that the player *self* will play.

Instance Methods

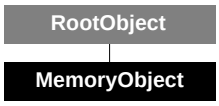
Inherited from `Clock`:

`addPeriodicCallback`	`clockAdded`	`pause`
`addRateCallback`	`clockRemoved`	`resume`
`addScaleCallback`	`effectiveRateChanged`	`timeJumped`
`addTimeCallback`	`forEachSlave`	`waitTime`
`addTimeJumpCallback`	`isAppropriateClock`	`waitUntil`

Inherited from `Player`:

`addMarker`	`goToBegin`	`playPrepare`
`eject`	`goToEnd`	`playUnprepare`
`fastForward`	`goToMarkerFinish`	`playUntil`
`getMarker`	`goToMarkerStart`	`resume`
`getNextMarker`	`pause`	`rewind`
`getPreviousMarker`	`play`	`stop`

MemoryObject

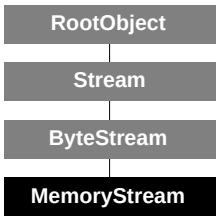


Class type: Core class (concrete)
Resides in: ScriptX and KMP executables
Inherits from: `RootObject`
Component: Object System Kernel

The `MemoryObject` class provides a template for creating an object that can be shared between the object world of the Kaleida Media Player and the C world defined by ScriptX loadable units. A memory object is used to exchange data between ScriptX and loadable extensions written in C.

`MemoryObject` is documented in the "Extending ScriptX" chapter of the *ScriptX Tools Guide*.

MemoryStream



Class type: Core class (concrete)
Resides in: ScriptX and KMP executables
Inherits from: `ByteStream`
Component: Streams

The `MemoryStream` class defines a subclass of `ByteStream` that represents a growable buffer in memory. Using an instance of `MemoryStream`, you can store and access arbitrary sequences of bytes in memory.

Creating and Initializing a New Instance

The following script is an example of how to create a new instance of the `MemoryStream` class:

```
myStream := new MemoryStream \
          initialSize:10 \
          growSize:10 \
          maxSize:100
```

The variable `myStream` contains a new instance of the `MemoryStream` class. The initial size of its memory buffer is 10 bytes, this buffer can grow by 10 bytes at a time, and the buffer's maximum size is 100 bytes. The `new` method uses keywords defined in `init`.

init

`init` *self* [`initialSize:`*integer*] [`growSize:`*integer*] [`maxSize:`*integer*] ⇨ *(none)*

self	`MemoryStream` object
`initialSize:`	`Integer` object
`growSize:`	`Integer` object
`maxSize:`	`Integer` object

Initializes the `MemoryStream` object *self*, applying the arguments as follows: `initialSize` represents its starting size, `growSize` represents the amount by which it grows when necessary, and `maxSize` represents the maximum size it can grow to. Do not call `init` directly on an instance—it is automatically called by the `new` method.

If you omit an optional keyword, its default is used. The defaults are:

`initialSize:500`
`growSize:500`
`maxSize:` limited by available memory

Instance Methods

Inherited from `Stream`:

`cursor`	`next`	`seekFromStart`
`flush`	`plug`	`setStreamLength`
`isAtFront`	`previous`	`streamLength`
`isPastEnd`	`read`	`write`
`isReadable`	`readReady`	`writeReady`

```
isSeekable              seekFromCursor
isWritable              seekFromEnd
```

Inherited from `ByteStream`:

```
fileIn                  readByte                writeString
pipe                    readReady
pipePartial             writeByte
```

The following instance methods are defined in `MemoryStream`:

flush (Stream)

`flush` *self* ⇨ `undefined`

Has no effect on instances of `MemoryStream`.

isReadable (Stream)

`isReadable` *self* ⇨ `true`

Returns `true`. All memory streams are readable.

isSeekable (Stream)

`isSeekable` *self* ⇨ `true`

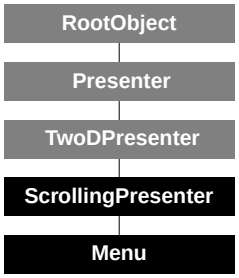
Returns `true`. All memory streams are seekable.

isWritable (Stream)

`isWritable` *self* ⇨ `true`

Returns `true`. All memory streams are writable.

Menu



Class type: Core class (concrete)
Resides in: ScriptX and KMP executables
Inherits from: `ScrollingPresenter`
Component: User Interface

The `Menu` class creates a selection of choices arranged in rows and columns by a row-column controller. A menu is a scrolling presenter that expects to contain other objects, typically actuators. A visible menu is associated with an invoker, which is an object the user clicks on that causes the menu to pop up. A menu's invoker is specified by the `invoker` instance variable. A menu can have more than one invoker; that is, there can be more than one way to pop up a menu. You can determine where the menu pops up, either to the right of the invoker, below the invoker, or at the mouse pointer, using the `placement` instance variable.

A `Menu` object has a default `RowColumnController` object that organizes the objects it contains into a single column, and an `ActuatorController` object that controls the selection of items from the list. The `RowColumnController` protocol allows a menu to grow to contain all of the `Actuator` objects, but it prevents it from growing outside the boundaries of the window. If a `Menu` object is too long or wide to fit into its window, it is clipped automatically.

A `Menu` object has an event interest that pops itself down if a `mouseDown` happens outside of itself and it currently does not have a submenu.

The `Menu` class creates its target presenter automatically, an instance of `TwoDSpace`. A menu serves as a proxy for its own space when adding and removing actuators. Any method calls and variables that a menu does not define itself are redirected to its target space. This means that the following expressions are equivalent:

```
myMenu.targetPresenter.size
myMenu.size
```

The following expressions also are equivalent, given that `cancelButton` is an actuator that is defined elsewhere:

```
prepend myMenu.targetPresenter cancelButton
prepend myMenu cancelButton
```

In the examples above, `size` is an instance variable defined by `Collection`, but it is redirected to the menu because its target presenter, an instance of `TwoDSpace`, is a collection. Since `TwoDSpace` is an indirect collection, it can also apply methods that are defined for its target collection, which is an instance of `Array`, by default. The `prepend` method, shown in the second example, is defined by `Sequence`. (An array is a sequence.)

Creating and Initializing a New Instance

The following script is an example of how to create a new instance of the Menu class:

```
myMenu := new Menu \
       placement:@menuDown
```

The variable myMenu contains the initialized menu. The value @menuDown specifies that this menu is a pulldown menu. The new method uses keywords defined in init.

To add an item to myMenu, call an appropriate collection method, which is redirected to the targetPresenter:

```
append myMenu myItem
```

init

init *self* [placement:*name*] [actuatorController:*actuatorController*]
[layoutController:*rowColumnController*] [fill:*brush*]
[stroke:*brush*] [horizScrollBar:*scrollBar*] [vertScrollBar:*scrollBar*]
[boundary:*rect*] [targetPresenter:*twoDSpace*] [target:*object*]
[stationary:*boolean*] ⇨ *(none)*

self	Menu object
placement:	NameClass object
actuatorController:	ActuatorController object
layoutController:	RowColumnController object

The superclass ScrollingPresenter uses the following keywords:

fill:	Brush object
stroke:	Brush object
horizScrollBar:	ScrollBar object
vertScrollBar:	ScrollBar object

Superclasses of ScrollingPresenter use the following keywords:

boundary:	Rect object
targetPresenter:	TwoDSpace object
target:	Any object (ignored by Menu)
stationary:	Boolean object

Initializes the Menu object *self*, applying the values supplied with the keywords to the instance variables of the same name. The init method automatically creates a RowColumnController object and an ActuatorController object attached to the menu space, or you can specify controllers for your menu using the controller keywords. Specify undefined for the value of the controller keywords if you do not want these controllers for your menu. Any controller that you specify for the value of the actuatorController keyword is passed on to the menu keyword of the ActuatorController instance. Do not call init directly on an instance—it is automatically called by the new method.

If you omit an optional keyword, its default value is used. The defaults are:

```
placement:@menuDown
actuatorController:@unsupplied
layoutController:@unsupplied
fill:undefined
stroke:undefined
horizScrollBar:undefined
vertScrollBar:undefined
boundary:(new Rect x2:0 y2:0)
```

```
targetPresenter:(new TwoDSpace)
target:undefined
stationary:false
```

The `targetPresenter` keyword, defined by `TwoPresenter`, is ignored, since a `Menu` object sets its own target presenter. A menu automatically creates an instance of `TwoDSpace` to act as its target presenter. The `target` keyword, defined by `Presenter`, is also ignored.

Instance Variables

Inherited from `Presenter`:

`presentedBy`	`subPresenters`	`target`

Inherited from `TwoDPresenter`:

`bBox`	`height`	`transform`
`boundary`	`IsImplicitlyDirect`	`width`
`clock`	`isTransparent`	`window`
`compositor`	`isVisible`	`x`
`direct`	`needsTickle`	`y`
`eventInterests`	`position`	`z`
`globalBoundary`	`stationary`	
`globalTransform`	`target`	

Inherited from `ScrollingPresenter`:

`clippingStencil`	`stroke`
`fill`	`targetPresenter`
`horizScrollBar`	`vertScrollBar`
`horizScrollBarDisplayed`	`vertScrollBarDisplayed`

Since a menu acts as a proxy for the instance of `TwoDSpace` that is its target presenter, the following instance variables are redirected to this space.

Redirected from `Space`:

`clock`	`protocols`	`tickleList`
`controllers`		

Redirected from `Collection`:

`bounded`	`maxSize`	`size`
`iteratorClass`	`minSize`	`uniformity`
`keyEqualComparator`	`mutable`	`uniformityClass`
`keyUniformity`	`mutableCopyClass`	`valueEqualComparator`
`keyUniformityClass`	`proprietored`	

Redirected from `IndirectCollection`:

`targetCollection`

Redirected from `TwoDMultiPresenter`:

`clock`	`fill`	`stroke`

The following instance variables are defined in `Menu`:

actuatorController

self.actuatorController (read-write) ActuatorController

Specifies an ActuatorController object that will be used to control the selection of items from the menu *self*. The class ActuatorController implements a protocol that allows a menu to pop down where appropriate. Any controller that you specify for the value of this instance variable is passed on to the menu keyword of the ActuatorController instance.

invoker

self.invoker (read-only) *(object)*

Specifies the object that last caused the menu *self* to pop up. If a menu has more than one invoker, then only the last object that invoked the menu is maintained in this instance variable. In most cases, the invoker is a PushButton object. To make an actuator be the invoker of a menu, set the menu instance variable on the Actuator object.

This variable is read-only for the class Menu. Subclasses of Menu that set their own invoker must define a setter method that links that invoker with a Menu object. Normally, this instance variable is set internally by the popup method.

layoutController

self.layoutController (read-write) RowColumnController

Specifies an instance of a RowColumnController that will be used to control the position of items from the menu *self*. By default, this controller sets up menu items in a single column.

placement

self.placement (read-write) NameClass

Determines how the menu *self* will pop up when activated by its invoker. By default, this value is set to @menuDown, indicating that the menu pops down from its invoker. Placement can also be set to @menuRight, which places the menu to the right of the invoker, or @menuAtPointer, which places the menu where the user clicks the mouse.

subMenu

self.subMenu (read-only) Menu

Specifies the currently popped up submenu of the menu *self* if there is one. Thus, *self* is a supermenu of subMenu. A menu may have more than one submenu, but only the currently popped up submenu is maintained in this instance variable. The value of this instance variable is undefined if the menu has no currently popped up submenu.

superMenu

self.superMenu (read-only) Menu

Specifies the menu from which the menu *self* was invoked. Thus, *self* is a submenu of superMenu. The value of this instance variable is undefined if the menu has no parent menu.

targetPresenter (ScrollingPresenter)

self.targetPresenter (read-write) TwoDSpace

Specifies the presenter to be scrolled inside the scrolling list *self*. Note that for the Menu class, this presenter is created automatically, and that it must be an instance of TwoDSpace.

Instance Methods

Inherited from TwoDPresenter:

adjustClockMaster	inside	show
createInterestList	localToSurface	surfaceToLocal
draw	notifyChanged	tickle
getBoundaryInParent	recalcRegion	
hide	refresh	

Inherited from ScrollingPresenter:

handleHorizScrollBar	layout
handleVertScrollBar	scrollTo

Since a menu acts as a proxy for the instance of TwoDSpace that is its target presenter, the following instance methods are redirected to this space.

Redirected from Collection:

add	forEach	iterate
addMany	forEachBinding	localEqual
addToContents	getAll	map
chooseAll	getAny	merge
chooseOne	getKeyAll	pipe
chooseOneBinding	getKeyOne	prin
deleteAll	getMany	removeAll
deleteBindingAll	getOne	removeOne
deleteBindingOne	hasBinding	setAll
deleteKeyAll	hasKey	setOne
deleteKeyOne	intersects	size
deleteOne	isEmpty	
emptyOut	isMember	

Redirected from IndirectCollection:

isAppropriateObject	objectAdded	objectRemoved

Redirected from TwoDMultiPresenter:

draw	findFirstInStencil	moveToBack
findAllAtPoint	isAppropriateObject	moveToFront
findAllInStencil	moveBackward	objectAdded
findFirstAtPoint	moveForward	objectRemoved

Since a TwoDSpace object is an indirect collection, you can also use any methods defined in the class specified by targetCollection. The target collection is typically an instance of Array, which inherits from Sequence, so the following instance methods are redirected to this space.

Accessible from LinearCollection:

chooseOneBackwards	deleteSecond	getMiddle
chooseOrdOne	deleteThird	getNth
deleteFifth	findRange	getNthKey
deleteFirst	forEachBackwards	getOrdOne
deleteFourth	getFifth	getRange
deleteLast	getFirst	getSecond
deleteNth	getFourth	getThird
deleteRange	getLast	pop

Accessible from Sequence:

addFifth	moveBackward	setFourth
addFirst	moveForward	setLast
addFourth	moveToBack	setNth
addNth	moveToFront	setSecond
addSecond	prepend	setThird
addThird	prependNew	sort

append	setFifth
appendNew	setFirst

The following instance methods are defined in Menu:

calculateSize

calculateSize *self* ⇨ Point

Resizes the bounding box of the Menu object *self* to show all items inside the target presenter. It returns a Point object that represents the new width and height of the menu. This method is called automatically by the popup method. Specialize this method if you do not want the menu to resize in this way.

hide

hide *self* ⇨ Boolean

Hides any submenus posted by the menu *self*, then hides the menu *self* and sets the values of invoker and supermenu to undefined. Returns true if it was successful in hiding the menus.

place

place *self point* ⇨*self*

self	Menu object
point	Point object

Determines where the menu *self* should be displayed, based on the menu's invoker and on the location of the event that caused the menu to be popped up. This event is an instance of MouseEvent; the coordinates of the mouse event with respect to the window supply the *point* parameter, which is the x-y location of the pointer when popup was called. This method is called automatically by the popup method.

This method uses the value of the invoker's bounds instance variable to calculate the menu's position, based on the value of the menu's placement instance variable and the value of *point*. If placement is @menuRight, then *point* is not used and the menu is placed to the right of the invoker presenter and adjusted if necessary to fit inside of the composited window area. If placement is @menuDown, then *point* is not used and the menu is placed below the invoker presenter and adjusted if necessary to fit inside of the composited window area. If placement is @menuAtPointer, then the manu's origin (its upper left corner by default) is placed at *point*. If placement does not contain one of these three values, the Menu class reports the exception invalidMenuPlacement. A subclass of Menu could define additional placements.

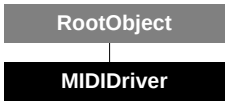
popup

popup *self invoker superMenu point* ⇨ Boolean

self	Menu object
invoker	Presenter object
superMenu	Menu object
point	Point object

Sets the invoker instance variable to *invoker*, sets the superMenu instance variable to *superMenu*, calls calculateSize *self*, calls place *self point*, enables the menu's actuatorController, and finally displays the menu *self*. The value of *invoker* is the presenter that was acted on to trigger the menu to pop up. The value of *point* is the location of the event that caused the menu to be popped up. This event is an instance of MouseEvent; the coordinates of the mouse event with respect to the window supply the *point* parameter, which is the x-y location of the pointer when popup was called. This method returns true if it was successful in presenting the menu.

MIDIDriver



Class type: Core class (concrete)
Resides in: ScriptX and KMP executables
Inherits from: `RootObject`
Component: Media Player

The `MIDIDriver` class represents the MIDI devices that exist in the system. When a `MIDIPlayer` instance prepares to play a MIDI piece, a driver can be provided to it, or it can choose a default driver.

This class acts as an interface to any MIDI hardware or software supported on the native system. It supports a simple interface that allows the client to send short (2 or 3-byte) or long (sysex) standard MIDI events.

The `MIDIDriver` class cannot be instantiated by the user by calling the `new` method. Instead, you use the `openMIDIDriverList` function to find the available MIDI drivers, then use the `openMIDIDriver` function to open a MIDI driver.

The global function `getMIDIDriverList` returns a list of pairs, where each pair corresponds to an installed MIDI driver. In each pair, the first element is a string of the driver's name, and the second element is a name such as `@internal` or `@external`.

The global function `openMIDIDriver` opens a MIDI driver, which is specified as a pair whose first element is a string of the driver's name, and the second element is a name such as `@internal` or `@external`. The function `openMIDIDriver` returns the opened `MIDDriver` object if it was successful, or `false` if it was not successful. (Note that just because a MIDI driver is installed does not mean it can be opened. If a driver for hardware is installed, but the hardware is not connected, the driver cannot be opened.)

You can use the following code to bind `Mdriver` to a MIDI driver that works on any MIDI-capable system. (If no working MIDI driver can be found, `Mdriver` ends up being undefined.)

```
global mdList := getMidiDriverList ()
global Mdriver := false
for md in mdlist until Mdriver do
    Mdriver := openMidiDriver md
```

See the Media Players chapter in the *ScriptX Components Guide* for more information on finding MIDI drivers.

On any ScriptX platform, MIDI capabilities may be provided by an external card and its accompanying driver. On MacOS, MIDI is generally implemented through software as part of QuickTime. Some MacOS systems may be equipped with a hardware device that loads its own driver, although most consumer systems are not.

Note – To insure cross-platform compatibility when you load an instance of `MIDIDriver`, set the value of `channelPolyphony` and call the `prepareDriver` method on it before sending it any messages. (Systems that implement MIDI through hardware can play MIDI streams without these steps.)

Class Methods

midiAvailable

self.`midiAvailable` ⇨ `Boolean`

Returns `true` if MIDI support is available on the system or `false` if not.

Instance Variables

channelPolyphony

self.`channelPolyphony` (read-write) `Array`

Specifies the maximum polyphony per channel for 16 channels. The value must be an array of 16 elements. The value of each elements indicates the polyphony (number of voices) for that channel.

You must set this instance variable before calling the `prepareDriver` method on the `MIDIDriver`.

Instance Methods

allNotesOff

`allNotesOff` *self* ⇨ `Boolean`

This sends the all-notes-off messages for all channels to the synthesizer to which the driver *self* interfaces.

The MIDI protocol encodes music in terms of note-on/note-off events. (It's a bit more sophisticated than that in reality but that's the basic principal). These events are sent to specific channels, each of which might represent a different instrument. There are 16 possible channels.

The all-notes-off message effectively acts as a note-off event for all notes of a channel. The `allNotesOff` method sends this message to all channels.

prepareDriver

`prepareDriver` *self* *obj* ⇨ `Boolean`

self	`MIDIDriver` object
obj	`Object` that supports the `playUnprepare` method, or `undefined`.

This method prepares the MIDI driver. If you are directly interacting with the MIDI driver (rather than going through a MIDI player) you must call this method after setting the MIDI driver's `channelPolyphony` instance variable, and before sending any messages to the driver. The *obj* argument should be an object that supports the `playUnprepare` method or `undefined`. If you call this method directly, you can usually pass the *obj* argument as `undefined`.

sendMIDIEvent

`sendMIDIEvent` *self* *midiEvent* *event* ⇨ `MIDIEvent`

self	`MIDIDriver` object
midiEvent	`MIDIEvent` object

This method sends a `MIDIEvent` instance to the MIDI driver *self*. The MIDI event can represent a standard MIDI message or a sysex message.

unPrepareDriver

`unPrepareDriver` *self* ⇨ `Boolean`

This method frees a driver's resources (usually the underlying hardware or software MIDI support) for use by another player. Normally this is handled by the MIDI Player but if you are accessing the driver directly be sure to unprepare it when you are done.

After callling the `unPrepareDriver` method on a driver, also call the `closeMIDIDriver` global function on it to close it if you have completely finished with it.

Global Functions

closeMidiDriver

`closeMidiDriver` *midiDriver* `OK`

This function closes a MIDI driver, which is specified as a pair whose first element is a string of the driver's name, and the second element is a name such as `@internal` or `@external`.

getMidiDriverList

`getMidiDriverList()` `Array`

This function returns a list of pairs, where each pair corresponds to an installed MIDI driver. In each pair, the first element is a string of the driver's name, and the second element is a name such as `@internal` or `@external`.

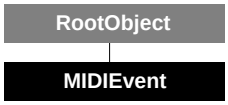
openMidiDriver

`openMidiDriver` *midiPair* `MIDIDriver` or `false`

This function opens a MIDI driver, which is specified as a pair whose first element is a string of the driver's name, and the second element is a name such as `@internal` or `@external`. The function returns the opened `MIDIDriver`, or `false` if the open operation was unsuccesful.

See the introduction to this chapter, or the Media Players chapter in the *ScriptX Components Guide* for more information on finding and opening MIDI drivers.

MIDIEvent



Class type: Core class (concrete)
Resides in: ScriptX and KMP executables
Inherits from: `RootObject`
Component: Media Player

The `MIDIEvent` class represents a MIDI message that can be sent to a MIDI driver. The messages includes 2-byte, 3-byte, system exclusive and meta- events.

To send a `MIDIEvent` to a MIDI driver, call the `sendMIDIEvent` method on a `MIDIDriver` instance, specifying the event to send.

Creating and Initializing a New Instance

When a MIDI file is imported into ScriptX, the importing process automatically creates a `MIDIStream` instance containing `MIDIEvent` instances. See the discussion of `MIDIPlayer` for information on importing a MIDI file.

You can also create instances of `MIDIEvent` by calling the new method on the class. The following script is an example of how to create a new instance of the `MIDIEvent` class:

```
myMIDIEvent := new MIDIEvent
```

In this example, the variable `myMIDIEvent` points to a new `MIDIEvent` instance. After creating the MIDI event, you would fill in the instance variables so that the new instance represents the kind of MIDI event you desire. To play the MIDI event, call the `sendMIDIEvent` method on a `MIDIDriver`, passing the MIDI event as an argument.

init

`init` *self* ⇨ *(none)*

self `MIDIEvent` object

Initializes the `MIDIEvent` object *self*. Do not call `init` directly on an instance—it is automatically called by the `new` method.

Instance Variables

data

self.`data` (read-write) `ByteString`

Specifies the raw bytes that represent the MIDI event *self*.

databyte1

self.`databyte1` (read-write) `Integer`

Specifies the second byte in the MIDI event *self*. This instance variable may be modified even if the event *self* is not a standard MIDI event (that is, even if it is a sysex- or meta-event). Note that this byte might be ignored when the MIDI piece plays, depending on the type of event specified by the status byte.

databyte2

self. `databyte2` (read-write) `Integer`

Specifies the third byte in the MIDI event *self*. This may be modified even if this is not a standard MIDI event (that is, even if it is a sysex- or meta-event). Note that this byte might be ignored when the MIDI piece plays, depending on the type of event specified by the status byte.

length

self. `length` (read-write) `Integer`

Specifies the number of bytes that represent the MIDI event *self*.

statusByte

self. `statusByte` (read-write) `Integer`

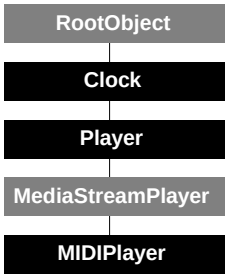
Specifies the first byte in the MIDI event *self*, also known as the status byte. This byte determines what type of MIDI event self is.

timeStamp

self. `timeStamp` (read-write) `Integer`

Specifies the time (in milliseconds) from the start of the piece that the event *self* should be played at. This instance variable is used by a `MIDIPlayer` object to schedule when to play MIDI events.

MIDIPlayer



Class type: Core class (concrete)
Resides in: ScriptX and KMP executables
Inherits from: `MediaStreamPlayer`
Component: Media Players

The `MIDIPlayer` class provides control of playback of MIDI data to a MIDI device.

Note – The current version of ScriptX supports only type 0 MIDI files.

You can only play MIDI on machines equipped with MIDI capability. Some systems provide MIDI capability through external hardware. MacOS systems supply MIDI capabilities through software emulation, although some MacOS systems may include a hardware interface. To insure cross-platform compatibility when you load an instance of `MIDIDriver`, set the value of `channelPolyphony` and call the `prepareDriver` method on the driver before sending it any messages. (Systems that implement MIDI through hardware can play MIDI streams without these steps.)

Creating and Initializing a New Instance

To create a `MIDIPlayer` instance, import a file containing digitized MIDI data. The importing process automatically creates a MIDI stream and a `MIDIPlayer` instance that can be used to play the MIDI sound.

The following script shows how to create an instance of `MIDIPlayer` by importing a MIDI file. The object `violaPlayer` can be used to play the MIDI sequence imported from the file `viola` on the ScriptX startup directory.

```
violaStream:= getstream theStartDir "viola" @readable
violaplayer:= importMedia theImportExportEngine violaStream \
       @MIDI @standard @player
```

This script shows an example of how to import a MIDI file as a `MIDIPlayer`. For more details of the arguments to the method `importMedia` on the global instance `theImportExportEngine`, please see either the “Media Stream Players” chapter in the *ScriptX Components Guide* or the chapter about importers in the *ScriptX Tools Guide.*

The `MIDIPlayer` class also supports the new method. However, users should import MIDI files to create new `MIDIPlayer` instances instead of calling the `new` method on the `MIDIPlayer` class. For the sake of completeness only, an example of using `new` is given here:

```
myPlayer := new MIDIPlayer \
        mediaStream:myMIDIStream \
        masterClock:topPlayer \
        driver:midiDriver1
```

The variable `myPlayer` points to the newly created MIDI player, which has the `MIDIStream` instance `myMIDIStream` as its media stream, the `Player` instance `topPlayer` as its master clock, and the `MIDIDriver` instance `MIDIDriver1` as its driver.

The `new` method uses the keywords defined in `init`.

init

`init` *self* [`mediaStream:`*stream*] [`masterClock:`*clock*]
[`driver:`*device*][`title:`*titleContainer*] ⇨ *(none)*

self	`MIDIPlayer` object
`mediaStream:`	`MIDIStream` object
`masterClock:`	`Clock` object
`driver:`	`MIDIDriver` object
`title:`	`TitleContainer` object to which to add the player.

Initializes the `MIDIStreamPlayer` object *self*, applying the arguments as follows: `mediaStream` sets the source of MIDI data, `masterClock` sets the player's master player, and `driver` sets the driver on which to play. The MIDI device is obtained from the `MIDIManager`. The player is added to the specified `title`. Do not call `init` directly on an instance—it is automatically called by the `new` method.

If you omit an optional keyword, its default value is used. The defaults are:

`mediaStream:`(none)
`masterClock:undefined` (top clock)
`driver:`(the default `MIDIDriver`)
`title:theScratchTitle`

Instance Variables

Inherited from `Clock`:

`callbacks`	`rate`	`ticks`
`effectiveRate`	`resolution`	`time`
`masterClock`	`scale`	`title`
`offset`	`slaveClocks`	

Inherited from `Player`:

`audioMuted`	`globalContrast`	`globalVolumeOffset`
`dataRate`	`globalHue`	`markerList`
`duration`	`globalPanOffset`	`status`
`globalBrightness`	`globalSaturation`	`videoBlanked`

Inherited from `MediaStreamPlayer`:

`frameRate`	`mediaStream`

The following instance variables are defined in `MIDIPlayer`:

driver

self`.driver` (read-write) `Sequence`

Specifies the list of `MIDIDriver` objects to which the MIDI player *self* is sending MIDI data.

mediaStream

self`.mediaStream` (read-write) `MediaStream`

This instance variable is inherited from the `MediaStreamPlayer` class. It specifies the media stream for the player. Setting this instance variable while the player is playing may cause the calling thread to block.

time

self.time (read-write) MediaStream

Inherited from the Clock class. Indicates the current time of the player *self*. Setting this instance variable while the player is playing may cause the calling thread to block.

volume

self.volume (read-write) Number

Specifies the local volume value for the sound controlled by the MIDI player *self*. The value is specified in dB (decibels), where a value of 0 dB represents unity gain. Values less than 0 attenuate the source sound, while values greater then 1 boost the source. dB is a relative measurement; every –6dB drop is equal to halving the volume, modeling human auditory perception.

The actual gain of the sound when it plays is determined by combining the local volume value with the global volume offset, which is passed down from the master player if any.

Instance Methods

Inherited from Clock:

addPeriodicCallback	clockAdded	pause
addRateCallback	clockRemoved	resume
addScaleCallback	effectiveRateChanged	timeJumped
addTimeCallback	forEachSlave	waitTime
addTimeJumpCallback	isAppropriateClock	waitUntil

Inherited from Player:

addMarker	goToBegin	playPrepare
eject	goToEnd	playUnprepare
fastForward	goToMarkerFinish	playUntil
getMarker	goToMarkerStart	resume
getNextMarker	pause	rewind
getPreviousMarker	play	stop

The following instance methods are defined in MIDIPlayer:

muteChannel

muteChannel *self channel flag* ⇨ Integer

self	MIDIPlayer object
channel	Integer object
flag	Boolean object

Mutes the specified *channel* for the MIDI player *self* if the *flag* is set to true. If *flag* is false, restores the channel to its initial volume. Returns the volume of the channel.

setChannelBank

setChannelBank *self channel bank* ⇨ Integer

self	MIDIPlayer object
channel	Integer object
bank	Integer object

Sets the instrument bank of the specified *channel* for the MIDI player *self*. The channel number should be a value between 1 and 16. The bank should be a 16-bit value.

setChannelPan

`setChannelPan` *self channel pan* ⇨ `Integer`

self	`MIDIPlayer` object
channel	`Integer` object
pan	`Integer` object

Sets the pan of the specified *channel* for the MIDI player *self*. The channel number should be a value between 1 and 16. The pan should be a value between 0 and 127.

setChannelProgram

`setChannelProgram` *self channel program* ⇨ `Integer`

self	`MIDIPlayer` object
channel	`Integer` object
program	`Integer` object

Sets the program (instrument) of the specified *channel* for the MIDI player *self*. The channel number should be a value between 1 and 16. The program should be a value between 0 and 127.

setChannelVolume

`setChannelVolume` *self channel volume* ⇨ `Integer`

self	`MIDIPlayer` object
channel	`Integer` object
volume	`Integer` object

Sets the volume of the specified *channel* for the MIDI player *self*. The channel number should be a value between 1 and 16. The volume should be a value between 0 and 127.

soloChannel

`soloChannel` *self channel flag* ⇨ `Integer`

self	`MIDIPlayer` object
channel	`ImmediateInteger` object
flag	`Boolean` object

Mutes all channels for the MIDI player *self* except the one specified if *flag* is true. Returns them to their initial volume if *flag* is false. Returns the volume of the channel.

transposeChannel

`transposeChannel` *self channel transposeVal* ⇨ `Integer`

self	`MIDIPlayer` object
channel	`Integer` object
transposeVal	`Integer` object

Transposes the *channel* of the MIDI player *self* up or down in pitch by adding *transposeVal* to all notes played on the *channel*. Transposition is cumulative, so that a subsequent transposition will be relative to the previous. Returns the initial transposition of the channel.

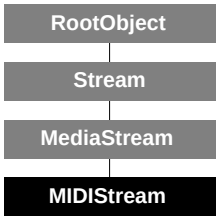
transposePiece

`transposePiece` *self transposeVal* ⇨ `Integer`

self	`MIDIPlayer` object
transposeVal	`Integer` object

Transposes the MIDI piece played by the MIDI player *self* up or down in pitch by adding *transposeVal* to all notes played on all channels. Transposition is cumulative, so that a subsequent transposition will be relative to the previous. Returns the initial transposition of the piece.

MIDIStream



Class type: Core class (concrete)
Resides in: ScriptX and KMP executables
Inherits from: `MediaStream`
Component: Media Players

The `MIDIStream` class provides services for reading streams of MIDI data. A `MIDIStream` contains a stream of MIDI events. The `read` method on `MIDIStream` instances reads `MIDIEvent` instances from the stream.

Creating and Initializing a New Instance

To create a `MIDIStream` instance, import a file containing an audio stream. The importing process automatically creates either a `MIDIStream` instance or a `MIDIPlayer` instance to play the MIDI stream. You can only import MIDI streams into ScriptX on machines equipped with MIDI capability.

The following script shows how to create an instance of `MIDIStream` by importing a MIDI file containing MIDI data. The file "RainSong" would reside on the ScriptX startup directory. To play the imported MIDI, you would need to create a `MIDIPlayer` object and set its `mediaStream` instance variable to the MIDI stream.

```
stream1 := getstream theStartDir "RainSong" @readable   \
rainStream := importMedia theImportExportEngine stream1 \
              @midi @standard @stream
```

This script shows an example of how to import a MIDI file as a `MIDIStream`. For more details of the arguments to the method `importMedia` on the global instance `theImportExportEngine`, please see either the “Media Stream Players” chapter in the *ScriptX Components Guide* or the chapter about importers in the *ScriptX Developer’s Guide.*

The `MIDIStream` class does support the `new` method, which takes a keyword argument of `inputStream`, which needs to be a byte stream. However, you would not normally call the `new` method on `MIDIStream` to create a new instance. Instead you would import a MIDI stream as described above. However, for the sake of completeness, the following script illustrates how to create a new instance of the `MIDIStream` class by calling the `new` method. For this example, you would need to have previously created the byte stream `myStream`, which should contain MIDI data.

```
myStream := new MidiStream \
              inputStream:myStream
```

The variable `myStream` points to the newly created MIDI stream, which has the `ByteStream` instance `myStream` as its input stream.

The `new` method uses the keywords defined in `init`.

init

`init` *self* [`inputStream:`*stream*] ⇨ *(none)*

self	`MIDIStream` object
`inputStream:`	`ByteStream` object

Initializes the `MIDIStream` object *self*, setting the `inputStream` argument as the source of data for the MIDI stream. Do not call `init` directly on an instance—it is automatically called by the `new` method.

If you omit an optional keyword, its default value is used. The defaults are:

`inputStream:undefined`

Instance Variables

Inherited from `MediaStream`:

`dataRate`	`rate`	`variableFrameSize`
`inputStream`	`sampleType`	
`markerList`	`scale`	

Instance Methods

Inherited from `Stream`:

`cursor`	`next`	`seekFromStart`
`flush`	`plug`	`setStreamLength`
`isAtFront`	`previous`	`streamLength`
`isPastEnd`	`read`	`write`
`isReadable`	`readReady`	`writeReady`
`isSeekable`	`seekFromCursor`	
`isWritable`	`seekFromEnd`	

Inherited from `MediaStream`:

`addMarker`	`isSeekable`	`isWritable`
`isReadable`		

The following instance methods are defined in `MIDIStream`:

read

`read` *self* ⇨ `MIDIEvent`

self	`MIDIStream` object

This method is inherited from the class `Stream` but is customized for the class `MIDIStream` to read and return the next MIDI event in the stream. The timestamp of the event that's read is the same as the cursor of the stream.

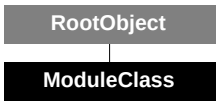
releaseObject

`releaseObject` *self midiEvent* ⇨ *(none)*

self	`MIDIStream` object
midiEvent	`MIDIEvent` object

Releases *MIDIEvent* object from the MIDI stream *self*. The MIDI event is sent back to the stream so it can be used again.

ModuleClass



Class type: Core class (concrete, sealed)
Resides in: ScriptX and KMP executables
Inherits from: `RootObject`
Component: Object System Kernel

ScriptX modules are instances of `ModuleClass`. The *ScriptX Language Guide* documents the use of modules in ScriptX.

An instance of `ModuleClass` contains a list of `NameBinding` objects. You can list the name bindings by coercing the module to any collection class:

```
(getModule @substrate) as Array
```

A ScriptX program is always running in a module. To get an existing `ModuleClass` object, use the `getModule` global function, which is defined on page 35 in Chapter , "Global Functions." To determine which module you are currently in, use the `currentModule` global function, defined in the same chapter.

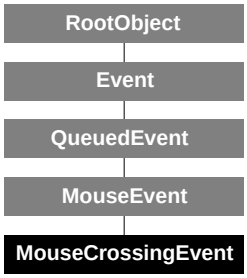
When you open ScriptX and open a Listener window, three instances of `ModuleClass` are created automatically: `@substrate`, `@scriptx`, and `@scratch`. and The `@substrate` module defines all ScriptX names. The `@scriptx` module acts as an interface for names defined in the substrate. Most user-defined modules use the `@scriptx` module. The `@scratch` module, which cannot be saved, is the default module in which a user program runs, unless it defines and runs in another module.

Although several generic functions defined by `ModuleClass` are visible in the scripter, developers should access modules only through the ScriptX language, and through the global functions `getModule` and `currentModule`, to assure compatibility with future versions of ScriptX.

Creating and Initializing a New Instance

You do not create an instance of `ModuleClass` directly. It is created automatically by the system when you use the module definition expression in the ScriptX language, as described in the *ScriptX Language Guide*.

MouseCrossingEvent



Class type: Core class (concrete)
Resides in: ScriptX and KMP executables
Inherits from: `MouseEvent`
Component: Events

The `MouseCrossingEvent` class provides an efficient mechanism to register when the mouse enters and exits from a 2D presenter. Every `TwoDPresenter` object is associated with an instance of `Stencil` that defines its boundary, stored in the instance variable named `boundary`. Each time the mouse travels across the boundary of a presenter that is visible, the system can potentially generate a mouse-crossing event.

It is possible for the user to jerk the mouse quickly across a presenter so quickly that the system does not detect a mouse-crossing event. However, once the system has registered a `MouseCrossingEvent` object with crossing type set to `@enter`, it is guaranteed to generate the matching event with crossing type set to `@leave`. On a particular presenter, mouse-crossing events are uniquely paired and matched. The mouse cannot enter a presenter again until it has left it.

Like the `MouseMoveEvent` class, `MouseCrossingEvent` is dependent on the threshold of the associated mouse device. The device generates a mouse crossing event only when the mouse moves a given number of pixels, set by the value of `threshold`. See the instance variable `threshold`, defined by the `MouseDevice` class.

Any interest not associated with a presenter has priority over interests that are stored by presenters. If an interest is not associated with a presenter, then the `MouseCrossingEvent` class generates events every time the mouse enters or leaves *any* presenter that is visible.

Creating and Initializing a New Instance

The following script creates a new instance of `MouseCrossingEvent`:

```
myMouseTrack:= new MouseCrossingEvent
```

The variable `myMouseTrack` contains an initialized instance of `MouseCrossingEvent`. The `new` method calls the `init` method defined by `MouseCrossingEvent`.

init

`init` *self* ⇨ *(none)*

self `MouseCrossingEvent` object

Initializes the `MouseCrossingEvent` object *self*. Do not call `init` directly on an instance—it is automatically called by the `new` method.

Class Variables

Inherited from Event:

interests	numInterests	

Inherited from QueuedEvent:

dispatchQueue		

Class Methods

Inherited from Event:

acquireQueueFromPool	relinquishQueueToPool
broadcastDispatch	signalDispatch

Instance Variables

Inherited from Event:

advertised	eventReceiver	timeStamp
authorData	matchedInterest	
device	priority	

Inherited from QueuedEvent:

secondaryDispatchStyle	secondaryRejectable

Inherited from MouseEvent:

buttons	localCoords	surfaceCoords
displayCoords	presenter	window
keyModifiers	screenCoords	

MouseCrossingEvent defines the following instance variables:

crossingType

self.crossingType (read-write) ⇨ NameClass

Indicates whether the mouse was entering or leaving the associated presenter. Takes on one of two possible values, @enter or @leave. A single event is associated with entering a presenter, and another with leaving.

If any receiver receives an event with crossingType set to @enter, then it is guaranteed to receive the matching event with crossingType set to @leave as soon as the mouse travels back across the presenter's boundary.

Instance Methods

Inherited from Event:

accept	isSatisfiedBy	sendToQueue
acquireRejectQueue	reject	signal
addEventInterest	relinquishRejectQueue	
broadcast	removeEventInterest	

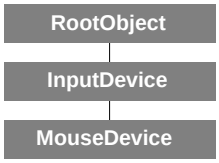
Inherited from QueuedEvent:

broadcast	secondarySignal	signal
secondaryBroadcast		

Inherited from MouseEvent:

broadcast	isButtonUp	isModifierInactive
isButtonDown	isModifierActive	isSatisfiedBy

MouseDevice



Class type: Core class (abstract)
Resides in: ScriptX and KMP executables
Inherits from: `InputDevice`
Component: Input Devices

`MouseDevice` is an abstract class that acts as an interface between the ScriptX Player and the native operating system. It is the underlying hardware that actually generates events. A `MouseDevice` instance—that is, an object that belongs to a subclass of `MouseDevice`—receives mouse events from the device driver and signals them as ScriptX events.

Since `MouseDevice` is abstract, it cannot be instantiated. However, calling `new` on `MouseDevice` will allow a future release of ScriptX to determine which subclass—`PhysicalMouse` or a virtual counterpart that provides the same functionality—is used to create the new instance. Since the current release of ScriptX is directed primarily to desktop computers, a title should assume for now that a physical mouse device is connected to the system.

Each instance of a subclass of `MouseDevice` defines the following:

- The current coordinates of the mouse.
- The mouse buttons pressed.
- The shape of the screen image of the mouse pointer and its associated hot spot.

Creating and Initializing a New Instance

Normally, you cannot invoke `new` on an abstract class. In the case of `MouseDevice`, however, the system allows you to do so. In the current release of ScriptX it creates an instance of `PhysicalMouse`.

```
mighty := new MouseDevice \
       threshold:(new Point x:2 y:2)
```

The variable `mighty` contains a the initialized instance of `PhysicalMouse`. The `threshold` keyword is set so that the device sends a `MouseMoveEvent` only if the mouse travels 2 pixels in either direction. The `new` method uses the keywords defined in `init`.

init

`init` *self* [`threshold:`*point*] [`deviceID:`*integer*] [`enabled:`*boolean*] ⇨ *(none)*

self	`MouseDevice` object
`threshold:`	`Point` object

Superclass `InputDevice` uses the following keywords:

`deviceID:`	`Integer` object
`enabled:`	`Boolean` object

Initializes the `MouseDevice` object *self*, applying the arguments as follows: the `threshold` keyword determines device sensitivity when the mouse is moved. The `deviceID` specifies an integer ID number that is unique to a particular subclass of

InputDevice; it allows a title to request a particular mouse if more than one mouse is attached to the system. The enabled keyword indicates whether the device is initially enabled. Do not call init directly on an instance—it is automatically called by the new method.

If you omit an optional keyword, the following defaults are used:

```
threshold:(new Point x:1 y:1)
deviceID:the first available ID number is supplied, beginning with 1
enabled:true
```

Class Methods

Inherited from InputDevice:

```
getDeviceFromList
```

Instance Variables

Inherited from InputDevice:

```
deviceID                focusable               focusManager
enabled
```

The following instance variables are defined in MouseDevice:

buttons

self.buttons (read-only) Array

Specifies the buttons currently being pressed on the mouse *self*. It returns an array of mouse buttons that are currently being pressed. The buttons instance variable is quite similar to keyModifiers, an instance variable on KeyboardDevice.

```
-- suppose "myMouse" is two-button mouse with both buttons pressed down
myMouse.buttons
⇨  #(@mouseButton1, @mouseButton2).
```

currentCoords

self.currentCoords (read-only) Point

Specifies the current coordinates in pixels, stated as Kaleida Media Player coordinates, of the pointer associated with the mouse *self*. For more information on coordinates, see "Coordinate Systems" in the "Spaces and Presenters" chapter of the *ScriptX Components Guide*.

keyModifiers

self.keyModifiers (read-only) Array

Specifies the current state of the modifier keys, returning a list of active keys as an Array object. The two kinds of modifier keys are called shift and state keys. A shift key is active if it is being pressed; a state key is active if it is currently toggled on. Possible values are @shift, @control, @alt. @command, @capLock, @numLock, and @scrollLock. For additional information about keys, see the KeyboardDevice class.

```
myMouse.keyModifiers -- with shift and control keys down
⇨  #(@shift, @control).
```

pointerType

self.`pointerType` (read-write) `Pointer` or `NameClass`

Specifies either a `Pointer` object for a custom pointer or a `NameClass` value that corresponds to a system-defined pointer. The pointer supplies a bitmap that indicates the position of the mouse device *self* on the screen.

Create an instance of `Pointer` to define a custom pointer, or specify a `NameClass` value to use one of the pointers that is defined by the underlying system. Possible `NameClass` values are `@ibeam`, `@help`, `@wait`, `@none` (hidden), and `@arrow`. The ScriptX Player uses one of the mouse pointers defined by the underlying operating system. For example, when the value of `pointerType` is `@wait`, Microsoft Windows and OS/2 display an hourglass, whereas MacOS displays a watch.

Note – The term "cursor" is used in more than one sense with many operating systems, including both Windows and MacOS. ScriptX uses "pointer" to distinguish the mouse pointer from a text cursor. A mouse pointer is a bitmap image that represents the mouse on a screen. A text cursor is a vertical bar that indicates an insertion point within a block of text.

---

threshold

self.`threshold` (read-write) `Point`

When the mouse *self* travels at least this number of pixels in the x and y direction from the position where the most recent `MouseMoveEvent` object was generated, a new `MouseMoveEvent` object is generated. Both x and y must be greater than 0.

Instance Methods

isButtonDown

`isButtonDown` *self buttonName* ⇨ `Boolean`

self	`MouseDevice` object
buttonName	`NameClass` object

Returns `true` if the button associated with *buttonName* on the mouse *self* is down, where *buttonName* may be `@mouseButton1`, `@mouseButton2`, or `@mouseButton3`.

Note the following distinctions: Calling `isButtonDown` on a mouse device indicates the current state of the device. Calling this method on a mouse event indicates the state of the mouse at the time that the event occurred. If the event represents an interest, it indicates a property of that event interest.

isButtonUp

`isButtonUp` *self buttonName* ⇨ `Boolean`

self	`MouseDevice` object
buttonName	`NameClass` object

Returns `true` if the button associated with *buttonName* on the mouse *self* is up, where *buttonName* may be `@mouseButton1`, `@mouseButton2`, or `@mouseButton3`.

Note the following distinctions: Calling `isButtonUp` on a mouse device indicates the current state of the device. Calling this method on a mouse event indicates the state of the mouse at the time that the event occurred. If the event represents an interest, it indicates a property of that event interest.

isModifierActive

`isModifierActive` *self keyName* ⇨`Boolean`

self	`MouseDevice` object
keyName	`NameClass` object representing a modifier key.

Returns `true` or `false`, depending on whether the modifier key *keyName* is active on the `MouseDevice` object *self*. See the `keyModifiers` instance variable for a list of possible values.

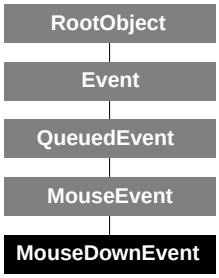
isModifierInactive

`isModifierInactive` *self keyName* ⇨`Boolean`

self	`MouseDevice` object
keyName	`NameClass` object representing a modifier key.

Returns `true` or `false`, depending on whether the modifier key *keyName* is inactive on the `MouseDevice` object *self*. See the `keyModifiers` instance variable for a list of possible values.

MouseDownEvent



Class type: Core class (concrete)
Resides in: ScriptX and KMP executables
Inherits from: `MouseEvent`
Component: Events

The `MouseDownEvent` class represents occurrences of the user pressing down a mouse button. Each instance represents a different occurrence.

Creating and Initializing a New Instance

The following script creates a new instance of the `MouseDownEvent` class:

```
minnie := new MouseDownEvent
```

The variable `minnie` contains an initialized instance of `MouseDownEvent`. The `new` method calls the `init` method defined by `MouseDownEvent`.

init

`init` *self* ⇨ *(none)*

self `MouseDownEvent` object

Initializes the `MouseDownEvent` object *self*. Do not call `init` directly on an instance—it is automatically called by the `new` method.

Class Variables

Inherited from `Event`:

`interests` `numInterests`

Inherited from `QueuedEvent`:

`dispatchQueue`

Class Methods

Inherited from `Event`:

`acquireQueueFromPool` `relinquishQueueToPool`
`broadcastDispatch` `signalDispatch`

Instance Variables

Inherited from `Event`:

`advertised` `eventReceiver` `timeStamp`

authorData	matchedInterest	
device	priority	

Inherited from QueuedEvent:

secondaryDispatchStyle	secondaryRejectable

Inherited from MouseEvent:

buttons	localCoords	surfaceCoords
displaySurface	presenter	window
keyModifiers	screenCoords	

Instance Methods

Inherited from Event:

accept	isSatisfiedBy	sendToQueue
acquireRejectQueue	reject	signal
addEventInterest	relinquishRejectQueue	
broadcast	removeEventInterest	

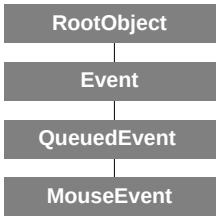
Inherited from QueuedEvent:

broadcast	secondarySignal	signal
secondaryBroadcast		

Inherited from MouseEvent:

broadcast	isButtonUp	isModifierInactive
isButtonDown	isModifierActive	isSatisfiedBy

MouseEvent



Class type: Core class (abstract)
Resides in: ScriptX and KMP executables
Inherits from: `QueuedEvent`
Component: Events

`MouseEvent` is an abstract class that represents any kind of mouse event. Its subclasses represent particular kinds of mouse events: `MouseMoveEvent`, `MouseDownEvent`, and `MouseUpEvent`. An instance of one of these subclasses holds information about the state of the event:

- When the event occurred—the `Event` class itself defines a `timeStamp` instance variable that is inherited by all events.
- Where the event occurred—its presenter, its display surface, and its x, y position, both locally within the presenter and in global coordinates.
- How the event occurred—which buttons were pressed or released by the user, or which buttons were down while the mouse moved.

Note – The following two phrases are used later to distinguish the properties and behavior of an event from those of an event interest:
- "Interest-only" means the instance *self* is used to express an event interest.
- "Event-only" means the instance *self* holds an actual user event.

Class Variables

Inherited from `Event`:

`interests` `numInterests`

Inherited from `QueuedEvent`:

`dispatchQueue`

Class Methods

Inherited from `Event`:

`acquireQueueFromPool` `relinquishQueueToPool`
`broadcastDispatch` `signalDispatch`

Instance Variables

Inherited from `Event`:

`advertised` `eventReceiver` `timeStamp`
`authorData` `matchedInterest`
`device` `priority`

Inherited from `QueuedEvent`:

`secondaryDispatchStyle` `secondaryRejectable`

The following instance variables are defined in `MouseEvent`:

buttons

self`.buttons` (read-write) `Array`

If the mouse event instance *self* is used as an interest, `buttons` indicates the set of mouse buttons that must be pressed for the interest to be satisfied. Leave empty if any combination of buttons would satisfy the interest. If it is used as an event, it indicates the set of buttons currently pressed.

displaySurface

self`.displaySurface` (read-write) `DisplaySurface`

An event-only instance variable. Specifies the display surface where the event was generated. This instance variable is normally set by the `MouseDevice` object that sends the event, but it can be set from the scripter to allow for playback or simulation of mouse events.

keyModifiers

self`.keyModifiers` (read-write) `Array`

If a `MouseEvent` instance is used as an interest, `keyModifiers` indicates the set of key modifiers that must be pressed for the interest to be satisfied. If it is used as an event, it indicates the set of modifiers currently pressed. The two kinds of modifier keys are called shift and state keys. A shift key is active if it is being pressed. A state key is active if it is currently toggled on. Possible values for modifier keys are `@shift`, `@control`, `@alt`, `@command`, `@capLock`, `@numLock`, and `@scrollLock`. For additional information about keys, see the `KeyboardDevice` class.

```
myKeyEvent.keyModifiers -- suppose the shift key was pressed
⇨ #(@shift)
myKeyinterest.keyModifiers := #(@control,@shift) -- set modifiers
⇨ #(@control,@shift)
```

localCoords

self`.localCoords` (read-only) `Point`

An event-only instance variable. Specifies the x and y coordinates (in pixels) of the point where the mouse event *self* was generated. These coordinates are relative to the upper-left corner of the local coordinates of the presenter specified by the `presenter` instance variable, and are defined upon delivery of the event. The default value of `localCoords`, if no event was delivered to an event interest in a concrete subclass of `MouseEvent`, is (0,0). See “Coordinate Systems” in the “Spaces and Presenters” chapter of the *ScriptX Components Guide*. See also `surfaceCoords` and `screenCoords`. To obtain the x or y coordinate, treat the result as a point:

```
self.localCoords.x -- the local x coordinate of mouseEvent self
```

presenter

self`.presenter` (read-write) `TwoDPresenter`

If the mouse event *self* is used as an interest, `presenter` specifies the 2D presenter within which a mouse event must occur for the event interest *self* to be satisfied. When you set this instance variable, the event interest will receive only events that occurred within the bounds of the presenter. This instance variable should be set before a script calls `addEventInterest`.

If the mouse event *self* represents an actual event instance, the `presenter` instance variable is set automatically when the event is matched to an event interest. In this case, `presenter` should be regarded as a read-only instance variable.

screenCoords

self`.screenCoords` (read-only) `Point`

An event-only instance variable. Specifies the x and y coordinates (in pixels) of the point where the mouse event *self* was generated. These coordinates are relative to the upper-left corner of the Kaleida Media Player and are defined upon delivery of the event. For the Macintosh, this origin is the corner of the screen; for Microsoft Windows and OS/2, the origin is the corner of the Kaleida Media Player window (which can be smaller than the screen).

The default value of `screenCoords`, if no event was delivered to an event interest in a concrete subclass of `MouseEvent`, is (0,0). With the `FullScreenWindow` class, `screenCoords` and `surfaceCoords` are the same. See "Coordinate Systems" in the "Spaces and Presenters" chapter of the *ScriptX Components Guide*. See also `localCoords` and `surfaceCoords`. To obtain the x or y coordinate, treat the result as a point:

```
self.screenCoords.x -- the global x coordinate of mouseEvent self
```

surfaceCoords

self`.surfaceCoords` (read-only) `Point`

(Shortcut for "display surface coordinates.") An event-only instance variable. Specifies the x and y coordinates (in pixels) of the point where the mouse event *self* was generated. The coordinates are defined upon delivery of the event. These coordinates are relative to the upper-left corner of the display surface that receives the mouse event, which, for all windows except `FullScreenWindow`, is the upper-left corner of the window. However, for instances of `FullScreenWindow`, the coordinates are relative to the upper-left corner of the display surface. (An instance of `FullScreenWindow` is actually a full-screen display surface, with a smaller-than-full-screen window that is refreshed each tick of its clock.) For `FullScreenWindow` instances, `screenCoords` and `surfaceCoords` are the same.

The default value of `surfaceCoords`, if no event was delivered to an event interest in a concrete subclass of `MouseEvent`, is (0,0). See "Coordinate Systems" in the "Spaces and Presenters" chapter of *ScriptX Components Guide*. See also `localCoords` and `screenCoords`. To obtain the x or y coordinate, treat the result as a point:

```
self.surfaceCoords.x -- the global x coordinate of mouseEvent self
```

window

self`.window` (read-only) `Window`

An event-only instance variable. Returns the window on which the mouse event *self* occurred.

Instance Methods

Inherited from `Event`:

`accept`	`isSatisfiedBy`	`sendToQueue`
`acquireRejectQueue`	`reject`	`signal`
`addEventInterest`	`relinquishRejectQueue`	
`broadcast`	`removeEventInterest`	

Inherited from QueuedEvent:

broadcast secondarySignal signal secondaryBroadcast

The following instance methods are defined in MouseEvent:

broadcast (QueuedEvent)

broadcast *self* ⇨ Exception

Reports the cantBroadcast exception if called on the mouse event *self*. Mouse events cannot be broadcast.

isButtonDown

isButtonDown *self buttonName* ⇨ Boolean

self	MouseEvent object
buttonName	NameClass object

Returns true for the mouse event *self* if the button associated with *buttonName* is down, where *buttonName* may be @mouseButton1, @mouseButton2, or @mouseButton3.

Note the following distinctions: Calling isButtonDown on a mouse event indicates the state of the mouse at the time that the event occurred. If the event instance *self* represents an interest, isButtonDown indicates a property of that event interest. This method is also defined, with the same syntax, on MouseDevice. Calling this method on a mouse device indicates the current state of the device.

isButtonUp

isButtonUp *self buttonName* ⇨ Boolean

self	MouseEvent object
buttonName	NameClass object

Returns true for the mouse event *self* if the button associated with *buttonName* is up, where *buttonName* may be @mouseButton1, @mouseButton2, or @mouseButton3.

Note the following distinctions: Calling isButtonUp on a mouse event indicates the state of the mouse at the time that the event occurred. If the event instance *self* represents an interest, it indicates a property of that event interest. This method is also defined, with the same syntax, on MouseDevice. Calling this method on a mouse device indicates the current state of the device.

isModifierActive

isModifierActive *self keyName* ⇨Boolean

self	MouseEvent object
keyName	NameClass object representing a modifier key.

Tests whether a modifier key is active. Returns true or false, depending on whether the modifier key *keyName* is active on the MouseEvent object *self*. See the keyModifiers instance variable for a list of possible values.

isModifierInactive

isModifierInactive *self keyName* ⇨Boolean

self	MouseEvent object
keyName	NameClass object representing a modifier key.

Tests whether a modifier key is inactive. Returns `true` or `false`, depending on whether the modifier key *keyName* is inactive on the `MouseEvent` object *self*. See the `keyModifiers` instance variable for a list of possible values.

isSatisfiedBy (Event)

`isSatisfiedBy` *self event* ⇨ `Boolean`

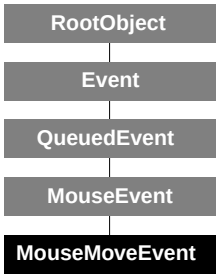
self	`Event` object that represents an event interest
event	`Event` object that represents an actual event

Tests whether the event interest *self* is satisfied by *event*. Do not call `isSatisfiedBy` directly from a script. It is called by the event system as a result of calling either `signal` or `broadcast`. The `MouseEvent` class specializes `isSatisfiedBy` to make several comparisons. It checks whether the device that generated the event is the same device that the event interest has registered an interest in. If the interest specifies a particular button or set of buttons, then `isSatisfiedBy` checks that the event occurred on one of those buttons.

If the matching event is delivered to an interest that is associated with a presenter, as is usual for mouse events, then `isSatisfiedBy` translates the instance variable `localCoords` for the matching event into the local coordinate system of the presenter to which the event is being delivered.

Among the classes that are descendants of `MouseEvent`, the class `MouseUpEvent` specializes `isSatisfiedBy`.

MouseMoveEvent



Class type: Core class (concrete)
Resides in: ScriptX and KMP executables
Inherits from: `MouseEvent`
Component: Events

The `MouseMoveEvent` class represents mouse movements.

Creating and Initializing a New Instance

The following script creates a new instance of the `MouseMoveEvent` class:

```
minnie := new MouseMoveEvent
```

The variable `minnie` contains an initialized instance of `MouseMoveEvent`. The `new` method calls the `init` method defined by `MouseMoveEvent`.

init

`init` *self* ⇨ *(none)*

self `MouseMoveEvent` object

Initializes the `MouseMoveEvent` object *self*. Do not call `init` directly on an instance—it is automatically called by the `new` method.

Class Variables

Inherited from `Event`:

`interests` `numInterests`

Inherited from `QueuedEvent`:

`dispatchQueue`

Class Methods

Inherited from `Event`:

`acquireQueueFromPool` `relinquishQueueToPool`
`broadcastDispatch` `signalDispatch`

Instance Variables

Inherited from `Event`:

`advertised` `eventReceiver` `timeStamp`

authorData	matchedInterest	
device	priority	

Inherited from QueuedEvent:

secondaryDispatchStyle	secondaryRejectable

Inherited from MouseEvent:

buttons	localCoords	surfaceCoords
displaySurface	presenter	window
keyModifiers	screenCoords	

Instance Methods

Inherited from Event:

accept	isSatisfiedBy	sendToQueue
acquireRejectQueue	reject	signal
addEventInterest	relinquishRejectQueue	
broadcast	removeEventInterest	

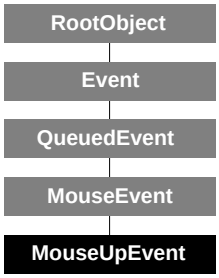
Inherited from QueuedEvent:

broadcast	secondarySignal	signal
secondaryBroadcast		

Inherited from MouseEvent:

broadcast	isButtonUp	isModifierInactive
isButtonDown	isModifierActive	isSatisfiedBy

MouseUpEvent



Class type: Core class (concrete)
Resides in: ScriptX and KMP executables
Inherits from: `MouseEvent`
Component: Events

The `MouseUpEvent` class represents the user releasing a mouse button.

Creating and Initializing a New Instance

The following script creates a new instance of the `MouseUpEvent` class:

```
mickey := new MouseUpEvent
```

The variable `mickey` contains an initialized instance of `MouseUpEvent`. The `new` method calls the `init` method defined by `MouseUpEvent`.

init

`init` *self* ⇨ *(none)*

self `MouseUpEvent` object

Initializes the `MouseUpEvent` object *self*. Do not call `init` directly on an instance—it is automatically called by the `new` method.

Class Variables

Inherited from `Event`:

`interests` `numInterests`

Inherited from `QueuedEvent`:

`dispatchQueue`

Class Methods

Inherited from `Event`:

`acquireQueueFromPool` `relinquishQueueToPool`
`broadcastDispatch` `signalDispatch`

Instance Variables

Inherited from `Event`:

`advertised` `eventReceiver` `timeStamp`

authorData	matchedInterest	
device	priority	

Inherited from QueuedEvent:

secondaryDispatchStyle	secondaryRejectable

Inherited from MouseEvent:

buttons	localCoords	surfaceCoords
displaySurface	presenter	window
keyModifiers	screenCoords	

The following instance variables are specialized by the MouseUpEvent class:

matchedInterest (Event)

self.matchedInterest (read-write) MouseDownEvent

If *self* represents an actual event, the instance variable matchedInterest has the same meaning as with other Event subclasses, defined by Event.

The MouseUpEvent class specializes matchedInterest for event interests. On event interests, it specifies an interest in MouseDownEvent that an interest in MouseUpEvent is paired with. Use matchedInterest to insure that a particular presenter receives a MouseUpEvent object when that event is paired logically with the most recent MouseDownEvent object that was delivered by the MouseDownEvent class. If the user has dragged the mouse outside of the presenter's boundary in the time since the mouse button was pressed, setting matchedInterest overrides the normal delivery mechanism. This can be used to ensure that the presenter that received a MouseDownEvent object is able to "clean up" when the mouse button is released.

For a sample script that demonstrates how matchedInterest is used, see "Using the 2D Graphics Component" in the "2D Graphics" chapter of the *ScriptX Components Guide*.

Instance Methods

Inherited from Event:

accept	isSatisfiedBy	sendToQueue
acquireRejectQueue	reject	signal
addEventInterest	relinquishRejectQueue	
broadcast	removeEventInterest	

Inherited from QueuedEvent:

broadcast	secondarySignal	signal
secondaryBroadcast		

Inherited from MouseEvent:

broadcast	isButtonUp	isModifierInactive
isButtonDown	isModifierActive	isSatisfiedBy

The following instance methods are specialized by the MouseUpEvent class:

isSatisfiedBy (Event)

isSatisfiedBy *self event* ⇨ Boolean

self	Event object that represents an event interest
event	Event object that represents an actual event

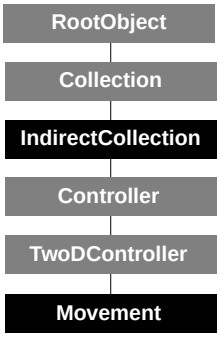
Tests whether the event interest *self* is satisfied by an *event*. Do not call isSatisfiedBy directly from a script. It is called by the event system as a result of calling either signal or broadcast. The MouseUpEvent class specializes isSatisfiedBy to make several

comparisons. It checks whether the device that generated the event is the same device that the event interest has registered an interest in. If the interest specifies a particular button or set of buttons, then `isSatisfiedBy` checks that the event occurred on one of those buttons.

Before a `MouseUpEvent` object is delivered to any interest stored on another presenter, `isSatisfiedBy` checks to see whether the instance variable `matchedInterest` for the interest *self* is pointing to the last interest processed by the `MouseDownEvent` class. See the definition of `matchedInterest`, defined by the `Event` class and specialized by `MouseUpEvent`.

If the matching event is delivered to an interest that is associated with a presenter, as is usual for all mouse events, then `isSatisfiedBy` translates the instance variable `localCoords` for the matching event into the local coordinate system of the presenter to which the event is being delivered.

Movement



Class type: Core class (concrete)
Resides in: ScriptX and KMP executables
Inherits from: `TwoDController`
Component: Controllers

The `Movement` class is a controller that moves one or more `Projectile` objects according to their velocity vectors. For each projectile in the controller's space, the `tickle` method calculates the distance it should have moved since the last tick according to its `velocity` instance variable. It then moves the projectile that distance. See the `Projectile` class for more details.

A `Movement` object is a collection of the `Projectile` objects it controls. These projectiles must also be in the space that the controller is controlling. Projectiles are either automatically or manually added to the movement controller, according to the `wholeSpace` instance variable. If `wholeSpace` is `false`, you can use the methods defined by `Collection` to add and remove objects from the movement controller. To ensure that only projectiles are added to a movement controller, the `protocols` instance variable is set to the `Projectile` class. See the `Controller` class for descriptions of `wholeSpace`, `protocols`, and other general properties of controllers.

`Movement` defines the `tickle` method to check all movement targets at every tick of the presenter's clock.

Creating and Initializing a New Instance

The following script is an example of how to create a new instance of the `Movement` class, after first creating its control space:

```
mySpace := new TwoDSpace boundary:(new Rect x2:200 y2:200)
myMover := new Movement space:mySpace
```

The variable `myMover` contains the initialized instance of `Movement`. This instance controls the movement of projectiles in `mySpace`. The `new` method uses the keywords defined in `init`.

init

`init` *self* [`space:`*space*] [`wholeSpace:`*boolean*] [`enabled:`*boolean*]
[`targetCollection:`*sequence*] ⇨ *(none)*

self `Movement` object

The superclass `Controller` uses the following keywords:

`space:`	`Space` object that holds projectiles
`wholeSpace:`	`Boolean` object
`enabled:`	`Boolean` object

The superclass `TwoDController` uses the following keyword:

`targetCollection:`	`Sequence` object (use with caution)

Initializes the `Movement` object *self*, applying the values supplied with the keywords to instance variables of the same name. Use discretion in changing the target collection; for more information, see the `TwoDController` class. Do not call `init` directly on an instance—it is automatically called by the `new` method.

If you omit an optional keyword, its default value is used. The defaults are:

```
space:undefined
wholeSpace:false
enabled:true
targetCollection:(new Array initialSize:4 growable:true)
```

Class Methods

Inherited from `Collection`:

`pipe`

Instance Variables

Inherited from `Collection`:

`bounded`	`maxSize`	`size`
`iteratorClass`	`minSize`	`uniformity`
`keyEqualComparator`	`mutable`	`uniformityClass`
`keyUniformity`	`mutableCopyClass`	`valueEqualComparator`
`keyUniformityClass`	`proprietored`	

Inherited from `IndirectCollection`:

`targetCollection`

Inherited from `Controller`:

`enabled`	`space`	`wholeSpace`
`protocols`		

The following instance variables are defined in `Movement`:

protocols (Controller)

self`.protocols` (read-write) `Array`

This instance variable initially contains the class `Projectile` for the interpolator *self*. This means that any object added to a `Movement` controller must have `Projectile` as one of its superclasses. See the `Controller` class for further description of this instance variable.

Instance Methods

Inherited from `Collection`:

`add`	`forEach`	`iterate`
`addMany`	`forEachBinding`	`localEqual`
`addToContents`	`getAll`	`map`

chooseAll	getAny	merge
chooseOne	getKeyAll	pipe
chooseOneBinding	getKeyOne	prin
deleteAll	getMany	removeAll
deleteBindingAll	getOne	removeOne
deleteBindingOne	hasBinding	setAll
deleteKeyAll	hasKey	setOne
deleteKeyOne	intersects	size
deleteOne	isEmpty	
emptyOut	isMember	

Inherited from `IndirectCollection`:

isAppropriateObject	objectAdded	objectRemoved

Inherited from `Controller`:

isAppropriateObject	tickle

Since a `Movement` controller is an indirect collection, you can also use any methods defined in the class specified by `targetCollection`. The target collection is typically an instance of `Array`, which inherits from `Sequence`, so the following instance methods are redirected to this controller.

Accessible from `LinearCollection`:

chooseOneBackwards	findRange	getNthKey
chooseOrdOne	forEachBackwards	getOrdOne
deleteFirst	getFirst	getRange
deleteLast	getLast	localEqual
deleteNth	getMiddle	localLT
deleteRange	getNth	pop

Accessible from `Sequence`:

addFifth	moveBackward	setFourth
addFirst	moveForward	setLast
addFourth	moveToBack	setNth
addNth	moveToFront	setSecond
addSecond	prepend	setThird
addThird	prependNew	sort
append	setFifth	
appendNew	setFirst	

The following instance methods are defined in `Movement`:

tickle (Controller)

`tickle` *self clock* ⇨ *self*

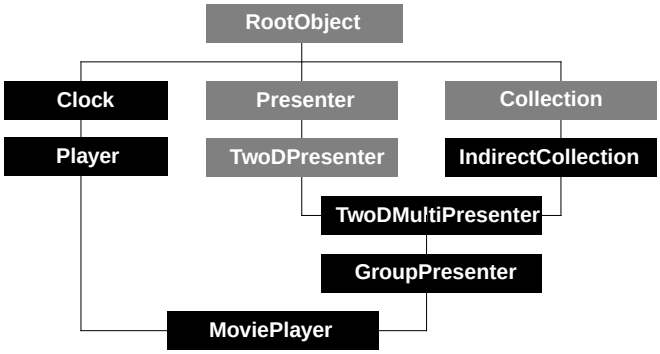
self	`Movement` object
clock	`Clock` object of the space being controlled

For each projectile in the controller's space, the `tickle` method calculates the distance the projectile should have moved according to its `velocity` instance variable, then moves the projectile that distance.

A callback calls this method on the `Movement` object *self*, supplying the space's clock as the value for *clock*. The callback calls this method once every tick of the space's clock.

For further details, refer to the section "The Ticklish Protocol" in the chapter "Controllers" in the *ScriptX Components Guide*.

MoviePlayer



Class type: Core class (concrete)
Resides in: ScriptX and KMP executables
Inherits from: `Player` and `GroupPresenter`
Component: Media Players

The `MoviePlayer` class provides methods for playing deinterleaved movies in ScriptX. When ScriptX imports a movie as a deinterleaved movie, it creates separate streams to contain the audio and video tracks. The importing process also creates appropriate players to play the streams, such as a `DigitalAudioPlayer` to play audio and `DigitalVideoPlayer` to play video.

The `MoviePlayer` instance acts as a master player, whose slave players are the players that play the individual tracks. When you call the `play` method on a `MoviePlayer` instance, it calls `play` on all its slave players to play the tracks of the movie. The `target` instance variable of a movie player holds an array of all the slave players that are needed to play the movie.

When a frame of the movie plays, data is needed from both the video stream and the audio stream. If the streams are not interleaved, the video data needed for a frame may be arbitrarily distant on the storage medium from the audio data needed for the same frame. When playing back movies with non-interleaved data from a hard disk, the extra search time required to seek to non-sequential positions is relatively small and does not significantly affect the speed of playback. However, the search time becomes significant if the movie is played back from a CD.

When the audio and video streams are interleaved, the video data and audio data required for a frame are located sequentially on the storage medium, thus minimizing the search time between each frame.

When importing a movie into ScriptX, you can choose to import it as an interleaved or non-interleaved movie. If you intend to play it back from hard disk, then you can import it as a non-interleaved movie (that is, import it as a `MoviePlayer`.) If you intend to play the imported movie on a CD, you should import it into an `InterleavedStreamPlayer` to preserve the interleaving.

The frame rate of a movie player is equal to smallest frame rate of all of its slaves players. Its boundary is the intersection of the boundaries all its slave players. Its scale is the smallest scale of all its slaves players.

Creating and Initializing a New Instance

To create a `MoviePlayer` instance, import a file containing a movie. The importing process automatically creates the `MoviePlayer` instance and all the other players and streams needed to play the movie.

The following script shows how to create an instance of `MoviePlayer` by importing a QuickTime file containing a digitized movie. The object `danceMovie` can be used to play the movie imported from the file `dance` on the ScriptX startup directory.

```
danceStream:= getstream theStartDir "dance" @readable
danceMovie:= importMedia theImportExportEngine danceStream\
          @movie @quicktime @player
```

This script shows an example of how to import an QuickTime file as a `MoviePlayer`. You can also import AVI files in the same manner, in which case you would need to change the `@quicktime` argument to `@avi`. For more details of the arguments to the method `importMedia` on the global instance `theImportExportEngine`, please see either the "Media Stream Players" chapter in the *ScriptX Components Guide* or the chapter about importers in the *ScriptX Tools Guide.*

After creating a `MoviePlayer` instance by importing a movie, append it to a visible surface such as a `Window` to use as its "screen". Call its `play` method to start it playing.

Class Methods

Inherited from `Collection`:

`pipe`

Instance Variables

Inherited from `Collection`:

`bounded`	`maxSize`	`size`
`iteratorClass`	`minSize`	`uniformity`
`keyEqualComparator`	`mutable`	`uniformityClass`
`keyUniformity`	`mutableCopyClass`	`valueEqualComparator`
`keyUniformityClass`	`proprietored`	

Inherited from `IndirectCollection`:

`targetCollection`

Inherited from `Presenter`:

`presentedBy`	`subPresenters`	`target`

Inherited from `TwoDPresenter`:

`bBox`	`height`	`transform`
`boundary`	`IsImplicitlyDirect`	`width`
`clock`	`isTransparent`	`window`
`compositor`	`isVisible`	`x`
`direct`	`needsTickle`	`y`
`eventInterests`	`position`	`z`
`globalBoundary`	`stationary`	
`globalTransform`	`target`	

Inherited from `TwoDMultiPresenter`:

`fill`	`stroke`

Inherited from `Clock`:

`callbacks`	`rate`	`ticks`
`effectiveRate`	`resolution`	`time`
`masterClock`	`scale`	`title`
`offset`	`slaveClocks`	

Inherited from `Player`:

`audioMuted`	`globalContrast`	`globalVolumeOffset`

dataRate	globalHue	markerList
duration	globalPanOffset	status
globalBrightness	globalSaturation	videoBlanked

The following instance variables are defined in MoviePlayer:

frameRate

self.frameRate (read-write) Number

The value of the frameRate instance variable on a movie player *self* is equal to smallest frameRate on all of its slaves players.

For each media stream player, frameRate specifies the number of frames per "second" that the player *self* plays the frames in its media stream. The frameRate reflects the true frame rate of the media while scale represents the tick rate used by the player to present that media.

Instance Methods

Inherited from Collection:

add	forEach	iterate
addMany	forEachBinding	localEqual
addToContents	getAll	map
chooseAll	getAny	merge
chooseOne	getKeyAll	pipe
chooseOneBinding	getKeyOne	prin
deleteAll	getMany	removeAll
deleteBindingAll	getOne	removeOne
deleteBindingOne	hasBinding	setAll
deleteKeyAll	hasKey	setOne
deleteKeyOne	intersects	size
deleteOne	isEmpty	
emptyOut	isMember	

Inherited from TwoDPresenter:

adjustClockMaster	inside	show
createInterestList	localToSurface	surfaceToLocal
draw	notifyChanged	tickle
getBoundaryInParent	recalcRegion	
hide	refresh	

Inherited from TwoDMultiPresenter:

draw	findFirstInStencil	moveToBack
findAllAtPoint	isAppropriateObject	moveToFront
findAllInStencil	moveBackward	objectAdded
findFirstAtPoint	moveForward	objectRemoved

Inherited from Clock:

addPeriodicCallback	clockAdded	pause
addRateCallback	clockRemoved	resume
addScaleCallback	effectiveRateChanged	timeJumped
addTimeCallback	forEachSlave	waitTime
addTimeJumpCallback	isAppropriateClock	waitUntil

Inherited from Player:

addMarker	goToBegin	playPrepare
eject	goToEnd	playUnprepare
fastForward	goToMarkerFinish	playUntil
getMarker	goToMarkerStart	resume

getNextMarker	pause	rewind
getPreviousMarker	play	stop

Since a MoviePlayer object is an indirect collection, you can also use any methods defined in the class specified by targetCollection. The target collection is typically an instance of Array, which inherits from Sequence, so the following instance methods are redirected to this player.

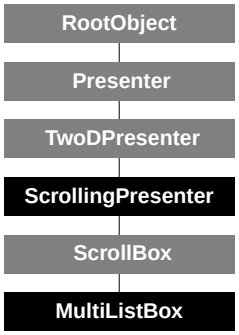
Accessible from LinearCollection:

chooseOneBackwards	findRange	getNthKey
chooseOrdOne	forEachBackwards	getOrdOne
deleteFirst	getFirst	getRange
deleteLast	getLast	localEqual
deleteNth	getMiddle	localLT
deleteRange	getNth	pop

Accessible from Sequence:

addFifth	moveBackward	setFourth
addFirst	moveForward	setLast
addFourth	moveToBack	setNth
addNth	moveToFront	setSecond
addSecond	prepend	setThird
addThird	prependNew	sort
append	setFifth	
appendNew	setFirst	

MultiListBox



Class type: Scripted class (abstract)
Resides in: `widgets.sxl`. Works with ScriptX and KMP executables
Inherits from: `ScrollBox`
Component: User Interface

The `MultiListBox` class defines a scrollable list of strings, one of which can be selected at a time, as shown in Figure 4-11. The list can be longer than the visible region, so that you need to scroll the list to bring the hidden part into view. It comes with an optional vertical scroll bar that automatically adjusts itself properly for a given amount of text.

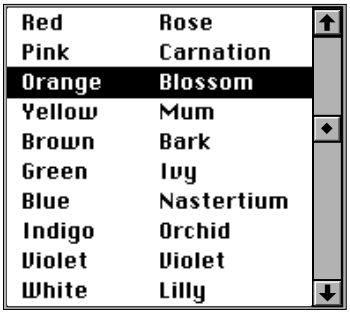


Figure 4-11: A MultiListBox object with numColumn set to 2.

Creating and Initializing a New Instance

The following script is an example of how to create a new instance of the `ScrollingPresenter` class:

```
myMultiListBox := new MultiListBox \
       list:#(#("Red", "Rose"), #("Pink", "Carnation"), \
           #("Orange", "Blossom"), #("Yellow", "Mum"), \
           #("Brown", "Bark"), #("Green", "Ivy")) \
       numColumns:2 \
       hasScrollBar:true \
       value:3 \
       boundary:(new Rect x2:250 y2:200)
```

The variable `myMultiListBox` contains the initialized multi-scrolling list box shown in Figure 4-11. The list box contains pairs of strings #("Red", "Rose"), #("Pink", "Carnation"), etc., and has a vertical scrollbar, which is automatically sized to fit inside the scrolling box's boundary of 250 x 200 pixels. The `value` keyword initially selects the 3rd item in the list. The `new` method uses the keywords defined in `init`.

init

`init` *self* [`list:`*sequence*] [`hasScrollBar:`*boolean*] [`value:`*integer*]
[`boundary:`*stencil*] [`font:`*fontContext*]
[`fill:`*brush*] [`stroke:`*brush*] ⇨ *(none)*

self	`ScrollListBox` object
`numColumns:`	`Integer` object for the number of columns
`list:`	`Sequence` object containing items in the list
`hasScrollBar:`	`Boolean` object indicating if scroll bar is present
`value:`	`Integer` object indicating initial item selected
`font:`	`FontContext` object of items in the list

Superclasses of `MultiListBox` use the following keywords:

`boundary:`	`Rect` object for the overall size of the box
`fill:`	`Brush` object (ignored by `ScrollListBox`)
`stroke:`	`Brush` object (ignored by `ScrollListBox`)
`targetPresenter:`	`TwoDPresenter` object (ignored by `ScrollListBox`)
`vertScrollBar:`	`ScrollBar` object (ignored by `ScrollListBox`)
`horizScrollBar:`	`ScrollBar` object (ignored by `ScrollListBox`)
`target:`	Any object (ignored by `ScrollListBox`)

Initializes the `ScrollListBox` object *self*, applying the values supplied with the keywords to the instance variables of the same name. Do not call `init` directly on an instance—it is called automatically by the `new` method.

If you omit an optional keyword, its default value is used. The defaults are:

```
numColumns:2
font:theSystemFont
hasScrollBar:false
list:#()
value:undefined
fill:undefined
stroke:undefined
targetPresenter:undefined
```

Instance Variables

Inherited from `Presenter`:

`presentedBy`	`subPresenters`	`target`

Inherited from `TwoDPresenter`:

`bBox`	`globalTransform`	`target`
`boundary`	`height`	`transform`
`clock`	`IsImplicitlyDirect`	`width`
`compositor`	`isTransparent`	`window`
`direct`	`isVisible`	`x`
`eventInterests`	`needsTickle`	`y`
`globalBoundary`	`position`	`z`
`globalRegion`	`stationary`	

Inherited from `ScrollBox`:

`authorData`	`lastLine`	`stepAmount`
`doubleClickAction`	`list`	`targetPresenter`
`doubleClickTime`	`numLines`	`value`
`fill`	`pageAmount`	`width`
`font`	`selectAction`	
`frame`	`selectedLine`	

Instance Methods

Inherited from `TwoDPresenter`:

`adjustClockMaster`	`hide`	`refresh`
`createInterestList`	`inside`	`show`
`draw`	`localToSurface`	`surfaceToLocal`
`getBoundaryInParent`	`notifyChanged`	`tickle`

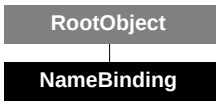
Inherited from `ScrollingPresenter`:

`handleHorizScroll`	`layout`	`scrollTo`
`handleVertScroll`		

Inherited from: `ScrollBox`:

`downReceiver`
`repeatScrollAction`
`selectLine`
`upReceiver`

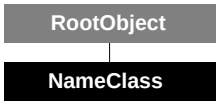
NameBinding



Class type: Core class (concrete)
Resides in: ScriptX and KMP executables
Inherits from: `RootObject`
Component: Object System Kernel

The `NameBinding` class represents the binding of an object to a variable name within a particular module. A `ModuleClass` object is a collection of name bindings. (`ModuleClass` does not inherit from `Collection`, and it does not implement most of the ScriptX Collection protocol, so the use of "collection" here is informal. See `ModuleClass` for more information.)

NameClass



Class type: Core class (concrete, sealed)
Resides in: ScriptX and KMP executables
Inherits from: `RootObject`
Component: Object System Kernel

The `NameClass` class represents all names used in ScriptX. Names are like strings in that they can contain any character, but they are different in that they are generally only stored once, in the system's name table. You can identify `NameClass` objects in that they start with an at (`@`) sign. Use of names in ScriptX is analogous to enumerated types in other languages. For instance, the following lines of code create two separate strings that are equal (`=`), but not identical (`==`).

```
a := "test"
b := "test"
a == b        -- returns false
```

In contrast, the variables below both refer to the identical name (notice the "`@`" signs). For these variables, both the equality and the identity tests return `true`.

```
a := @test
b := @test
a == b        -- returns true
```

Names are used to represent values. For example, a `ScrollBar` object can have one of two orientations, horizontal or vertical. Rather than representing these with numbers, they are represented as names: `@horizontal` and `@vertical`. Prefixing a name token with the "at" sign is equivalent to calling `internDownCase` on the string representation of the token.

Creating and Initializing a New Instance

You can identify `NameClass` objects in that they start with an "`@`" sign. There are two ways to create a `NameClass` instance:

- Type the "at" sign (`@`) followed by the name:

  ```
  Format:       @name
  Example:      @test
  ```

- To convert a string to a name, call either the `intern` or `internDowncase` class method on `NameClass`, passing in the string:

  ```
  Format:       intern NameClass "name"
  Example:      intern NameClass "test"
  ```

Either technique creates and interns the `NameClass` object `@test`. In the first case, the name object is interned at compile time—that is, when your ScriptX code is compiled into bytecode. In the second case, interning takes place at runtime—`intern` enters the name into the system hash table and returns a new name object only if none already exists.

`NameClass` is a special system class used for uniquely identifying symbols. You may not override its methods, nor can you create subclasses of `NameClass`. If you call `new` on `NameClass`, the `init` method is invoked on the newly allocated object.

init

`init` *self* `name:`*string* ⇨ *(none)*

self	`NameClass` class
`name`	`String` object

Initializes the `NameClass` object *self*, where `name` specifies the string to be stored in the `name` object. The value supplied with `name` is *not* interned, so that no other name (even if made up of the identical characters), will ever be identical with it. You should only call `new` on `NameClass` in the special circumstances where uniqueness is required. Do not call `init` directly on an instance—it is automatically called by the `new` method.

Class Methods

intern

`intern` *self nameString* ⇨ `NameClass`

self	`NameClass` class
nameString	`String` object

Gets the name table from `NameClass`, and searches through the table for a name whose string representation equals *nameString*. If a match is found, the matching `NameClass` object is returned. Otherwise, a new entry in the name table is created with the text of *nameString* and a new `NameClass` object is returned. Interning takes place at runtime.

internDowncase

`internDowncase` *self nameString* ⇨ `NameClass`

self	`NameClass` class
nameString	`String` object

Behaves like `intern`, except that the *nameString* is converted to lowercase letters before it is interned. Case is retained for printing. Interning takes place at runtime.

isInterned

`isInterned` *self nameString* ⇨ `NameClass`

self	`NameClass` class
nameString	`String` object

Returns the `NameClass` object if there is an entry in the name table whose string representation is equal to *nameString*; otherwise it returns `empty`.

isInternedDowncase

`isInternedDowncase` *self nameString* ⇨ `NameClass`

self	`NameClass` class
nameString	`String` object

Behaves like `isInterned`, except that the *nameString* is converted to lowercase letters before a match is attempted.

Instance Methods

copy (RootObject)

`copy` *self* ⇨ `NameClass`

Returns a copy of a the `NameClass` object self, if *self* is not interned. Otherwise, `copy` returns the same object.

hashOf

`hashOf` *self* ⇨ `ImmediateInteger`

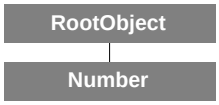
Returns a positive `ImmediateInteger` object representing a 29-bit hash value for the string *self*. The `hashOf` generic function is most commonly used to hash keys for items to be stored in a hash table.

stringOf

`stringOf` *self* ⇨ `String`

Returns a `String` object, the equivalent of *self* as a string.

Number



Class type: Core class (abstract, sealed)
Resides in: ScriptX and KMP executables
Inherits from: `RootObject`
Component: Numerics

The `Number` class represents all of the various types of numbers that can be used in ScriptX.

The `Number` class has the following four operators:

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

These symbols substitute for substrate methods that are hidden from the scripter level.

ScriptX provides several global numeric constants, which are defined in the "Global Constants and Variables" chapter of this volume. The constants `e`, `pi`, `piDiv2`, and `sqrt2` provide standard values for commonly used mathematical constants. The global constants `nan`, `negInf`, and `posInf` represent infinite and uncountable magnitudes in ScriptX expressions. They are useful in ranges and comparisons.

Instance Methods

In the following methods, *self* and *num2* are instances of a subclass of `Number`. All trigonometric methods require and return values in radians, not degrees.

abs

`abs` *self* ⇨ `Number`

Returns the absolute value of the number *self*.

acos

`acos` *self* ⇨ `Float`

Returns the arc cosine of the number *self*, expressed in radians. The argument *self* must also be expressed in radians. See `degToRad`.

asin

`asin` *self* ⇨ `Float`

Returns the arc sine of the number *self*, expressed in radians. The argument *self* must also be expressed in radians. The range for *self* is `-1` to `+1` inclusive. See `degToRad`.

atan

`atan` *self* ⇨ `Float`

Returns the arc tangent of the number *self*, expressed in radians. The argument *self* must also be expressed in radians. See `degToRad`.

atan2

`atan2` *self num2* ⇨ `Float`

Returns the inverse of the four-quadrant tangent of the numbers *self* and *num2*. The result is in radians.

ceiling

`ceiling` *self* ⇨ `Integer`

Rounds the number *self* up to the next higher integer (toward positive infinity; for example, `4.5` becomes `5` and `-4.5` becomes `-4`.Compare with `floor`.

cos

`cos` *self* ⇨ `Float`

Returns the cosine of the number *self*, expressed in radians. The argument *self* must also be expressed in radians. See `degToRad`.

cosh

`cosh` *self* ⇨ `Float`

Returns the hyperbolic cosine of the number *self*. The argument *self* must be expressed in radians.

degToRad

`degToRad` *self* ⇨ `Float`

Converts the degree measure *self* to radians.

exp

`exp` *self* ⇨ `Float`

Raises e (2.718 . . .) to the power *self*, which is e^{self}.

floor

`floor` *self* ⇨ `Integer`

Converts the number *self* to the next lower integer (toward negative infinity). For example, `4.5` becomes `4`. and `-4.5` becomes `-5`. Compare with `ceiling`.

frac

`frac` *self* ⇨ `Number`

Returns the fractional portion of the number *self*. Also see `trunc`.

ln

`ln` *self* ⇨ `Float`

Returns the natural logarithm of the number *self*.

inverse

`inverse` *self* ⇨ `Float`

Returns the inverse of the number *self*, that is, `1/`*self*.

log

`log` *self int* ⇨ `Float`

Returns the logarithm of the number *self* in base *int*, which is log $_{int}$ *self*.

max

`max` *self num2* ⇨ `Number`

Returns the larger of the numbers *self* or *num2*.

min

`min` *self num2* ⇨ `Number`

Returns the smaller of the numbers *self* or *num2*.

mod

`mod` *self num2* ⇨ `Number`

Returns the result of the number *self* modulo *num2*, which has the same sign as *num2*. The value for *num2* can never be zero. The result of `mod` is the same as:

self `-` `(floor (`*self* `/`*num2*`)*` *num2*`)`

For example, `mod -1 4` returns `3`, and `mod -4.5 -2` returns `-0.5`.

morph

`morph` *self resultClass arg* ⇨ `Number`

self	`Number` object
resultClass	Any class name
arg	`normal` (reserved for future use)

Returns the result of the number *self* coerced to the class specified in *resultClass*. Use `normal` as the value of *arg*. For numbers, `morph` is the same as `coerce`.

negate

`negate` *self* ⇨ `Number`

Returns the negation of the number *self*.

power

`power` *self num2* ⇨ `Float`

Raises the number *self* to the *num2* power, which is $self^{num2}$.

radToDeg

`radToDeg` *self* ⇨ `Float`

Converts the radian measure *self* to degrees.

rand

`rand` *self* ⇨ `Number`

Returns a pseudo-random number with a value greater than or equal to 0 and less than the number *self*. The random number has the same class as the number *self*.

The `rand` method uses a hidden random state generator to generate the numbers. The `rand` method is defined to behave like the following function, where `hiddenRS` is a hidden `RandomState` generator whose seed is unspecified:

```
function (rand n -> random hiddenRS n)
```

Use the `rand` method for casual needs of random numbers. The `rand` method cannot guarantee to return either the same sequence or a different sequence of numbers every time after starting ScriptX. For example, if two threads are using `rand`, then it depends on the timing of the thread switching which of the two threads gets the next random number, since the same generator is used for all threads.

If you need repeatability of random numbers you should use the `RandomState` class, which allows you to specify a starting seed.

Furthermore, if you need to guarantee non-repeatability of random numbers every time after starting ScriptX, you should create your own `RandomState` object and seed it with some changing value, such as date and time.

rem

`rem` *self num2* ⇨ `Number`

Returns the remainder of the number *self* divided by the number *num2*. It has the same sign as *self* when *num2* is positive, and the opposite sign when *num2* is negative. The value for *num2* can never be zero.

For example, `rem –1 4` results in `-1`, and `rem –4.5 –2` results in `–0.5`.

round

`round` *self* ⇨ `Number`

Rounds the number *self* to the appropriate integer. Postive numbers are rounded upward if the fractional part is 0.5 or higher, and downward if it is less than 0.5. Negative numbers are rounded downward if the fractional part is –0.5 or less, and upward if it is greater than –0.5. For example, 4.5 becomes 5 and –4.5 becomes –5. Notice that `round` is equivalent to the following:

```
fn round1 x -> (
    if x > 0 then
        if (abs (frac x)) >= 0.5 then return ceiling x
        else return floor x
    else
        if (abs (frac x)) >= 0.5 then return floor x
        else return ceiling x
)
-- a more efficient but less readable version of round1
fn round2 x ->
    if xor (abs(frac x) >= 0.5) (x > 0) then floor x else ceiling x
```

sin

`sin` *self* ⇨ `Float`

Returns the sine of the number *self*, expressed in radians. The argument *self* must also be expressed in radians. See `degToRad`.

sinh

`sinh` *self* ⇨ `Float`

Returns the hyperbolic sine of the number *self*. The argument *self* must be expressed in radians.

sqrt

`sqrt` *self* ⇨ `Float`

Returns the square root of the number *self*.

tan

`tan` *self* ⇨ `Float`

Returns the tangent of the number *self*, expressed in radians. The argument *self* must also be expressed in radians. See `degToRad`.

tanh

`tanh` *self* ⇨ `Float`

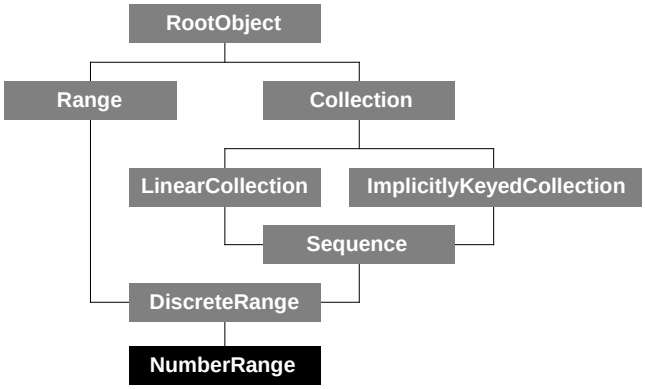
Returns the hyperbolic tangent of the number *self*. The argument *self* must be expressed in radians.

trunc

`trunc` *self* ⇨ `Number`

Returns the integer portion of the number *self*; the decimal portion is truncated (hence its name). See also `frac`.

NumberRange



Class type: Core class (concrete)
Resides in: ScriptX and KMP executables
Inherits from: `DiscreteRange`
Component: Collections

A `NumberRange` object is a discrete range of numbers.

Creating and Initializing a New Instance

The following script is an example of how to create a new instance of the `NumberRange` class.

```
myRange := new NumberRange \
       lowerBound:(-2 * pi) \
       upperBound:(2 * pi) \
       increment:(pi / 2)
```

The variable `myRange` contains the initialized number range. The instance contains the 9 numbers from approximately -6.28 to +6.28 at intervals of about 1.57:

-6.283, -4.712, -3.141, -1.570, 0, 1.570, 3.141, 4.712, 6.283

The `new` method uses the keywords defined by the `init` method.

init

`init` *self* `lowerBound:`*number* `upperBound:`*number* [`increment:`*number*]

⇨ *(none)*

self	`NumberRange` object
`lowerBound:`	`Number` object
`upperBound:`	`Number` object
`increment:`	`Number` object

Initializes the `NumberRange` object *self*, applying the values supplied with the keywords to the instance variables of the same name. Do not call `init` directly on an instance—it is automatically called by the `new` method.

If you omit an optional keyword, its default value is used. The only default is:

`increment:1`

Note – For negative increments, the `lowerBound` and `upperBound` are reversed—set the starting value to `lowerbound` and the ending value to `upperBound`. For example:

```
init self lowerbound:10 upperBound:0 increment:-1
```

For ranges with negative increments, you must supply an increment; the increment is optional only for ranges with positive increments.

Class Methods

Inherited from `Collection`:

`pipe`

Instance Variables

Inherited from `Collection`:

`bounded`	`maxSize`	`size`
`iteratorClass`	`minSize`	`uniformity`
`keyEqualComparator`	`mutable`	`uniformityClass`
`keyUniformity`	`mutableCopyClass`	`valueEqualComparator`
`keyUniformityClass`	`proprietored`	

Inherited from `Range`:

`includesLower`	`lowerBound`	`valueClass`
`includesUpper`	`size`	
`increment`	`upperBound`	

The following instance variables are redefined in `NumberRange`:

includesLower (Range)

self`.includesLower` (read-only) `Boolean`

Returns `true`, and cannot be changed.

includesUpper (Range)

self`.includesUpper` (read-only) `Boolean`

Returns `true`, and cannot be changed.

valueClass (Range)

self`.valueClass` (read-only) *(class)*

Returns `Number`, and cannot be changed.

Instance Methods

Inherited from `Collection`:

`add`	`forEach`	`iterate`
`addMany`	`forEachBinding`	`localEqual`
`addToContents`	`getAll`	`map`
`chooseAll`	`getAny`	`merge`
`chooseOne`	`getKeyAll`	`pipe`
`chooseOneBinding`	`getKeyOne`	`prin`
`deleteAll`	`getMany`	`removeAll`
`deleteBindingAll`	`getOne`	`removeOne`
`deleteBindingOne`	`hasBinding`	`setAll`

deleteKeyAll	hasKey	setOne
deleteKeyOne	intersects	size
deleteOne	isEmpty	
emptyOut	isMember	

Inherited from LinearCollection:

chooseOneBackwards	findRange	getNthKey
chooseOrdOne	forEachBackwards	getOrdOne
deleteFirst	getFirst	getRange
deleteLast	getLast	localEqual
deleteNth	getMiddle	localLT
deleteRange	getNth	pop

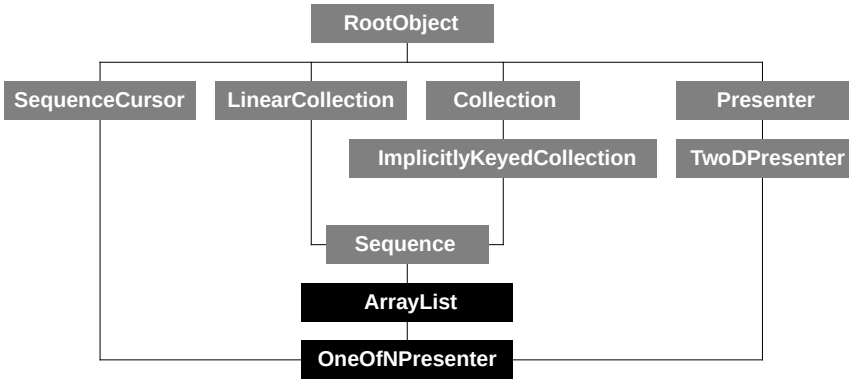
Inherited from Sequence:

addFifth	moveBackward	setFourth
addFirst	moveForward	setLast
addFourth	moveToBack	setNth
addNth	moveToFront	setSecond
addSecond	prepend	setThird
addThird	prependNew	sort
append	setFifth	
appendNew	setFirst	

Inherited from Range:

withinRange

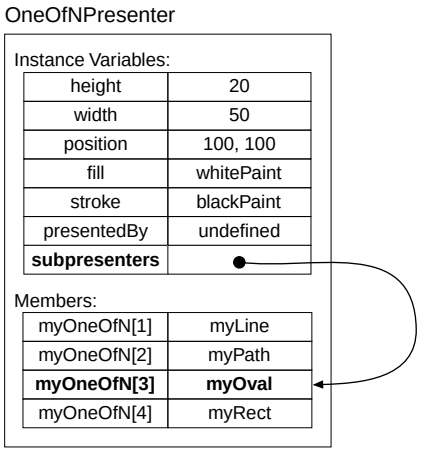
OneOfNPresenter



Class type: Core class (concrete)
Resides in: ScriptX and KMP executables
Inherits from: `SequenceCursor`, `ArrayList` and `TwoDPresenter`
Component: Spaces and Presenters

A `OneOfNPresenter` object is an ordered list of presenters, only one of which is shown at a time. A `OneOfNPresenter` object changes its screen size to contain what it's displaying.

For example, the following figure represents an instance of `OneOfNPresenter` (not all instance variables are shown) with four instances of `TwoDShape` as members—a line, path, oval, and rectangle. Notice that the `subpresenters` instance variable points to the one shape that the `OneOfNPresenter` object currently displays—in this case, `myOval` has the privilege of being displayed.



The members of an instance of `OneOfNPresenter` are held in an `ArrayList` object. When an array is stored in a storage container, normally, it is stored as one clump, where all members must be loaded into memory together. This could be a problem if the members take up a lot of memory, such as full-screen bitmaps. Since only one presenter is ever meant to be displayed at a time, it does not make sense to require they all be loaded into memory at once.

To solve this problem, when you add a `OneOfNPresenter` instance to a library, title, or accessory container, the instance automatically adds each of its members (subobjects) separately to the storage container, using `addNewToStorageContainer`. (If the subobject has already been added to a storage container, the subobject is not moved.) This enables the subobjects to be loaded into memory separately. This is a good memory-saving technique that you can use in cases other than `OneOfNPresenter`.

Using a OneOfNPresenter object that holds a set of TwoDMultiPresenter objects, you can create a multilayered sequence of spaces within spaces—similar to cards and backgrounds in a HyperCard or ToolBook stack.

OneOfNPresenter inherits methods from the SequenceCursor class for moving about in the list of subpresenters—methods such as goTo, forward and backward.

Creating and Initializing a New Instance

The following script is an example of how to create a new instance of the OneOfNPresenter class:

```
myOneOfN := new OneOfNPresenter
```

This script creates a new instance of OneOfNPresenter. You then add objects to this group using the methods available to the ArrayList class. The new method uses the same keyword arguments as init.

After creating an instance of OneOfNPresenter, you can add a 2D presenter to it, but then you must call goTo to show it, such as:

```
append myOneOfN myPresenter
goTo myOneOfN 1
```

init

init *self* [initialSize:*integer*] [growSize:*integer*] [stationary:*boolean*] ⇨
(none)

self	OneOfNPresenter object

Superclasses of OneOfNPresenter use these keywords:

initialSize:	Integer object
growSize:	Integer object
boundary:	(ignored by OneOfNPresenter)
target:	(ignored by OneOfNPresenter)
stationary:	Boolean object

Initializes the OneOfNPresenter object *self*, applying the arguments as follows: initialSize sets the initial number of empty slots in *self*, growSize determines the amount by which *self* can grow. Do not call init directly on an instance—it is automatically called by the new method.

If you omit an optional keyword, its default is used. The defaults are:

```
initialSize:20
growSize:20
stationary:false
```

Class Methods

Inherited from Collection:

pipe

Instance Variables

Inherited from Presenter:

presentedBy	subPresenters	target

Inherited from TwoDPresenter:

bBox	height	transform
boundary	IsImplicitlyDirect	width
clock	isTransparent	window
compositor	isVisible	x
direct	needsTickle	y
eventInterests	position	z
globalBoundary	stationary	
globalTransform	target	

Inherited from Collection:

bounded	maxSize	size
iteratorClass	minSize	uniformity
keyEqualComparator	mutable	uniformityClass
keyUniformity	mutableCopyClass	valueEqualComparator
keyUniformityClass	proprietored	

Inherited from SequenceCursor:

cursor

The following instance variables are defined in OneOfNPresenter:

purge

self.purge (read-write) Boolean

When true, causes the currently displayed presenter in the one-of-n-presenter, *self*, to be purged from memory when the next presenter is displayed. When false, causes the presenters to remain in memory. The default is true.

Set purge to true when the presenters are large and memory is tight; set it to false when you want to switch quickly between presenters and memory is not a problem.

Instance Methods

Inherited from TwoDPresenter:

adjustClockMaster	inside	show
createInterestList	localToSurface	surfaceToLocal
draw	notifyChanged	tickle
getBoundaryInParent	recalcRegion	
hide	refresh	

Inherited from Collection:

add	forEach	iterate
addMany	forEachBinding	localEqual
addToContents	getAll	map
chooseAll	getAny	merge
chooseOne	getKeyAll	pipe
chooseOneBinding	getKeyOne	prin
deleteAll	getMany	removeAll
deleteBindingAll	getOne	removeOne
deleteBindingOne	hasBinding	setAll
deleteKeyAll	hasKey	setOne
deleteKeyOne	intersects	size
deleteOne	isEmpty	
emptyOut	isMember	

Inherited from LinearCollection:

chooseOneBackwards	findRange	getNthKey
chooseOrdOne	forEachBackwards	getOrdOne

deleteFirst	getFirst	getRange
deleteLast	getLast	localEqual
deleteNth	getMiddle	localLT
deleteRange	getNth	pop

Inherited from Sequence:

addFifth	moveBackward	setFourth
addFirst	moveForward	setLast
addFourth	moveToBack	setNth
addNth	moveToFront	setSecond
addSecond	prepend	setThird
addThird	prependNew	sort
append	setFifth	
appendNew	setFirst	

Inherited from SequenceCursor:

backward	forward	goTo

The following instance methods are defined in OneOfNPresenter:

draw (TwoDPresenter)

draw *self surface clip* ⇨ *self*

self	OneOfNPresenter object
surface	DisplaySurface object
clip	Region object for parent's clip area

Tells the OneOfNPresenter object *self* to render its current image onto *surface*, with clipping defined by the stencil *clip*. This method is usually called by the 2D compositor, which knows what surface the image should be rendering onto.

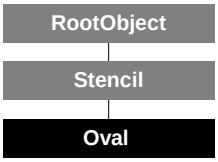
goTo (SequenceCursor)

goTo *self cursor* ⇨ *(object)*

self	OneOfNPresenter object
cursor	Integer object

Displays the member of the OneOfNPresenter object *self* whose position is specified by *cursor*. For more information about cursors, see the SequenceCursor class.

Oval



Class type: Core class (concrete)
Resides in: ScriptX and KMP executables
Inherits from: `Stencil`
Component: 2D Graphics

`Oval` is a subclass of `Stencil` that is used to render ovals and circles, which are a special case of oval. When an oval is rendered on a display surface, its shape is defined by the bounding rectangle in which it is inscribed. This bounding rectangle is represented by its upper-left and lower-right vertices. The boundary of the oval touches the midpoint of each edge of this bounding rectangle.

Note that `Oval` objects are not presenters—to display an oval, create an instance of `TwoDShape` using an `Oval` object as the *boundary* argument.

Creating and Initializing a New Instance

The following script is an example of how to create a new instance of the `Oval` class:

```
anOval := new Oval \
          x1:0 \
          y1:0 \
          x2:50 \
          y2:50
```

The variable `anOval` contains the new `Oval` instance. The upper left corner of its bounding box is positioned at 0,0, and the lower right corner of its bounding box is positioned at 50, 50. The `new` method uses the keywords defined in `init`.

init

`init` *self* [`x1:`*number*] [`y1:`*number*] [`x2:`*number*] [`y2:`*number*] ⇨ *(none)*

self	`Oval` object
`x1:`	`Number` object
`y1:`	`Number` object
`x2:`	`Number` object
`y2:`	`Number` object

Initializes the `Oval` object *self*, applying the arguments as follows: `x1:` and `y1:` represent one corner of the defining rectangle; `x2:` and `y2:` represent the opposite corner. Do not call `init` directly on an instance—it is automatically called by the `new` method.

If you omit an optional keyword, its default value is used. The defaults are:

```
x1:0
y1:0
x2:1
y2:1
```

Instance Variables

Inherited from `Stencil`:

`bBox`

The following instance variables are defined in `Oval`:

height

self`.height` (read-write) `Number`

Specifies the height of the oval *self*. Setting `height` resets `y2` and keeps `x1`, `y1`, and `x2` constant.

width

self`.width` (read-write) `Number`

Specifies the width of the oval *self*. Setting `width` resets `x2` and keeps `x1`, `y1`, and `y2` constant.

x1

self`.x1` (read-write) `Number`

Specifies the x coordinate of the upper-left corner of the bounding box.

x2

self`.x2` (read-write) `Number`

Specifies the x coordinate of the lower-right corner of the bounding box.

y1

self`.y1` (read-write) `Number`

Specifies the y coordinate of the upper-left corner of the bounding box.

y2

self`.y2` (read-write) `Number`

Specifies the y coordinate of the lower-right corner of the bounding box.

Instance Methods

Inherited from `Stencil`:

`inside`	`onBoundary`	`transform`
`intersect`	`subtract`	`union`

The following instance methods are defined in `Oval`:

copy

`copy` *self* ⇨ `Oval`

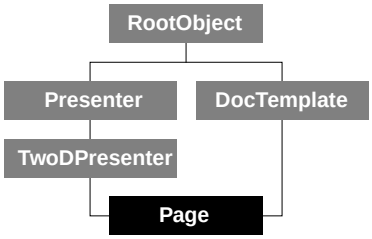
Creates and returns a copy of the oval *self*.

moveToZero

`moveToZero` *self* ⇨ `Oval`

Repositions the oval *self* so that the values of `x1` and `y1` are 0, adjusting the values of `x2` and `y2` to maintain their original offset.

Page



Class type: Core class (concrete)
Resides in: ScriptX and KMP executables
Inherits from: `TwoDPresenter` and `DocTemplate`
Component: Document Templates

The `Page` class represents pages in a document. The design of a `Page` is determined by the frame of the page, which is made of a `PageTemplate` or a `PageLayer` object. (A `PageTemplate` is itself a collection of `PageLayer` objects.)

When a page is displayed, its `changePage` method is called, which causes all the elements in the page layers on the page to update the data they display.

`Page` instances are an integral part of a document template hierarchy. A `Page` must be used in conjunction with `Document`, `PageTemplate`, `PageLayer` and `PageElement` instances. To display a page, first create the page using `PageTemplate`, `PageLayer` and `PageElement` instances to determine its design. Create a document, append the page to it, append the document to visible space, and go to the relevant page in the document.

Creating and Initializing a New Instance

The following script is an example of how to create a new instance of the `Page` class.

```
myPage := new Page \
          frame:standardTemplate \
          boundary:(new Rect x2:640 y2:480) \
          target:(p -> #(getHeading(p), getDescription(p)))
```

The variable `myPage` contains the initialized `Page` instance, which uses a rectangular boundary. The variable `standardTemplate` points to a frame for the page, which could be a `PageTemplate` containing one `PageLayer` for the background of the page and one `PageLayer` for the elements that display the text on the page. The `target` of `myPage` is an expression that generates an array containing text for the heading and text for the description to be displayed on the page. (Assume that the functions `getHeading` and `getDescription` have been defined to get the heading and description respectively.) The expression in the `target` instance variable is evaluated each time the page is displayed if any of the `PageElement` objects on the page need to access the page's `data` instance variable to find out what to present.

After creating the page, you would append it to a `Document`.

The `new` method uses the keywords defined in `init`.

init

`init` *self* `frame:`*pageTemplate* [`boundary:`*stencil*] [`target:`*object*] ⇨ *(none)*

self	`Page` object
`frame:`	`PageTemplate` or `PageLayer` object
`boundary:`	`Stencil` object defining region in which object is rendered
`target:`	An expression or object that can be used to generate or point to data for the document
`stationary:`	`Boolean` object

Initializes the `Page` object *self*, applying the keyword arguments as follows: `frame` is a `PageTemplate` or `PageLayer` object that acts as a container for other elements of the page; `boundary` is a `Stencil` object that defines the region in which the page will be rendered; and `target` is an object or expression that evaluates to the data needed by the page elements on a page when the page is displayed.

Although `boundary` is not a required keyword argument, the page will not be visible unless it has a boundary. Do not call `init` directly on an instance—it is automatically called by the `new` method.

If you omit an optional keyword, its default value is used. The defaults are:

```
boundary:undefined
target:undefined
stationary:false
```

Instance Variables

Inherited from `Presenter`:

`presentedBy`	`subPresenters`	`target`

Inherited from `TwoDPresenter`:

`bBox`	`height`	`transform`
`boundary`	`IsImplicitlyDirect`	`width`
`clock`	`isTransparent`	`window`
`compositor`	`isVisible`	`x`
`direct`	`needsTickle`	`y`
`eventInterests`	`position`	`z`
`globalBoundary`	`stationary`	
`globalTransform`	`target`	

The following instance variables are defined in `Page`:

data

self`.data` (read-only) *(object)*

Specifies data that can be used by page elements on the page *self*.

The value of `data` is determined by evaluating the `target` instance variable. The target of a `Page` object may be any constant value, in which case `target` and `data` are one and the same, or it may be a function such which is evaluated to produce the data. The data must make sense to the objects contained within the document. Ultimately, `data` can be used by the data reference within `PageElement` objects.

The value of `data` is calculated each time it is accessed, for example, each time a page is displayed where the page elements on the page refer to the `data` instance variable of the page. Note that this instance variable is read-only—you cannot modify it directly; it is automatically set when you get its value.

frame

self.frame (read-write) PageTemplate or PageLayer

Specifies the design layout for the page *self*. The value of this variable is a PageTemplate or PageLayer object. A PageTemplate is itself a collection of PageLayer objects. Each PageLayer object specifies a layer of design for the page.

target

self.target (read-write) (*function or expression*)

This instance variable is inherited from Presenter, but is used slightly differently in a page. The target of a page *self* is a function or expression that evaluates to the value for the data instance variable of the page. The value in the target instance variable of a document is evaluated only if the PageElement instances on a Page need to access the data instance variable on the page to find out what to present. If target is a function, the argument *self* is automatically passed into it.

In other words, whenever the data instance variable is accessed, target is evaluated and the result is put into data.

Instance Methods

Inherited from TwoDPresenter:

adjustClockMaster	inside	show
createInterestList	localToSurface	surfaceToLocal
draw	notifyChanged	tickle
getBoundaryInParent	recalcRegion	
hide	refresh	

Inherited from DocTemplate:

find parent	findNthParent
getNthParentData	getParentData

The following instance methods are defined in Page:

changePage

changePage *self newPage* ⇨ *(none)*

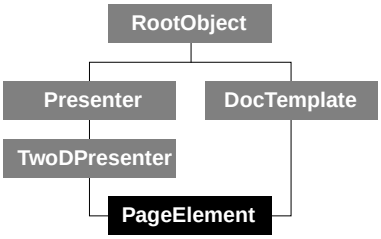
self	Page object
newPage	Page object

This method is called when a page opens. You do not need to call this method directly, since it is called automatically as necessary. However, you can specialize this method if you want to change the actions that occur when a page opens. Each time a page in a document opens, the document calls its changePage method, and also calls the changePage method on the relevant page, page template, page layers, and page elements. The default behavior of changePage is to cause the objects on the page to dynamically update their data.

On the Page class, the variables *self* and *newPage* are identical. The second argument is required to provide a consistent calling sequence for the changePage method on all of the classes that inherit from DocTemplate.

Specialize the changePage method to modify the behavior when a page in a document opens, for example, if the opening of the page should automatically start a sound or movie playing. The default definition of changePage causes all the PageElement objects on the page to evaluate the data they are to display. Thus, when modifying the changePage method, be sure to include a call to nextMethod to get the inherited behavior.

PageElement



Class type: Core class (concrete)
Resides in: ScriptX and KMP executables
Inherits from: `TwoDPresenter` and `DocTemplate`
Component: Document Templates

A `PageElement` object represents a single visual element on a page. The `element` of a `PageElement` is a `TwoDPresenter` object that contains a single presenter. The `target` is a function or expression that evaluates to the data to be presented by presenter on the page.

Each time a page is displayed, all the `PageElement` objects on the page re-evaluate their `target`, thus enabling the content of the page to be dynamically determined.

A `PageElement` acts as a proxy for its presenter and takes on its boundary and position.

`PageElement` instances are an integral part of a document template hierarchy. A `PageElement` must be used in conjunction with `Document`, `Page`, and `PageLayer` instances. A `PageElement` must be appended to a `PageLayer` which is used in the frame of a page in a document.

Creating and Initializing a New Instance

The following script is an example of how to create a new instance of the `PageElement` class:

```
myPageElement1 := new PageElement \
                  presenter:myTextPresenter \
                  boundary:(new Rect x1:40 y1:40 x2:600 y2:440) \
                  target:("I like to be in Bodfish." as Text)
```

The variable `myPageElement1` contains the initialized `PageElement` that presents a `TextPresenter`. In this case, the `target` of the `TextPresenter` is a constant. On every page that uses this page element, the page element displays the text `I like to be in Bodfish`.

`PageElement` instances containing constant targets,as shown in the previous example, are useful for background elements of a page, that do not change their content from page to page. However, for page elements whose content varies from page to page, you would normally supply an expression for the target, where the expression dynamically evaluates the content each time the page is displayed. For example:

```
myPageElement2 := new PageElement \
                  presenter:myTextPresenter \
                  boundary:(new Rect x1:40 y1:40 x2:600 y2:440) \
                  target:(e -> getfirst (getParentData e Page))
```

In this example, the variable myPageElement2 points to a PageElement that presents a TextPresenter. In this case, the the target of the TextPresenter is determined by retrieving the first value in the data instance variable of the page containing the page element. (This value in turn is determined by evaluating the expression in the page's target instance variable.)

After creating a page element, you should append it to a page layer, which in turn should be used as the frame for a page in a document.

The new method uses the keywords defined in init.

init

init *self* presenter:*twoDPresenter* [boundary:*stencil*] [target:*object*] ⇨ *(none)*

self	PageElement object
presenter:	A TwoDPresenter object, such as TextPresenter, TwoDShape, MoviePlayer, PushButton, that presents something on the page.
boundary:	Stencil object that determines the region of the element (if it is not superceded by the target.)
target:	An expression or object that either is, or evaluates to, the object to be presented by the page element.
stationary:	Boolean object

Initializes the PageElement object *self*, applying the arguments as follows: presenter is a TwoDPresenter object that contains a single presenter; boundary is a Stencil object that defines the region in which the page element will be rendered; and target is an object or function that is evaluated to produce the data to be presented by the PageElement. Do not call init directly on an instance—it is automatically called by the new method.

If you omit an optional keyword, its default value is used. The defaults are:

```
boundary:boundary of presenter, a TwoDPresenter object
target:undefined
stationary:false
Instance Variables
```

Inherited from Presenter:

presentedBy	subPresenters	target

Inherited from TwoDPresenter:

bBox	height	transform
boundary	IsImplicitlyDirect	width
clock	isTransparent	window
compositor	isVisible	x
direct	needsTickle	y
eventInterests	position	z
globalBoundary	stationary	
globalTransform	target	

The following instance variables are defined in PageElement:

data

self.data (read-only) (*object*)

Specifies the data presented by the PageElement object *self*. The value of data is the result of evaluating the target instance variable, which is inherited from the class Presenter. The target of a PageElement object may be any constant value, in which

case `target` and `data` are one and the same, or it may be a function which is evaluated to produce the data. `PageElement` objects frequently make use of the `data` instance variable in parent objects, instances of `Page`, `PageTemplate`, and `PageLayer`.

presenter

self`.presenter` (read-write) `TwoDPresenter`

Specifies the presenter that the `PageElement` object *self* contains. Its value must be a `TwoDPresenter` object that contains a single presenter, such as a `TwoDShape`, `TextPresenter`, `MoviePlayer`, `PushButton`, and so on.

target

self`.target` (read-write) (*object or expression*)

This instance variable is inherited from `Presenter`, but is used slightly differently in a page element. The `target` of a page element *self* is an object or function that evaluates to the value for the `data` instance variable of the page element, where the `data` instance variable determines the content to be displayed by the page element. (Thus, the `data` instance variable acts as an intermediary that is not present on most presenters that are not in a document template hierarchy.)

Instance Methods

Inherited from `TwoDPresenter`:

`adjustClockMaster`	`inside`	`show`
`createInterestList`	`localToSurface`	`surfaceToLocal`
`draw`	`notifyChanged`	`tickle`
`getBoundaryInParent`	`recalcRegion`	
`hide`	`refresh`	

Inherited from `DocTemplate`:

`find parent`	`findNthParent`
`getNthParentData`	`getParentData`

The following instance methods are defined in `PageElement`:

changePage

`changePage` *self newPage* ⇨ *(none)*

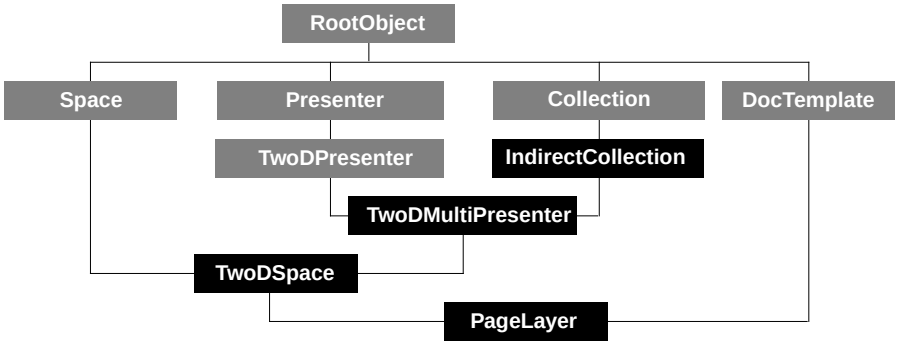
self	`PageElement` object
newPage	`Page` object

This method is called when a page that uses the page element *self* opens. You do not need to call this method directly, since it is called automatically as necessary. However, you can specialize this method if you want to change the actions that occur when a page opens. The default behavior of `changePage` is to cause the objects on the page to dynamically update their data.

In the `PageElement` class, the `changePage` method evaluates the expression in the element's `target` instance variable and sets the element's presenter's target to the resultant value.

If you specialize the `changePage` method for a new subclass, be sure to call `nextMethod` on the superclass, since the default `changePage` method makes sure that the objects used on the page dynamically update their data.

PageLayer



Class type: Core class (concrete)
Resides in: ScriptX and KMP executables
Inherits from: `TwoDSpace` and `DocTemplate`
Component: Document Templates

A `PageLayer` object is a 2D space that provides a layer of design for a page. It contains `PageElement` objects, where each `PageElement` object contributes an element to the design, such as a text box or a picture box. A `PageLayer` may represent a look that is shared by many pages, such as a background image, or a look that is unique to a single page.

The layout of a page can be determined by one or more page layers. If a page uses multiple page layers, the page layers are collected into a single page template that serves as the frame for the page.

Presenters and page elements may be placed on only one page layer. Page layers may be placed in many page templates or pages.

When a page is displayed, its page layers are rendered as a stack, with the first `PageLayer` on the bottom and the last one on top.

`PageLayer` instances are an integral part of a document template hierarchy. A `PageLayer` must be used in conjunction with `Document`, `Page`, `PageTemplate` and `PageElement` instances.

Creating and Initializing a New Instance

The following script is an example of how to create a new instance of the `PageLayer` class:

```
myPageLayer := new PageLayer \
                  boundary:myPageTemplate.boundary
```

The variable `myPageLayer` contains the initialized `PageLayer` instance, whose boundary is the same as `myPageTemplate`, which would be a `PageTemplate` object to which this page layer will be appended later.

After creating `myPageLayer`, you would append `PageElement` objects to it to specify the design of the layer, then use it as the frame for a page in a document. You might optionally append it to a `PageTemplate` object, in this case `myPageTemplate`, and use the page template as the frame for a page.

The `new` method uses the keywords defined in `init`.

init

`init` *self* [`boundary:`*stencil*] [`target:`*object*] [`fill:`*brush*] [`stroke:`*brush*] ⇨ *(none)*

self	`PageLayer` object
`boundary:`	`Stencil` object
`target:`	An expression or object that can be used to generate or point to data for the page layer
`stationary:`	`Boolean` object

Superclasses of `PageLayer` use the following keywords:

`fill:`	`Brush` object (that fills in the background of the page layer)
`stroke:`	`Brush` object (used for the border of the page layer)
`targetCollection:`	collection of targets for a `TwoDMultiPresenter`.

Initializes the `PageLayer` object *self*, applying the arguments as follows: `boundary` is a `Stencil` object that defines the region of the page layer, and `target` is an object or expression that evaluates to the data needed by the page elements on the page layer when the page containing the page layer is opened.

For performance reasons, when creating an instance of `PageLayer` (actually, any instance of `TwoDMultiPresenter` or its subclasses), you should not specify the `targetCollection` keyword, so it can create its default collection. Such presenters require collections that can be traversed quickly when drawing and handling events. Performance could suffer if you change the `targetCollection` keyword argument to something other than the default. Do not call `init` directly on an instance—it is automatically called by the `new` method.

If you omit an optional keyword, its default value is used. The defaults are:

```
boundary:undefined
target:undefined
fill:undefined
stroke:undefined
stationary:false
```

Class Methods

Inherited from `Collection`:

`pipe`

Instance Variables

Inherited from `DocTemplate`:

`data`

Inherited from `Collection`:

`bounded`	`maxSize`	`size`
`iteratorClass`	`minSize`	`uniformity`
`keyEqualComparator`	`mutable`	`uniformityClass`
`keyUniformity`	`mutableCopyClass`	`valueEqualComparator`
`keyUniformityClass`	`proprietored`	

Inherited from `IndirectCollection`:

`targetCollection`

Inherited from `Presenter`:

`presentedBy`	`subPresenters`	`target`

Inherited from TwoDPresenter:

bBox	height	transform
boundary	IsImplicitlyDirect	width
clock	isTransparent	window
compositor	isVisible	x
direct	needsTickle	y
eventInterests	position	z
globalBoundary	stationary	
globalTransform	target	

Inherited from TwoDMultiPresenter:

clock	fill	stroke

Inherited from Space:

clock	controllers	protocols

Inherited from TwoDSpace:

protocols

The following instance variables are defined in PageLayer:

data

self.data (read-only) *(object)*

Specifies data that can be used by PageElements on the page layer *self*.

The value of data is determined by evaluating the target instance variable. The target of a PageLayer object may be any constant value, in which case target and data are one and the same, or it may be a function which is evaluated to produce the data. The data must make sense to the objects contained within the document. Ultimately, data can be used by the data reference within PageElement objects.

The value of data is calculated each time it is needed, for example, each time a page is displayed where the page elements on the page refer to the data instance variable of the page layer.

Instance Methods

Inherited from DocTemplate:

find parent	findNthParent
getNthParentData	getParentData

Inherited from Collection:

add	forEach	iterate
addMany	forEachBinding	localEqual
addToContents	getAll	map
chooseAll	getAny	merge
chooseOne	getKeyAll	pipe
chooseOneBinding	getKeyOne	prin
deleteAll	getMany	removeAll
deleteBindingAll	getOne	removeOne
deleteBindingOne	hasBinding	setAll
deleteKeyAll	hasKey	setOne
deleteKeyOne	intersects	size
deleteOne	isEmpty	
emptyOut	isMember	

Inherited from IndirectCollection:

isAppropriateObject	objectAdded	objectRemoved

Inherited from TwoDPresenter:

adjustClockMaster	inside	show
createInterestList	localToSurface	surfaceToLocal
draw	notifyChanged	tickle
getBoundaryInParent	recalcRegion	
hide	refresh	

Inherited from TwoDMultiPresenter:

draw	findFirstInStencil	moveToBack
findAllAtPoint	isAppropriateObject	moveToFront
findAllInStencil	moveBackward	objectAdded
findFirstAtPoint	moveForward	objectRemoved

Inherited from Space:

isAppropriateObject	objectAdded	objectRemoved

Since a PageLayer object is an indirect collection, you can also use any methods defined in the class specified by targetCollection. The target collection is typically an instance of Array, which inherits from Sequence, so the following instance methods are redirected to a page layer.

Accessible from LinearCollection:

chooseOneBackwards	findRange	getNthKey
chooseOrdOne	forEachBackwards	getOrdOne
deleteFirst	getFirst	getRange
deleteLast	getLast	localEqual
deleteNth	getMiddle	localLT
deleteRange	getNth	pop

Accessible from Sequence:

addFifth	moveBackward	setFourth
addFirst	moveForward	setLast
addFourth	moveToBack	setNth
addNth	moveToFront	setSecond
addSecond	prepend	setThird
addThird	prependNew	sort
append	setFifth	
appendNew	setFirst	

The following instance methods are defined in PageLayer:

changePage

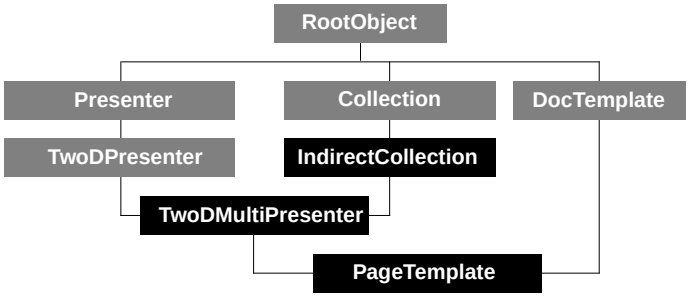
changePage *self newPage* ⇨ *(none)*

self	PageLayer object
newPage	Page object

This method is called when a page that uses the page layer *self* opens. You do not need to call this method directly, since it is called automatically as necessary. However, you can specialize this method if you want to change the actions that occur when a page opens. The default behavior of changePage is to cause the objects on the page to dynamically update their data.

Specialize the changePage method to modify the behavior when a page layer is displayed when a page opens. The default definition of changePage for a page layer causes all the PageElement objects presented by the page layer to evaluate the data they are to display. Thus, when modifying the changePage method, be sure to include a call to nextMethod to get the inherited behavior.

PageTemplate



Class type: Core class (concrete)
Resides in: ScriptX and KMP executables
Inherits from: `TwoDMultiPresenter` and `DocTemplate`
Component: Document Templates

A `PageTemplate` object specifies the design for a page in a document. A `PageTemplate` is a collection of `PageLayer` objects, where each page layer contributes a layer of design to the overall layout. A page template could be used to create a common look shared by many pages.

The same `PageTemplate` object can be used on many different pages within a document.

A `PageTemplate` object can be used anywhere that a `PageLayer` object is used, as a container for `PageLayer` objects and other `PageTemplate` objects.

`PageTemplate` instances are an integral part of a document template hierarchy. A `PageTemplate` object must be used in conjunction with `Document`, `Page`, `PageLayer` and `PageElement` instances.

Creating and Initializing a New Instance

The following script is an example of how to create a new instance of the `PageTemplate` class:

```
myPageTemplate := new PageTemplate \
                  boundary:(new Rect x1:0 y1:0 x2:640 y2:480)
```

The variable `myPageTemplate` points to a `PageTemplate` object with the given rectangular boundary. To use `myPageTemplate` as a page template, you would need to append `PageLayer` objects to it, then specify `myPageTemplate` as the `frame` for one or more pages in a document.

The `new` method uses the keywords defined in `init`.

init

`init` *self* [`boundary:`*stencil*] [`target:`*object*] ⇨ *(none)*

self	`PageTemplate` object
`boundary:`	`Stencil` object of the region for the page template
`target:`	An expression or object that can be used to generate or point to data for the page template.
`stationary:`	`Boolean` object

Initializes the `PageTemplate` object *self*, applying the arguments as follows: `boundary` is a `Stencil` object that defines region for the page template, and `target` is an object or expression that evaluates to the data needed by the page elements on the page layer in the page template when the page containing the page template is opened.

If you omit an optional keyword, its default value is used. The defaults are:

```
boundary:(new Rect x2:0 y2:0)
target:undefined
stationary:false
```

Class Methods

Inherited from `Collection`:

`pipe`

Instance Variables

Inherited from `Collection`:

`bounded`	`maxSize`	`size`
`iteratorClass`	`minSize`	`uniformity`
`keyEqualComparator`	`mutable`	`uniformityClass`
`keyUniformity`	`mutableCopyClass`	`valueEqualComparator`
`keyUniformityClass`	`proprietored`	

Inherited from `IndirectCollection`:

`targetCollection`

Inherited from `Presenter`:

`presentedBy`	`subPresenters`	`target`

Inherited from `TwoDPresenter`:

`bBox`	`height`	`transform`
`boundary`	`IsImplicitlyDirect`	`width`
`clock`	`isTransparent`	`window`
`compositor`	`isVisible`	`x`
`direct`	`needsTickle`	`y`
`eventInterests`	`position`	`z`
`globalBoundary`	`stationary`	
`globalTransform`	`target`	

Inherited from `TwoDMultiPresenter`:

`fill`	`stroke`

The following instance variables are defined in `PageTemplate`:

data

self`.data` (read-only) *(object)*

Specifies data that can be used by `PageElements` on the page using the page template *self*.

The value of `data` is determined by evaluating the `target` instance variable. The target of a `PageTemplate` object may be any constant value, in which case `target` and `data` are one and the same, or it may be a function which is evaluated to produce the data. The data must make sense to the objects contained within the document. Ultimately, `data` can be used by the data reference within `PageElement` objects.

The value of `data` is calculated each time it is needed, for example, each time a page is displayed where the page elements on the page layers in the page's page template refer to the `data` instance variable of the page template.

Instance Methods

Inherited from DocTemplate:

find parent	findNthParent	
getNthParentData	getParentData	

Inherited from Collection:

add	forEach	iterate
addMany	forEachBinding	localEqual
addToContents	getAll	map
chooseAll	getAny	merge
chooseOne	getKeyAll	pipe
chooseOneBinding	getKeyOne	prin
deleteAll	getMany	removeAll
deleteBindingAll	getOne	removeOne
deleteBindingOne	hasBinding	setAll
deleteKeyAll	hasKey	setOne
deleteKeyOne	intersects	size
deleteOne	isEmpty	
emptyOut	isMember	

Inherited from IndirectCollection:

isAppropriateObject	objectAdded	objectRemoved

Inherited from TwoDPresenter:

adjustClockMaster	inside	show
createInterestList	localToSurface	surfaceToLocal
draw	notifyChanged	tickle
getBoundaryInParent	recalcRegion	
hide	refresh	

Inherited from TwoDMultiPresenter:

draw	findFirstInStencil	moveToBack
findAllAtPoint	isAppropriateObject	moveToFront
findAllInStencil	moveBackward	objectAdded
findFirstAtPoint	moveForward	objectRemoved

Since a PageTemplate controller is an indirect collection, you can also use any methods defined in the class specified by targetCollection. The target collection is typically an instance of Array, which inherits from Sequence, so the following instance methods are redirected to this controller.

Accessible from LinearCollection:

chooseOneBackwards	findRange	getNthKey
chooseOrdOne	forEachBackwards	getOrdOne
deleteFirst	getFirst	getRange
deleteLast	getLast	localEqual
deleteNth	getMiddle	localLT
deleteRange	getNth	pop

Accessible from Sequence:

addFifth	moveBackward	setFourth
addFirst	moveForward	setLast
addFourth	moveToBack	setNth
addNth	moveToFront	setSecond
addSecond	prepend	setThird
addThird	prependNew	sort
append	setFifth	
appendNew	setFirst	

The following instance methods are defined in `PageTemplate`:

changePage

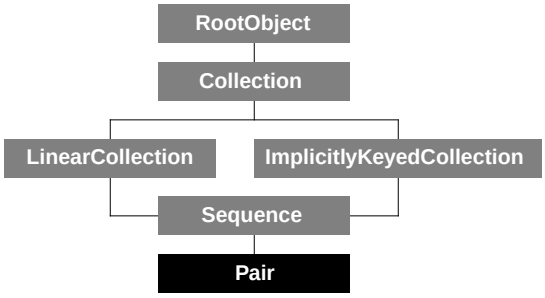
changePage *self newPage* ⇨ *(none)*

self	`PageTemplate` object
newPage	`Page` object

This method is called when a page that uses the page template *self* opens. You do not need to call this method directly, since it is called automatically as necessary. However, you can specialize this method if you want to change the actions that occur when a page opens. The default behavior of changePage is to cause the objects on the page to dynamically update their data.

Each time a page in a document opens, the document calls its changePage method, and also calls the changePage method on the relevant page, page template, page layers, and page elements. If you specialize the changePage method for a new subclass, be sure to call `nextMethod` on the superclass, since the default changePage method makes sure that the objects used on the page dynamically update their data.

Pair



Class type: Core class (concrete)
Resides in: ScriptX and KMP executables
Inherits from: `Sequence`
Component: Collections

The `Pair` class represents arrays of two values. A `Pair` object is created with a fixed size of 2. You cannot add (`addNth`) or delete (`deleteNth`) items, but you can set their values (`setNth`). Adding a value to a `Pair` object causes it to report the exception `bounded`.

Also see the `Single`, `Triple`, and `Quad` classes.

Creating and Initializing a New Instance

The following script is an example of how to create a new instance of the `Pair` class:

```
myPair := new Pair values:#(1,2)
```

The variable `myPair` contains the initialized pair, which contains the numbers 1 and 2. The `new` method uses the keywords defined in `init`.

init

`init` *self* [`values:`*collection*] ⇨ *(none)*

self `Pair` object

Superclass `Collection` uses the following keyword:

`values:` `Collection` object of size 2

Initializes the `Pair` object *self*, setting the two items in the list to the members of the `values` collection. Do not call `init` directly on an instance—it is automatically called by the `new` method.

If you omit an optional keyword, its default value is used. The defaults are:

```
values: #(undefined, undefined)
```

Class Methods

Inherited from `Collection`:

`pipe`

Instance Variables

Inherited from `Collection`:

`bounded`	`maxSize`	`size`
`iteratorClass`	`minSize`	`uniformity`

keyEqualComparator	mutable	uniformityClass
keyUniformity	mutableCopyClass	valueEqualComparator
keyUniformityClass	proprietored	

Instance Methods

Inherited from Collection:

add	forEach	iterate
addMany	forEachBinding	localEqual
addToContents	getAll	map
chooseAll	getAny	merge
chooseOne	getKeyAll	pipe
chooseOneBinding	getKeyOne	prin
deleteAll	getMany	removeAll
deleteBindingAll	getOne	removeOne
deleteBindingOne	hasBinding	setAll
deleteKeyAll	hasKey	setOne
deleteKeyOne	intersects	size
deleteOne	isEmpty	
emptyOut	isMember	

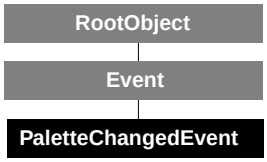
Inherited from LinearCollection:

chooseOneBackwards	findRange	getNthKey
chooseOrdOne	forEachBackwards	getOrdOne
deleteFirst	getFirst	getRange
deleteLast	getLast	localEqual
deleteNth	getMiddle	localLT
deleteRange	getNth	pop

Inherited from Sequence:

addFifth	moveBackward	setFourth
addFirst	moveForward	setLast
addFourth	moveToBack	setNth
addNth	moveToFront	setSecond
addSecond	prepend	setThird
addThird	prependNew	sort
append	setFifth	
appendNew	setFirst	

PaletteChangedEvent



Class type: Core class (concrete)
Resides in: ScriptX and KMP executables
Inherits from: `Event`
Component: Events

The `PaletteChangedEvent` class is used to notify interested event receivers of a change in the hardware color palette. A change in pixel-depth is equivalent to a palette change. If the hardware color palette changes for a given screen, one instance of `PaletteChangedEvent` is broadcast to all interested receivers for each instance of `DisplaySurface` that is currently attached to a 2D compositor.

A developer might want to receive this class of event in order to maintain an alternate set of bitmaps or media images for different pixel depths in which the user might choose to run a title.

A developer does not normally need to create an instance of this class. `TwoDCompositor` objects maintain interests in palette-changed events. For example, the instance of `TwoDCompositor` that is associated with an instance of `DisplaySurface` automatically creates and maintains interests in this class of events. This compositor receives the event and updates the `colorMap` instance variable on the display surface.

An interest in `PaletteChangedEvent` is usually associated with a particular display surface. If the `displaySurface` instance variable is set on an interest, the `isSatisfiedBy` method accepts only events that are associated with that display surface.

Creating and Initializing a New Instance

The following script creates a new instance of the `PaletteChangedEvent` class:

```
myPCEvent := new PaletteChangedEvent
```

The variable `myPCEvent` contains an initialized `PaletteChangedEvent` instance. The `new` method calls the `init` method defined by `PaletteChangedEvent`.

init

`init` *self* ⇨ *(none)*

self `PaletteChangedEvent` object

Initializes the `PaletteChangedEvent` object *self*. Do not call `init` directly on an instance—it is automatically called by the `new` method.

Class Variables

Inherited from `Event`:

`interests` `numInterests`

Class Methods

Inherited from Event:

acquireQueueFromPool	relinquishQueueToPool
broadcastDispatch	signalDispatch

Instance Variables

Inherited from Event:

advertised	eventReceiver	timeStamp
authorData	matchedInterest	
device	priority	

The following instance variables are defined in PaletteChangedEvent:

The following two phrases are used below to refer to the type of event:
- "Interest-only" means the instance *self* is used to express an event interest.
- "Event-only" means the instance *self* holds an actual user event.

bitsPerPixel

self.bitsPerPixel (read-write) Integer

Event-only instance variable. Think of the PaletteChangedEvent object *self* as a message to notify interested receivers that the user or the system has changed the hardware color palette. This instance variable indicates the pixel depth of the new ColorMap object for the DisplaySurface object referred to by the displaySurface instance variable. Possible values are any pixel depths that are supported by both the underlying hardware and the Kaleida Media Player. Typical values are 2, 4, 8, 16, and 24.

colorMap

self.colorMap (read-write) ColorMap

Event-only instance variable. Think of the PaletteChangedEvent object *self* as a message to notify interested receivers that the user or the system has changed the hardware color palette. This instance variable indicates the new ColorMap object for the DisplaySurface object referred to by the displaySurface instance variable.

displaySurface

self.displaySurface (read-write) DisplaySurface

Indicates the display surface for which the PaletteChangedEventobject *self* was broadcast. When the hardware palette of a screen changes, a PaletteChangedEvent instance is broadcast for each DisplaySurface object on the screen.

Instance Methods

Inherited from Event:

accept	isSatisfiedBy	sendToQueue
acquireRejectQueue	reject	signal
addEventInterest	relinquishRejectQueue	
broadcast	removeEventInterest	

The following instance methods are defined in PaletteChangedEvent:

isSatisfiedBy (Event)

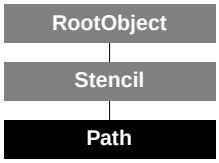
`isSatisfiedBy` *self event* ⇨ `Boolean`

self	`Event` object that represents an event interest
event	`Event` object that represents an actual event

The method `isSatisfiedBy` tests whether an event interest is satisfied by an event. Do not call `isSatisfiedBy` directly from a script. It is called by the event system as a result of calling either `signal` or `broadcast`.

The class `PaletteChangedEvent` specializes `isSatisfiedBy` to test whether the given *event* occurred on the same display surface as the event interest *self*.

Path



Class type: Core class (concrete)
Resides in: ScriptX and KMP executables
Inherited from: `Stencil`
Component: 2D Graphics

The `Path` class provides a way to build multi-segmented, multi-contoured paths or shapes. A path can include straight lines, arcs, curves, and splines. A path can have multiple contours (or subpaths). A path can be open or closed. Figure 4-12 shows some sample paths.

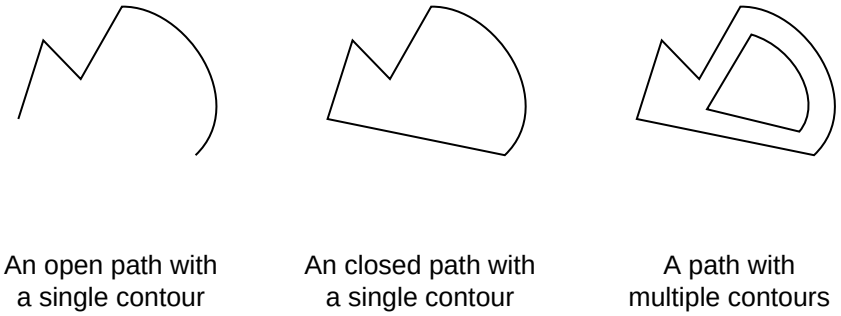


Figure 4-12: Sample paths

Since a `Path` object is a `Stencil`, it must be put in a presenter before it can be viewed. You must specify a value for the `stroke` instance variable for the presenter to make the path visible. (By default, the stroke value is always `undefined`.)

You can specify the value for the `fill` instance variable of a presenter that presents a closed path. However, specifying a `fill` value for a presenter that presents an open path may lead to unpredictable consequences.

If a path that crosses itself is filled, the filling toggles each time a line is crossed, as illustrated in Figure 4-13, which shows an unfilled path and the same path filled.

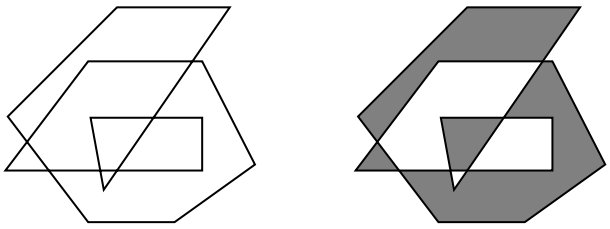


Figure 4-13: If a path crosses itself, filling alternates each side of a crossed line.

Building A Path

To create a new path, call the `new` method on the class `Path`. The `Path` class behaves a little differently to other ScriptX classes. To build the path, you must call a sequence of methods, instead of providing keywords to the `new` method and setting instance variables on the new object, as you would with most other classes. After building a path, you can modify it by adding new segments (lines, curves, arcs, or splines). You cannot modify existing parts of the path.

After calling the `new` method on the class `Path`, you need to specify the starting point of the path by calling its `moveTo` method. This starting point becomes the "current point." When a path is extended by drawing a line, an arc, a curve or a spline, the extension is always drawn starting from the current point. After a path is extended, the end of the path becomes the new current point.

To move the current point to another position without drawing the connecting line, use the `moveTo` method. The point to which the path is moved becomes the starting point for a new contour (or subpath).

You can use the `closePath` method to draw a line from the current point to the starting point for the current contour (or subpath).

When you make changes directly to a `Path` object, the presenter presenting the path does not update to show the changes. To update the appearance of the shape when the path changes, set the `changed` instance variable on the shape to `true` after making the change to the path. (You will need to do this each time you change the path and want to see the effect of the change.)

Drawing Line Segments

To add a line segment to the end of a path, use the `lineTo` method.

Drawing Arcs

To add an arc to the end of a path, use the methods `arc`, `arcn`, or `arcTo`. Each of these methods is discussed in the API method listing later in the chapter.

When adding an arc to a path, you need to specify angle arguments. 0 degrees corresponds to the 3 o'clock position, and increases going clockwise. That is, the 6 o'clock position is 90 degrees; the 9 o'clock position is 180 degrees; and the 12 o'clock position is 270 degrees. Figure 4-14 illustrates how angles work for drawing arcs.

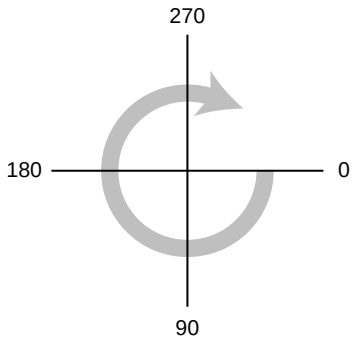


Figure 4-14: Angles for Arcs

Drawing Curves

To draw a curve at the end of a path, use the method `curveTo`, which draws a cubic Bezier curve (which is the same kind of curve that the class `Curve` draws).

Drawing Splines

You can add cubic splines, B-splines, and quadratic splines to a path. To draw a spline, first call a method that draws the first segment of the appropriate kind of spline, such as `startCubicSpline`, `startBSpline`, and `startQuadSpline`. To add further segments to the spline, use one of the methods `cubicSplineTo`, `bSplineTo`, and `quadSplineTo`, depending on the kind of spline to be extended. After starting a spline of a particular kind, you can add further segments of the same kind of spline. If you want to change the kind of spline, you need to call a method to start that kind of spline.

Quadratic and cubic splines are made of a series of quadratic Bezier curves and cubic Bezier curves respectively. The curves are connected so that their tangents are continuous. Quadratic splines are used in TrueType and QuickDraw GX. B-splines are the smoothest of all three splines (up to the 2nd derivative) based on cubic equations. Note that the start and end points of a B-spline are different from the supplied control points.

When you draw a spline, ScriptX draws from the current point in the path to the end point in the spline. If you extend the spline, the previous end point becomes a control point in a continuing spline, and the previously drawn path alters its shape to accommodate the change in the spline. Figure 4-15 illustrates this concept. In the top path, the point marked as 3 is the end of the path, and is the current point. However, when the spline is extended, as shown in the bottom path, the path no longer passes through point 3. The path has changed shape because the spline was extended.

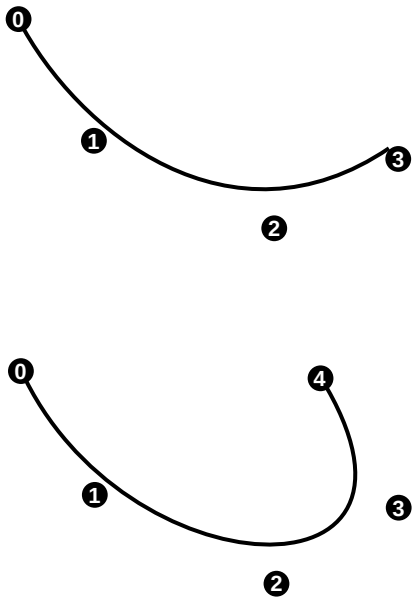


Figure 4-15: When a spline is extended, the path changes its shape

Sample Script for Building a Path

The following sample script builds a path that depicts a face, as shown in Figure 4-16.

```
global p := new Path

global s := new Twodshape boundary:p stroke:blackbrush

-- Make the left eye
moveto p 0 0
lineto p 60 0
lineto p 60 20
lineto p 0 20
lineto p 0 15
lineto p 6 15
lineto p 5 20
lineto p 0 20
closepath p

-- make the right eye
moveto p 100 0
lineto p 160 0
lineto p 160 20
lineto p 100 20
lineto p 100 15
lineto p 106 15
lineto p 105 20
lineto p 100 20
closepath p

-- make the nose
moveto p 80 40
lineto p 40 80
lineto p 80 80

-- make the mouth
moveto p 20 100
curveto p 60 160 100 160 140 100
curveto p 100 140 60 140 20 100

-- show the face in a window
w := new window
show w
append w s
```

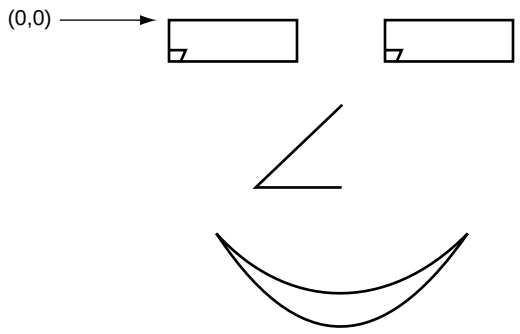


Figure 4-16: A path that depicts a face

Instance Variables

Inherited from `Stencil`:

`bbox`

The following instance variables are defined in `Path`:

height

self`.height` (read-write) `Float`

Returns the height of the `Path` object *self*. When the value of `height` is set, the path is rescaled automatically and its width is set accordingly, using the value of `realBbox` to determine the object's new dimensions. The object's position, that is, its top left corner, is unchanged in the process.

insideRule

self`.insideRule` (read-write) `NameClass`

Specifies the rule used to determine which parts of a path to fill when the presenter presenting the path *self* is filled. Currently, the only valid setting is `@evenOdd`. With this rule, a point is inside the area to be filled if the number of path crossings from that point to the outside of the path is odd. A point is outside if the number of crossings is even. (More simply put, the fill state toggles as each line is crossed.)

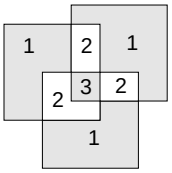


Figure 17: Even-odd rule

length

self`.length` (read-only) `Number`

Returns the length of the `Path` object *self*.

realBbox

self.realBbox (read-only) Rect

Returns a Rect object that is the actual bounding rectangle of the path *self*, including its line width. Compare realBbox with bbox, an instance variable that is inherited from Stencil. The value of bbox reflects the bounding rectangle of the path's control points, including the line width. Hence, the area of the rectangle returned by bbox is greater than or equal to that of the rectangle returned by realBbox. When the height or width is rescaled, realBbox rather than bbox is used to determine the path's new dimensions.

width

self.width (read-write) Float

Returns the width of the Path object *self*. When the value of width is set, the path is rescaled automatically and its width is set accordingly, using the value of realBbox to determine the object's new dimensions. The object's position, that is, its top left corner, is unchanged in the process.

Instance Methods

Inherited from Stencil:

inside	subtract
intersect	transform
onBoundary	union

The following methods are defined in Path:

arc

arc *self x y r angle1 angle2* ⇨ Path

self	The Path object.
x, *y*	The x and y values of the point at the center of a circle of radius *r*.
r	The radius of the arc.
angle1	The starting angle of the arc. The value is from 0 to 360.
angle2	The ending angle of the arc. The value is from 0 to 360.

This method draws an arc of a circle whose center is at (*x*, *y)* and whose radius is *r*. The arc starts at *angle1* and finishes at *angle2* . (0 degrees starts at the 3 o'clock position, and increases going clockwise.) This method also draws a line from the current point of the path to the start of the arc.

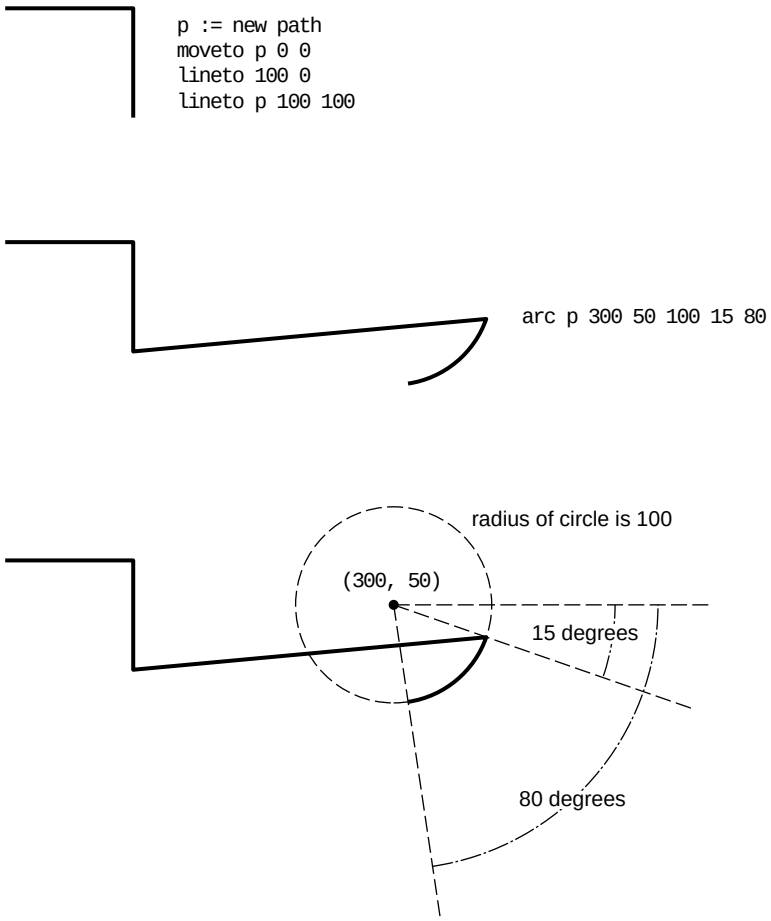


Figure 4-18: Illustration of the arc method

arcn

`arcn` *self x y r angle1 angle2* ⇨ `Path`

This method does the same as `arc`, except that the angles are measured in a counter-clockwise direction.

arcTo

`arcTo` *self x1 y1 x2 y2 r* ⇨ `Path`

self	The `Path` object.
x1, y1	The x and y values of the end point of a line that starts at the current point of the path. This line specifies the tangent of the starting point of the arc.
x,2 y2	The x and y values of the end point of a line that starts at (*x1, y1*). This line specifies the tangent of the end point of the arc.
r	The radius of the arc.

This method draws an arc of radius *r*. The arc is tangent to the lines from the current point to (*x1, y1*) and from (*x1, y1*) to (*x2, y2*) as illustrated in Figure 4-19. The method also draws a line from the current point to the start of the arc (the start of the arc lies on the line from the current point to *x1, y1*).

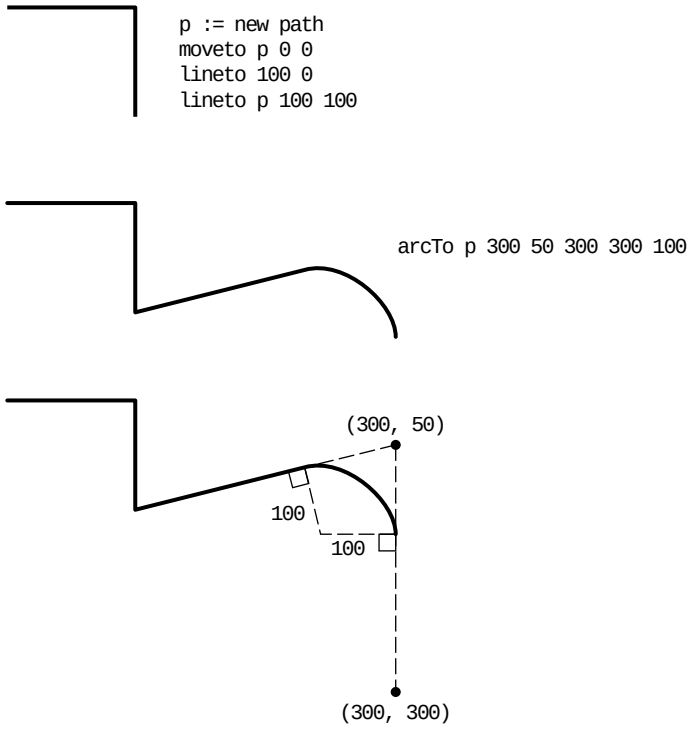


Figure 4-19: The arcTo method

bSplineTo

`bSplineTo` *self x y* ⇨ `Path`

self	The `Path` object.
x1, y1	The x and y values of the new end point of the B-spline.

Extends a B-spline on the path *self* by adding a control point (*x*, *y*). If there are `n` control points, `P1`, `P2`, `P3` ... `Pn`, the end point is given by `(Pn-2 + 4Pn-1 + Pn)/6`, and this becomes the current point.

Before extending a B-spline, the spline must be started with the method `startBSpline`.

Figure 4-20 on page 511 shows a sample B-spline.

closePath

`closePath` *self* ⇨ `Path`

Closes the current subpath of the path *self* by adding a line segment from the current point to the first point of the subpath. When the current subpath is closed or empty, this method has no effect. After a subpath is closed, the first point of the closed subpath becomes the current point. Even if the first and last points on a path coincide, you should call the `closePath` method before calling the `fill` method on a path, because filling an open path may have unpredictable results.

cubicSplineTo

`cubicSplineTo` *self x y* ⇨ `Path`

self	The `Path` object.
x y	The x and y values of the new end point of the cubic spline.

Extends a cubic spline on the path *self* to point (*x*, *y*). This method turns the current point in the path into a control point (instead of an end point) for the spline, and the path changes shape accordingly. The point (*x*,*y*) becomes the current point.

Before extending a cubic spline, the spline must be started with the method `startCubicSpline`.

Figure 4-21 on page 512 shows a sample cubic spline.

curveTo

`curveTo` *self x1 y1 x2 y2 x3 y3* ⇨ `Path`

self	The `Path` to which to add the curve.
x1, *y1*	The x and y values of the first control point in the curve.
x2, *y2*	The x and y values of the second control point in the curve.
x3, *y3*	The x and y values of the end of the curve.

Draws a cubic Bézier curve from the current point of the path *self* to the point (*x3*, *y3*) with two control points (*x1*, *y1*) and (*x2*, *y2*). See the discussion of the `Curve` class for an explanation of how the control points affect the curve.

getAngle

`getAngle` *self d* ⇨ `Number`

self	`Path` object
d	`Number` between 0.0 and 1.0 that indicates how far along the curve to get the angle.

Returns the tangent angle (in radians) on the path at distance *d* along the path.

getCurvature

`getCurvature` *self d* ⇨ `Number`

self	`Path` object
d	`Number` between 0.0 and 1.0 that indicates how far along the curve to get the angle.

Returns the curvature of the path at distance *d* along the path. The curvature is positive if the tangent angle increases, that is, the curve at the point is heading in a clockwise direction. The curvature is negative if the curve is heading in a counter-clockwise direction. The curvature is zero if the path is straight. The inverse of the curvature gives the radius of the tangential circle.

getPoint

`getPoint` *self d* ⇨ `Point`

self	`Path` object
d	`Number` between 0.0 and 1.0 that indicates how far along the path to get the point.

Returns the point that lies on the path at distance *d* along the path. If *d* is 0, returns the starting point of the path. If *d* is 1, returns the finishing point. If *d* is 0.25, returns the point that is a quarter of a way along the distance of the path. If *d* is 0.5, returns the point that is half way along the distance of the path, and so on.

The following code shows how to create a group space containing a previously-created `Path` object and also a small round shape. When the group space is displayed in a window, the small round shape will appear at the point half way along the distance of the path.

```
g1 := new GroupSpace
s1 := new twoDShape boundary:path1 stroke:(new brush color:bluecolor)
s2 := new twoDShape boundary:(new oval x2:5 y2:5) fill:blackbrush
p1 := getPoint path1 0.5
s2.x := p1.x
s2.y := p1.y
append g1 s1
append g1 s2
```

join

`join` *self p1 combination todo* ⇨ `Path`

self	A `Path` object.
p1	A `Path` object.
combination	A `NameClass` object specifying which points to join. The value must be one of `@startToStart`, `@startToEnd`, `@endToStart`, or `@endToEnd`.
todo	A `NameClass` object whose value must be either `@create` or `@mutate`.

Joins a `Path` object *p1*, to the path *self*, modifying *self*. The *combination* argument specifies which points of the path to join. The joining points are connected by a straight line if they are different.

The *todo* argument specifies whether a new path is created and returned (`@create`) or whether the path *self* is modified and returned (`@mutate`).

lineTo

`lineTo` *self x y* ⇨ `Path`

self	`Path` object
x	`Number` object
y	`Number` object

Appends a straight line to the path *self*. from the current point to the point (*x*, *y*), which becomes the current point. If the path is empty (has no last point), then this method reports the exception `@badParamater`.

moveTo

`moveTo` *self x y* ⇨ `Path`

self	`Path` object
x	`Number` object
y	`Number` object

Starts a new subpath of the path *self*. Moves the current point of the path *self* to (*x*, *y*) without drawing the line from the previous current point.

newPath

`newPath` *self* ⇨ `Path`

Resets the contents of the path *self* to a zero length path.

quadSplineTo

`quadSplineTo` *self x y* ⇨ `Path`

self	The `Path` object.
x, y	The x and y values of the new end point of the B-spline.

Extends a quadratic spline on the path *self* to point (*x, y*). This method turns the current point in the path into a control point (instead of an end point) for the spline, and the path changes shape accordingly. The point (*x,y*) becomes the current point.

Before extending a quadratic spline, the spline must be started with the method `startQuadSpline`.

Figure 4-22 on page 513 shows a sample quadratic spline.

startBSpline

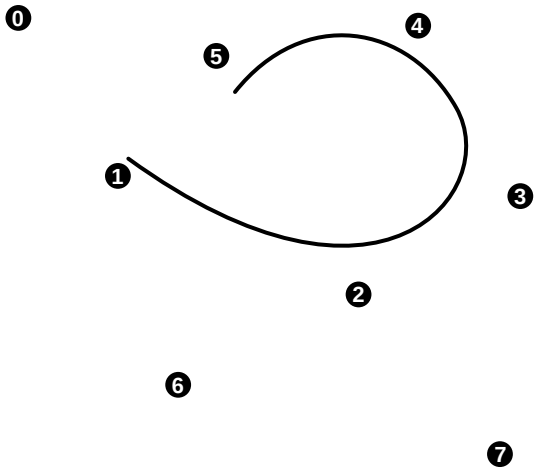
`startBSpline` *self x1 y1 x2 y2 x3 y3* ⇨ `Path`

self	The `Path` object.
x1, y1	The x and y values of the first control point of the B-spline.
x2, y2	The x and y values of the second control point of the B-spline.
x3. y3	The end point for the B-spline.

Draws a B-spline curve. This method uses the current point, `P0`, and supplied points `P1` (*x1, y1*), `P2` (*x2, y2*), and `P3` (*x3,y3*) as control points.The B-spline starts at `(P0 + 4P1 + P2)/6` and ends at `(P1 + 4P2 + P3)/6`, making the endpoint the current point. If there is a previous line or curve, a line is added between the end point of the previous segment and the start point of the B-spline.

Use this method to start a spline. Use the `bSplineTo` method to continue a B-spline.

Figure 4-20 shows a sample B-spline.




```
p2 := new path
moveTo p2 0 0

startBspline p2 50 80 170 140 250 90
BSplineTo p2 200 5
BSplineTo p2 100 20
BSplineTo p2 80 185
BSplineTo p2 240 220
```

Figure 4-20: Sample B-spline

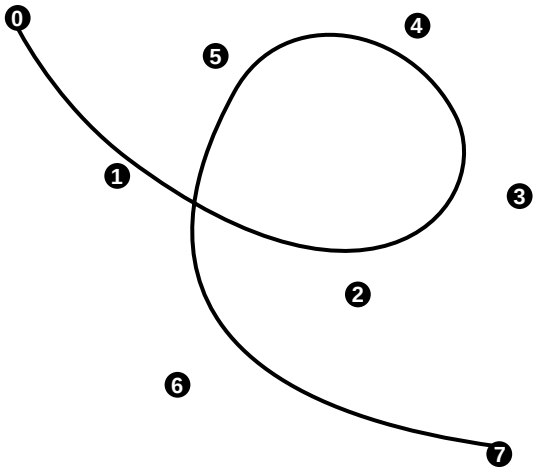
startCubicSpline

`startCubicSpline` *self x1 y1 x2 y2 x3 y3* ⇨ `Path`

self	The `Path` object.
x1, y1	The x and y values of the first control point of the cubic spline.
x2, y2	The x and y values of the second control point of the cubic spline.
x3. y3	The end point for the cubic spline.

Draws a cubic Bezier spline curve from the current point of the `Path` *self* to (*x3, y3*), using (*x1, y1*) and (*x2, y2*) as the control points for the spline. Use this method to start a spline. Use the `cubicSplineTo` method to continue a cubic spline.

Figure 4-21 shows a sample cubic spline. Figure 4-23 shows both a cubic spline and a quadratic spline that use the same control points.




```
p1 := new path
moveto p1 0 0
startCubicSpline p1 50 80 170 140 250 90
cubicSplineTo p1 200 5
cubicSplineTo p1 100 20
cubicSplineTo p1 80 185
cubicSplineTo p1 240 220
```

Figure 4-21: A sample cubic spline

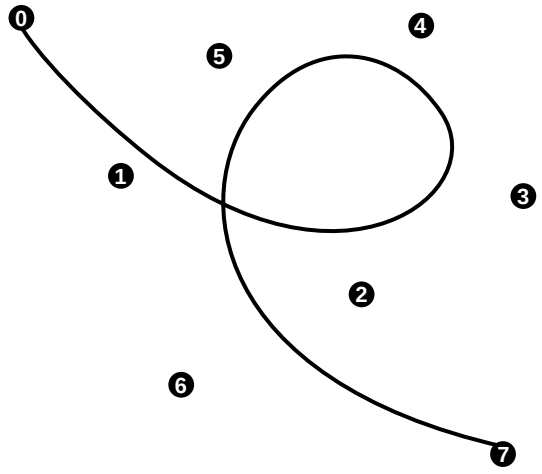
startQuadSpline

`startQuadSpline` *self x1 y1 x2 y2* ⇨ `Path`

self	The `Path` object.
x1, y1	The x and y values of the first control point of the B-spline.
x2, y2	The x and y values of the second control point of the B-spline.

Draws a TrueType style spline cuve (that uses quadratic Bézier control points) from the current point of the `Path` *self* to (*x2, y2*) , using (*x1, y1*) as the control point for the spline. Use this method to start a spline. Use the `quadSplineTo` method to continue a quad spline.

Figure 4-22 shows a sample quadratic spline. Figure 4-23 shows both a cubic spline and a quadratic spline that use the same control points.




```
p1 := new path
moveto p1 0 0
startQuadSpline p1 50 80 170 140
quadSplineTo 250 90
quadSplineTo p1 200 5
quadSplineTo p1 100 20
quadSplineTo p1 80 185
quadSplineTo p1 240 220
```

Figure 4-22: A sample quadratic spline

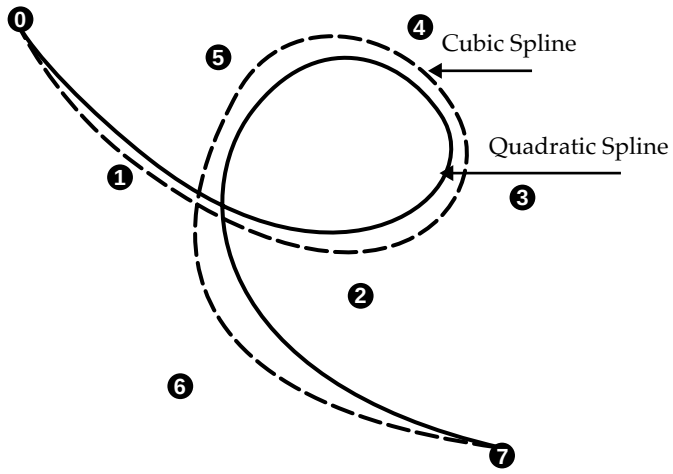
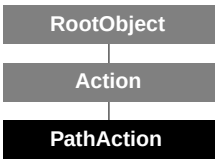


Figure 4-23: Comparison of sample cubic and quadratic splines

PathAction



Class type: Core class (concrete)
Resides in: ScriptX and KMP executables
Inherits from: `Action`
Component: Animation

The `PathAction` class represents an action that sets the position of a target object to a given location at a specified time, under the control of an action list player. To be triggered, an instance of `PathAction` needs to be added to the action list of an action list player, then the player needs to be played.

Unlike with `DeltaPathAction`, the object is moved to an absolute position, not relative to the current position. This action causes an abrupt change in position, unlike `InterpolateAction`, which causes a smooth change.

Creating and Initializing a New Instance

The following script is an example of how to create a new instance of the `PathAction` class:

```
myAction := new PathAction \
              destPosition:(new Point x:50 y:50) \
              targetNum:2 \
              time:60
```

The variable `myAction` holds an initialized instance of `PathAction`. This instance specifies that, at the player's time of 60 ticks, the player's second target (`targetNum:2`) moves to the point 50, 50. The `new` method uses the keywords defined in `init`.

init

`init` *self* [`destPosition:`*point*] [`targetNum:`*integer*] [`time:`*integer*] ⇨ *(none)*

self	`PathAction` object
`destPosition:`	`Point` object representing the position for the target object to move to

Superclass `Action` uses the following keywords:

`targetNum:`	`Integer` object indicating which object in the target list of the player to move
`time:`	`Integer` object representing the time in ticks to trigger the action

Initializes the `PathAction` object *self*, applying its arguments to the instance variables of the same name. At the time set by `time`, this action will cause the target set by `targetNum` to move to the position set by `destPosition`. Do not call `init` directly on an instance—it is automatically called by the `new` method.

If you omit an optional keyword, its default value is used. The defaults are:

```
targetNum:0
time:0
destPosition:undefined
```

Instance Variables

Inherited from `Action`:

`authorData` `targetNum` `time`
`playOnly`

The following instance variables are defined in `PathAction`:

destPosition

self`.destPosition` (read-write) `Point`

Specifies the destination position of the target object when the path action *self* is triggered.

Instance Methods

Inherited from `Action`:

`trigger`

The following instance method is defined in `PathAction`:

trigger (Action)

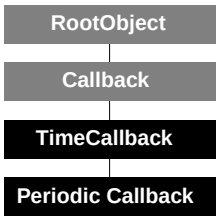
`trigger` *self target player* ⇨ *target*

self	`PathAction` object
target	Any object
player	`ActionListPlayer` object

Causes the path action *self* to move the presenter specified by *target* to the x,y coordinates specified by the `destPosition` instance variable. This method is called by an `ActionListPlayer` object at the time specified by the `time` instance variable. This action list player is passed in as the third argument *player*.

When the action list player calls trigger, the value for *target* is automatically determined by taking the `targetNum` instance variable and finding the object in the corresponding slot in the player's target list. If the `targetNum` instance variable is out of range (that is, less than 1 or greater than the size of the target list), the value of the *target* is `empty`. Also, if there is no *target* in a certain slot in the target list yet, the value of *target* is `undefined`.

PeriodicCallback



Class type: Core class (concrete)
Resides in: ScriptX and KMP executables
Inherits from: `TimeCallback`
Component: Clocks

The `PeriodicCallback` class is used to perform actions when the clock reaches certain time intervals.

You never create an instance of `PeriodicCallback` directly. Instead, you request a callback from a clock using the `addPeriodicCallback` method defined by the `Clock` class.

Instance Variables

Inherited from `Callback`:

`authorData`	`onceOnly`	`script`
`condition`	`order`	`target`
`label`	`priority`	

Inherited from `TimeCallback`:

`condition`	`timeIsTicks`

The following instance variables are defined in `PeriodicCallback`:

condition (Callback)

self`.condition` (read-write) `NameClass`

This instance variable, defined by `Callback`, determines the condition under which the callback will be activated. The following are valid values for `PeriodicCallback` objects:

`@forward`
`@backward`
`@either` (default)

skipIfLate

self`.skipIfLate` (read-write) `Boolean`

Determines whether the script function associated with the `PeriodicCallback` object *self* should be called even when the associated clock has passed the appropriate time period. When `true`, the function won't be called if it is behind schedule; instead, it will be called at the next appropriate time. When `false`, the callback will attempt to "catch up" when it falls behind, calling its function repeatedly until it does so.

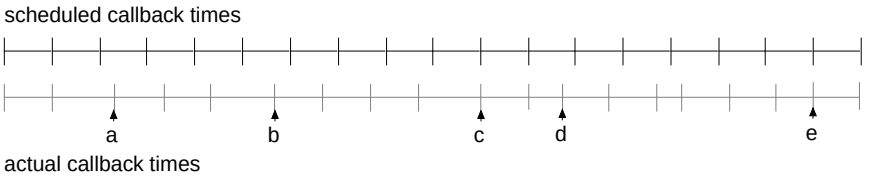


Figure 4-24: Effect of setting skipIfLate instance variable to false

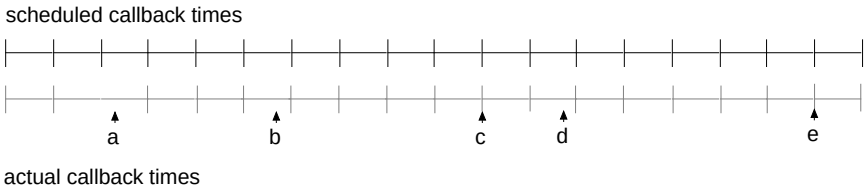


Figure 4-25: Effect of setting skipIfLate instance variable to true

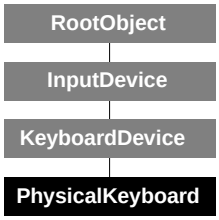
Note that when this variable is set to `false`, the callback may never catch up, which means the callback may become continually out of synch with its clock. By default, the value of this instance variable is `false`.

Instance Methods

Inherited from `Callback`:

`cancel`

PhysicalKeyboard



Class type: Core class (concrete)
Resides in: ScriptX and KMP executables
Inherits from: `KeyboardDevice`
Component: Input Devices

The `PhysicalKeyboard` class communicates with the keyboard driver, which in turn receives events from the actual hardware keyboard device.

Creating and Initializing a New Instance

As a rule, developers should create an instance of `PhysicalKeyboard` by calling `new` on the `KeyboardDevice` class. The `KeyboardDevice` class checks automatically for a hardware device attached to the system.

The following script is an example of how to create a new instance of the `PhysicalKeyboard` class:

```
myKeyboard := new PhysicalKeyboard \
                autoRepeat:true
```

The variable `myKeyboard` contains the initialized instance of `PhysicalKeyboard`. If the attached hardware keyboard supports it, `autoRepeat` will be set to `true`. The `new` method uses the keywords defined in `init`.

init

`init` *self* [`autoRepeat:`*boolean*] [`deviceID:`*integer*] [`enabled:`*boolean*]

⇨ *(none)*

self	`PhysicalKeyboard` object
`autoRepeat:`	`Boolean` object

Superclasses of `PhysicalKeyboard` use the following keywords:

`deviceID:`	`Integer` object
`enabled:`	`Boolean` object

Initializes the `PhysicalKeyboard` object *self*, applying the arguments as follows: `autoRepeat` enables or disables hardware auto-repeat keys, `deviceID` is used to choose the device, and `enabled` activates the device if it is set to `true`. This method also adds the new instance to a private list kept by `KeyboardDevice`. Each subclass of `InputDevice` maintains its own list of devices, so the device ID is unique only for that subclass. Since the current version of ScriptX supports one physical keyboard, this keyword is generally set to its default value. Do not call `init` directly on an instance—it is automatically called by the `new` method.

If you omit an optional keyword, its default value is used. The defaults are:

```
autoRepeat:false
deviceID:1
enabled:true
```

Class Methods

Inherited from `InputDevice`:

`getDeviceFromList`

Instance Variables

Inherited from `InputDevice`:

`deviceID` `focusable` `focusManager`
`enabled`

Inherited from `KeyboardDevice`:

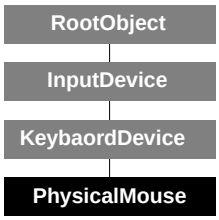
`autoRepeat` `focusManager` `keyModifiers`

Instance Methods

Inherited from `KeyboardDevice`:

`existKey` `getKeyString` `isModifierInactive`
`getKeyName` `isModifierActive`

PhysicalMouse



Class type: Core class (concrete)
Resides in: ScriptX and KMP executables
Inherits from: `MouseDevice`
Component: Input Devices

The `PhysicalMouse` class communicates with the mouse driver, which in turn receives events from the actual hardware mouse device. ScriptX supports three mouse buttons. Since many hardware mice do not actually have three physical buttons, the remaining buttons can be implemented by using a modifier key (such as **Shift**, **Alt**, **Ctrl**, **Option**, or **Command**) in combination with a physical mouse button. For example, given a one-button mouse, extra "buttons" might be defined as holding down one or more modifier keys while pressing the first mouse button.

Creating and Initializing a New Instance

As a rule, developers should create an instance of `PhysicalMouse` by calling `new` on the `MouseDevice` class. The `MouseDevice` class checks automatically for a hardware device attached to the system.

The following script is an example of how to create a new instance of the `PhysicalMouse` class.

```
mighty := new PhysicalMouse \
          threshold:(new Point x:2 y:2)
```

The variable `mighty` contains the initialized `PhysicalMouse` object. Its threshold is set to be sensitive to moves of at least 2 pixels in either direction. The `new` method uses the keywords defined in `init`.

init

`init` *self* [`threshold:`*point*] [`deviceID:`*integer*] [`enabled:`*boolean*] ⇨ *(none)*

self	`PhysicalMouse` object
`threshold:`	`Point` object

Superclass `MouseDevice` uses the following keywords:

`deviceID:`	`Integer` object
`enabled:`	`Boolean` object

Initializes the `PhysicalMouse` object *self*, applying the arguments as follows: The `threshold` determines device sensitivity when the mouse is moved. The `deviceID` is an integer ID number that is unique to a particular subclass of `InputDevice`; it allows a title to request a particular mouse if more than one mouse is attached to the system. The `enabled` keyword activates the device if set to `true`. This method also adds the new instance to a private list kept by `MouseDevice`. Do not call `init` directly on an instance—it is automatically called by the `new` method.

If you omit an optional keyword, its default value is used. The defaults are:

```
threshold:(new Point x:1 y:1)
```

```
deviceID:1
enabled:true
```

If you create a subclass of `PhysicalMouse` that overrides the `init` method, be sure to invoke its superclass's `init` method using the `nextMethod` call.

Note – Authors are encouraged to call `new` on `MouseDevice` rather than on `PhysicalMouse`. Read the description of `new` on the `MouseDevice` class to see if that method applies.

Class Methods

Inherited from `InputDevice`:

`getDeviceFromList`

Instance Variables

Inherited from `InputDevice`:

`deviceID`	`focusable`	`focusManager`
`enabled`		

Inherited from `MouseDevice`:

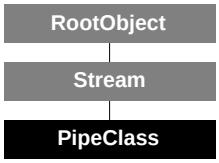
`buttons`	`keyModifiers`	`threshold`
`currentCoords`	`pointerType`	

Instance Methods

Inherited from `MouseDevice`:

`isButtonDown`	`isModifierActive`	`isModifierInactive`
`isButtonUp`		

PipeClass



Class type: Core class (concrete)
Resides in: ScriptX and KMP executables
Inherits from: `Stream`
Component: Threads

The `PipeClass` class provides a convenient way for threads to pass objects around to each other. A pipe is a conduit for communication between processes, often used as a synchronization device. `PipeClass` is one of two classes of pipes in the ScriptX core classes; the other class, `BytePipe`, is used for passing streams of data between threads.

Pipes use gates internally—a thread suspends or blocks execution while waiting on a pipe. The thread system blocks an active thread that tries to read from a pipe that is empty, or write to one that is full. The value of the `status` instance variable on the suspended thread is `@waiting`. If a `PipeClass` object is unbounded in size, it will never cause a thread to suspend on writing.

`PipeClass` objects may be "broken." Breaking a pipe means that writes are no longer allowed. If a script attempts to write to a broken pipe, it reports the `brokenPipe` exception. A reading from an empty and broken pipe also reports the `brokenPipe` exception. ScriptX provides several methods to determine the size and integrity of a pipe.

`PipeClass` objects may have a reading thread associated with them. If this thread is defined, the pipe will call `threadActivate` on it whenever new objects are written to the pipe (`threadActivate` is a method on the `Thread` class). ScriptX reports the `threadProhibited` exception if the thread is not runnable—for example, if the thread is waiting on a pipe or gate. This technique is used internally in the ScriptX core classes by the `EventQueue` class, a subclass of `PipeClass`.

When an instance of `PipeClass` is instantiated, ScriptX automatically creates a sequence that will store the queue of objects in the pipe. In the current version of ScriptX, this sequence is an `Array`. It is possible to directly manipulate this queue. In the following example, the `acquireQueue` method returns a reference to this sequence, allowing a script to examine and perhaps rearrange or modify objects in the pipe. This method obtains a lock, so that no other thread can read from or write to the pipe. A script must explicitly relinquish the queue back to the pipe before it can be processed further:

```
theSequence := acquireQueue thePipe
-- operations on theSequence
relinquishQueue thePipe
```

`PipeClass` objects implement the Stream protocol. Since they are non-seekable `Stream` objects, the exception `cantSeek` is reported when a script invokes a method that is implemented only for seekable streams. For additional information, see the class definition of `Stream`.

Creating and Initializing a New Instance

The following script is an example of how to create a new instance of the `PipeClass` class:

```
myPipe := new PipeClass \
              maxSize:10 \
              thread:myThread \
              label:"pipe numero uno"
```

The variable `myPipe` contains the initialized `PipeClass` object, which has a maximum size of 10 and is attached to `myThread`. The `new` method uses the keywords defined in `init`.

init

`init` *self* [`maxSize:`*integer*] [`thread:`*thread*] [`label:`*object*] ⇨ *(none)*

self	`PipeClass` object
`maxSize:`	`Integer` object representing maximum number of objects
`thread:`	`Thread` object that can read from the pipe
`label:`	Any object, used for debugging

Initializes the `PipeClass` object *self*, applying the values supplied with the keywords to instance variables of the same name. Do not call `init` directly on an instance—it is automatically called by the `new` method.

If you omit an optional keyword, its default value is used. The defaults are:

```
label:undefined
maxSize:posInf
thread:undefined
```

Instance Variables

broken

self`.broken` (read-only) `Boolean`

If the value of `broken` is `true`, the pipe has been explicitly broken.

label

self`.label` (read-write) *(object)*

Specifies a label for `PipeClass` object *self*. This label can be any object; it is displayed when you print the object, and is useful for debugging.

maxSize

self`.maxSize` (read-only) `Integer`

The maximum allowed number of objects in the `PipeClass` object *self*, or `0` if *self* is unbounded in size.

size

self`.size` (read-only) `Integer`

The current size, or number of items, in the `PipeClass` object *self*.

thread

self`.thread` (read-write) `Thread`

A thread to activate when the `PipeClass` object *self* is not empty. Can be used to read objects from the thread.

Instance Methods

Inherited from `Stream`:

`cursor`	`next`	`seekFromStart`
`flush`	`plug`	`setStreamLength`
`isAtFront`	`previous`	`streamLength`
`isPastEnd`	`read`	`write`
`isReadable`	`readReady`	`writeReady`
`isSeekable`	`seekFromCursor`	
`isWritable`	`seekFromEnd`	

The following instance methods are defined in `PipeClass`:

acquireQueue

`acquireQueue` *self* ⇨ `Sequence`

Used to gain low-level access to the queue of objects, a `Sequence` object, in the `PipeClass` object *self*. Calling `acquireQueue` on a pipe blocks other threads that attempt to read from or write to the pipe until `relinquishQueue` is invoked. A thread that acquires this queue owns a lock; it is able to read from and write to the pipe, as well as manipulate the queue directly.

breakPipe

`breakPipe` *self* ⇨ *(none)*

Breaks the `PipeClass` object *self*.

isReadable (Stream)

`isReadable` *self* ⇨ `true`

Returns `true`, since a `PipeClass` object is always a readable stream. Though it is always readable, there are not necessarily objects in the pipe to be read. Call the method `readReady` to determine whether or not there are objects in the pipe to be read.

isSeekable (Stream)

`isSeekable` *self* ⇨ `false`

Returns `false`, since a `PipeClass` object is always an unseekable stream. For a discussion of seekable streams, see the class definition of `Stream`.

isWritable (Stream)

`isWritable` *self* ⇨ `true`

Returns `true`, since a `PipeClass` object is always a writable stream. Although it is always writable, it may already be full, or it may be broken. Call the method `writeReady` to determine whether or not an object can be written to the pipe in its current state.

isPastEnd (Stream)

`isPastEnd` *self* ⇨ `Boolean`

Returns `true` only when the `pipe` is broken and empty.

pipeSize

`pipeSize` *self* ⇨ `Integer`

Returns the number of items waiting to be read inside the pipe *self*. Since this method acquires a lock on a pipe, it can potentially block.

pipeSizeOrFail

`pipeSizeOrFail` *self* ⇨ `Integer` or `empty`

Attempts to determine the number of objects in the pipe *self*. If it is able to determine the size, it returns an integer value. It may return `empty` if it cannot determine the size, as when a `RegularThread` object has a lock on the pipe.

read (Stream)

`read` *self* ⇨ *(object)*

Reads one object from the pipe *self* and returns that object.

readNowOrFail

`readNowOrFail` *self* ⇨ *(object)*

Attempts to read from the pipe *self*. If it reads without suspending, it returns the value read; otherwise, it returns `empty`.

readReady (Stream)

`readReady` *self* ⇨ `Integer`

Returns the number of objects that are ready to be read from the pipe *self*.

relinquishQueue

`relinquishQueue` *self* ⇨ *(none)*

Relinquishes low-level access to the queue of objects, a `Sequence` object, in the pipe *self*. This method balances a call to `acquireQueue`.

write (Stream)

`write` *self value* ⇨ `Boolean`

self	`PipeClass` object
value	Any object

Writes the object *value* to the pipe *self*; returns `true` if it succeeds.

writeNowOrFail

`writeNowOrFail` *self value* ⇨ `Boolean`

self	`PipeClass` object
value	Any object

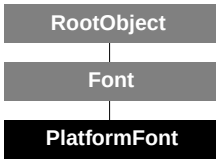
Attempts to write the object *value* to the pipe *self*. If it writes successfully, it returns `true`; otherwise, it returns `false`. This operation is for processes that cannot block while writing to a pipe.

writeReady (Stream)

`writeReady` *self* ⇨ `Integer`

Returns the number of objects that are ready to be written to the pipe *self*.

PlatformFont



Class type: Core class (concrete)
Resides in: ScriptX and KMP executables
Inherits from: `Font`
Component: Text and Fonts

The `PlatformFont` class provides an object interface to the underlying font technology and available fonts on multiple platforms. `PlatformFont` objects specify the font name (typeface) only; font variations such as bold or italic are specified through attributes to `Text` or `TextPresenter` objects.

Creating and Initializing a New Instance

The following script is an example of how to create a new instance of the `PlatformFont` class:

```
myFont := new PlatformFont \
              macintoshName:"Helvetica" \
              windowsName:"Arial" \
              os2Name:"Helvetica"
```

The `myFont` variable contains the initialized font. The font that is used differs depending on the platform ScriptX is currently running on (Helvetica for Macintosh, Arial for Microsoft Windows, or Helvetica for OS/2). The `new` method uses keywords defined by the `init` method.

Note – If the keyword `os2Name` has not yet been implemented, you can use just the keywords `name`, `macintoshName`, and `windowsName`. If you do not specify a font name for the `name` keyword, you must supply font names for both `macintoshName` and `windowsName`. If you do use the `name` keyword, the other two keywords are optional.

---

init

`init` *self* `name:`*string* [`macintoshName:`*string*] [`windowsName:`*string*]
[`os2Name:`*string*] ⇨ *(none)*

`init` *self* [`name:`*string*] `macintoshName:`*string* `windowsName::`*string*
`os2Name:`*string* ⇨ *(none)*

self	`PlatformFont` object
`name:`	`String` object representing the name of a typeface
`macintoshName:`	`String` object representing the name of a typeface on the Macintosh platform
`windowsName:`	`String` object representing the name of a typeface on the Windows platform
`os2Name:`	`String` object representing the name of a typeface on the OS/2 platform

Initializes the PlatformFont object *self*, using the typeface specified by name when that is the only parameter supplied. If macintoshName, windowsName, or OS2Name is supplied in addition to name, any one of them takes precedence over name. Only name may be specified by itself. You can use multiple keywords to avoid differences in font names across platforms.

Do not call init directly on an instance—it is automatically called by the new method.

If you omit an optional keyword, its default is used. The following are the default fonts for the platforms listed:

```
macintoshName:"Geneva" (for U.S. versions; for other countries, it is the
                        application font, as opposed to the system font)
windowsName:"Arial"
os2Name:"Helvetica"
```

Class Variables

Inherited from Font:

default

Instance Variables

The following instance variables are defined in PlatformFont:

name

self.name (read-only) String

The name of the current font as a String object.

macintoshName

self.macintoshName (read-only) String

The name of the font, as specified by the macintoshName keyword to the init method. If the macintoshName keyword was not specified, then *self*.macintoshName has the same value as *self*.name.

windowsName

self.windowsName (read-only) String

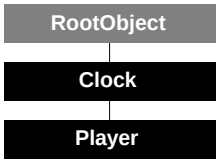
The name of the font, as specified by the windowsName keyword to the init method. If the windowsName keyword was not specified, then *self*.windowsName has the same value as *self*.name.

os2Name

self.os2Name (read-only) String

The name of the font, as specified by the os2Name keyword to the init method. If the os2Name keyword was not specified, then *self*.os2Name has the same value as *self*.name.

Player



Class type: Core class (concrete)
Resides in: ScriptX and KMP executables
Inherits from: `Clock`
Component: Players

The `Player` class provides basic methods for controlling any media that can be "played." Such media may include media streams (such as sound and video files), external devices (such as CD-Rom devices), or any time-based presentation (such as animations or transitions). The methods defined in the `Player` class include `play`, `stop`, `fastForward`, `rewind`, and so on.

Methods to control specific media or animation types are implemented in subclasses of `Player`. For example, the class `DigitalAudioPlayer` defines methods for controlling audio volume.

Since the `Player` class inherits from `Clock`, instances of `Player` and its subclasses can be arranged in a timing hierarchy. A timing hierarchy consists of players that are synchronized together so that one player can be used to control multiple players. When one player acts as a controller for another, it is said to be the "master player," and the players it controls are its "slave players." For example, a single player could be used to control the timing of one player that plays a sound track and another that plays an animation. (See the "Clocks" chapter in the *ScriptX Components Guide* for more on timing hierarchies).

All subclasses of `Player` should support the ability to use any player as a master player. For newly defined subclasses of `Player`, ensure that each method that overrides a `Player` method maintains correct interaction within the timing hierarchy by invoking the superclass's method.

The class `Player` has instance variables, such as `globalVolumeOffset`, for parameters that affect all audio and video players in a timing hierarchy. Such parameters are referred to as "global parameters." You can control the effects of all players in a timing hierarchy by changing global parameters on the master player. For example you could make all audio players in a timing hierarchy play more quietly by changing the value of the `globalVolumeOffset` instance variable on the master player.

The value of an instance variable for a global parameter (such as `globalVolumeOffset`) is combined with the value of the instance variable for the local parameter (such as `volume`) to produce the actual value for a parameter. For example, the volume at which a `DigitalAudioPlayer` actually plays is determined by combining the value of the `volume` instance variable (defined on the class `DigitalAudioPlayer`) and the `globalVolumeOffset` instance variable (defined on the class `Player`).

When you change a global parameter on a master player, the change affects any slave players to which the change is relevant. If a slave player cannot respond to the change, it ignores it.

In the current release of ScriptX, the instance variables that affect visual global attributes of players, such as `globalBrightness` and `globalContrast`, are not hooked up. For example, if you change the value of the `globalBrightness` instance variable of a player, there will be no change in the brightness of the media presented by the player or its slave players.

Creating Player Subclasses

When creating subclasses of `Player`, you should not redefine the methods that change the rate of the player, such as `play`, `stop`, `pause`, `fastforward` and `rewind`. Instead, if the new player class needs to specialize the way it presents its data, define the `effectiveRateChanged` method to check the `effectiveRate` instance variable and take the appropriate action. Also, if the player distinguishes between the stopped and paused state, then when the effective rate is 0 the `effectiveRateChanged` method should also check the `status` instance variable to find out if the player is stopped or paused.

Also do not redefine methods that change the time of the player, such as `gotoBegin` and `gotoEnd`. Instead, redefine the `timeJumped` method inherited from the class `Clock`.

See the section on the `Clock` class for more information about the methods `effectiveRateChanged` and `timeJumped`. See also the "Clocks" chapter in the *ScriptX Components Guide*.

Creating and Initializing a New Instance

To create a player that can play a media or sequence, create an instance of the appropriate subclass. For example, to create a player to play audio, create an instance of `DigitalAudioPlayer`, or to create a player to play an animation create an instance of `ActionListPlayer`.

If you want to use a player solely as a controller of other players, then create an instance of the class `Player`.

The following script is an example of how to create a new instance of the `Player` class:

```
myPlayer := new Player \
              masterClock:clock1 \
              scale:1
```

In this example, the variable `myPlayer` points to an instance of `Player` whose master clock is `clock1` and whose scale is 1.

To specify this player as a controller for another player, you would put it as the value of the `masterClock` instance variable on the player (or players) it is to control. The `masterClock` instance variable is inherited from the class `Clock`.

The `new` method uses the keywords defined in `init`.

init

`init` *self* [`masterClock:`*clock*] [`scale:`*number*] [`title:`*titleContainer*] ⇨ *(none)*

self	`Player` object

The following keywords are used by superclasses of `Player`:

`masterClock:`	`Clock` object to use as the master clock.
`scale:`	`Number` object to use as the scale for the clock. If you give the value as a non-integer number, the value is rounded down to the nearest integer.
`title:`	`TitleContainer` object to which to add the player.

Initializes the `Player` object *self*, putting the `masterClock` as the master clock of this player and setting the scale of the clock to the given scale value. The player is added to the specified `title`. Do not call `init` directly on an instance—it is automatically called by the `new` method.

If you omit an optional keyword, its default value is used. The defaults are:

```
masterClock:undefined
scale:1
title:theScratchTitle
```

Instance Variables

Inherited from `Clock`:

`callbacks`	`rate`	`ticks`
`effectiveRate`	`resolution`	`time`
`masterClock`	`scale`	`title`
`offset`	`slaveClocks`	

The following instance variables are defined in `Player`:

audioMuted

self`.audioMuted` (read-write) `Boolean`

Controls whether or not the player *self* and its slave players currently play sound (if they have sound capability). If the value is `true`, then no sound is played (though the player's `time` instance variable continues to increment its value indicating that time continues to tick). If the value is `false`, then sound is played normally. Note that the setting of this variable affects only sound media presented by a player and its slave players.

If the player is a master player, the value of this instance variables affects its slave players too.

dataRate

self`.dataRate` (read-only) `Integer`

Specifies the number of bytes per second that the player *self* needs to read in order not to skip presenting some of its media.

duration

self`.duration` (read-only) `Integer`

Gets the duration of the media or sequence played by the player *self*. A duration of `-1` implies that the media or sequence has an indeterminate duration.

globalBrightness

self`.globalBrightness` (read-write) `Fixed`

Specifies the global brightness setting of the player *self* and its slave players. It is used in presenting visual media and is expressed as a percentage of the local brightness setting. This setting affects only external devices such as VCRs. (See the `Player` class introduction for more on global parameters.) Currently this instance variable is not used, so changing its value for any kind of players has no effect.

globalContrast

self.globalContrast (read-write) Fixed

Specifies the global contrast setting of the player *self* and its slave players. It is used in presenting visual media and is expressed as a percentage of the local contrast setting. This setting affects only external devices such as VCRs. (See the Player class introduction for more on global parameters). Currently this instance variable is not used, so changing its value for any kind of players has no effect.

globalHue

self.globalHue (read-write) Fixed

Specifies the global hue setting of the player *self* and its slave players. It is used in presenting visual media and is expressed as a percentage of the local hue setting. This setting affects only external devices such as VCRs. (See the Player class introduction for more on global parameters). Currently this instance variable is not used, so changing its value for any kind of players has no effect.

globalPanOffset

self.globalPanOffset (read-write) Fixed

Specifies an adjustment to the placement of the sound on the sound state of the player *self* and its slave players. The actual pan value used to present sound to the user is determined by adding the global pan offset value to the local pan value. The local pan value is specified by the pan instance variable, which is defined on subclasses of Player that play sound, such as DigitalAudioPlayer. A slave player gets the value for globalPanOffset from its master player.

The value for globalPanOffset can be a fixed number between 0.0 and 1.0 inclusive, where 0 is far left and 1.0 is far right.

The globalPanOffset instance variable only affects players in a timing hierarchy that have a volume instance variable.

globalSaturation

self.globalSaturation (read-write) Fixed

Specifies the global saturation setting of the player *self* and its slave players. It is used in presenting visual media and is expressed as a percentage of the local saturation setting. This setting affects only external devices such as VCRs. (See the Player class introduction for more on global parameters). Currently this instance variable is not used, so changing its value for any kind of players has no effect.

globalVolumeOffset

self.globalVolumeOffset (read-write) Fixed

Specifies the global volume setting of the player *self* and its slave players, expressed as an offset in dB (decibels) from the local volume setting.

A player determines the actual volume for playing sound by adding the global volume offset value to the local volume value. For example, if the local volume of a player is -6 and the global volume offset is 6, the effective volume is 0.

The local volume value is specified by the volume instance variable, which is defined on subclasses of Player that play sound, such as DigitalAudioPlayer. A slave player gets the value for its globalVolumeOffset instance variable from its master player.

The globalVolumeOffset instance variable only affects instances of classes that have a volume instance variable, which currently is the DigitialAudioPlayer class.

markerList

self.markerList (read-write) SortedKeyedArray

Specifies the array of markers associated with the player *self*.

The media stream being played may also have markers, but the player's marker list does not include markers in the media stream.

To add a marker to the player, use the player's addMarker method, which in turn modifies the array held in the markerList instance variable. The array is keyed by the start times of the markers.

Until a marker is explicitly added to a player by calling the player's addMarker method, the contents of this variable is undefined.

status

self.status (read-only) NameClass

Specifies the current status of the Player object *self*. By examining the current status and the values of videoBlanked and audioMuted, the author can determine just what the player is doing at any given time.

The status can be any of the following:

@playForward	Rate = 1.0
@stop	Stopped, not ready (Rate = 0)
@pause	Stopped, ready to play (Rate = 0)
@playReverse	Rate = -1.0
@slowReverse	Rates between –1.0 and 0
@fastReverse	Rate < -1
@slowForward	Rates between 0 and 1.0
@fastForward	Rate > 1.0

videoBlanked

self.videoBlanked (read-write) Boolean

Determines whether or not the image presented by a player *self* (such as a VideoPlayer) and its slave players is visible or not. If videoBlanked is true, then no image is presented (though the player's time instance variable continues to increment its value indicating that time continues to tick). If false, then an image is presented normally. Note that videoBlanked affects visual media presented by this player and its slaves.

If the player is a master player, the value of this instance variables affects its slave players too.

Instance Methods

Inherited from Clock:

addPeriodicCallback	clockAdded	pause
addRateCallback	clockRemoved	resume
addScaleCallback	effectiveRateChanged	timeJumped
addTimeCallback	forEachSlave	waitTime
addTimeJumpCallback	isAppropriateClock	waitUntil

The following instance methods are defined in Player:

addMarker

`addMarker` *self marker* ⇨ `Marker`

self	`Player` object
marker	`Marker` object

Adds *marker* to the player, *self*. The marker is added to the array of markers stored in the player's `markerList` instance variable. This method returns the given marker if the addition is successful, `undefined` if it is not.

eject

`eject` *self* ⇨ `Boolean`

Ejects the media from the player *self*. This method is only useful to players that correspond to physical devices, such the loadable class `CDPlayer` that can be used to play an audio compact disc in a CD-Rom drive.

fastForward

`fastForward` *self* ⇨ `Boolean`

Fast forwards the media or sequence associated with the player *self*. If the media is playing at the time, the player attempts to present media as it fast forwards. In fast-forward, the `rate` of the player is set to `5`. If the player *self* is a master player, this method also calls `fastForward` on its slave players.

getMarker

`getMarker` *self time markerID* ⇨ `Marker`

self	`Player` object
time	`Number` object that indicates the number of ticks in the player's scale.
markerID	`NameClass` object. The value must be one of `@inMarker`, `@prevMarker`, or `@nextmarker`.

Returns the marker of the player *self* that encloses or is closest to the given *time*. The *markerID* parameter can take one of three values: `@inMarker`, `@prevMarker` or `@nextMarker`.

Each marker marks a time range. If one or more markers on the player contain the given time, then the marker whose start time is closest to the given time is returned, regardless of the *markerID*. If no marker contains the given time, the returned value depends on the value of *markerID*.

The following list summarizes the results of `getMarker` according to the value of *markerID*:

- If *markerID* is `@inMarker`, and if one or more markers on the player contain the given time, then the marker whose start time is closest to the given time is returned. If no marker contains the given time, this method returns `undefined`.
- If *markerID* is `@prevMarker`, if one or more markers on the player contain the given time, then the marker whose start time is closest to the given time is returned. If no marker contains the given time, this method returns the previous marker whose start time is closest to that time or `undefined` if there is no previous marker.
- If *markerID* is `@nextMarker`, and if one or more markers on the player contain the given time, then the marker whose start time is closest to the given time is returned. If no marker contains the given time, this method returns the next marker after that time. If there is no next marker, this method returns the marker with the latest start time, or `undefined` if the player has no markers at all.

getNextMarker

`getNextMarker` *self marker* ⇨ `Marker`

self	`Player` object
marker	`Marker` object

Returns the marker following *marker* in the player *self*'s marker list. This method returns `undefined` if there is no marker following *marker*. (The markers in the list are sorted by start time.)

getPreviousMarker

`getPreviousMarker` *self marker* ⇨ `Marker`

self	`Player` object
marker	`Marker` object

Returns the marker before *marker* in the `Player` object *self*'s marker list. This method returns `undefined` if there is no marker before *marker*. (The markers in the list are sorted by time.)

goToBegin

`goToBegin` *self* ⇨ `Integer`

Positions the player *self* at the beginning of the media it is playing. When the `gotoBegin` method is called on a player, its `time` instance variable is set to 0, which corresponds to the beginning of the player's media or sequence.

If the player *self* is a master player, this method also sets the master's slave players back to the beginning.

This method returns the tick count corresponding to the beginning of the player's media or sequence, which is 0.

goToEnd

`goToEnd` *self* ⇨ `Integer`

Positions the player *self* at the end of the media it is playing. When the `gotoEnd` method is called on a player, its `time` instance variable is set to the value of the player's `duration` instance variable. (The value of the `duration` instance variable of a player indicates the length in time of the media or sequence played by the player.)

If the player *self* is a master player, this method also sets the slave players to the end of their media.

goToMarkerFinish

`goToMarkerFinish` *self marker* ⇨ `Integer`

self	`Player` object
marker	`Marker` object

Positions the player *self* at the time indicated by the finish time of the *marker*. (A marker holds its finish time in ticks in its `finish` instance variable.)

More specifically, this method sets the `ticks` instance variable of the player *self* to the value specified by the `finish` instance variable of the marker. The method also sets the player's `time` instance variable to the corresponding finish time. It returns the value in the marker's `finish` instance variable.

If the player *self* is a master player, this method also sets the time of its slave players to the time indicated by the finish time of the given marker, taking offsets into account.

goToMarkerStart

`goToMarkerStart` *self marker* ⇨ `Integer`

self	`Player` object
marker	`Marker` object

Positions the player *self* at the time indicated by the start time of *marker*. A marker holds its start time in ticks in its `start` instance variable.

More specifically, this method sets the `ticks` instance variable of the player *self* to the value specified by the `start` instance variable to the marker. It also sets the player's `time` instance variable to the corresponding start time. The method returns the value in the marker's `start` instance variable.

If the player *self* is a master player, this method also sets the time of its slave players to the time indicated by the start time of the given marker, taking offsets into account.

pause

`pause` *self* ⇨ `Boolean`

Temporarily stops the player *self* from playing its associated media or sequence. By default, this is identical to `stop`, but some `Player` subclasses may be able to take actions within this method that will make continuing from the paused state less time-consuming than coming to a full stop, then restarting.

If the player *self* is a master player, also pauses its slave players.

play

`play` *self* ⇨ `Boolean`

Begins playing the media or sequence associated with the player *self*.

This method sets the rate to 1. If the player *self* is a master player, this method also starts its slave players playing.

playPrepare

`playPrepare` *self nextRate* ⇨ `Boolean`

self	`Player` object to prepare for playing
nextRate	`Number` representing the next rate for the player

Prepares the player *self* to play its media. In media-production terms, this may be thought of as "pre-roll." Once this method has been invoked, its `play` and `stop` methods may be called at will.

The *nextRate* argument provides a hint about the rate at which the player will play once it starts. A rate of 1 means play forward at normal speed. This hint allows the player to adjust how it pre-rolls the media to prepare it appropriately for the specified rate. However, the player's rate is set explicitly by calling its `play` method (which always sets the rate to 1) or by setting its `rate` instance variable.

Usually you call `playPrepare` with a *nextRate* argument of 1, then you call `play` to start the player playing. If for some reason you want it to play at a rate other than 1, call `playPrepare` with the desired *nextRate* argument, then instead of calling the `play` method, set the `rate` instance variable directly to *nextRate*.

This method may be overridden in a subclass to fill buffers, initialize hardware, locate codecs, and to perform any other initialization that takes time or acquires resources required by a player or its stream before starting play.

If the player *self* is a master player, this method also prepares the slave players for playing.

playUnprepare

`playUnprepare` *self* ⇨ `Boolean`

Releases the resources that player *self* needs to play its associated media or sequence.

Subclasses may override this method to free hardware, buffers, or other resources the player may have allocated with `playPrepare`.

Before calling `playUnprepare`, you should call the player's `stop` method. Once `playUnprepare` has been called, `playPrepare` must be called again before the player can be played.

If the player *self* is a master player, this method also unprepares the slave players.

Note that you should call `playUnprepare` only when you are ready to get rid of the player. It is not necessary to call `playUnprepare` once for every time you call `playPrepare`; in fact, it is much less efficient to do so. This is true because `playPrepare` allocates buffers and then pre-rolls media into those buffers. If a buffer has already been allocated, `playPrepare` just pre-rolls media into it without having to allocate a new one. The method `playUnprepare` deallocates the buffers which `playPrepare` has allocated. Each time you call `playPrepare` after having called `playUnprepare`, new buffers have to be allocated. If you call `playUnprepare` only at the end, however, you avoid unnecessary deallocation and reallocation of buffers.

playUntil

`playUntil` *self timeValue* ⇨ `Boolean`

self	`Player` object to stop
timeValue	`Number` of ticks to stop playing at

Plays the media or sequence associated with the player *self* until the given *timeValue* is reached, then stops.

If the player *self* is a master player, this method also plays the slave players.

resume

`resume` *self* ⇨ `Boolean`

Resumes the player *self* playing after it has been temporarily paused. The `resume` method must be called as many times as `pause` was called before the player will resume playing.

If the player *self* is a master player, also resumes its slave players.

rewind

`rewind` *self* ⇨ `Boolean`

Rewinds the media or sequence associated with the player *self*. If the player is playing at the time, the player should attempt to present media as it rewinds. `Rewind` sets the `rate` instance variable of the player to `-5`.

If the player *self* is a master player, this method also rewinds its slave players.

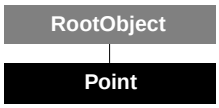
stop

`stop` *self* ⇨ `Boolean`

Stops playing the media or sequence associated with the player *self*.

If the player *self* is a master player, this method also calls stops its slave players.

Point



Class type: Core class (concrete, sealed)
Resides in: ScriptX and KMP executables
Inherits from: `RootObject`
Component: 2D Graphics

The `Point` class defines a single point in a 2D coordinate system. Stroking a point can produce results outside the pixel it represents, but filling a point affects only a single pixel.

Note that `Point` objects are not presenters—nor are they stencils. To display a point, coerce a `Point` instance to a `Region`, and create an instance of `TwoDShape` using the coerced point as the `boundary` argument. For example:

```
myPoint := new Point x:10 y:10
myShape := new twoDShape boundary:(myPoint as Region)
```

Creating and Initializing a New Instance

The following script is an example of how to create a new instance of the `Point` class:

```
aPoint := new Point \
              x:10 \
              y:10
```

The variable `aPoint` contains a new `Point` object positioned at the coordinates 10, 10. The new method uses the keywords defined in `init`.

init

`init` *self* [`x:`*number*] [`y:`*number*] ⇨ *(none)*

self	`Point` object
`x:`	`Number` object
`y:`	`Number` object

Initializes the `Point` object *self*, applying the arguments as follows: `x` sets the x coordinate and `y` sets the y coordinate of the point. Do not call `init` directly on an instance—it is automatically called by the `new` method.

If you omit an optional keyword, its default value is used. The defaults are:

```
x:0
y:0
```

Instance Variables

x

self`.x` (read-write) `Number`

Specifies the x coordinate of the point *self*.

y

self.y (read-write) Number

Specifies the y coordinate of the point *self*.

Instance Methods

copy

copy *self* ⇨ Point

Creates and returns a copy of the point *self*.

invTransform

invTransform *self matrix result* ⇨ Point

self	Point instance
matrix	TwoDMatrix representing the transformation
result	Name representing the desired result: @mutate or @create

Transforms the point *self*, by the inverse of the 2D matrix *matrix*. The value returned depends on the value of *result*.

To inversely transform a point, you first call the appropriate transformation methods on *matrix* and then call this method on the point *self*.

transform

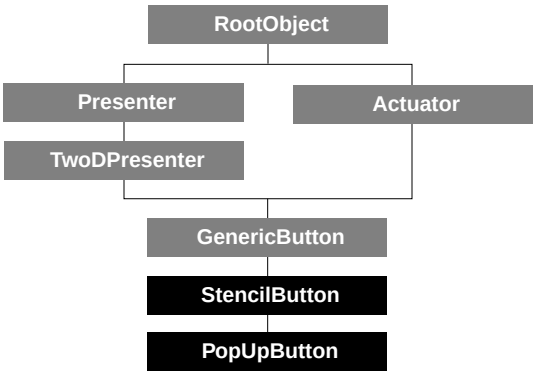
transform *self matrix result* ⇨ Point

self	Point instance
matrix	TwoDMatrix representing the transformation
result	Name representing the desired result: @mutate or @create

Transforms the point *self*, by the 2D matrix *matrix*. The value returned depends on the value of *result*.

To transform a point, you first call the appropriate transformation methods on *matrix* and then call this method on the point *self*.

PopUpButton



Class type: Scripted class (concrete)
Resides in: widgets.sxl. Works with ScriptX and KMP executables
Inherits from: StencilButton
Component: User Interface

PopUpButton is a user interface Widget Kit class that provides a framed button whose appearance is defined by a stencil and a Frame object that gives the button a three dimensional look. A PopUpButton object is similar to a StencilButton object, but PopUpButton takes a menu argument and displays that menu when the PopUpButton is activated with the mouse.

The clipping boundary of a PopUpButton object is calculated automatically as the smallest rectangle that encloses the stencil and the button frame.

Creating and Initializing a New Instance

The following script is an example of how to create a new instance of the PopUpButton class:

```
myPopUpButton := new PopUpButton \
          menu:myPopUpMenu value:@two
```

The variable myPopUpButton contains an initialized PopUpButton object. The new method uses the keywords defined in init.

init

`init` *self* `menu:`*popupMenu* [`boundary:`*stencil*] [`value:`*index*] ⇨ *(none)*

self	PopUpButton object
menu:	PopUpMenu object
boundary:	Stencil object
value:	Collection value (index) object

Initializes the PopUpButton object *self*, applying the values supplied with the keywords to the instance variables of the same name. Supplies a stencil for the button, takes the frame and font for the button from the menu, calculates a boundary for the button if necessary, and displays the menu item selected by value on the button. Do not call init directly on an instance — it is automatically called by the new method.

If you omit the boundary keyword argument, the default dimensions used for the button are the width and height of a single menu item. If you omit the value keyword argument, the first item in the menu is displayed on the button by default.

The PopUpButton class inherits the stencil keyword from StencilButton. The stencil media/PopUp.bmp is automatically applied to a PopUpButton object.

Instance Variables

Inherited from `Actuator`:

`enabled`	`pressed`	`toggledOn`
`menu`		

Inherited from `Presenter`:

`presentedBy`	`subPresenters`	`target`

Inherited from `TwoDPresenter`:

`bBox`	`height`	`transform`
`boundary`	`IsImplicitlyDirect`	`width`
`clock`	`isTransparent`	`window`
`compositor`	`isVisible`	`x`
`direct`	`needsTickle`	`y`
`eventInterests`	`position`	`z`
`globalBoundary`	`stationary`	
`globalTransform`	`target`	

Inherited from `GenericButton`:

`activateAction`	`pressAction`	`releaseAction`
`authorData`		

Inherited from `StencilButton`:

`border`	`recessedFrame`	`stencil`
`buttonFrame`		

The following instance variables are defined in `PopUpButton`:

boundary

self`.boundary` (read-write) `Frame`

Specifies the boundary of the `PopUpButton` object *self*.

menu

self`.menu` (read-write) `String`

Specifies the menu to be displayed by the `PopUpButton` object *self*.

value

self`.value` (read-write) `String`

Specifies the menu item to be displayed on the `PopUpButton` object *self*.

Instance Methods

Inherited from `Actuator`:

`activate`	`press`	`toggleOff`
`multiActivate`	`release`	`toggleOn`

Inherited from `TwoDPresenter`:

`adjustClockMaster`	`inside`	`show`
`createInterestList`	`localToSurface`	`surfaceToLocal`
`draw`	`notifyChanged`	`tickle`
`getBoundaryInParent`	`recalcRegion`	
`hide`	`refresh`	

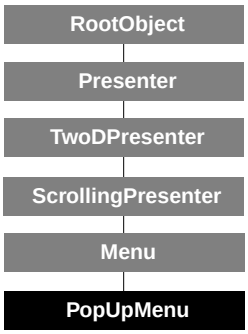
Inherited from `GenericButton`:

`activate`	`press`	`release`
`multiActivate`		

Inherited from `StencilButton`:

`draw`	`recalcRegion`

PopUpMenu



Class type: Scripted class (concrete)
Resides in: widgets.sxl. Works with ScriptX and KMP executables
Inherits from: Menu
Component: User Interface

PopUpMenu is a user interface Widget Kit class that provides a framed box whose appearance is defined by a ListSelection object within a ListBox object and a Frame object that gives the box a three dimensional look. A PopUpMenu object must be used along with a PopUpButton object. The PopUpButton object activates the PopUpMenu object.

The clipping boundary of a PopUpMenu object is calculated automatically as the smallest rectangle that encloses the ListBox.

Creating and Initializing a New Instance

The following script is an example of how to create a new instance of the PopUpMenu class:

```
myPopUpMenu := new PopUpMenu \
          list:#(@one:"First",@two:"Second",@three:"Third") \
          width:104 \
          actuatorController:undefined \
          layoutController:undefined
```

The variable myPopUpMenu contains an initialized PopUpMenu object. The new method uses the keywords defined in init.

init

```
init self width:integer [ boundary:stencil ] [ list:collection ]
    [ font:font ] [ placement:name ] [ actuatorController:actuatorController ]
    [layoutController:rowColumnController ] [ fill:brush ] [ stroke:brush ]
    [ horizScrollBar:scrollBar ] [ vertScrollBar:scrollBar ]
    [ targetPresenter:twoDSpace ] [ target:object ]
    [ stationary:boolean ]                                    ⇨ (none)
```

self	PopUpMenu object
width:	Integer object
boundary:	Stencil object
list:	Collection object
font:	Font object

The superclass Menu uses the following keywords:

placement:	NameClass object
actuatorController:	ActuatorController object
layoutController:	RowColumnController object

The superclass ScrollingPresenter uses the following keywords:

fill:	Brush object
stroke:	Brush object
horizScrollBar:	ScrollBar object
vertScrollBar:	ScrollBar object

Superclasses of ScrollingPresenter use the following keywords:

targetPresenter:	TwoDSpace object
target:	Any object (ignored by Menu)
stationary:	Boolean object

Initializes the PopUpMenu object *self*, applying the values supplied with the keywords to the instance variables of the same name. Do not call init directly on an instance — it is automatically called by the new method.

If you omit one of the keyword arguments, the following defaults are used:

```
boundary:(new Rect x2:width y2:font.leading + 2)
list:#()
font:theSystemFont
placement:@menuDown
actuatorController:@unsupplied
layoutController:@unsupplied
fill:undefined
stroke:undefined
horizScrollBar:undefined
vertScrollBar:undefined
targetPresenter:(new TwoDSpace)
target:undefined
stationary:false
```

You should not provide a value for boundary. If you do not provide a value for boundary, then the boundary of the new PopUpMenu object is calculated automatically as the smallest rectangle that encloses the width, one row of the list, and the menu frame.

Instance Variables

Inherited from Presenter:

presentedBy	subPresenters	target

Inherited from TwoDPresenter:

bBox	globalRegion	transform
boundary	globalTransform	width
changed	height	window
clock	imageChanged	x
compositor	isVisible	y
direct	position	z
eventInterests	stationary	
globalBoundary	target	

Inherited from ScrollingPresenter:

clippingStencil	stroke

```
fill                          targetPresenter
horizScrollBar                vertScrollBar
horizScrollBarDisplayed       vertScrollBarDisplayed
```

Inherited from Menu:

```
actuatorController    placement          targetPresenter
invoker               subMenu
layoutController      superMenu
```

Since a menu acts as a proxy for the instance of TwoDSpace that is its target presenter, the following instance variables are redirected to this space.

Redirected from Space:

```
clock                 protocols          tickleList
controllers
```

Redirected from Collection:

```
bounded               maxSize            size
iteratorClass         minSize            uniformity
keyEqualComparator    mutable            uniformityClass
keyUniformity         mutableCopyClass   valueEqualComparator
keyUniformityClass    proprietored
```

Redirected from IndirectCollection:

```
targetCollection
```

Redirected from TwoDMultiPresenter:

```
clock                 fill               stroke
```

The following instance variables are defined in PopUpMenu:

font

self.font (read-write) Font

Specifies the font in which to render the text of the items in the list of the PopUpMenu object *self*.

frame

self.frame (read-write) Frame

Specifies the Frame object that delimits the PopUpMenu object *self*. A default Frame object is supplied that gives the menu a slight three-dimensional appearance.

list

self.list (read-write) Collection

Specifies the collection of menu options to be displayed on the PopUpMenu object *self*. By default, each item in the collection is displayed on a separate row, in a single column, in the menu.

selectAction

self.selectAction (read-write) *(function)*

Specifies the function that is called when the PopUpMenu object *self* receives a mouse up event. This function is called automatically by the upReceiver method. You can write a function to perform any action. This function has two arguments:

```
funcName authorData selectedLine
```

authorData	Data in the `authorData` instance variable
selectedLine	Data in the `selectedLine` instance variable

Although any global function, anonymous function, or method can be assigned to `activateAction`, there are differences in how different classes of functions are dispatched. For information on functions and function dispatch, see the "Object System Kernel" chapter of the *ScriptX Components Guide*.

selectedLine

self`.selectedLine` (read-write) `Integer`

Specifies which menu option is currently selected in the `PopUpMenu` object *self*, and therefore which menu option is currently displayed on the associated `PopUpButton` object (see also the `value` keyword for the `PopUpButton` class).

selection

self`.selection` (read-write) `ListSelection`

Specifies the `ListSelection` object that contains the `list` of the `PopUpMenu` object *self*.

stepAmount

self`.stepAmount` (read-write) `Integer`

Used in selecting a new menu option. Defined to be the same as the `leading` of the `font` of the `PopUpMenu` object *self*.

Instance Methods

Inherited from `TwoDPresenter`:

`adjustClockMaster`	`inside`	`show`
`createInterestList`	`localToSurface`	`surfaceToLocal`
`draw`	`notifyChanged`	`tickle`
`getBoundaryInParent`	`recalcRegion`	
`hide`	`refresh`	

Inherited from `ScrollingPresenter`:

`handleHorizScrollBar`	`layout`
`handleVertScrollBar`	`scrollTo`

Inherited from `Menu`:

`calculateSize`	`place`	`popup`
`hide`		

Since a menu acts as a proxy for the instance of `TwoDSpace` that is its target presenter, the following instance methods are redirected to this space.

Redirected from `Collection`:

`add`	`forEach`	`iterate`
`addMany`	`forEachBinding`	`localEqual`
`addToContents`	`getAll`	`map`
`chooseAll`	`getAny`	`merge`
`chooseOne`	`getKeyAll`	`pipe`
`chooseOneBinding`	`getKeyOne`	`prin`
`deleteAll`	`getMany`	`removeAll`
`deleteBindingAll`	`getOne`	`removeOne`
`deleteBindingOne`	`hasBinding`	`setAll`
`deleteKeyAll`	`hasKey`	`setOne`

deleteKeyOne	intersects	size
deleteOne	isEmpty	
emptyOut	isMember	

Redirected from IndirectCollection:

isAppropriateObject	objectAdded	objectRemoved

Redirected from TwoDMultiPresenter:

draw	findFirstInStencil	moveToBack
findAllAtPoint	isAppropriateObject	moveToFront
findAllInStencil	moveBackward	objectAdded
findFirstAtPoint	moveForward	objectRemoved

Since a TwoDSpace object is an indirect collection, you can also use any methods defined in the class specified by targetCollection. The target collection is typically an instance of Array, which inherits from Sequence, so the following instance methods are redirected to this space.

Accessible from LinearCollection:

chooseOneBackwards	deleteSecond	getMiddle
chooseOrdOne	deleteThird	getNth
deleteFifth	findRange	getNthKey
deleteFirst	forEachBackwards	getOrdOne
deleteFourth	getFifth	getRange
deleteLast	getFirst	getSecond
deleteNth	getFourth	getThird
deleteRange	getLast	pop

Accessible from Sequence:

addFifth	moveBackward	setFourth
addFirst	moveForward	setLast
addFourth	moveToBack	setNth
addNth	moveToFront	setSecond
addSecond	prepend	setThird
addThird	prependNew	sort
append	setFifth	
appendNew	setFirst	

The following instance methods are defined in PopUpMenu:

repeatScrollAction

repeatScrollAction *self targetPresenter direction* ⇨ *(none)*

self	PopUpMenu object
targetPresenter	ListBox object
direction	@up or @down

Responds to vertical movement of the mouse across the PopUpMenu object *self* (the mouse button must be down in order to display the PopUpMenu object) by highlighting a menu option either above or below the currently selected menu option. This method is called by the tickle method, and this method in turn calls the scrollTo method.

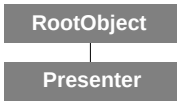
upReceiver

upReceiver *self theInterest event* ⇨ *(none)*

self	PopUpMenu object
theInterest	ListBox object
event	@up or @down

Responds to release of the mouse button by discontinuing tickles of the `PopUpMenu` object *self* and making the currently highlighted menu option the selected menu option that is displayed on the associated `PopUpButton` object. Finally, this method hides the `PopUpMenu` object so that only the `PopUpButton` object is displayed.

Presenter



Class type: Core class (abstract)
Resides in: ScriptX and KMP executables
Inherits from: `RootObject`
Component: Spaces and Presenters

The `Presenter` class provides a simple protocol for establishing relationships among graphic objects. It provides both a target to be presented and the means for forming a presentation hierarchy among related targets. However, `Presenter` does not provide any specific implementation for presentation; subclasses of `Presenter` provide the necessary implementation for drawing themselves (using the graphic compositor) or for processing events.

A presenter is contained in no more than one presentation space at a time. It may, however, be in other nonpresentation spaces at the same time.

Each instance of a `Presenter` subclass exists in a presentation hierarchy, which is traversed when composing each frame. A presenter specifies its position in this hierarchy with the instance variables `presentedBy` and `subpresenters`.

Creating and Initializing a New Instance

Because `Presenter` is an abstract class, you cannot create an instance of `Presenter`, nor is it useful to call the `init` method on `Presenter` directly. However, you should call `init` from any subclass of `Presenter` that overrides `init`, for proper initialization of instances of the subclass.

init

`init` *self* [`target:`*object*] ⇨ *(none)*

self	`Presenter` object
`target:`	Any object

Initializes the `Presenter` object *self*, applying the value supplied with `target` to the instance variable of the same name. Do not call `init` directly on an instance—it is automatically called by the `new` method.

If you omit an optional keyword, its default value is used. The defaults are:

```
target:undefined
```

The following instance variables are defined in `Presenter`:

Instance Variables

presentedBy

self`.presentedBy` (read-write) `Presenter`

Specifies the `Presenter` object that is currently presenting the presenter *self*. These two presenters have a parent/child relationship in the presentation hierarchy. The value of this instance variable is `undefined` for the top presenter in a presentation hierarchy. Note that a `Presenter` object can be presented by only one `Presenter` object at a time.

For example, if a window `myWindow` contains a pushbutton, you can say the pushbutton is presented by the window. The pushbutton's `presentedBy` instance variable a reference to `myWindow`. One consequence of this relationship is that if you move the window, the pushbutton moves with it.

In general, you should not set `presentedBy`, even though it can be written to. This instance variable is automatically and properly set when a presenter is added to or removed from the list specified by the `subpresenters` instance variable. Explicitly setting this variable can be a source of problems unless you fully understand the organization of the presentation hierarchy and take care to maintain the necessary relationships between presenters.

subpresenters

self`.subpresenters` (read-write) `Collection`

Specifies the collection of presenters contained in the presenter *self*. When a presenter has no subpresenters, the value of `subpresenters` is `undefined`.

For example, an instance of `TwoDSpace` might contain a video player and pushbuttons for making it play and stop. The video player and pushbuttons are subpresenters of the 2D space.

In general the presenter *self* is always presented (a 2D space has a background fill and a stroke); the subpresenters collection determines which additional objects are to be presented, and in what order.

`Presenter` is abstract and does not make use of the `subpresenters` instance variable—it leaves the particular implementation to its subclasses. Subclasses must determine the order in which to present the subpresenters. The `Presenter` class does not do any sorting of the subpresenters.

For example, in the case of graphic presenters with multiple subpresenters, that is, `TwoDMultiPresenter`, the order of subpresenters determines their visual depth, or z-order; a subpresenter at the front of the list displays in front of a presenter at the back of the list.

Every list of subpresenters forms a branch of a presentation hierarchy. Any subpresenter can also have subpresenters, extending the hierarchy deeper still.

Any object in the subpresenters list is displayed; objects not in the subpresenters list are not displayed (except targets may be displayed). Each concrete subclass of `Presenter` manages how objects are moved into and out of the subpresenters list. For example, all objects added to an instance of `TwoDMultiPresenter` are also subpresenters, and are displayed together; however, for an object added to a `OneOfNPresenter` instance, the `goTo` method moves only one object into the subpresenters list at a time, and only that object is displayed.

The number of subpresenters that a presenter can have depends on its class. In general, presenters that inherit from `TwoDMultiPresenter` can have multiple subpresenters, while presenters that don't inherit from `TwoDMultiPresenter` are limited to either zero or one subpresenter. Presenters that inherit from `TwoDMultiPresenter` are called *containers* because they can contain more than one subpresenter at a time.

The simplest noncontainer presenters, such as `TwoDShape`, need no subpresenter because there is nothing special going on—there is only one presenter to be displayed, and it does not get switched with other presenters.

More complex noncontainer presenters, such as `OneOfNPresenter` and `CostumedPresenter`, require one subpresenter. `OneOfNPresenter` maintains a list of objects to present, but can present only one at a time—that one is held in the `subpresenters` instance variable. `CostumedPresenter` can change the object it is presenting—again, the object being presented is held in `subpresenters`.

Container presenters, such as `TwoDMultiPresenter`, `TwoDSpace`, `Window`, `TwoDShape`, `GroupPresenter`, and `MoviePlayer`, can have multiple subpresenters, which are displayed at the same time.

If you subclass off of `TwoDPresenter` rather than `TwoDMultiPresenter`, the resulting class will by default have no subpresenters—you must add the extra functionality required to handle subpresenters.

target

self`.target` (read-write) *(object)*

Specifies the object that the `Presenter` object *self* is presenting. The target need not be a presenter. When `target` is not needed, its value should be `undefined`.

For example, an instance of `Text` is the target for a `TextPresenter` object. By itself, an instance of `Text` cannot draw—it requires a presenter such as `TextPresenter` to display it.

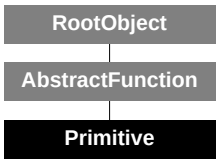
When creating an instance of certain `presenter` classes, if you omit the `target` keyword argument, a default target is created. These presenter classes and their default objects are shown in the following list:

`TextPresenter`	`Text` object
`TextEdit`	`Text` object
`CostumedPresenter`	`TwoDPresenter` object
`TwoDShape`	`Stencil` object
`TransitionPlayer`	`TwoDShape` object

For a `PushButton` object, on the other hand, the default target is `undefined`, because it has its own presenters in its instance variables `pressedPresenter`, `releasedPresented`, and `disabledPresenter`.

You can make the `target` instance variable of a presenter set its display properties by overriding `targetSetter` in its `init` method. For example, in making a bar chart, you could define a `Bar` class that changes the height of its bar based on the value of its target number.

Primitive



Class type: Core class (concrete, sealed)
Resides in: ScriptX and KMP executables
Inherits from: `AbstractFunction`
Component: Object System Kernel

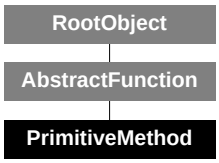
A global function that is defined in the substrate is compiled to create an instance of `Primitive`. By contrast, global functions that are defined in the scripter are implemented as `ByteCodeMethod` objects. Global functions are called identically, whether they are implemented as primitives or as bytecode methods.

For more information on ScriptX function dispatch, see the "Object System Kernel" chapter of the *ScriptX Components Guide*.

Creating and Initializing a New Instance

You cannot define an instance of `Primitive` in the scripter. It is possible to create a `Primitive` object in C, using the Extending ScriptX API. For information on the extending ScriptX, see the *ScriptX Developer's Guide*.

PrimitiveMethod



Class type: Core class (concrete, sealed)
Resides in: ScriptX and KMP executables
Inherits from: `AbstractFunction`
Component: Object System Kernel

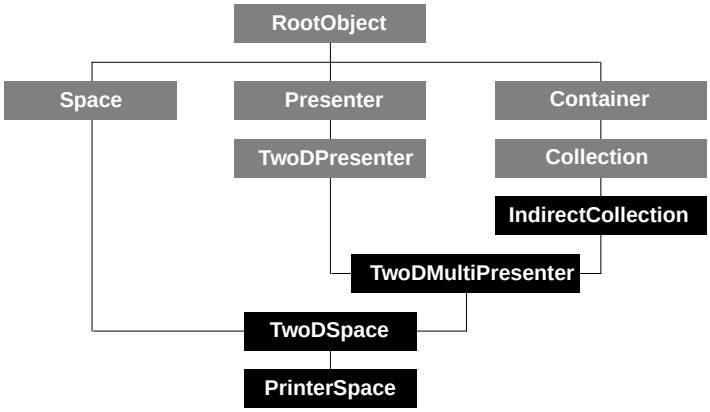
A method that is defined in the substrate is compiled to create an instance of `PrimitiveMethod`. By contrast, methods that are defined in the scripter are implemented as `ByteCodeMethod` objects. A `PrimitiveMethod` object can be called directly, however, it is usually called indirectly, by calling the generic function it implements a method for.

For more information on ScriptX function dispatch, see the "Object System Kernel" chapter of the *ScriptX Components Guide*.

Creating and Initializing a New Instance

You cannot define an instance of `PrimitiveMethod` in the scripter. It is possible to create a `PrimitiveMethod` object in C, using the Extending ScriptX API to create a generic. For information on the extending ScriptX, see the *ScriptX Developer's Guide*.

PrinterSpace



Class type: Loadable class (concrete)
Resides in: `printing.lib`. Works with ScriptX and KMP executables
Inherits from: `TwoDSpace`
Component: Printing

The `PrinterSpace` class prints out the state of a presentation hierarchy. A `PrinterSpace` instance, like a `Window` instance, can be made the parent space of a presentation hierarchy. The `printFrame` method can then be used to take a snapshot of the presentation. As with any `TwoDSpace`, a `PrinterSpace` instance has a fixed boundary and clips its sub-presenters.

You have to be careful when you print the presentation hierarchy associated with a `Window`, since a title may use a `Window` that is a different size from the `PrinterSpace` page. See the Printing component in the *ScriptX Components Guide* for information on how to do this.

Creating and Initializing a New Instance

The following script is an example of how to create a new instance of the `PrinterSpace` class:

```
p := new PrinterSpace
```

The variable `p` contains the new `PrinterSpace` object. The `new` method uses keywords defined by the `init` method.

For performance reasons, when creating an instance of `PrinterSpace` — actually, any instance of `TwoDMultiPresenter` or its subclasses — you should not specify the `targetCollection` keyword. Allow the class to create its default collection. Such a presenter requires a target collection that can be traversed quickly when drawing subpresenters or handing events. Performance could suffer if you change `targetCollection` to something other than what is generated by default.

init

`init` *self* [`deviceName:`*string*] [`surface:`*printerSurface*] ⇨ *(none)*

self	`PrinterSpace` object
`deviceName:`	`String` object
`surface:`	`PrinterSurface` object

There are no required keyword arguments. This initialization takes either a `deviceName` keyword, from which it creates a `PrinterSurface`, or an actual `PrinterSurface` instance. If neither keyword is supplied, a default printer is chosen. Possible values for `deviceName` can be obtained from the `getPrinterNameList` global function. Do not call `init` directly on an instance—it is automatically called by the `new` method.

Class Methods

Inherited from `Collection`:
`pipe`

Instance Variables

Inherited from `Space`:

`clock`	`controllers`	`protocols`

Inherited from `Presenter`:

`presentedBy`	`subPresenters`	`target`

Inherited from `TwoDPresenter`:

`bBox`	`globalRegion`	`transform`
`boundary`	`globalTransform`	`width`
`changed`	`height`	`window`
`clock`	`imageChanged`	`x`
`compositor`	`isVisible`	`y`
`direct`	`position`	`z`
`eventInterests`	`stationary`	
`globalBoundary`	`target`	

Inherited from `Collection`:

`bounded`	`maxSize`	`size`
`iteratorClass`	`minSize`	`uniformity`
`keyEqualComparator`	`mutable`	`uniformityClass`
`keyUniformity`	`mutableCopyClass`	`valueEqualComparator`
`keyUniformityClass`	`proprietored`	

Inherited from `IndirectCollection`:
`targetCollection`

Inherited from `TwoDMultiPresenter`:

`clock`	`fill`	`stroke`

Inherited from `TwoDSpace`:
`protocols`

The following instance variables are defined in `PrinterSpace`:

effectiveResolution

self`.effectiveResolution` (read-write) `Pair`

The effective resolution of the space represented by the `PrinterSpace` as a pair of numbers in dots per inch. The first number in the pair is the horizontal resolution, and the second is the vertical resolution. This resolution is computed from the `transform` instance variable of the `PrinterSpace` and the `physicalResolution` of the `PrinterSurface`.

While the `physicalResolution` instance variable on `PrinterSurface` tells you the physical resolution of the underlying print device, the `effectiveResolution` instance variable on `PrinterSpace` represents the effective resolution of the space represented by the `PrinterSpace`. In other words, it takes into account the transform applied to the

space. For example, if the `transform` instance variable of the `PrinterSpace` is the identity matrix, the effective resolution of the `PrinterSpace` space is the same as that of the underlying device. If the `transform` instance variable performs a 2x scaling, the effective resolution is half that of the device.

firstPage

self`.firstPage` (read-write) `Integer`

The first page to print; the default value is `1`. All flushed pages before this page are ignored. This instance variable can be changed by the end user when the printer dialog is invoked. Same as for `PrinterSurface`.

lastPage

self`.lastPage` (read-write) `Integer` or `Name`

The last page to print; the default value is `@all`, which means that there is no upper limit to the number of pages to be printed. All flushed pages after this page are ignored. This instance variable can be changed by the end user when the printer dialog is invoked. Same as for `PrinterSurface`.

orientation

self`.orientation` (read-only) `Name`

The orientation of the paper: either `@portrait` or `@landscape`. Same as for `PrinterSurface`.

paperBoundary

self`.paperBoundary` (read-only) `Rect`

The actual dimensions of the paper. The `boundary` instance variable represents the dimensions of the *printable area* of the paper. Both `boundary` and `paperBoundary` are represented in the same coordinate system with the origin at the top-left corner of the printable area (at the top left of the area defined by the `boundary` instance variable). Same as for `PrinterSurface`.

surface

self`.surface` (read-write) `PrinterSurface`

The `PrinterSurface` associated with this object.

Instance Methods

Inherited from `Space`:

`isAppropriateObject`	`objectAdded`	`objectRemoved`

Inherited from `TwoDPresenter`:

`addChangedRegion`	`getBoundaryInParent`	`show`
`adjustClockMaster`	`hide`	`windowToLocal`
`createInterestList`	`localToWindow`	
`draw`	`refresh`	

Inherited from `Container`:

`adopt`	`evalReferent`	`selectContext`
`contextOf`	`referentOf`	`setReferent`
`contexts`	`seekToReferent`	
`disown`	`selectAll`	

Inherited from `Collection`:

add	emptyOut	isMember
addMany	expand	iterate
addManyValues	forEach	map
chooseAll	forEachBinding	merge
chooseOne	getAll	pipe
chooseOneBinding	getAny	prin
contract	getKeyAll	reductionFor
deleteAll	getKeyOne	removeAll
deleteBindingAll	getMany	removeOne
deleteBindingOne	getOne	setAll
deleteKeyAll	hasBinding	setOne
deleteKeyOne	hasKey	
deleteOne	isEmpty	

Inherited from `IndirectCollection`:

isAppropriateObject	objectAdded	objectRemoved

Inherited from `TwoDMultiPresenter`:

draw	findFirstInStencil	moveToBack
findAllAtPoint	isAppropriateObject	moveToFront
findAllInStencil	moveBackward	objectAdded
findFirstAtPoint	moveForward	objectRemoved

Since a `PrinterSpace` object is an indirect collection, you can also use any methods defined in the class specified by `targetCollection`. The target collection is typically an instance of `Array`, which inherits from `Sequence`, so the following instance methods are redirected to a printer:

Accessible from `LinearCollection`:

chooseOneBackwards	deleteSecond	getMiddle
chooseOrdOne	deleteThird	getNth
deleteFifth	findRange	getNthKey
deleteFirst	forEachBackwards	getOrdOne
deleteFourth	getFifth	getRange
deleteLast	getFirst	getSecond
deleteNth	getFourth	getThird
deleteRange	getLast	pop

Accessible from `Sequence`:

addFifth	moveBackward	setFourth
addFirst	moveForward	setLast
addFourth	moveToBack	setNth
addNth	moveToFront	setSecond
addSecond	prepend	setThird
addThird	prependNew	sort
append	setFifth	
appendNew	setFirst	

The following instance methods are defined in `PrinterSpace`:

flushDocument

`flushDocument` *self* ⇨ *self*

Indicates the end of a document. If the document was being spooled, the spooled information is sent to a printer or printer queue. This method must be called for actual printing to occur. Same as in `PrinterSurface`.

flushPage

`flushPage` *self* ⇨ *self*

Indicates the end of a page. Flushes out the contents of the current page as determined by the printer and printing method. Same as in `PrinterSurface`.

getMargin

`getMargin` *self units* ⇨ `Quad`

self	`PrinterSpace` object
units	`Name` object

Finds the effective margins on the `PrinterSpace`, based on the current value of the `boundary` instance variable. This method returns a collection of four values: the margins on the left, top, right, and bottom, in that order. The *units* parameter specifies what units the margin values represent: either `@inches`, `@cm`, `@points`, or `@pixels`.

printerDialog

`printerDialog` *self* ⇨ `Boolean`

Displays a machine-specific dialog that can be used to change settings for the current printing job. Changes made in this dialog box are reflected in the `firstPage` and `lastPage` instance variables. Returns `true` if the dialog is confirmed, and returns `false` if it is cancelled. Printing should not continue if it returns `false`. Same as in `PrinterSurface`.

printFrame

`printFrame` *self* ⇨ *self*

Takes a snapshot of the presentation hierarchy hooked up to the `PrinterSpace` by calling the `draw` method on the top `TwoDPresenter`.

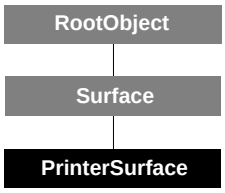
setMargin

`setMargin` *self margin units* ⇨ `Quad`

self	`PrinterSpace` object
margin	`Quad` object
units	`Name` object

Sets the `boundary` of the `PrinterSpace` to incorporate a margin. The *margin* parameter is a collection of four values: the margins on the left, top, right, and bottom, in that order. The *units* parameter specifies what units the margin values represent: either `@inches`, `@cm`, `@points`, or `@pixels`.

PrinterSurface



Class type: Loadable class (concrete)
Resides in: `printing.lib`. Works with ScriptX and KMP executables
Inherits from: `Surface`
Component: Printing

The `PrinterSurface` class provides the low-level mechanism for rendering stencils. The use of `PrinterSurface` is discouraged — in most cases, you should use only the `PrinterSpace` class. Whenever necessary, instance variables and methods from the `PrinterSurface` class are mirrored in the `PrinterSpace` class.

Like its screen counterpart, `DisplaySurface`, the `PrinterSurface` class implements `fill`, `stroke`, and `transfer` methods to render stencils to the printer. The `printing` loadable module adds methods to the individual `Stencil` classes to actually perform the rendering.

Creating and Initializing a New Instance

The following script is an example of how to create a new instance of the `PrinterSurface` class:

```
Psurface:= new PrinterSurface
```

The variable `Psurface` contains the new `PrinterSurface` object. The `new` method uses keywords defined by the `init` method. The surface has a boundary that represents the printable area of the page.

init

`init` *self* [`deviceName:`*string*] ⇨ *(none)*

self	`PrinterSurface` object
`deviceName:`	`String` object

Creates a `PrinterSurface` object that represents the printer specified by the `deviceName` keyword. If the keyword is omitted, a default printer is chosen. Possible values for `deviceName` can be obtained from the `getPrinterNameList` global function. Do not call `init` directly on an instance—it is automatically called by the `new` method.

Instance Variables

Inherited from `Surface`:
`boundary`

The following instance variables are defined in `PrinterSurface`:

availableResolutions

self`.availableResolutions` (read-only) `Array`

The resolutions available for a printer as an array of pairs of numbers. Any of the elements (number pairs) in this array can be used to set `physicalResolution`.

boundary (Surface)

self.boundary (read-only) Rect

This instance variable is the same as defined in the Surface class except that it is read-only.

physicalResolution

self.physicalResolution (read-write) Pair

The physical resolution of the underlying print device as a pair of numbers in dots per inch. The first number in the pair is the horizontal resolution, and the second is the vertical resolution.

While the physicalResolution instance variable on PrinterSurface tells you the physical resolution of the underlying print device, the effectiveResolution instance variable on PrinterSpace represents the effective resolution of the space represented by the PrinterSpace. In other words, it takes into account the transform applied to the space. For example, if the transform instance variable of the PrinterSpace is the identity matrix, the effective resolution of the PrinterSpace space is the same as that of the underlying device. If the transform instance variable performs a 2x scaling, the effective resolution is half that of the device.

firstPage

self.firstPage (read-write) Integer

The first page to print; the default value is 1. All flushed pages before this page are ignored. This instance variable can be changed by the end user when the printer dialog is invoked. Same as for PrinterSpace.

lastPage

self.lastPage (read-write) Integer or Name

The last page to print; the default value is @all, which means that there is no upper limit to the number of pages to be printed. All flushed pages after this page are ignored. This instance variable can be changed by the end user when the printer dialog is invoked. Same as for PrinterSpace.

orientation

self.orientation (read-only) Name

The orientation of the paper: either @portrait or @landscape. Same as for PrinterSpace.

paperBoundary

self.paperBoundary (read-only) Rect

The actual dimensions of the paper. The boundary instance variable represents the dimensions of the *printable area* of the paper. Both boundary and paperBoundary are represented in the same coordinate system with the origin at the top-left corner of the printable area (at the top left of the area defined by the boundary instance variable). Same as for PrinterSpace.

Instance Methods

Inherited from Surface:

fill stroke transfer

The following instance methods are defined in `PrinterSurface`:

flushDocument

`flushDocument` *self* ⇨ *self*

Indicates the end of a document. If the document was being spooled, the spooled information is sent to a printer or printer queue. This method must be called for actual printing to occur. Same as in `PrinterSpace`.

flushPage

`flushPage` *self* ⇨ *self*

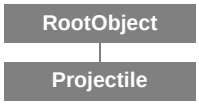
Indicates the end of a page. Flushes out the contents of the current page as determined by the printer and printing method. Same as in `PrinterSpace`.

printerDialog

`printerDialog` *self* ⇨ `Boolean`

Displays a machine-specific dialog that can be used to change settings for the current printing job. Changes made in this dialog box are reflected in the `firstPage` and `lastPage` instance variables. Returns `true` if the dialog is confirmed, and returns `false` if it is cancelled. Printing should not continue if it returns `false`. Same as in `PrinterSpace`.

Projectile



Class type: Core class (abstract)
Resides in: ScriptX and KMP executables
Inherits from: `RootObject`
Component: Controllers

The `Projectile` class is an abstract mixin class. It adds the physical properties `velocity`, `acceleration`, and `elasticity` to any object, and an `audioPlayer` that plays a sound when it collides with another object.

A projectile can allow objects to be controlled by `Bounce`, `Gravity`, or `Movement` controllers.

Creating and Initializing a New Instance

You never create an instance by calling `new` on the `Projectile` class. Instead, you mix it in when defining another class, then override the `init` instance method of the new class as described below.

init

`init` *self* ⇨ *(none)*

self `Projectile` object

Initializes the `Projectile` object *self*. Do not call init directly on an instance—it is automatically called by the new method.

Instance Variables

acceleration

self`.acceleration` (read-write) `Point`

Specifies the acceleration vector for the projectile *self*. The vector has x and y components, giving it a direction. By default a projectile has an acceleration of 0, 0.

audioPlayer

self`.audioPlayer` (read-write) `AudioPlayer`

Specifies the audio player used to play a sound associated with the projectile *self*. What the sound is used for depends on the controller. For the bounce controller, the sound occurs when the projectile hits the edge of the space.

elasticity

self`.elasticity` (read-write) `Number`

Specifies the amount of elasticity the projectile *self* has when bouncing off edges of a space or other obstacles. Can be a number between 0 and 1, inclusive, where 0 means lose all energy in a collision and 1 means perfectly elastic, lose no energy. (Numbers greater than 1 mean gain energy on collision.)

velocity

self.`velocity` (read-write) `Point`

Specifies the velocity vector for the projectile *self*. The vector has x and y components, giving it a direction. By default a projectile has a velocity of 2, 2.

Instance Methods

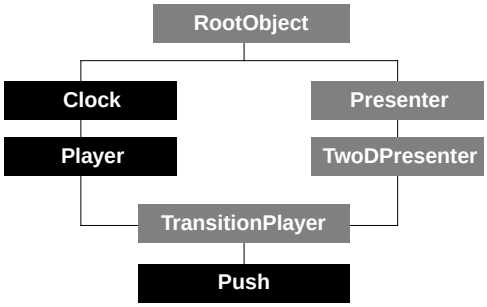
makeSound

`makeSound` *self amplitude* ⇨ *self*

self	`Projectile` object
amplitude	`Number` object

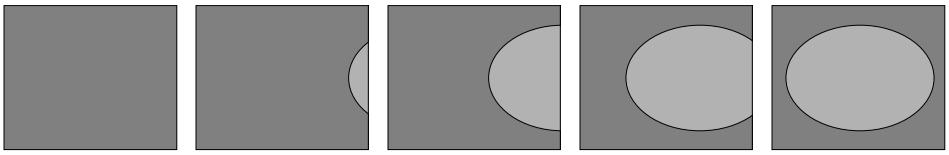
Causes the projectile *self* to make a sound at the given *amplitude*. The amplitude is a number greater than or equal to 0.

Push



Class type: Loadable class (concrete)
Resides in: `ltrans.lib`. Works with ScriptX and KMP executables
Inherits from: `TransitionPlayer`
Component: Transitions

The `Push` transition player provides a visual effect that causes the target to gradually appear, as if pushed onto the screen from the given direction. The `@left` direction playing forward is shown below. The target appears when the transition is played forward, and disappears when played backward (transition's rate set to -1).



Directions: `@right`, `@left`, `@up`, `@down`

Rate: Can play forward or backward.

For a side-by-side illustrations of all transitions, see the Transitions chapter in the *ScriptX Components Guide*.

Creating and Initializing a New Instance

The following script is an example of how to create a new instance of the `Push` class:

```
myTransition := new Push \
                    duration:60 \
                    direction:@left \
                    target:myShape
```

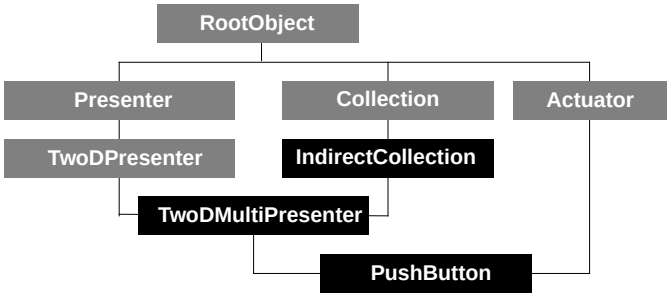
The variable `myTransition` contains the initialized transition. The transition reveals the image `myShape` being pushed to the left, and has a duration of 60 ticks.

You determine which space the transition will take effect in by adding this instance into that space. Then, when you play the transition player, `myShape` is transitioned into that space.

The `new` method uses the keywords defined in `init`.

NOTE – For the instance variables and methods, see the `BarnDoor` class.

PushButton



Class type: Core class (concrete)
Resides in: ScriptX and KMP executables
Inherits from: `TwoDMultiPresenter` and `Actuator`
Component: User Interface

The `PushButton` class provides the basic functionality for presenting and controlling buttons as elements of the user interface. An actuator controller associated with the button receives and processes mouse events. When pushbuttons are pressed, released, or activated, they can invoke an action, a function that is defined by a script.

The clipping boundary of a `PushButton` object is calculated automatically as the smallest rectangle that encloses all its contained presenters. Each of these presenters has its `x` and `y` set to (0, 0). For more information about presenters, see the "Spaces and Presenters" chapter of the *ScriptX Components Guide*.

Creating and Initializing a New Instance

The following script creates a new instance of the `PushButton` class. It first defines two objects that act as stencils for the targets of simple presenters. One is used to define the target of both the pressed and released presenters; the other, to define the target of the disabled presenter.

```
myBounds := new Rect x1:30 y1:30 x2:80 y2:50 -- stencil object
theCheck := new Line x1:30 y1:30 x2:80 y2:50 -- stencil object
myButton := new PushButton \
                pressedPresenter:(new TwoDShape boundary:myBounds \
                fill:blackBrush)\
                releasedPresenter:(new TwoDShape boundary:myBounds \
                fill:whiteBrush stroke:blackBrush)\
                disabledPresenter:(new TwoDShape boundary:theCheck
                fill:blackBrush)
```

The variable `myButton` contains an initialized `PushButton` object, which will control the presenters that are specified by its keyword arguments. The `new` method uses the keywords defined by the `init` method.

init

`init` *self* [`pressedPresenter:`*twoDPresenter*]
[`releasedPresenter:`*twoDPresenter*]
[`disabledPresenter:`*twoDPresenter*]
[`fill:` *brush*] [`stroke:` *brush*] ⇨ *(none)*

self	`PushButton` object
`pressedPresenter:`	`TwoDPresenter` object
`releasedPresenter:`	`TwoDPresenter` object
`disabledPresenter:`	`TwoDPresenter` object

Superclasses of `PushButton` use these keywords:

`fill:`	`Brush` object
`stroke:`	`Brush` object
`boundary:`	(Ignored by `PushButton`)
`target:`	(Ignored by `PushButton`)
`targetCollection:`	`Sequence` object (use carefully)
`stationary:`	`Boolean` object

Initializes the `PushButton` object *self*, applying the values supplied with the keywords to the instance variables of the same name. Do not call `init` directly on an instance—it is automatically called by the `new` method.

If you omit an optional keyword, its default value is used. The defaults are:

```
pressedPresenter:undefined
releasedPresenter:undefined
disabledPresenter:undefined
fill:undefined
stroke:undefined
boundary:(new Rect x2:0 y2:0) if all subpresenters are undefined
target:undefined
targetCollection:(new Array initialSize:14 growable:true)
stationary:false
```

The `PushButton` class inherits a number of keywords from its superclasses that are generally not used.

A pushbutton's appearance is generally determined by the appearance of its three subpresenters, but the `fill` and `stroke` keywords can be used separately to provide a fill and stroke for the entire button. For example, a pushbutton could use the `fill` keyword to specify a background color. If the three subpresenters target instances of bitmap, this fill color will show through anyplace where the bitmap's invisible color appears.

Since a pushbutton calculates its own boundary as the smallest `Rect` object that encloses its subpresenters, the `boundary` keyword is ignored.

The `target` keyword is ignored, but it is not explicitly overridden. It can be used by a subclass of `PushButton` at the developer's discretion. A subclass could use the `target` keyword to set the target on subpresenters. For example, if the three subpresenters of `PushButton`—the pressed, released, and disabled presenters—were all instances of `TextPresenter`, the `target` keyword could be used to supply a `String` object as the common target of all three. Note that `target` is an instance variable defined by `Presenter`. A presenter's target is a reference to data that it presents.

The `targetCollection` keyword should generally be ignored. Each subclass of `TwoDMultiPresenter` sets its own target collection, chosen for efficiency. Use discretion in changing the target collection; for more information, see the definition of the `TwoDMultiPresenter` class.

Class Methods

Inherited from `Collection`:

`pipe`

Instance Variables

Inherited from `Actuator`:

`enabled`	`pressed`	`toggledOn`
`menu`		

Inherited from `Collection`:

`bounded`	`maxSize`	`size`
`iteratorClass`	`minSize`	`uniformity`
`keyEqualComparator`	`mutable`	`uniformityClass`
`keyUniformity`	`mutableCopyClass`	`valueEqualComparator`
`keyUniformityClass`	`proprietored`	

Inherited from `IndirectCollection`:

`targetCollection`

Inherited from `Presenter`:

`presentedBy`	`subPresenters`	`target`

Inherited from `TwoDPresenter`:

`bBox`	`height`	`transform`
`boundary`	`IsImplicitlyDirect`	`width`
`clock`	`isTransparent`	`window`
`compositor`	`isVisible`	`x`
`direct`	`needsTickle`	`y`
`eventInterests`	`position`	`z`
`globalBoundary`	`stationary`	
`globalTransform`	`target`	

Inherited from `TwoDMultiPresenter`:

`clock`	`fill`	`stroke`

The following instance variables are defined in `PushButton`:

activateAction

self`.activateAction` (read-write) *(function)*

Specifies the default function that is called when the pushbutton *self* is activated. Initially this instance variable is `undefined`. You can write a function to perform any action. This function has two arguments:

`funcName` *authorData self*

authorData	data in the `authorData` instance variable
self	the `PushButton` object *self* to which the action is attached

Although any global function, anonymous function, or method can be assigned to `activateAction`, there are differences in how different classes of functions are dispatched. For information on functions and function dispatch, see the "Object System Kernel" chapter of the *ScriptX Components Guide*.

authorData

self.authorData (read-write) *(object)*

Used as an argument for the function that is specified by activateAction, pressAction, and releaseAction. This instance variable can be any object.

disabledPresenter

self.disabledPresenter (read-write) TwoDPresenter

Specifies the TwoDPresenter object to be presented when the pushbutton *self* is disabled. The presenter specified by disabledPresenter is displayed in front of all of the pushbutton's other presenters when enabled is set to false. If the value of disabledPresenter is undefined, then the user receives no visual clue that the button is not enabled.

multiActivateAction

self.multiActivateAction (read-write) *(function)*

Specifies the default function that is called when the pushbutton *self* receives a given number of multiple clicks, each within the system's double-click threshold. See the definition of multiActivate, a method defined by Actuator. Initially this instance variable is undefined. You can write a function to perform any action. This function has three arguments:

```
funcName authorData self n
```

authorData	data in the authorData instance variable
self	the PushButton object *self* to which the action is attached
n	an Integer object, representing the number of times the pushbutton *self* was activated

Although any global function, anonymous function, or method can be assigned to multiActivateAction, there are differences in how different classes of functions are dispatched. For information on functions and function dispatch, see the "Object System Kernel" chapter of the *ScriptX Components Guide*.

pressAction

self.pressAction (read-write) *(function)*

Specifies the default function that is called when the pushbutton *self* is being pressed. The function has two arguments, as shown above in activateAction.

Although any global function, anonymous function, or method can be assigned to pressAction, there are differences in how different classes of functions are dispatched. For information on functions and function dispatch, see the "Object System Kernel" chapter of the *ScriptX Components Guide*.

pressedPresenter

self.pressedPresenter (read-write) TwoDPresenter

Specifies the TwoDPresenter object to be presented when the pushbutton *self* is pressed. If the value of pressedPresenter is undefined, then the pushbutton has no pressed appearance.

releaseAction

self.releaseAction (read-write) *(function)*

Specifies the default function to call when the pushbutton *self* has been pressed and is released without being activated. Typically, this happens when the user has moved the mouse pointer away from the button before releasing it. The function has two arguments, as previously shown in the activateAction instance variable.

Although any global function, anonymous function, or method can be assigned to releaseAction, there are differences in how different classes of functions are dispatched. For information on functions and function dispatch, see the "Object System Kernel" chapter of the *ScriptX Components Guide*.

releasedPresenter

self.releasedPresenter (read-write) TwoDPresenter

Specifies the TwoDPresenter object to be presented when the pushbutton *self* is not pressed. If the value of releasedPresenter is empty, then the pushbutton has no released appearance. (It may still be visible if its disabled presenter is defined.)

Instance Methods

Inherited from Actuator:

activate	press	toggleOff
multiActivate	release	toggleOn

Inherited from Collection:

add	forEach	iterate
addMany	forEachBinding	localEqual
addToContents	getAll	map
chooseAll	getAny	merge
chooseOne	getKeyAll	pipe
chooseOneBinding	getKeyOne	prin
deleteAll	getMany	removeAll
deleteBindingAll	getOne	removeOne
deleteBindingOne	hasBinding	setAll
deleteKeyAll	hasKey	setOne
deleteKeyOne	intersects	size
deleteOne	isEmpty	
emptyOut	isMember	

Inherited from IndirectCollection:

isAppropriateObject	objectAdded	objectRemoved

Inherited from TwoDPresenter:

adjustClockMaster	inside	show
createInterestList	localToSurface	surfaceToLocal
draw	notifyChanged	tickle
getBoundaryInParent	recalcRegion	
hide	refresh	

Inherited from TwoDMultiPresenter:

draw	findFirstInStencil	moveToBack
findAllAtPoint	isAppropriateObject	moveToFront
findAllInStencil	moveBackward	objectAdded
findFirstAtPoint	moveForward	objectRemoved

Since a PushButton object is an indirect collection, you can also use any methods defined in the class specified by targetCollection. The target collection is typically an instance of Array, which inherits from Sequence, so the following instance methods are redirected to a pushbutton.

Accessible from LinearCollection:

chooseOneBackwards	findRange	getNthKey
chooseOrdOne	forEachBackwards	getOrdOne
deleteFirst	getFirst	getRange
deleteLast	getLast	localEqual
deleteNth	getMiddle	localLT
deleteRange	getNth	pop

Accessible from Sequence:

addFifth	moveBackward	setFourth
addFirst	moveForward	setLast
addFourth	moveToBack	setNth
addNth	moveToFront	setSecond
addSecond	prepend	setThird
addThird	prependNew	sort
append	setFifth	
appendNew	setFirst	

The following instance methods are defined in PushButton:

activate (Actuator)

`activate` *self* ⇨ *self*

Tells the pushbutton *self* that it has been activated, specializing the activate method, defined by Actuator. If the PushButton object *self* is enabled, activate displays the TwoDPresenter object that is specified by releasedPresenter. If the released presenter is undefined, no released appearance is displayed. (The button may have other visible presenters that are defined.) activate then calls the method handleActivate on the pushbutton.

calculateSize

`calculateSize` *self* ⇨ Point

Calculates the correct bounding box of the PushButton object *self*. By default, the size of the button is the largest width and height of all its contained presenters. This method returns a Point object that contains the new width and height. This method is called automatically each time any of the contained presenters changes. If the width or height of the button is changed after this method is called, then the subpresenters will be clipped to the new width and height.

handleActivate

`handleActivate` *self* ⇨ *self*

Calls the function specified by the activateAction instance variable. The two arguments to that function are authorData and *self*.

handleMultiActivate

`handleMultiActivate` *self n* ⇨ *self*

self	PushButton object
n	Integer object, the number of clicks

Calls the function specified by the `multiActivateAction` instance variable. The calling sequence is the same as for `handleActivate`, with the addition of a third argument *n* that indicates the number of clicks within the system's double-click threshold.

handlePress

`handlePress` *self* ⇨*self*

Calls the function specified by the `pressAction` instance variable. The two arguments to that function are `authorData` and *self*.

handleRelease

`handleRelease` *self* ⇨*self*

Calls the function specified by the `releaseAction` instance variable. The two arguments to that function are `authorData` and *self*.

layout

`layout` *self* ⇨*self*

Sets the correct locations of all presenters that the `PushButton` object *self* presents. This method is automatically called each time any of these presenters changes.

press (Actuator)

`press` *self* ⇨*self*

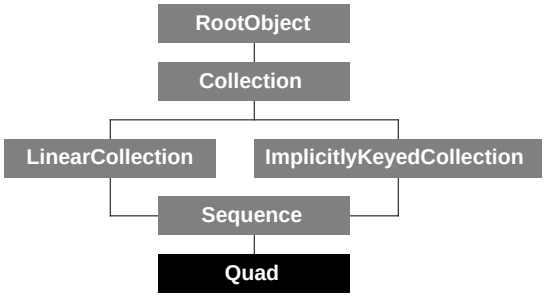
Tells the pushbutton *self* that it has been pressed, specializing the `press` method, defined by `Actuator`. If the `PushButton` object *self* is enabled, `press` displays the `TwoDPresenter` object that is specified by `pressedPresenter`. If the pressed presenter is `undefined`, no pressed appearance is displayed. (The button may have other visible presenters that are defined.) `press` then calls the method `handlePress` on the pushbutton.

release (Actuator)

`release` *self* ⇨*self*

Tells the pushbutton *self* that it has been released, specializing the `release` method, as defined by `Actuator`. If the `PushButton` object *self* is enabled, `release` displays the `TwoDPresenter` object that is specified by `releasedPresenter`. If the released presenter is `undefined`, no released appearance is displayed. (The button may have other visible presenters that are defined.) `release` then calls the method `handleRelease` on the pushbutton.

Quad



Class type: Core class (concrete)
Resides in: ScriptX and KMP executables
Inherits from: Sequence
Component: Collections

The Quad class represents arrays of four values. A Quad object is created with a fixed size of 4. You cannot add (addNth) or delete (deleteNth) items, but you can set their values (setNth).

Also see the Single, Pair, and Triple classes.

Adding a value to a Quad object causes it to report the exception bounded.

Creating and Initializing a New Instance

The following script is an example of how to create a new instance of the Quad class:

```
myQuad := new Quad values:#(1,2,3,4)
```

The variable myQuad contains the initialized quad, which contains the numbers 1, 2, 3, and 4. The new method uses the keywords defined in init.

init

`init` *self* ⇨ *(none)*

self Quad object

Initializes the Quad object *self*, setting each of the four items in the list to undefined. Do not call init directly on an instance—it is automatically called by the new method.

Class Methods

Inherited from Collection:

pipe

Instance Variables

Inherited from Collection:

bounded	maxSize	size
iteratorClass	minSize	uniformity
keyEqualComparator	mutable	uniformityClass
keyUniformity	mutableCopyClass	valueEqualComparator
keyUniformityClass	proprietored	

Instance Methods

Inherited from `Collection`:

`add`	`forEach`	`iterate`
`addMany`	`forEachBinding`	`localEqual`
`addToContents`	`getAll`	`map`
`chooseAll`	`getAny`	`merge`
`chooseOne`	`getKeyAll`	`pipe`
`chooseOneBinding`	`getKeyOne`	`prin`
`deleteAll`	`getMany`	`removeAll`
`deleteBindingAll`	`getOne`	`removeOne`
`deleteBindingOne`	`hasBinding`	`setAll`
`deleteKeyAll`	`hasKey`	`setOne`
`deleteKeyOne`	`intersects`	`size`
`deleteOne`	`isEmpty`	
`emptyOut`	`isMember`	

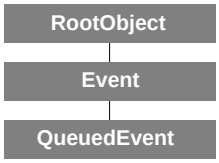
Inherited from `LinearCollection`:

`chooseOneBackwards`	`findRange`	`getNthKey`
`chooseOrdOne`	`forEachBackwards`	`getOrdOne`
`deleteFirst`	`getFirst`	`getRange`
`deleteLast`	`getLast`	`localEqual`
`deleteNth`	`getMiddle`	`localLT`
`deleteRange`	`getNth`	`pop`

Inherited from `Sequence`:

`addFifth`	`moveBackward`	`setFourth`
`addFirst`	`moveForward`	`setLast`
`addFourth`	`moveToBack`	`setNth`
`addNth`	`moveToFront`	`setSecond`
`addSecond`	`prepend`	`setThird`
`addThird`	`prependNew`	`sort`
`append`	`setFifth`	
`appendNew`	`setFirst`	

QueuedEvent



Class type: Core class (abstract)
Resides in: ScriptX and KMP executables
Inherits from: `Event`
Component: Events

Subclasses of `QueuedEvent` are used to describe both system- and user-defined events that need to be delivered sequentially and chronologically. When events of this type are first signaled or broadcast, they are placed on a dispatch queue. The event is not actually dispatched to the interested parties until it reaches the end of the dispatch queue. The secondary dispatch is handled using the `secondarySignal` and `secondaryBroadcast` methods.

These are the subclasses of `QueuedEvent` that are defined at system startup:

- `MouseEvent` is the abstract superclass of all mouse events. Its subclasses among the core classes are `MouseDownEvent`, `MouseUpEvent`, `MouseMoveEvent`, and `MouseCrossingEvent`.
- `KeyboardEvent` is the abstract superclass of all keyboard events. Its subclasses among the core classes are `KeyboardDownEvent` and `KeyboardUpEvent`.

Class Variables

Inherited from `Event`:

`interests` `numInterests`

The following class variables are defined in `QueuedEvent`:

dispatchQueue

self`.dispatchQueue` (read-write) `EventDispatchQueue`

Specifies the primary dispatch queue to use for events of the class *self*. All events that are generated by input devices, such as mouse events and keyboard events, belong to subclasses of `QueuedEvent`. The system creates a primary dispatch queue, an instance of `EventDispatchQueue`, which is stored in a global variable, `theUIEventDispatchQueue`, to handle such events. Normally, all user interface events should pass through this queue. The class variable `dispatchQueue` is available to allow for creation of a subclass of `QueuedEvent` that uses some other primary queue to process queued events.

Class Methods

Inherited from `Event`:

`acquireQueueFromPool` `relinquishQueueToPool`
`broadcastDispatch` `signalDispatch`

Instance Variables

Inherited from `Event`:

`advertised` `eventReceiver` `timeStamp`

authorData	matchedInterest
device	priority

The following instance variables are defined in QueuedEvent:

secondaryDispatchStyle

self.secondaryDispatchStyle (read-write) NameClass

A name indicating the style of dispatch used when this event leaves a dispatch queue. Possible values are @signalDispatchStyle and @broadcastDispatchStyle.

secondaryRejectable

self.secondaryRejectable (read-write) NameClass

Indicates whether or not the event will be rejectable when it is finally dispatched.

Instance Methods

Inherited from Event:

accept	isSatisfiedBy	sendToQueue
acquireRejectQueue	reject	signal
addEventInterest	relinquishRejectQueue	
broadcast	removeEventInterest	

The following instance methods are defined in QueuedEvent:

broadcast (Event)

broadcast *self* ⇨ Boolean

Sets secondaryDispatchStyle to @broadcastDispatchStyle, sets secondaryRejectable to false, and places the event *self* on the class's event dispatch queue. It returns true if delivery is successful.

secondaryBroadcast

secondaryBroadcast *self* ⇨ Boolean

Do not call from the scripter. This method is called when the event *self* leaves the primary dispatch queue, returning true if delivery is successful. It delivers the event using the rules normally used by broadcast. It is visible to the scripter to be overriden.

secondarySignal

secondarySignal *self rejectable* ⇨ Boolean

self	QueuedEvent object
rejectable	Boolean, indicating asynchronous delivery

Do not call from the scripter. This method is called when the event *self* leaves the primary dispatch queue, returning true if delivery is successful. If the value of *rejectable* is true, then delivery is asynchronous. It delivers the event using the rules normally used by signal. It is visible to the scripter to be overriden.

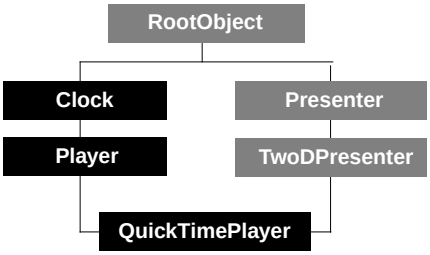
signal (Event)

signal *self rejectable* ⇨ Boolean

self	QueuedEvent object
rejectable	Boolean

Sets `secondaryDispatchStyle` to `@signalDispatchStyle`, sets `secondaryRejectable` to the value of *rejectable*, and places the event *self* on the class's event dispatch queue. If the value of *rejectable* is `true`, then delivery is asynchronous. It returns `true` if delivery is successful.

QuickTimePlayer



Class type: Loadable class (concrete)
Resides in: `qtplayer.lib`. Works with ScriptX and KMP executables
Inherits from: `Player` and `TwoDPresenter`
Component: Media Players

Kaleida Labs provides two loadable classes that can be used to play movies directly from hard disk without importing them into ScriptX. These classes are the `VFWPlayer` and the `QuickTimePlayer`. These classes provide ScriptX object wrappers to the Video For Windows runtime and the QuickTime runtime, respectively.

The `QuickTimePlayer` works on Windows and Macintosh computers. It can play QuickTime files directly, without importing them. The native QuickTime extension is responsible for doing everything from reading the video information off disk, deinterleaving it, decompressing video frames and displaying them on the screen. The ScriptX `QuickTimePlayer` class simply provides the movie with an area of screen real-estate on which to display the video and supports behavior similar to that of the ScriptX `MoviePlayer` class.

The `QuickTimePlayer` is provided as a loadable class and is not part of the ScriptX Core Classes. It must be explicitly loaded before it can be used.

The `QuickTimePlayer` classes inherits from `TwoDPresenter` and `Player`.

Be cautious when building a title that uses the `QuickTimePlayer` class. A title that depends on this player will not be portable to different platforms, and will only work so long as the files for the movies played by the player reside in the place where the player expects to find them. Note also that instances of loadable media player classes cannot be saved to the object store.

Loading the QuickTimePlayer Class

`QuickTimePlayer` is a scripted class that needs to be loaded dynamically before it can be used. The script files required to load the `QuickTimePlayer` class reside in a directory called `qtplayer` in a directory called `loadable`.

Loading the QuickTimePlayer Class

To load the `QuickTimePlayer` class on a Macintosh system, open the `loadme.sx` file in the `LOADABLE/QTPLAYER` directory using the **Open Title** command in the ScriptX **File** menu.

Before loading the `QuickTimePlayer` class on a Windows system, ensure that the following two criteria have been met:

1. The file `QTWPLYR.DLL` must reside in the ScriptX directory.

   Copy `QTWPLYR.DLL` from the `C:...\LOADABLE\QTPLAYER` directory to the ScriptX directory

2. QuickTime for Windows must be installed and the files for it must reside either in the Windows system directory, (`C:\WINDOWS\SYSTEM`) or in the ScriptX directory.

Then open the `loadme.sx` file in the `/LOADABLE/QTPLAYER` directory using the **Open Title** command in the ScriptX **File** menu.

Using the Loadable Media Player Classes

After loading a loadable Media Player Class, you can use it to play the appropriate media on the appropriate platform without importing the media into ScriptX.

Create an instance of the appropriate class by calling the `new` method on the appropriate class.

The following script is an example of how to create a new instance of the `QuickTimePlayer` class by calling the `new` method. :

```
myPlayer := new QuickTimePlayer \
       dir:theStartDir \
       filename:"RodDance" \
       masterClock:topPlayer
```

The variable `myPlayer` points to the newly created quicktime player, which can be used to play the movie "`RodDance`" that resides in the ScriptX startup directory. The player's master clock is `topPlayer`.

The `new` method uses the keywords defined in `init`.

You can play the movie associated with a loadable media player instance by using the usual `Player` and `MoviePlayer` methods. For example:

```
-- append the player to a visible window
global w := new windows boundary:myPlayer.bbox
show w
append w myPlayer

-- Play the movie associated with myPlayer from beginning to end
gotobegin myPlayer
playUntil myPlayer myPlayer.duration
```

init

`init` *self* [`dir:`*dirRep*] [`filename:`*string*] [`masterClock:`*topPlayer*] [`title:`*titleContainer*] ⇨ *self*

self	`QuickTimePlayer` object.
`dir:`	`DirRep` object for the directory containing the movie to be played.
`filename:`	`String` object for the name of the QuickTime file containing the movie to be played.

The `Clock` superclass uses the following keywords:

`masterClock:`	`Clock` object to be used as the master player
`scale:`	(Ignored by the loadable media player classes)
`title`:	(Ignored by the loadable media player classes. Loadable media player instances cannot be saved to title containers.)

Initializes the loadable media player object *self* so that it can play the movie in the file specified by `filename` which resides in the directory specified by `dir`. If supplied, `masterClock` is used as the master clock for the new player.

If you omit an optional keyword its default value is used. The defaults are:

```
dir:theStartDir
filename:undefined
masterClock:undefined (which means the root hardware clock)
```

Instance Variables

Inherited from Clock:

callbacks	rate	ticks
effectiveRate	resolution	time
masterClock	scale	title
offset	slaveClocks	

Inherited from Player:

audioMuted	globalContrast	globalVolumeOffset
dataRate	globalHue	markerList
duration	globalPanOffset	status
globalBrightness	globalSaturation	videoBlanked

Inherited from Presenter:

presentedBy	subPresenters	target

Inherited from TwoDPresenter:

bBox	globalBoundary	target
boundary	globalRegion	transform
changed	globalTransform	width
clock	height	window
compositor	imageChanged	x
direct	position	y
eventInterests	stationary	z

Instance Methods

Inherited from Clock:

addPeriodicCallback	clockAdded	pause
addRateCallback	clockRemoved	resume
addScaleCallback	effectiveRateChanged	timeJumped
addTimeCallback	forEachSlave	waitTime
addTimeJumpCallback	isAppropriateClock	waitUntil

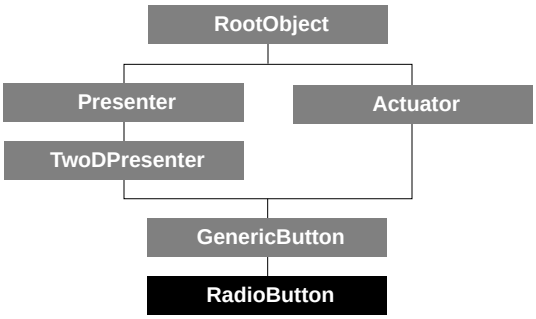
Inherited from Player:

addMarker	goToBegin	playPrepare
eject	goToEnd	playUnprepare
fastForward	goToMarkerFinish	playUntil
getMarker	goToMarkerStart	rewind
getNextMarker	pause	stop
getPreviousMarker	play	

Inherited from TwoDPresenter:

addChangedRegion	draw	refresh
adjustClockMaster	getBoundaryInParent	windowToLocal
createInterestList	localToWindow	

RadioButton



Class type: Scripted class (abstract)
Resides in: widgets.sxl. Works with ScriptX and KMP executables
Inherits from: GenericButton
Component: User Interface

RadioButton is a user interface Widget Kit class that provides a framed button whose appearance is defined by a stencil object that changes appearance when the button is selected, a text object that is displayed to the right of the button stencil, and a frame that encloses both the button stencil and the text. Clicking the mouse anywhere on or within the radio button's frame selects or deselects the button.

Three bitmaps give the button stencil different appearances for different mouse events. These bitmaps are stored in the media directory. The radio.bmp bitmap provides the appearance for the button stencil when the radio button is not selected; this is round with a black outline and gray fill. The radioC.bmp bitmap provides the appearance for the button stencil when the radio button is selected; this is the same as the unselected appearance but with a black disk in the center of the button stencil. The radioDP.bmp bitmap provides the appearance for the button stencil when the mouse button has been clicked on the radio button but has not yet been released; this is the same as the unselected appearance but with a black ring around the edge of the button stencil.

The clipping boundary of a RadioButton object is calculated automatically as a rectangle that encompasses the text stencil and the button stencil.

Creating and Initializing a New Instance

The following script is an example of how to create a new instance of the RadioButton class:

```
myRadioButton := new RadioButton \
       text:"On/Off" \
       frame:(new Frame)
```

The variable myRadioButton contains an initialized RadioButton object. The new method uses the keywords defined in init.

init

`init` *self* [`fill:`*brush*] [`font:`*font*] [`text:`*string*]
[`boundary:`*stencil*] [`frame:`*frame*] ⇨ *(none)*

self	`RadioButton` object
`fill:`	`Brush` object
`font:`	`Font` object
`text:`	`String` object
`boundary:`	`Stencil` object
`frame:`	`Frame` object

Initializes the `RadioButton` object *self*, applying the values supplied with the keywords to the instance variables of the same name. Creates a new `TextStencil` object to display the specified text in the specified font, and calculates a boundary that encompasses the text stencil and the button stencil. Do not call `init` directly on an instance — it is automatically called by the `new` method.

If you omit one of the keyword arguments, the following defaults are used:

```
fill:whiteBrush
font:theSystemFont
text:"Hello"
boundary:unsupplied
frame:undefined
```

You should not provide a value for `boundary`. If you do not provide a value for `boundary`, then the boundary of the new `RadioButton` object is calculated automatically as a rectangle that encompasses the text stencil and the button stencil.

Instance Variables

Inherited from `Actuator`:

`enabled`	`pressed`	`toggledOn`
`menu`		

Inherited from `Presenter`:

`presentedBy`	`subPresenters`	`target`

Inherited from `TwoDPresenter`:

`bBox`	`height`	`transform`
`boundary`	`IsImplicitlyDirect`	`width`
`clock`	`isTransparent`	`window`
`compositor`	`isVisible`	`x`
`direct`	`needsTickle`	`y`
`eventInterests`	`position`	`z`
`globalBoundary`	`stationary`	
`globalTransform`	`target`	

Inherited from `GenericButton`:

`activateAction`	`pressAction`	`releaseAction`
`authorData`		

The following instance variables are defined in `RadioButton`:

fill

self.fill (read-write) Brush

Specifies the brush, if any, used to fill the boundary (behind the button stencil and the text stencil) of the RadioButton object *self*. If this variable is set to undefined, the fill is transparent. Its default is whiteBrush.

font

self.font (read-write) Font

Specifies the font in which to render the text of the RadioButton object *self*.

frame

self.frame (read-write) Frame

Specifies the Frame object that is displayed around the RadioButton object *self*.

text

self.text (read-write) String

Specifies the string of characters to be displayed to the right of the RadioButton object *self*.

Instance Methods

Inherited from Actuator:

activate	press	toggleOff
multiActivate	release	toggleOn

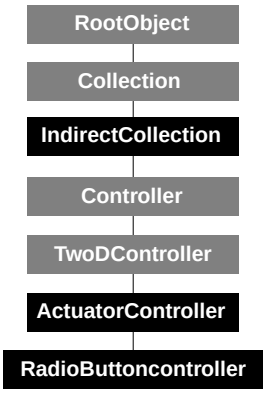
Inherited from TwoDPresenter:

adjustClockMaster	inside	show
createInterestList	localToSurface	surfaceToLocal
draw	notifyChanged	tickle
getBoundaryInParent	recalcRegion	
hide	refresh	

Inherited from GenericButton:

activate	press	release
multiActivate		

RadioButtonController



Class type: Core class (concrete)
Resides in: ScriptX and KMP executables
Inherits from: `ActuatorController`
Component: User Interface

The `RadioButtonController` class is used to control actuators so that only one actuator can be toggled on at once. These actuators operate as a group of "radio buttons." Radio buttons are common in user interface design. If the users toggles on one button in a group, then the button that was formerly chosen is toggled off automatically. (If you do not desire this behavior, use an `ActuatorController` object instead.)

`Toggle` objects are often used with a radio button controller because they can show two different states—toggled on and toggled off. Note that a toggle can appear any way you define it—for example, you could display a toggle as a bitmap image of a police officer holding up a stop sign when toggled off and a go sign when toggled on.

As an actuator controller, `RadioButtonController` manages a set of interests in mouse events, defined by `ActuatorController`. `RadioButtonController` specializes the `processActivate` method defined by `ActuatorController`. For a useful table that demonstrates how these event interests are mapped to the `activate`, `multiActivate`, `press`, and `release` methods defined by all actuators, see the class definition for `ActuatorController`.

Creating and Initializing a New Instance

The following script creates a new instance of `RadioButtonController`, after first creating a space:

```
mySpace := new TwoDSpace boundary:(new rect x2:120 y2:120)
myController := new RadioButtonController \
                  space:mySpace \
                  wholeSpace:true
```

This example creates a radio button controller, which will control actuators that are added to the space `mySpace`. The `new` method uses keywords defined in `init`.

init

`init` *self* [`space:`*space*] [`wholeSpace:`*boolean*] [`enabled:`*boolean*]
[`targetCollection:`*sequence*] ⇨ *(none)*

self `RadioButtonController` object

The superclass `Controller` uses the following keywords:

`space:`	`Space` object
`wholeSpace:`	`Boolean` object
`enabled:`	`Boolean` object

The superclass `TwoDController` uses the following keyword:

`targetCollection:`	`Sequence` object (use with caution)

Initializes the `RadioButtonController` object *self*, applying the keyword arguments to instance variables of the same name. Use discretion in changing the target collection; for more information, see the definition of the `TwoDController` class. Do not call `init` directly on an instance—it is automatically called by the `new` method.

If you omit an optional keyword, its default value is used. The defaults are:

```
space:undefined
wholeSpace:false
enabled:true
targetCollection:(new Array initialSize:14 growable:true)
```

Class Methods

Inherited from `Collection`:

`pipe`

Instance Variables

Inherited from `Collection`:

`bounded`	`maxSize`	`size`
`iteratorClass`	`minSize`	`uniformity`
`keyEqualComparator`	`mutable`	`uniformityClass`
`keyUniformity`	`mutableCopyClass`	`valueEqualComparator`
`keyUniformityClass`	`proprietored`	

Inherited from `IndirectCollection`:

`targetCollection`

Inherited from `Controller`:

`enabled`	`space`	`wholeSpace`
`protocols`		

Inherited from `ActuatorController`:

`activateInterest`	`pressInterest`	`releaseInterest`
`doubleClickTime`	`protocols`	

The following instance variables are defined in `RadioButtonController`:

chosenToggle

self`.chosenToggle` (read-write) `Actuator`

Contains the `Actuator` object that is currently toggled on—one actuator among the actuators that the radio button controller *self* is controlling. If this controller is not controlling anything, then the value of `chosenToggle` is `undefined`.

Setting a new value for `chosenToggle` automatically activates the chosen actuator and calls `toggleOff` on the actuator that was previous chosen.

Instance Methods

Inherited from Collection:

add	forEach	iterate
addMany	forEachBinding	localEqual
addToContents	getAll	map
chooseAll	getAny	merge
chooseOne	getKeyAll	pipe
chooseOneBinding	getKeyOne	prin
deleteAll	getMany	removeAll
deleteBindingAll	getOne	removeOne
deleteBindingOne	hasBinding	setAll
deleteKeyAll	hasKey	setOne
deleteKeyOne	intersects	size
deleteOne	isEmpty	
emptyOut	isMember	

Inherited from IndirectCollection:

isAppropriateObject	objectAdded	objectRemoved

Inherited from Controller:

isAppropriateObject	tickle

Inherited from ActuatorController:

processActivate	processPress	processRelease

Since a RadioButtonController object is an indirect collection, you can also use any methods defined in the class specified by targetCollection. The target collection is typically an instance of Array, which inherits from Sequence, so the following instance methods are redirected to this controller.

Accessible from LinearCollection:

chooseOneBackwards	findRange	getNthKey
chooseOrdOne	forEachBackwards	getOrdOne
deleteFirst	getFirst	getRange
deleteLast	getLast	localEqual
deleteNth	getMiddle	localLT
deleteRange	getNth	pop

Accessible from Sequence:

addFifth	moveBackward	setFourth
addFirst	moveForward	setLast
addFourth	moveToBack	setNth
addNth	moveToFront	setSecond
addSecond	prepend	setThird
addThird	prependNew	sort
append	setFifth	
appendNew	setFirst	

The following instance methods are defined in RadioButtonController:

objectAdded (IndirectCollection)

objectAdded *self key actuator* ⇨*self*

self	RadioButtonController object
key	Ordinal position of the object added
actuator	Actuator object

Do not call this method directly from the scripter. It is visible to the scripter so that it can be overridden in a subclass of `RadioButtonController`. When an *actuator* is added to the collection of objects controlled by the radio button controller *self*, `objectAdded` is triggered automatically. Since the target collection of a radio button controller is an array, an implicitly keyed collection, the actuator's key indicates its ordinal position in the collection after it is added. If `wholeSpace` is `true`, this method is invoked when an actuator is added to the control space. This method sets the value of `toggledOn` for the actuator being added to the space to `false`.

objectRemoved (IndirectCollection)

`objectRemoved` *self actuator* ⇨*self*

self	`RadioButtonController` object
actuator	`Actuator` object

Do not call this method directly from the scripter. It is visible to the scripter so that it can be overridden in a subclass of `RadioButtonController`. When an *actuator* is removed from the collection of objects controlled by the radio button controller *self*, `objectRemoved` is triggered automatically. If the actuator being removed from the space was the currently chosen toggle, this method sets the value of `chosenToggle` to return the first object it controls and calls `activate` on that object.

processActivate (ActuatorController)

`processActivate` *self interest event* ⇨*self*

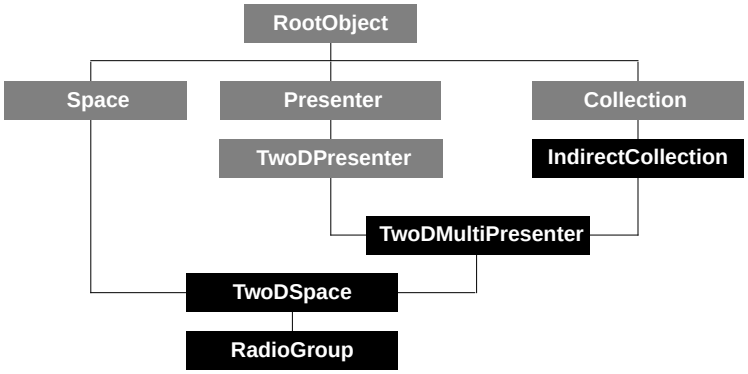
self	`RadioButtonController` object
interest	`MouseEvent` object that matches the `activateInterest`.
event	`MouseEvent` object that matches *interest*

Do not call this method directly from the scripter. It is visible to the scripter so that it can be overridden in a subclass of `RadioButtonController`. When an actuator controlled by the radio button controller *self* is activated, the `processActivate` method is triggered automatically. This method is an event receiver. Its event interest is assigned to the instance variable `activateInterest`, defined by `ActuatorController`. As an event receiver, it receives and processes events that match that interest.

If `wholeSpace` is `true`, `processActivate` looks through the collection of actuators controlled by the radio button controller *self* to determine which actuator was hit. If `wholeSpace` is `false`, `processActivate` determines which actuator was hit by examining the `presenter` instance variable of *event*.

If another actuator is currently toggled on, `processActivate` calls `toggleOff` on that object, and then calls `activate` on the actuator that was hit. This new actuator automatically becomes the currently chosen toggle. (See the `Actuator` and `Toggle` classes in the *ScriptX Class Reference*.) If no actuator is found—that is, if no hit occurred—then `processActivate` rejects the event.

RadioGroup



Class type: Scripted class (concrete)
Resides in: `widgets.sxl`. Works with ScriptX and KMP executables
Inherits from: `TwoDSpace`
Component: User Interface

`RadioGroup` is a user interface Widget Kit class that provides a box that displays a set of `RadioButton` objects. Only one of these buttons in the group can be selected at any given time.

The clipping boundary of a `RadioGroup` object is calculated automatically as a rectangle that encompasses the entire set of radio buttons.

Creating and Initializing a New Instance

The following script is an example of how to create a new instance of the `RadioGroup` class:

```
myRadioGroup := new RadioGroup \
          itemList:#(@one:"One", @two:"Two", @three:"Three")
```

The variable `myRadioGroup` contains an initialized `RadioGroup` object, which in this case is a box that displays three radio buttons in a vertical format. The `new` method uses the keywords defined in `init`.

init

`init` *self* [`boundary:`*stencil*] [`font:`*font*] [`itemList:`*collectionOfItems*]
[`itemClass:`*buttonClass*] [`itemBoundary:`*stencil*] [`fill:`*brush*]
[`orientation:`*verticalOrHorizontal*] ⇨ *(none)*

self	`RadioGroup` object
`controller:`	`RadioButtonController` object
`itemList:`	`Array` object
`value:`	`Integer` object
`frame:`	`Frame` object

Superclasses of `RadioGroup` use the following keywords:

`boundary:`	`Stencil` object
`fill:`	`Brush` object
`stroke:`	`Brush` object
`scale:`	`Integer` object
`target:`	Ignored by `RadioGroup`
`targetCollection:`	`Sequence` object

Initializes the RadioGroup object *self*, applying the values supplied with the keywords to the instance variables of the same name. Creates a box to display all the buttons in itemList, and calculates a boundary that encompasses all those buttons. Do not call init directly on an instance — it is automatically called by the new method.

If you omit one of the keyword arguments, the following defaults are used:

```
boundary:unsupplied
font:theSystemFont
itemList:unsupplied
itemClass:RadioButton
itemBoundary:unsupplied
fill:whiteBrush
orientation:@vertical
```

You should not provide a value for boundary. If you do not provide a value for boundary, then the boundary of the new RadioGroup object is calculated automatically as a rectangle that encompasses all the buttons in the group.

Class Methods

Inherited from Collection:

pipe

Instance Variables

Inherited from Collection:

bounded	maxSize	size
iteratorClass	minSize	uniformity
keyEqualComparator	mutable	uniformityClass
keyUniformity	mutableCopyClass	valueEqualComparator
keyUniformityClass	proprietored	

Inherited from IndirectCollection:

targetCollection

Inherited from Presenter:

presentedBy	subPresenters	target

Inherited from TwoDPresenter:

bBox	height	transform
boundary	IsImplicitlyDirect	width
clock	isTransparent	window
compositor	isVisible	x
direct	needsTickle	y
eventInterests	position	z
globalBoundary	stationary	
globalTransform	target	

Inherited from TwoDMultiPresenter:

clock	fill	stroke

Inherited from Space:

clock	controllers	tickleList

Inherited from TwoDSpace:

protocols

The following instance variables are defined in RadioGroup:

controller

self.controller (read-write) RadioButtonController

Specifies the RadioButtonController object that will control the RadioGroup object *self*.

frame

self.frame (read-write) Frame

Specifies the Frame object that encompasses the RadioGroup object *self*.

itemList

self.itemList (read-write) Array

Specifies the objects that are listed in the RadioGroup object *self*.

value

self.value (read-write) Integer

Specifies which item in itemList in the RadioGroup object *self* is selected by default, if any.

Instance Methods

Inherited from Collection:

add	forEach	iterate
addMany	forEachBinding	localEqual
addToContents	getAll	map
chooseAll	getAny	merge
chooseOne	getKeyAll	pipe
chooseOneBinding	getKeyOne	prin
deleteAll	getMany	removeAll
deleteBindingAll	getOne	removeOne
deleteBindingOne	hasBinding	setAll
deleteKeyAll	hasKey	setOne
deleteKeyOne	intersects	size
deleteOne	isEmpty	
emptyOut	isMember	

Inherited from IndirectCollection:

isAppropriateObject	objectAdded	objectRemoved

Inherited from TwoDPresenter:

adjustClockMaster	inside	show
createInterestList	localToSurface	surfaceToLocal
draw	notifyChanged	tickle
getBoundaryInParent	recalcRegion	
hide	refresh	

Inherited from TwoDMultiPresenter:

draw	findFirstInStencil	moveToBack
findAllAtPoint	isAppropriateObject	moveToFront
findAllInStencil	moveBackward	objectAdded
findFirstAtPoint	moveForward	objectRemoved

Inherited from Space:

isAppropriateObject	objectAdded	objectRemoved

Since a Window object is an indirect collection, you can also use any methods defined in the class specified by targetCollection. The target collection is typically an instance of Array, which inherits from Sequence, so the following instance methods are redirected to a window:

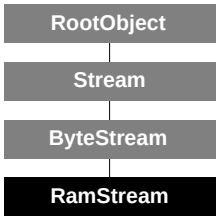
Accessible from LinearCollection:

chooseOneBackwards	deleteSecond	getMiddle
chooseOrdOne	deleteThird	getNth
deleteFifth	findRange	getNthKey
deleteFirst	forEachBackwards	getOrdOne
deleteFourth	getFifth	getRange
deleteLast	getFirst	getSecond
deleteNth	getFourth	getThird
deleteRange	getLast	pop

Accessible from Sequence:

addFifth	moveBackward	setFourth
addFirst	moveForward	setLast
addFourth	moveToBack	setNth
addNth	moveToFront	setSecond
addSecond	prepend	setThird
addThird	prependNew	sort
append	setFifth	
appendNew	setFirst	

RamStream



Class type: Core class (concrete)
Resides in: ScriptX and KMP executables
Inherits from: `ByteStream`
Component: Streams

The `RamStream` class provides a way to hold data in a RAM buffer, generally for some temporary data.

Creating and Initializing a New Instance

The following script is an example of how to create a new instance of the `RamStream` class:

```
myBuffer := new RamStream \
              maxSize:1000
```

The variable `myBuffer` contains the new `RamStream`. Its maximum size is 1000 bytes. The `new` method uses the keywords defined in `init`.

init

`init` *self* [`max:` *integer*] ⇨ *(none)*

self	`RamStream` object
`maxSize:`	`Integer` object representing maximum size of buffer

Initializes the `RamStream` object *self*, applying the value supplied with the `maxSize` keyword as the maximum size of the buffer represented. Do not call `init` directly on an instance—it is automatically called by the `new` method.

If you omit an optional keyword, its default value is used. The defaults are:

`maxSize:500`

Instance Methods

Inherited from `Stream`:

`cursor`	`next`	`seekFromStart`
`flush`	`plug`	`setStreamLength`
`isAtFront`	`previous`	`streamLength`
`isPastEnd`	`read`	`write`
`isReadable`	`readReady`	`writeReady`
`isSeekable`	`seekFromCursor`	
`isWritable`	`seekFromEnd`	

Inherited from `ByteStream`:

`fileIn`	`readByte`	`writeString`
`pipe`	`readReady`	
`pipePartial`	`writeByte`	

The following instance methods are implemented by `RamStream`:

flush (Stream)

`flush` *self* ⇨ `true`

Inherited from `Stream`, this method is overridden in `RamStream` to return `true`.

isReadable (Stream)

`isReadable` *self* ⇨ `true`

Inherited from `Stream`, this method is overridden in `RamStream` to return `true`.

isSeekable (Stream)

`isSeekable` *self* ⇨ `true`

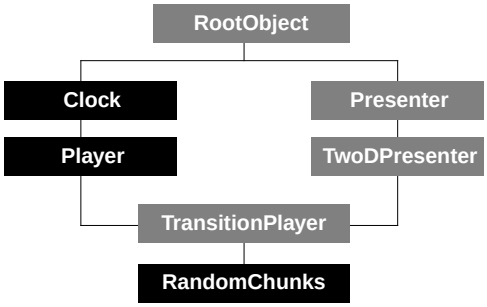
Inherited from `Stream`, this method is overridden in `RamStream` to return `true`.

isWritable (Stream)

`isWritable` *self* ⇨ `true`

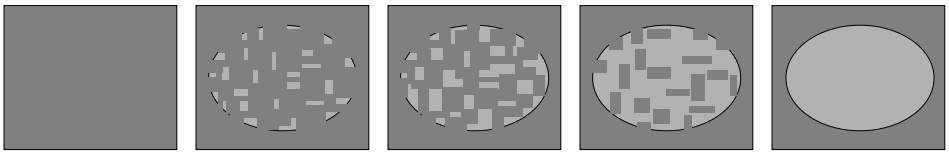
Inherited from `Stream`, this method is overridden in `RamStream` to return `true`.

RandomChunks



Class type: Loadable class (concrete)
Resides in: `ltrans.lib`. Works with ScriptX and KMP executables
Inherits from: `TransitionPlayer`
Component: Transitions

The `RandomChunks` transition player provides a visual effect where the target gradually appears as random-size squares, rectangles, columns, or rows. The `@rects` direction playing forward is shown below. The target appears when the transition is played forward, and disappears when played backward (transition's rate set to -1).



Directions: `@squares`, `@rects`, `@columns`, `@rows`

Rate: Must be zero or positive. Cannot play backward.

For a side-by-side illustrations of all transitions, see the Transitions chapter in the *ScriptX Components Guide*.

Creating and Initializing a New Instance

The following script is an example of how to create a new instance of the `RandomChunks` class:

```
myTransition := new RandomChunks \
                    duration:60 \
                    direction:@squares \
                    target:myShape
```

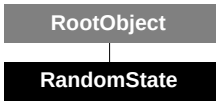
The variable `myTransition` contains the initialized transition. The transition reveals the image `myShape` in squares in a random distribution, and has a duration of 60 ticks.

You determine which space the transition will take effect in by adding this instance into that space. Then, when you play the transition player, `myShape` is transitioned into that space.

The `new` method uses the keywords defined in `init`.

NOTE – For the instance variables and methods, see the `BarnDoor` class.

RandomState



Class type: Core class (concrete, sealed)
Resides in: ScriptX and KMP executables
Inherits from: `RootObject`
Component: Numerics

The `RandomState` class lets you create a random-state generator for generating random numbers. You initialize the random-state generator with a seed. To obtain a random number, call `random` on the random-state generator, supplying a maximum value for the result. Note that a `RandomState` object is not itself a number—it generates numbers.

The seed you specify when creating a random-state generator determines a repeatable sequence of random numbers. That is, reseeding a generator with the same seed (using `randomize`), or creating a new generator with the same seed produces the same sequence of random numbers. For example, this script creates two sequences of random numbers. The random numbers produced from the two print statements are identical and include numbers from the set (0, 1, 2, 3, 4, 5, 6, 7, 8, 9) but never 10:

```
rs := new RandomState seed:100 -- use the seed 100
for 5 do (print (random rs 10)) -- prints one set of numbers

randomize rs 100 -- use the same seed 100
for 5 do (print (random rs 10)) -- prints the same set
```

To produce a different sequence of random numbers each time a program is run, you can seed the random-state generator with a number derived from the current date and time.

Creating and Initializing a New Instance

The following script is an example of how to create a new instance of the `RandomState` class:

```
randomGen := new RandomState seed:10.5
random randomGen -5.2
```

The variable `randomGen` contains the initialized random state generator seeded with an initial seed of 10. The second statement uses this random state generator to return a random number between 0 and –5.2. The `new` method uses the keywords defined in `init`.

init

`init` *self* [`seed:`*number*] ⇨ *(none)*

self	`RandomState` object
`seed:`	`Number` to initialize the random state generator

Initializes the `RandomState` object *self*, where `seed` specifies the number used to start the random sequence (not the first value generated). The sign of the seed is ignored, but it can be any kind of number.

To obtain a random number, use the `random` method on this random-state generator. Each seed generates a unique, but repeatable, sequence of random numbers. Using the same seed value for another generator (or for `randomize`) will produce the same sequence of random numbers. Do not call `init` directly on an instance—it is automatically called by the `new` method.

If you omit an optional keyword, its default value is used:

```
seed:1
```

Instance Methods

random

`random` *self max* ⇨ `Number`

self	`RandomState` object
max	`Number` object, the limit to the random number

Returns a random number from the random-number generator *self*. The random number returned has the same class and sign as the number specified by *max*, with a value between 0 and *max*, including 0 but not including *max*. For example, if value *max* is a negative float, then the random number returned is a negative float.

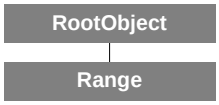
randomize

`randomize` *self seed* ⇨ `Number`

self	`RandomState` object to be randomized
seed	`Number` object to re-initialize the generator

Re-initializes the random-number generator *self* with the value *seed*. This method has the same effect as creating a new `RandomState` object with this seed. The sequence of numbers generated for a particular seed is deterministic and repeatable.

Range



Class type: Core class (abstract)
Resides in: ScriptX and KMP executables
Inherits from: `RootObject`
Component: Collections

The `Range` class provides a protocol for objects that incorporate a range or set of values. Each element that a range incorporates must be an instance of its value class. Unlike collections, in which each element is a separate object, a range stores only information about its upper and lower bounds, and its increment.

Since `Range` is an abstract class, it does not define an `init` method. Each concrete subclass of `Range` defines its own `init` method.

Instance Variables

includesLower

self`.includesLower` (read-only) `Boolean`

Is `true` if the lower bound is part of the range *self*. This instance variable is read-only for immutable subclasses. Although all subclasses of `Range` in the core classes are immutable, a script could create a mutable subclass.

includesUpper

self`.includesUpper` (read-only) `Boolean`

Is `true` if the upper bound is part of the range *self*. This instance variable is read-only for immutable subclasses. Although all subclasses of `Range` in the core classes are immutable, a script could create a mutable subclass.

increment

self`.increment` (read-only) `Number`

Specifies the increment (step-size) of the range *self*. The value of `increment` can be the special value `@continuous`, indicating that all possible values between the upper and lower bounds are elements of the range.

lowerBound

self`.lowerBound` (read-only) *(object)*

Specifies the lower bound of the range *self*. The class of the object specified by `lowerBound` must be the same as the class returned by the `valueClass` method.

Specifies the lower bound of the range *self*. The value of `lowerBound` must be an instance of the class specified by `valueClass`. If the value class is `Number`, the value of `lowerBound` can be `posInf` (positive infinity) or `negInf` (negative infinity).

In ranges that are not continuous, the values of `upperBound` and `increment` interact such that setting this value can actually set the upper bound to something less. For example, given a range with `lowerBound` set to 0 and `upperBound` set to 13, setting `increment` to 3 will actually leave the upper bound at 12.

size

self.`size` (read-only) `Integer`

Specifies the number of elements in the range *self*. For ranges that are unbounded, and for continuous ranges, the value of `size` is not defined. If the number of elements is countable and bounded, the value of `size` is an `Integer` object.

upperBound

self.`upperBound` (read-only) *(object)*

Specifies the upper bound of the range *self*. The value of `upperBound` must be an instance of the class specified by `valueClass`. If the value class is `Number`, the value of `upperBound` can be `posInf` (positive infinity) or `negInf` (negative infinity).

In ranges that are not continuous, the values of `upperBound` and `increment` interact such that setting this value can actually set the upper bound to something less. For example, given a range with `lowerBound` set to 0 and `upperBound` set to 13, setting `increment` to 3 will actually leave the upper bound at 12.

valueClass

self.`valueClass` (read-only) *(class)*

Specifies a class that restricts that set of values which are allowed within a range. Values within the range *self* must be instances of its value class. For example, all values within a continuous number range must be numbers.

Instance Methods

withinRange

`withinRange` *self value* ⇨ `Boolean`

self	`Range` object
value	Any object

Returns `true` if *value* is within the range *self*; otherwise, it returns `false`.

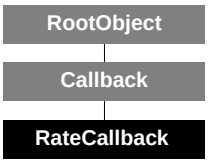
Subclasses Must Implement

Subclasses of `Range` must implement the following methods. They will report an exception if called but not implemented.

```
includesLowerGetter
includesUpperGetter
incrementGetter
init
lowerBoundGetter
sizeGetter
upperBoundGetter
valueClassGetter
withinRange
```

Subclasses of `Range` may define setter methods for instance variables in the Range protocol, or set initial values through the `init` method. Instance variables defined by ranges are writable only if the range is mutable, and if a setter method is defined for the variable. Although it is possible to create a range that is mutable, all ranges defined in the core classes are immutable.

RateCallback



Class type: Core class (concrete)
Resides in: ScriptX and KMP executables
Inherits from: `Callback`
Component: Clocks

The `RateCallback` class is used to represent actions scheduled to be performed when the clock's rate changes.

You never create an instance of `RateCallback` directly. Instead, you request a callback from a clock using the `addRateCallback` method defined by the `Clock` class.

The `order` instance variable, inherited from `Callback`, has no effect with a `RateCallback` instance.

Instance Variables

Inherited from `Callback`:

`authorData`	`onceOnly`	`script`
`condition`	`order`	`target`
`label`	`priority`	

The following instance variables are defined in `RateCallback`:

condition

self`.condition` (read-write) `NameClass`

Determines the condition under which the rate callback *self* will be activated. The following values are valid for `RateCallback` objects:

```
@lessThan
@greaterThan
@equal
@notEqual
@lessThanOrEqual
@greaterThanOrEqual
@change (default)
```

rate

self`.rate` (read-write) `NameClass`

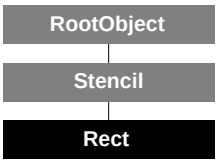
The value used in comparisons by the callback condition when the rate changes.

Instance Methods

Inherited from `Callback`:

`cancel`

Rect



Class type: Core class (concrete)
Resides in: ScriptX and KMP executables
Inherits from: `Stencil`
Component: 2D Graphics

The `Rect` class is a subclass of `Stencil` used to represent areas of a coordinate system and to render rectangular images. The area represented by an image rendered by a `Rect` instance is determined by its upper-left and lower-right corner points. In the ScriptX imaging model, the largest pixel values defining a rectangle are excluded from the resulting image. For example, if a rectangle is defined by the corner points 0,0 and 20,20, pixels with x or y values of 20 aren't included in the image area.

Note that `Rect` objects are not presenters—to display a rectangle, create an instance of `TwoDShape` using a `Rect` object as the *boundary* argument.

Creating and Initializing a New Instance

The following script is an example of how to create a new instance of the `Rect` class:

```
myRect := new Rect \
          x1:0 \
          y1:0 \
          x2:50 \
          y2:50
```

The variable `myRect` contains the new `Rect` object. Its upper left corner is at 0,0 and its lower right corner is at 50,50. Notice that `x1` and `y1` default to `0` and so are optional here. The `new` method uses the keywords defined in `init`.

init

`init` *self* [`x1:`*number*] [`y1:`*number*] [`x2:`*number*] [`y2:`*number*] ⇨ *(none)*

self	`Rect` object
`x1:`	`Number` object
`y1:`	`Number` object
`x2:`	`Number` object
`y2:`	`Number` object

Initializes the `Rect` object *self*, applying the arguments as follows: `x1` and `y1` represent one corner of the rectangle, `x2` and `y2` represent the opposite corner. The values supplied with the keywords are applied to instance variables of the same name. Do not call `init` directly on an instance—it is automatically called by the `new` method.

If you omit an optional keyword, its default value is used. The defaults are:

```
x1:0
y1:0
x2:0
y2:0
```

Instance Variables

Inherited from `Stencil`:

`bBox`

The following instance variables are defined in `Rect`:

height

self`.height` (read-write) `Number`

Specifies the height of the rectangle *self*. Setting `height` resets `y2` and keeps `x1`, `y1`, and `x2` constant.

width

self`.width` (read-write) `Number`

Specifies the width of the rectangle *self*. Setting `width` resets `x2` and keeps `x1`, `y1`, and `y2` constant.

x1

self`.x1` (read-write) `Number`

Specifies the x coordinate of the upper-left corner of the rectangle *self*.

x2

self`.x2` (read-write) `Number`

Specifies the x coordinate of the lower-right corner of the rectangle *self*.

y1

self`.y1` (read-write) `Number`

Specifies the y coordinate of the upper-left corner of the rectangle *self*.

y2

self`.y2` (read-write) `Number`

Specifies the y coordinate of the lower-right corner of the rectangle *self*.

Instance Methods

Inherited from `Stencil`:

`inside`	`onBoundary`	`transform`
`intersect`	`subtract`	`union`

The following instance methods are defined in `Rect`:

copy

`copy` *self* ⇨ `Rect`

Creates and returns a copy of the rectangle *self*.

invTransform

`invTransform` *self matrix mutateOrCreate* ⇨ `Path`

self	`Stencil` to transform
matrix	`TwoDMatrix` object representing the transformation
mutateOrCreate	`NameClass` object specifying the desired result: `@mutate` or `@create`

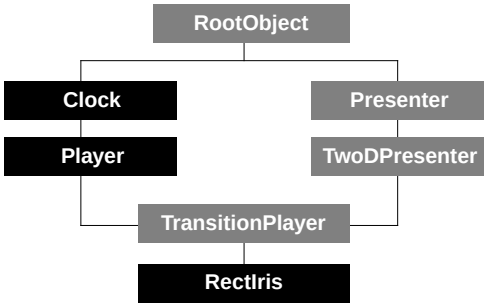
Transforms the `Rect` *self* by the inverse of the `TwoDMatrix` object *matrix*. The returned value is a new `Path` obejct. (It seems that when transforming or inversely transforming a Rect object, the result is always a new `Path` obje ct, regardless of the value of *mutateOrÇreate*.)

moveToZero

`moveToZero` *self* ⇨ `Rect`

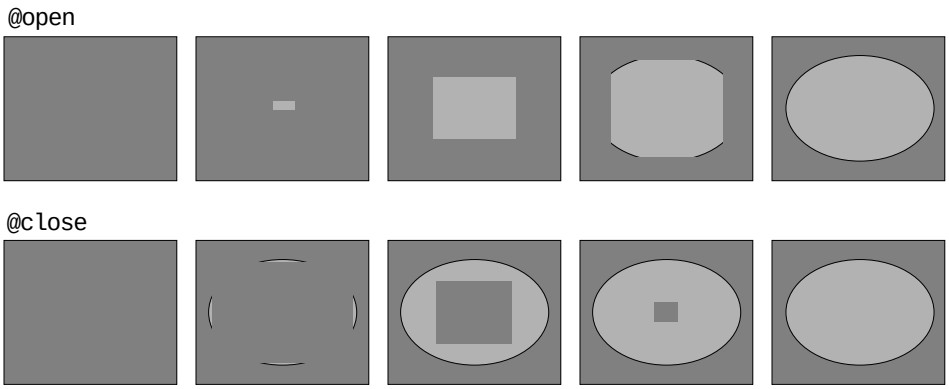
Repositions the rectangle *self* so that the values of `x1` and `y1` are 0, adjusting the values of `x2` and `y2` to maintain their original offset.

RectIris



Class type: Loadable class (concrete)
Resides in: `ltrans.lib`. Works with ScriptX and KMP executables
Inherits from: `TransitionPlayer`
Component: Transitions

The `RectIris` transition player provides a visual effect where the target gradually appears either from the center outward or from the edges inward, as shown below. The iris's ratio of width-to-height is the same as the target's width-to-height. The target appears when the transition is played forward, and disappears when played backward (transition's rate set to -1).



Directions: `@open`, `@close`

Rate: Can play forward or backward.

For a side-by-side illustrations of all transitions, see the Transitions chapter in the *ScriptX Components Guide*.

Creating and Initializing a New Instance

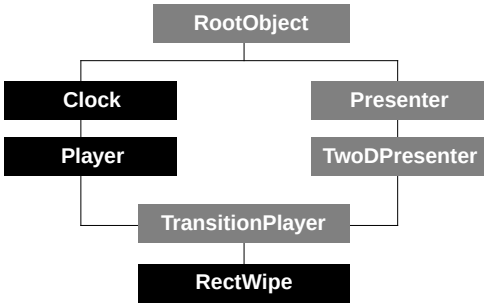
The following is an example of how to create a new instance of the `RectIris` class:

```
myTransition := new RectIris \
                    duration:60 \
                    direction:@open \
                    target:myShape
```

The transition reveals the image `myShape` from the center first, in a rectangular shape, and has a duration of 60 ticks. You determine which space the transition will take effect in by adding this instance into that space. Then, when you play the transition, `myShape` is transitioned into that space.

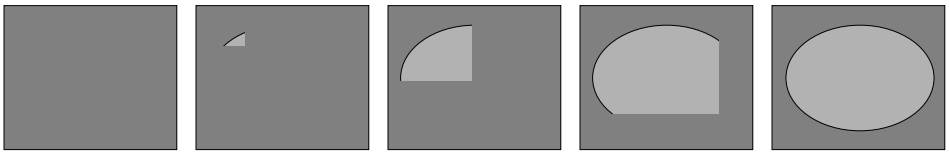
NOTE – For the instance variables and methods, see the `BarnDoor` class.

RectWipe



Class type: Loadable class (concrete)
Resides in: `ltrans.lib`. Works with ScriptX and KMP executables
Inherits from: `TransitionPlayer`
Component: Transitions

The `RectWipe` transition player provides a visual effect where the target gradually appears, wiped into view toward the given direction. The `@southeast` direction playing forward is shown below. The target appears when the transition is played forward, and disappears when played backward (transition's rate set to -1).



Directions: `@northeast`, `@southeast`, `@southwest`, `@southeast`

Rate: Can play forward or backward.

For a side-by-side illustrations of all transitions, see the Transitions chapter in the *ScriptX Components Guide*.

Creating and Initializing a New Instance

The following script is an example of how to create a new instance of the `RectWipe` class:

```
myTransition := new RectWipe \
                    duration:60 \
                    direction:@southeast \
                    target:myShape
```

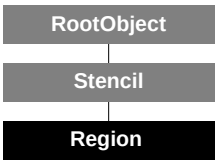
The variable `myTransition` contains the initialized transition. The transition reveals the image `myShape` in a rectangular shape toward the southeast direction, and has a duration of 60 ticks.

You determine which space the transition will take effect in by adding this instance into that space. Then, when you play the transition player, `myShape` is transitioned into that space.

The `new` method uses the keywords defined in `init`.

NOTE – For the instance variables and methods, see the `BarnDoor` class.

Region



Class type: Core class (concrete)
Resides in: ScriptX and KMP executables
Inherits from: `Stencil`
Component: 2D Graphics

`Region` is a general concrete subclass of `Stencil`, used within ScriptX to represent rendered images. `Regions` are often returned by operations on stencils, such as `union`, `intersect`, and `transform`, when the result can't be represented efficiently by another `Stencil` subclass. Regions can be used in many places where a stencil is required.

Creating and Initializing a New Instance

You seldom create an instance of `Region` directly. One exception is in the `TwoDPresenter` method `addChangedRegion`; if you override this method, you may need to create and return a new `Region`. The following script is an example of how to create a new instance of the `Region` class:

```
myRegion := new Region
```

Instance Variables

Inherited from `Stencil`:

`bBox`

The following instance variables are defined in `Region`:

height

self`.height` (read-only) `Number`

Specifies the height of the region *self*.

width

self`.width` (read-only) `Number`

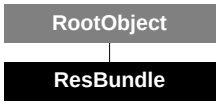
Specifies the width of the region *self*.

Instance Methods

Inherited from `Stencil`:

`inside`	`onBoundary`	`transform`
`intersect`	`subtract`	`union`

ResBundle



Class type: Core class (concrete)
Resides in: ScriptX and KMP executables
Inherits from: `RootObject`
Component: Files

The `ResBundle` class represents Macintosh resource files. Importing media from resource files is performed by first invoking `new` on the `ResBundle` class, then using the `getOneStream` method to open a stream for a specific resource in the bundle. Once the stream has been created from the resource file, you can import the resource using an instance of the `importExportEngine`. See the chapter on importers in the *Developer's Guide* for more on importing streams of media data into ScriptX.

Creating and Initializing a New Instance

The following sample script creates a new instance of the `ResBundle` class:

```
myBundle := new ResBundle \
       dir:startdir \
       path:"filename"
```

The variable `myBundle` contains the new `ResBundle` object. This object represents a resource file whose name is "filename", found in the directory represented by the `dirRep` global `startDir`. The `new` method uses the keywords defined in `init`.

init

`init` *self* `dir:`*dirRep* `path:`*collectionOrString* [`mode:`*name*] ⇨ *(none)*

self	`ResBundle` object
`dir:`	`DirRep` object
`path:`	`KeyedCollection` or `String` object
`mode:`	`NameClass` object representing file access mode

Initializes and opens a new `ResBundle` object *self*, or opens a new file for an instance of `ResBundle`, applying the keyword arguments, as follows:

- The keyword `dir` specifies a `DirRep` object representing a complete or partial path to the directory containing the resource file.
- The keyword `path` can take as its argument an object representing the subdirectory path and filename of the new `ResBundle` object. The argument can be either a `String` or `KeyedCollection` in one of the following forms:

```
"media folder/bitmap folder/someImageFile"
#("media folder","bitmap folder","someImageFile")
```

- The keyword `mode` specifies the access mode for the resource file represented by the `ResBundle` instance. The argument may be one of `@readable`, `@writable`, or `@readwrite`. By default, the file is open in readable mode.

Do not call `init` directly on an instance—it is automatically called by the `new` method.

Instance Variables

streams

self.streams (read-only) Array

An array of all streams open for access to resources within the file represented by the ResBundle *self*.

writable

self.writable (read-only) Boolean

Specifies whether the file associated with the resource bundle *self* is opened for writing.

Instance Methods

closeBundle

closeBundle *self* ⇨ *(none)*

Closes the file for the resource bundle *self*. You must explicitly plug any stream opened on the resource bundle before invoking this method; otherwise, it reports an exception.

getOneStream

getOneStream *self* [type:*string*] [name:*string* | ID:*number* | index:*number*] ⇨ ResStream

self	ResBundle instance supplying the stream
type:	String object specifying the type of the resource
name:	String object, the name of the resource
ID:	Number object, the identity of the resource
index:	Number object, the array index of the resource

Creates and returns a stream for reading a specified resource from the file represented by ResBundle object *self*. The resource type is specified by the type keyword. Use one of the alternate keywords name, ID, or index to specify the particular resource.

getStreamList

getStreamList *self* [type:*string*] ⇨ Array

self	ResBundle instance
type:	String object specifying resource type

Returns an array whose contents represent all resources of the type specified in the *string* argument to the type keyword. This elements in this array are triples in the form #(*type*, *ID*, *name*), where *type* represents the type of a resource, *ID* represents the identity of that resource, and *name* represents the name of that resource.

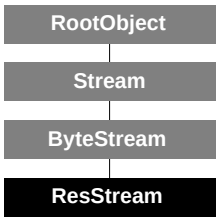
makeOneStream

makeOneStream *self* [type:*string*] [name:string | ID:*number*] ⇨ ResStream

self	ResBundle instance creating the stream
type:	String object specifying the type of the resource
name:	String object representing the name of the resource
ID:	Number object representing the identity of the resource

Creates and returns a stream for writing the specified resource to the file represented by the resource bundle *self*. The resource type is specified by the type keyword. Use one of the alternate keywords name or ID to specify the particular resource within the file.

ResStream



Class type: Core class (concrete)
Resides in: ScriptX and KMP executables
Inherits from: `ByteStream`
Component: Streams

The `ResStream` class defines the streams returned by instance methods of the `ResBundle` class from Macintosh system resource files.

Instance Methods

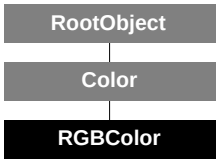
Inherited from `Stream`:

`cursor`	`next`	`seekFromStart`
`flush`	`plug`	`setStreamLength`
`isAtFront`	`previous`	`streamLength`
`isPastEnd`	`read`	`write`
`isReadable`	`readReady`	`writeReady`
`isSeekable`	`seekFromCursor`	
`isWritable`	`seekFromEnd`	

Inherited from `ByteStream`:

`fileIn`	`readByte`	`writeString`
`pipe`	`readReady`	
`pipePartial`	`writeByte`	

RGBColor



Class type: Core class (concrete)
Resides in: ScriptX and KMP executables
Inherits from: `Color`
Component: 2D Graphics

The `RGBColor` class represents colors in a general RGB or grayscale color space where each color component (red, green, and blue) is represented by 8 bits of information. For grayscale color spaces, all three color components are set to the same value.

ScriptX provides eight global instances of `RGBColor`: `blackColor`, `whiteColor`, `blueColor`, `redColor`, `greenColor`, `cyanColor`, `magentaColor`, and `yellowColor`. These instances are defined in the chapter "Global Constants and Variables."

Creating and Initializing a New Instance

The following script is an example of how to create a new instance of the `RGBColor` class:

```
myColor := new RGBColor \
           red:255 \
           green:0 \
           blue:0
```

The variable `myColor` contains the new `RGBColor` object. This object has a red value of 255, a green value of 0, and a blue value of 0. The `new` method uses the keywords defined in `init`.

init

`init` *self* `red:`*number* `green:`*number* `blue:`*number* ⇨ *(none)*

self	`RGBColor` object
`red:`	`Number` object
`green:`	`Number` object
`blue:`	`Number` object

Initializes the `RGBColor` object *self*, applying the arguments as follows: `red` represents the red value, `green` represents the green value, and `blue` represents the blue value. These values may be floats between 0.0 and 1.0 or integers between 0 and 255, where the least significant bits are thrown away as appropriate to fit the corresponding component. However, upon instantiation, all values are converted to integers between 0 and 255. To modify the corresponding instance variables after instantiation, use integer values. Do not call `init` directly on an instance—it is automatically called by the `new` method.

The `RGBColor` `init` method has no optional keywords.

Instance Variables

blue

self.blue (read-write) Integer

Specifies the integer value of the blue component of the RGB color *self*. Valid values are in the range of 0 to 255.

green

self.green (read-write) Integer

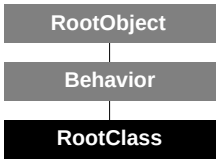
Specifies the integer value of the green component of the RGB color *self*. Valid values are in the range of 0 to 255.

red

self.red (read-write) Integer

Specifies the integer value of the red component of the RGB color *self*. Valid values are in the range of 0 to 255.

RootClass



Class type: Core class (concrete)
Resides in: ScriptX and KMP executables
Inherits from: `Behavior`
Component: Object System Kernel

The class named `RootClass` is the superclass of all metaclasses. It contains the methods that are common to all metaclasses. Moreover, it is deemed by convention to represent all the classes in the system. Its class method `new` creates a class (unlike the `new` method in the `RootObject` class that creates an instance of a class).

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 ScriptX 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.

Creating and Initializing a New Instance

You create a new class object by invoking `new` on the class `RootClass`. However, the mechanism exported to the scripting language is the class construction construct. An example of its use is shown below:

```
class CoercedArray (Array)
    inst vars
        coerceTo
    inst methods
        method init self #rest args #key coerceTo: -> (
            self.coerceTo := coerceTo
            return (apply nextMethod self args)
        )
        method append self val ->
            return (apply nextMethod self (coerce val self.coerceTo))
end
```

This example creates a new class named `CoercedArray`. It is defined to inherit from `Array`, it has a single instance variable, `coerceTo`, and it specializes the methods `init` and `append`. Instances of the class will coerce all elements appended to the list to the same class before storing the value in the list.

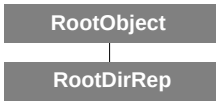
To create an instance of `RootClass`, use the class constructor form as shown in the previous example. See *ScriptX Language Guide* for the full syntax.

Instance Methods

Inherited from `Behavior`:

`canClassDo`	`getSupers`	`methodBinding`
`getDirectSubs`	`isDirectSub`	`new`
`getDirectSupers`	`isMemberOf`	
`getSubs`	`isSub`	

RootDirRep



Class type: Core class (abstract)
Resides in: ScriptX and KMP executables
Inherits from: `RootObject`
Component: Files

The `RootDirRep` class is an abstract class that provides an object that represents the root directory of file systems that represents individual storage devices as volumes (such as the Macintosh System) or drives (such as DOS). On these systems, ScriptX includes a file-system-specific subclass of `RootDirRep` to implement machine-dependent behavior such as `HFSRootDirRep` (Macintosh) and `FATRootDirRep` (Windows).

A global constant, `theRootDir`, is created at startup by the ScriptX runtime environment to represent the root directory of a platform's file system. You can use the methods defined by the `DirRep` class to access actual files and directories through `theRootDir`.

RootObject

RootObject

Class type: Core class (abstract)
Resides in: ScriptX and KMP executables
Inherits from: *none*
Component: Object System Kernel

`RootObject` is the root superclass of all classes; therefore, it has no superclass.

All classes (and metaclasses) inherit from the `RootObject` class, and so the `RootObject` class contains behavior common to all objects. The `RootObject` class is part of the metaclass hierarchy.

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 ScriptX 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.

Instance Methods

afterInit

`afterInit` *self arg1 arg2 . . . key1*`:`*value1 key2*`:`*value2 . . .* ⇨ *(none)*

self	Any class
arg1 arg2 . . .	Any objects
key1 key2 . . .	Keywords defined in the `init` or `afterInit` methods
value1 key2 . . .	Objects appropriate to the specified *key*

Do not call this method directly; it is automatically called by the `new` method. This method performs some post-initialization work on the object *self*, such as building internal structures dependent on settings derived from `init`, or assigning values supplied with keywords. The calling syntax shown above for `afterInit` is the same as for `init`, defined on page 615.

When you call the `new` method on a class, the method creates an instance of the class, calls `init` to initialize the instance, and then calls `afterInit`. Any keywords supplied with the `new` method are passed along for both `init` and `afterInit` to use.

The `afterInit` method is empty in `RootObject` and in most core classes. One of the few core classes that implements `afterInit` is `Collection`, where it allows you to supply the initial key and value pairs of a collection.

When you create a new class, you may want to implement `afterInit` to operate on the initialized object. For example, when subclassing a window, you might add objects to that window. This implementation should also call `apply nextMethod` to allow superclasses to perform their post-initialization. (Note that few core classes have an implementation for the `afterInit` method.)

If you provide an implementation for `afterInit`, define the method using the `#rest` and `#key` keywords as follows:

`afterInit` *self* `#rest` *args* `#key` *key1*`:`*value1 key2*`:`*value2 . . .*

Like the `init` and `new` methods, the `afterInit` method can take keyword arguments. You can supply keyword arguments in any order. Keyword arguments for some classes are optional and may be omitted. You should provide default values in the method definition for optional keyword arguments. See the "Positional and Keyword Arguments in Methods" section in the "Information Common to All Classes" chapter of this manual, and see the *ScriptX Language Guide* manual for more information about `#rest` and `#key` arguments.

You should call `apply nextMethod` in the body of `afterInit` to pass the call to superclasses up the inheritance hierarchy. This is described in detail in the *ScriptX Language Guide*.

afterLoading

`afterLoading` *self stream* ⇨ *(object)*

self	Any object
stream	A storage stream object

This method is invoked automatically on the object *self* after it has been loaded from a storage container by the `inflate` method. If you try to load an object that is already in memory, `afterLoading` is not called. Override this method to initialize objects loaded from storage containers.

Loading takes place in two steps. The `inflate` method initializes objects referred to by the inflated object's instance variables. The `afterLoading` method provides a place to access those instance variables and manipulate them through their methods and instance variables. Override this method to perform custom preparation on *self* and the objects it refers to.

The *stream* argument can only be used with `afterLoadingIfNecessary`. This argument provides object store information to that method.

See the chapter "Title Management" in the *ScriptX Architecture and Components Guide* for more information on how objects are added to and stored in a container.

afterLoadingIfNecessary

`afterLoadingIfNecessary` *self stream* ⇨ *(object)*

self	Any object or class
stream	A storage stream where objects are written

This method is just like `load` or any other method in that it causes *self* to be loaded. The `afterLoadingIfNecessary` method is only provided for backward compatibility with earlier versions of ScriptX. You should discontinue use of this method.

allIvNames

`allIvNames` *self* ⇨ Sequence

Returns a sequence of names of all direct and inherited slot instance variables on the object or class *self*. In contrast, see the method `ivNames` (also defined in `RootObject`), which is similar but does not include inherited slot instance variables. A slot instance variable is one that has a value cell, in contrast to a virtual instance variable, which has a getter method that determines the value rather than getting it from a value cell.

Also note that `allIvNames` returns the slot instance variables it finds, whether documented or not. Some slot instance variables have no getter or setter methods, and are therefore not accessible by any means in ScriptX.

For more information on instance variables, see "Instance and Class Variables" in the Object System Kernel chapter of the *ScriptX Architecture and Components Guide*, or see the *ScriptX Language Guide*.

canObjectDo

`canObjectDo` *self generic* ⇨ `Boolean`

self	Any object or class
generic	`Generic` object

Asks the object *self* if it can execute *generic*, which represents an instance of `Generic`. The `Generic` class, a subclass of `AbstractFunction`, is a substrate class that represents generic functions. It is not defined in this volume.

canStore

`canStore` *self container* ⇨ `Boolean`

self	Any object to add
container	`StorageContainer` object

The `canStore` method is called by the object store while tracing through the object hierarchy of *self* and its subobjects to determine whether each object should be stored. For example, some objects may already be stored in other containers and therefore cannot be stored in this *container*. Developers should specialize `canStore` to return `false` if a particular object should not be stored or is not storeable. The default implementation in `RootObject` returns `true`. See the chapter "Title Management" in the *ScriptX Architecture and Components Guide* for more information about how objects are added to and stored in a container.

copy

`copy` *self* ⇨ *(object)*

Creates a copy of *self*, if the class of *self* implements the Copy protocol.

When `copy` is called, it first creates a new instance of the class of *self*. It then walks the precedence vector of the class. (The precedence vector is the list of classes returned by `getSupers`, with the given class in front. For more information, see the discussion of multiple inheritance in the *ScriptX Language Guide*.)

For each class in the precedence vector that defines real instance variables, `copy` checks to see if the class defines an `initCopy` method. If so, then it calls `initCopy`.

If the object is a scripted class that does not specialize `initCopy`, then `copy` performs the default action for scripted classes—it makes a shallow copy of the object's real instance variables. If the object is an instance of a substrate class that does not specialize `initCopy`, then `copy` reports an exception.

Although the normal mechanism for specializing the Copy protocol is to specialize `initCopy`, many of the core classes specialize `copy` to return the object *self*. The *ScriptX Class Reference* does not document every class that implements either `copy` or `initCopy`. In general, collections, ranges, and events define an `initCopy` method. Numbers, names, and many of the classes in the 2D Graphics component specialize `copy`. To determine which classes can be copied, use the generic functions `getDirectGenerics` or `getAllGenerics`.

The following script prints an alphabetized list of classes that can be copied because they specialize or inherit a specialization of `copy`. (This script may take a minute to return a result.)

```
getSubs RootObject | (x -> if isMember (getAllGenerics x) copy \
       then x else empty) | SortedArray | print
```

The following script prints an alphabetized list of all classes that can be copied because they specialize `initCopy`.

```
getSubs RootObject | (x -> if isMember (getDirectGenerics x) initCopy \
    then x else empty) | SortedArray | print
```

The union of these two lists contains all the ScriptX core classes that can be copied. See also `initCopy`, defined on page 616.

Note – Certain classes may not appear in this list until they are explicitly loaded. See the note under `getSupers` and `getSubs`, defined by `Behavior`.

deflate

`deflate` *self stream* ⇨ *(object)*

self	Any object or class
stream	A storage stream where objects are written

This method is called by the `update` method to actually write the object *self* to the storage stream *stream*, along with its instance variables. This method is also invoked when you call `update` on a storage container. Override this method to customize the storage of objects in a container. A class's `deflate` method should correspond to its `inflate` method; instance variables should be written to *stream* in this method in the same order that they are retrieved by `inflate`.

Within this method, you can store objects with custom techniques or by using the default object store mechanism. For example, you may want to store some instance variables by storing selected data from them, then initializing them explicitly when they are inflated.

Also see the global functions `defaultDeflate` and `deflateSubObjectReference`.

getAllGenerics

`getAllGenerics` *self* ⇨ Sequence

self	Any class

Returns an array with all the generic functions defined on the given class of *self* and all its superclasses, including setter and getter generic functions.

getAllMethods

`getAllMethods` *self* ⇨ Sequence

self	Any class

Returns an array with all the methods defined on the given class of *self* and all of its superclasses, including setter and getter methods.

getClass

`getClass` *self* ⇨ *(class)*

self	Any class

Returns this object's class, where *self* is any object or class.

getClassName

`getClassName` *self* ⇨ `String`

self	Any class

Returns a `String` object containing the name of the class of *self*, where *self* is any object or class. If *self* is a class, `getClassName` returns its corresponding metaclass.

getDirectGenerics

`getDirectGenerics` *self* ⇨ Sequence

self	Any class

Returns an array with the generic functions defined directly on the given class *self*, including setter and getter generic functions. It does not include functions inherited from superclasses of the given class.

getDirectMethods

`getDirectMethods` *self* ⇨ Sequence

self	Any class

Returns an array with the methods defined directly on the given class *self*, including setter and getter methods. It does not include methods inherited from superclasses of the given class.

inflate

`inflate` *self stream* ⇨ *(object)*

self	Any object or class
stream	A storage stream where objects are written

This method is invoked automatically when the object *self* is loaded from a storage container. This method retrieves the object from the storage stream *stream* and initializes it, along with its instance variables. Override this method to customize the initialization of objects retrieved from a storage container. A class's `inflate` method should correspond to its `deflate` method; instance variables should be retrieved from *stream* in this method in the same order that they were written by `deflate`.

Within this method, you can initialize objects with custom techniques or by using the default object store mechanism. For example, you may want to initialize some instance variables by instantiating them directly. For other instance variables, you may call `inflateSubObjectReference` (a global function) to have the default inflation mechanism handle them. Do not attempt to manipulate the objects assigned to instance variables within this method; use `afterLoading` instead.

Loading takes place in two steps. The `inflate` method initializes objects referred to by an object's instance variables. The `afterLoading` method provides a place to access those instance variables and manipulate them through their methods and instance variables.

Also see the global functions `defaultInflate` and in`flateSubObjectReference`.

init

`init` *self arg1 arg2 . . . key1:value1 key2:value2 . . .* ⇨ *(none)*

self	Any class
arg1 arg2 . . .	Any objects
key	Keyword defined in the `init` or `afterInit` methods
value	An object appropriate to the specified *key*

Do not call this method directly; it is automatically called by the `new` method. This method initializes the object *self* by applying default values to instance variables and setting other private internal states. This method is empty in `RootObject` and is implemented in subclasses.

The `init` method is called automatically by `new`. Any keywords supplied with the `new` generic function are passed along for both `init` and `afterInit` to use.

When you create a new class, you may want to implement `init` for that class to set default values for instance variables that the class defines. Any implementation should call `apply nextMethod` to allow superclasses to perform their initialization. Within the implementation of `init`, you should not call any methods on *self*, the object being initialized; you should call methods on *self* only after the object is initialized, such as in the `afterInit` method.

Optional keywords are indicated in this manual by square brackets:

`init` *self* [`data:`*byteString*] [`colormap:`*colormap*] `bBox:`*rect*.

If you provide an implementation for `init`, define the method using the `#rest` and `#key` keywords as follows:

`init` *self* `#rest` *args* `#key` *key1*`:`*value1* *key2*`:`*value2* `. . .`

Like the `new` method, the `init` method takes keyword arguments. You can supply keyword arguments in any order. Some keyword arguments are optional and may be omitted. You should provide default values in the method definition for optional keyword arguments. See the "Positional and Keyword Arguments in Methods" section in the "Information Common to All Classes" chapter of this manual, and see the *ScriptX Language Guide* manual for more information about `#rest` and `#key` arguments.

initCopy

`initCopy` *self* ⇨ *(object)*

Do not call `initCopy` directly. It is called automatically by `copy`, a generic function that, together with `initCopy`, comprises the Copy protocol. Substrate classes that can be copied specialize either `copy` or `initCopy`. As a general rule, classes that specialize `initCopy` create a shallow copy of their instance variables. Scripted classes do not normally specialize `initCopy`. They can be copied by default. For more information, see the definition of `copy`, defined on page 613.

Note – Do not use `nextMethod` within an `initCopy` method; `copy` visits every class in the precedence vector automatically.

isAKindOf

`isAKindOf` *self theClass* ⇨ `Boolean`

self	Any object
theClass	Any class

Returns `true` if *self* is an instance of the class *theClass*, or of any subclass of *theClass*.

isComparable

`isComparable` *self other* ⇨ `Boolean`

self	Any object or class
other	Any object or class

Returns `true` if *self* and *other* are comparable objects. The `isComparable` generic function is intended for use on instances of classes. It is one of four generics that are used to define all of the comparison functions. By default, *self* and *other* are comparable if (`getClass` *self* `==` `getClass` *other*), but some classes specialize this method. For example, all subclasses of `Number` are comparable.

ivNames

`ivNames` *self* ⇨ `SortedArray`

Returns a sequence of names of the direct slot instance variables defined in the object or class *self*. Contrast this with the method `allIvNames`, which also returns slots that are inherited. If *self* is an object, then the information returned corresponds to a given instance of a class only. A slot instance variable is one that has a value cell, in contrast to a virtual instance variable, which has a getter method that determines the value rather than getting it from a value cell.

Also note that `ivNames` returns the slot instance variables it finds, whether documented or not. Some slot instance variables have no getter or setter methods, and are therefore not accessible by any means in ScriptX.

For more information on instance variables, see "Instance and Class Variables" in the Object System Kernel chapter of the *ScriptX Components Guide*, or see the *ScriptX Language Guide*.

ivTypes

`ivTypes` *self* ⇨ `SortedArray`

Returns a sequence of `Quad` objects containing information on all real instance variables, (those that are implemented as slots) on the object or class *self*. This method corresponds with the method `ivNames`, but the information returned by `ivTypes` is more extensive. Each quad has the following elements:

1. The name of an instance variable on the object or class.
2. A representation of metadata on that variable as a string.
3. The offset in bytes from the beginning of the class to that instance variable.
4. An integer, representing metadata flags on the variable. The flags in Table 4-2 indicate the structure of actual instance variables. They are observable as the fourth element of each quad returned by the method `ivTypes`.

Table 4-8: Structure element flags observable from the scripter

Name	Mask	Interpretation
`READ_ONLY`	`0x10`	Is this element read-only in the scripter?
`VISIBLE`	`0x40`	Is this element visible or hidden to the scripter?
`TRANSIENT`	`0x80`	Is this element persistent or transient?

load

`load` *self* ⇨ *(none)*

Touches the object *self* in a storage container, causing the stored object to be loaded into memory. This method calls `afterLoading` unless *self* was already in memory. (Thus, you can call `load` on an object muliple times, and `afterloading` is called only when the object is actually loaded.)

Use `load` to preload objects into memory and avoid the delay of sequentially loading many large objects when changing scenes, for example. See the chapter "Title Management" in the *ScriptX Architecture and Components Guide* for more information on object storage and retrieval.

The `load` method actually accepts multiple arguments, but ignores all its arguments except the first one, which enables the following function to work as a `startupAction`:

```
(tc -> forEach tc load undefined)
```

Also see the global function `isInMemory`.

localEqual

`localEqual` *self other* ⇨ `Boolean`

self	Any object
other	Any object

Compares *self* and *other*, returning `true` if they have the same value. By default, `localEqual` is the same as `eq`, which indicates that *self* and *other* are the same object. Many classes in the core classes specialize or inherit a specialized implementation of `localEqual`, including numbers and collections. This generic function generally assumes that *self* and *other* are comparable. It is one of the four generic functions that are used to define all the comparison functions.

localLt

`localLt` *self other* ⇨ `Boolean`

self	Any object
other	Any object

Compares *self* and *other*, returning `true` if the value of *self* is less than the value of *other*. By default, `localLt` reports an exception—a class must specialize `localLt` to provide for the ordering of objects. As with `localEqual`, This generic function generally assumes that *self* and *other* are comparable. The `localLt` method is one of the four generic functions that are used to define all the comparison functions.

morph

`morph` *self targetClass option* ⇨ *(class)*

self	Any object or class
targetClass	Any class
option	An object, to be used as an argument

Casts the object *self* from its class to the class *targetClass*. The global function `coerce` is implemented by two generic functions: `morph` and `newFrom`. The syntax for `morph` is the same as for `coerce`, except that `morph` accepts an extra coercion control argument, *option*, that is dependent on the target class. Classes that implement a specialized version of `morph` can use this argument for any purpose. Any implementation of `morph` should accept `@normal` as a value for this argument.

In general, the source class defines `morph` to call `new` on the target class, supplying the appropriate arguments. The `morph` method then converts instance variables from the source class to the target class, as appropriate.

If `morph` succeeds in coercing to the target class, it should not call `nextMethod`. If `morph` fails to coerce, it should either call `nextMethod` or report the `cantCoerce` exception. This insures that if both a `morph` and a `newFrom` class method exist for a given class, the `morph` method takes precedence over `newFrom`.

For more information, see “Coercing Objects” in the “Object System Kernel” chapter of the *ScriptX Components Guide*.

prin

`prin` *self mode stream* ⇨ *(none)*

self	Any object or class
mode	`NameClass` object
stream	`ByteStream` object that is writable

Prints a representation of the object *self* to *stream*, formatting it using one of the printing styles specified by *mode*. The value of *mode* represents one of the ScriptX printing styles and can be one of `@normal`, `@complete`, `@unadorned`, or `@debug`. For information on printing styles, see the *ScriptX Language Guide*.

The `debug` stream, a system constant, represents the standard output stream used by the Listener window. (The `debug` stream should not be confused with the `@debug` printing mode, one of the possible values for the *mode* argument.)

The generic function `prin` is one of two generics that is used to define all the printing functions, the other being `recurPrin`. For more information, see "Printing Objects" in the "Object System Kernel" chapter of the *ScriptX Components Guide*.

recurPrin

`recurPrin` *self format stream state* ⇨ *(none)*

self	Any object or class
format	`NameClass` object
stream	`ByteStream` object that is writable
state	Recursive print state

This generic function has a calling signature that is identical to `prin`, with the addition of a fourth argument. This generic exists to be specialized in classes that contain recursive structures; it is not meant to be called from the scripter. For objects that do not contain recursive structures, `recurPrin` calls `prin` by default. Together with `prin`, it is one of the two methods that is used to define all the global printing functions. For more information, see "Printing Objects" in the "Object System Kernel" chapter of the *ScriptX Architecture and Components Guide*.

removeMethod

`removeMethod` *self method* ⇨ *(none)*

self	Any class
method	Method or generic function

Finds the specified method, either given as a method or as the generic function it specializes, and removes it from the method table and cache of the class *self*. The `removeMethod` method is implemented only for specialized instances of classes: instances that define additional methods for an object.

removeMethods

`removeMethods` *self* ⇨ *(none)*

self	Any class

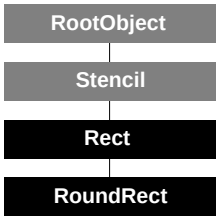
Removes all cache entries and clears out method tables for the class *self*. The `removeMethods` method is implemented only for specialized instances of classes: instances that define additional methods for an object.

update

`update` *self* ⇨ *(none)*

Saves the object *self* along with all subobjects of *self* (which are any objects reachable from *self* through instance variables, function arguments, or any other means) to the storage container that *self* belongs to, if the `mode` of the storage container is `@create` or `@update`. If *self* has not been added to any storage container, or if the `mode` of the storage container is not `@create` or `@update`, then this method fails.

RoundRect



Class type: Core class (concrete)
Resides in: ScriptX and KMP executables
Inherits from: `Rect`
Component: 2D Graphics

`RoundRect` is a subclass of `Rect` that renders rectangular areas with rounded corners. The shape rendered by a `RoundRect` instance is represented by its upper-left and lower-right corner points and the x and y radii of its rounded corners.

Note that `RoundRect` objects are not presenters—to display a rounded rectangle, create an instance of `TwoDShape` using a `RoundRect` object as the *boundary* argument.

Creating and Initializing a New Instance

The following script is an example of how to create a new instance of the `RoundRect` class:

```
myRoundRect := new RoundRect \
       x1:0 \
       y1:0 \
       x2:50 \
       y2:50 \
       rx:10 \
       ry:5
```

The variable `myRoundRect` contains the initialized `RoundRect` object. The upper-left corner of its bounding box is at 0,0 and the lower-right corner of its bounding box is at 50,50. The corners are rounded with a radius of 10 relative to the x axis and a radius of 5 relative to the y axis. Notice that `x1` and `y1` default to `0` and so are optional here. The `new` method uses the keywords defined in `init`.

init

`init` *self* [`x1:`*number*] [`y1:`*number*] [`x2:`*number*] [`y2:`*number*]
[`rx:`*number*] [`ry:`*number*] ⇨ *(none)*

self	`RoundRect` object
`x1:`	`Number` object
`y1:`	`Number` object
`x2:`	`Number` object
`y2:`	`Number` object
`rx:`	`Number` object
`ry:`	`Number` object

Initializes the `RoundRect` object *self*, applying the arguments as follows: `x1` and `y1` represent one corner of the rounded rectangle, `x2` and `y2` represent the opposite corner, and `rx` and `ry` represent the x and y radii of the rounded corners. Do not call `init` directly on an instance—it is automatically called by the `new` method.

If you omit an optional keyword, its default value is used. The defaults are:

```
x1:0
y1:0
x2:0
y2:0
rx:0
ry:0
```

Instance Variables

Inherited from `Stencil`:

`bBox`

Inherited from `Rect`:

`height`	`x1`	`y1`
`width`	`x2`	`y2`

The following instance variables are defined in `RoundRect`:

height (Rect)

self`.height` (read-write) `Number`

Specifies the height of the round rectangle. Setting `height` resets `y2` and keeps `x1`, `y1`, and `x2` constant.

radius

self`.radius` (read-write) `Number`

Specifies the radius of the corners of the rounded rectangle. When this value is set explicitly, the corners of the `RoundRect` instance are rounded by this radius in both dimensions.

rx

self`.rx` (read-write) `Number`

Specifies the x-axis radius of the corners of the rounded rectangle. When `rx` and `ry` are non-zero, the corners of the `RoundRect` instance are rounded in the x dimension by this value.

ry

self`.ry` (read-write) `Number`

Specifies the y-axis radius of the corners of the rounded rectangle. When `rx` and `ry` are nonzero, the corners of the `RoundRect` instance are rounded in the y dimension by the value of `ry`.

width (Rect)

self`.width` (read-write) `Number`

Specifies the width of the round rectangle. Setting `width` resets `x2` and keeps `x1`, `y1`, and `y2` constant.

x1 (Rect)

self`.x1` (read-write) `Number`

Specifies the x coordinate of the upper-left corner of the rounded rectangle.

x2 (Rect)

self.x2 (read-write) Number

Specifies the x coordinate of the lower-right corner of the rounded rectangle.

y1 (Rect)

self.y1 (read-write) Number

Specifies the y coordinate of the upper-left corner of the rounded rectangle.

y2 (Rect)

self.y2 (read-write) Number

Specifies the y coordinate of the lower-right corner of the rounded rectangle.

Instance Methods

Inherited from Stencil:

inside	onBoundary	union
intersect	subtract	

Inherited from Rect:

copy	moveToZero

The following instance methods are defined in RoundRect:

transform

transform *self matrix result* ⇨ *(object)*

self	RoundRect object
matrix	TwoDMatrix representing the transformation
result	Name representing the desired result: @mutate or @create

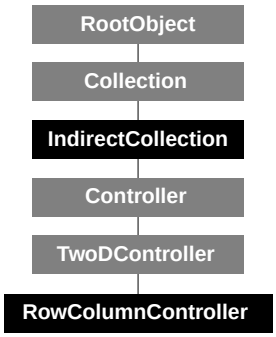
Transforms an instance of the RoundRect class *self,* by the TwoDMatrix *matrix*. The value returned depends on the specified result:

@mutate – transforms *self* directly

@create – creates a copy of *self*, then transforms and returns the copy

The result returned by a complex transformation matrix, such as one representing both a scale and rotate operation, may be an instance of Region. In such cases, the *result* argument is ignored and a copy is returned. To transform a rounded rectangle, you first call the appropriate transformation methods on *matrix*, and then call this method on the RoundRect instance.

RowColumnController



Class type: Core class (concrete)
Resides in: ScriptX and KMP executables
Inherits from: `TwoDController`
Component: User Interface

The `RowColumnController` class is a collection that arranges its members into rows and columns. A controller is a collection of the objects it controls. It traverses this collection of objects and lays them out one after another in its space in a grid pattern.

A `RowColumnController` object cannot explicitly place an object into a given row or column. Instead, the order of the objects in the list and the controller's layout information determine the layout.

Objects are laid out and rendered according to their position in the presentation hierarchy. If z ordering is used, then objects with a higher `z` value are laid out first. Ordering can be modified by using collection operations, or by setting the `z` value. For more information on the presentation hierarchy and z ordering of presenters, see the "Spaces and Presenters" chapter of the *ScriptX Components Guide*.

If the value of `wholeSpace` for this controller is set to `true`, then the controller will resize its space to be the smallest size needed to encompass the controlled targets. It does not automatically adjust its layout when a target resizes itself. If the value of `wholeSpace` is set to `false`, then it will not resize the space.

Creating and Initializing a New Instance

The following script creates a new instance of the `RowColumnController` class, after first creating a space for it to control:

```
mySpace := new TwoDSpace (new Rect x1:0 y1:0 x2:600 y2:400)
myRCController := new RowColumnController \
                  layoutOrder: @columnMajor \
                  numRows:10 \
                  space:mySpace
```

The variable `myRCController` contains an initialized instance of `RowColumnController` which controls the space specified by `mySpace`. Any objects you add to this controller are arranged in 1 column with 10 rows. The `new` method uses keyword arguments defined in `init`.

init

`init` *self* [`layoutOrder:`*name*] [`numColumns:`*number*]
[`numRows:`*number*] [`targetCollection:`*sequence*]
[`enabled:`*boolean*] [`space:`*space*] [`wholeSpace:`*boolean*] ⇨ *(none)*

self	`RowColumnController` object
`layoutOrder:`	`NameClass` object
`numColumns:`	`Number` object
`numRows:`	`Number` object

The superclass `TwoDController` uses the following keyword:

`targetCollection:`	`Sequence` object (use with caution)

The superclass `Controller` uses the following keywords:

`enabled:`	`Boolean` object
`space:`	`Space` object
`wholeSpace:`	`Boolean` object

Initializes the `RowColumnController` object *self*, applying the keyword arguments to instance variables of the same name. Use discretion in changing the target collection; for more information, see the definition of the `TwoDController` class. Do not call `init` directly on an instance—it is automatically called by the `new` method.

If you omit an optional keyword, its default value is used. The defaults are:

```
layoutOrder:@rowMajor
numColumns:1
numRows:1
targetCollection:(new Array initialSize:14 growable:true)
enabled:true
space:undefined
wholeSpace:undefined
```

Class Methods

Inherited from `Collection`:

`pipe`

Instance Variables

Inherited from `Collection`:

`bounded`	`maxSize`	`size`
`iteratorClass`	`minSize`	`uniformity`
`keyEqualComparator`	`mutable`	`uniformityClass`
`keyUniformity`	`mutableCopyClass`	`valueEqualComparator`
`keyUniformityClass`	`proprietored`	

Inherited from `IndirectCollection`:

`targetCollection`

Inherited from `Controller`:

`enabled`	`space`	`wholeSpace`
`protocols`		

The following instance variables are defined in `RowColumnController`:

columnWidths

self.columnWidths (read-write) Sequence

Specifies a sequence of numbers describing the width of individual columns managed by the row-column controller *self*. If a particular entry is negative, is not a number, or is missing, then that column's width will default according to the value of the defaultWidth instance variable.

defaultHeight

self.defaultHeight (read-write) NameClass

Describes how the default row height is determined for rows managed by the row-column controller *self*. Possible values are @highest (computes the highest object in the row and uses that as the height of the row), or @last (uses the height of the last row with an explicit height).

defaultWidth

self.defaultWidth (read-write) NameClass

Describes how the default column width is determined for columns managed by the row-column controller *self*. Possible values are @widest (computes the widest object in the column and uses that as the width of the column), or @last (uses the width of the last column with an explicit width).

layoutOrder

self.layoutOrder (read-write) NameClass

Describes the order in which objects will be inserted into rows and columns managed by the row-column controller *self*. Possible values are @rowMajor (fill the rows first) and @columnMajor (fill the columns first).

numColumns

self.numColumns (read-write) Number

Specifies the number of columns in the layout of model objects arranged by the row-column controller *self* when the value of layoutOrder is @rowMajor.

numRows

self.numRows (read-write) Number

Specifies the number of rows in the layout of model objects managed by the row-column controller *self* when the value of layoutOrder is @columnMajor.

rowHeights

self.rowHeight (read-write) Sequence

Specifies a sequence of numbers describing the heights of individual rows managed by the row-column controller *self*. If a particular entry is negative, is not a number, or is missing, then that row's height will default according to the value of the defaultHeight instance variable.

Instance Methods

Inherited from `Collection`:

`add`	`forEach`	`iterate`
`addMany`	`forEachBinding`	`localEqual`
`addToContents`	`getAll`	`map`
`chooseAll`	`getAny`	`merge`
`chooseOne`	`getKeyAll`	`pipe`
`chooseOneBinding`	`getKeyOne`	`prin`
`deleteAll`	`getMany`	`removeAll`
`deleteBindingAll`	`getOne`	`removeOne`
`deleteBindingOne`	`hasBinding`	`setAll`
`deleteKeyAll`	`hasKey`	`setOne`
`deleteKeyOne`	`intersects`	`size`
`deleteOne`	`isEmpty`	
`emptyOut`	`isMember`	

Inherited from `IndirectCollection`:

`isAppropriateObject`	`objectAdded`	`objectRemoved`

Inherited from `Controller`:

`isAppropriateObject`	`tickle`

Since a `RowColumnController` object is an indirect collection, you can also use any methods defined in the class specified by `targetCollection`. The target collection is typically an instance of `Array`, which inherits from `Sequence`, so the following instance methods are redirected to this controller.

Accessible from `LinearCollection`:

`chooseOneBackwards`	`findRange`	`getNthKey`
`chooseOrdOne`	`forEachBackwards`	`getOrdOne`
`deleteFirst`	`getFirst`	`getRange`
`deleteLast`	`getLast`	`localEqual`
`deleteNth`	`getMiddle`	`localLT`
`deleteRange`	`getNth`	`pop`

Accessible from `Sequence`:

`addFifth`	`moveBackward`	`setFourth`
`addFirst`	`moveForward`	`setLast`
`addFourth`	`moveToBack`	`setNth`
`addNth`	`moveToFront`	`setSecond`
`addSecond`	`prepend`	`setThird`
`addThird`	`prependNew`	`sort`
`append`	`setFifth`	
`appendNew`	`setFirst`	

The following instance methods are defined in `RowColumnController`:

layout

`layout` *self* ⇨*self*

Organizes model objects that are controlled by the row-column controller *self* in a grid layout according to the values of the instance variables `layoutOrder`, `rowHeight`, `columnWidth`, `defaultHeight`, and `defaultWidth`.

objectAdded (IndirectCollection)

`objectAdded` *self key modelObject* ⇨*self*

self	`RowColumnController` object
key	Ordinal position of the object added
modelObject	Any object

Do not call this method directly from the scripter. It is visible to the scripter so that it can be overridden in a subclass of `RowColumnController`. Since the `RowColumnController` class inherits from `IndirectCollection`, this method is triggered automatically when a model object is added. As defined by `RowColumnController`, `objectAdded` calls `objectAdded` on superclasses and then triggers its own `layout` method.

For more information on `objectAdded` and `objectRemoved`, see the "Controllers" chapter of the*ScriptX Components Guide* and the class definitions of `TwoDController` and `Controller`.

objectRemoved (IndirectCollection)

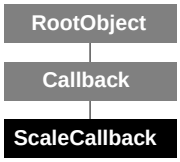
`objectRemoved` *self modelObject* ⇨*self*

self	`RowColumnController` object
modelObject	Any object

Do not call this method directly from the scripter. It is visible to the scripter so that it can be overridden in a subclass of `RowColumnController`. Since the `RowColumnController` class inherits from `IndirectCollection`, this method is invoked automatically when a model object is removed. As defined by `RowColumnController`, `objectRemoved` calls `objectRemoved` on superclasses and then triggers its own `layout` method.

For more information on `objectAdded` and `objectRemoved`, see the "Controllers" chapter of the *ScriptX Components Guide* and the class definitions of `TwoDController` and `Controller`.

ScaleCallback



Class type: Core class (concrete)
Resides in: ScriptX and KMP executables
Inherits from: `Callback`
Component: Clocks

The `ScaleCallback` class is used to perform actions when the clock's scale changes.

You never create an instance of `ScaleCallback` directly. Instead, you request a callback from a clock using the `addScaleCallback` method defined by the `Clock` class.

The `order` instance variable, inherited from `Callback`, has no effect with a `ScaleCallback` instance.

Instance Variables

Inherited from `Callback`:

`authorData`	`onceOnly`	`script`
`condition`	`order`	`target`
`label`	`priority`	

The following instance variables are defined in `ScaleCallback`:

condition

self`.condition` (read-write) `NameClass`

Determines the condition under which the scale callback will be activated. Valid values for `condition` are `@lessThan`, `@greaterThan`, `@equal`, `@notEqual`, `@lessThanOrEqual`, `@greaterThanOrEqual`, and `@change`. The default value is `@change`.

scale

self`.scale` (read-write) `NameClass`

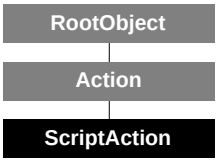
Specifies the value used in comparisons by the callback condition when the scale changes.

Instance Methods

Inherited from `Callback`:

`cancel`

ScriptAction



Class type: Core class (concrete)
Resides in: ScriptX and KMP executables
Inherits from: `Action`
Component: Animation

The `ScriptAction` class represents an action that will execute a script you specify at a certain time, under the control of an action list player. To be triggered, an instance of `ScriptAction` needs to be added to the action list of an action list player, then the player needs to be played.

You specify the script in the `script` instance variable. Actually, the "script" must be a ScriptX function defined with three arguments: an action, a target, an `ActionListPlayer` object.

```
function myFunc action target player -> ( -- body of function )
```

Here is an example function (where `alp` is the `ActionListPlayer` object):

```
function myFunc action target alp -> ( alp.time := 15 )
```

- The *action* parameter is similar to the *self* argument in a method—it is the `ScriptAction` object itself that this function is being called from.
- The *target* parameter is the object currently being acted on, from `targets` in the `ActionListPlayer` object.
- The *player* parameter is the `ActionListPlayer` object that is playing this action. This allows scripts to affect the player, so that loops, jumps, or pause commands are easy to implement.

You normally should not use `ScriptAction` for adding or removing an object from the target list—use `TargetListAction` for that. Using `TargetListAction` enables the script to correctly create and dispose of objects during fast forward and rewind.

Note – It is important to set the `playOnly` instance variable (inherited from `Action`) to `true` for certain actions that you want to happen only while playing, and not while "fast forwarding" to reconstruct the state of the world during a time jump. Otherwise, you may not be able to fast forward past certain points. See `playOnly` in the `Action` class for details.

Creating and Initializing a New Instance

The following script is an example of how to create a new instance of the `ScriptAction` class:

```
function widenFunc action target player ->
       (target.width := 2 * target.width)
myAction := new ScriptAction \
              script:widenFunc \
              targetNum:2 \
              time:20
```

The variable `myAction` holds an initialized instance of `ScriptAction`. This instance executes the function `widenFunc` at the player's time of 20 ticks. This function widens the player's second target (`targetNum:2`) by multiplying its width by 2. The `new` method uses the keywords defined in `init`.

init

`init` *self* [`script:`*function*] [`targetNum:`*integer*] [`time:`*integer*] ⇨ *(none)*

self	`ScriptAction` object
`script:`	`Function` object containing the script to be executed when this action is triggered

Superclass `Action` uses the following keywords:

`targetNum:`	`Integer` object indicating which object in the target list of the player to be passed into the function
`time:`	`Integer` object representing the time in ticks to trigger the action

Initializes the `ScriptAction` object *self*, applying the arguments to the instance variables of the same name. At the time specified by `time`, this script action will cause the function specified by `script` to execute. If an invalid script is passed as the script, this method reports an `invalidScript` exception. Do not call `init` directly on an instance—it is automatically called by the `new` method.

If you omit an optional keyword, its default value is used. The defaults are:

```
targetNum:0
time:0
script:undefined
```

Instance Variables

Inherited from `Action`:

`authorData` `targetNum` `time` `playOnly`

The following instance variables are defined in `ScriptAction`:

script

self`.script` (read-write) *(function)*

Specifies the function to execute when the script action *self* is triggered. The function takes three arguments: an action, the target, and the `ActionListPlayer` object. The script action calls this function as:

```
myFunc  action  target  player
```

Given an action list player `alp` and the script action `myAction`, you can manually call this script as follows:

```
myAction.script myAction alp.targets[targetNum] alp
```

Although any global function, anonymous function, or method can be assigned to `script`, there are differences in how different classes of functions are dispatched. For information on functions and function dispatch, see the "Object System Kernel" chapter of the *ScriptX Components Guide*.

Instance Methods

Inherited from `Action`:

`trigger`

The following instance method is defined in `ScriptAction`:

trigger (Action)

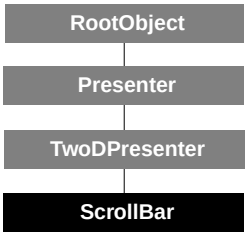
`trigger` *self target player* ⇨ *(object)*

self	`ScriptAction` object
target	Any object
player	`ActionListPlayer` object

Causes the script action *self* to execute at the time specified by the `time` instance variable. This method is called by an `ActionListPlayer` object. This action list player is passed in as the third argument *player*. This method returns the object returned by evaluating the function.

When the action list player calls trigger, the value for *target* is automatically determined by taking the `targetNum` instance variable and finding the object in the corresponding slot in the player's target list. If the `targetNum` instance variable is out of range (that is, less than 1 or greater than the size of the target list), the value of the *target* is `empty`. Also, if there is no *target* in a certain slot in the target list yet, the value of *target* is `undefined`.

ScrollBar



Class type: Core class (concrete)
Resides in: ScriptX and KMP executables
Inherits from: `TwoDPresenter`
Component: User Interface

`ScrollBar` is a user interface class that allows the user to set a numeric value by direct manipulation of a scroll bar, as shown in Figure 26. This value can then be used to control other objects. The `ScrollingPresenter` class uses scroll bars to allow a user to view portions of a target presenter.

A `ScrollBar` object, like other user interface objects, has no intrinsic appearance. In ScriptX, a scroll bar without stencils and without fill and stroke colors is not visible; you determine the "look" of a scroll bar by the stencils, fill, and stroke you attach to it. Figure 26 depicts the visible components of a `ScrollBar` object.

If you omit the increment stencil and decrement stencil, the scroll bar serves as a slider. A scroll bar can be oriented either horizontally (as shown) or vertically.

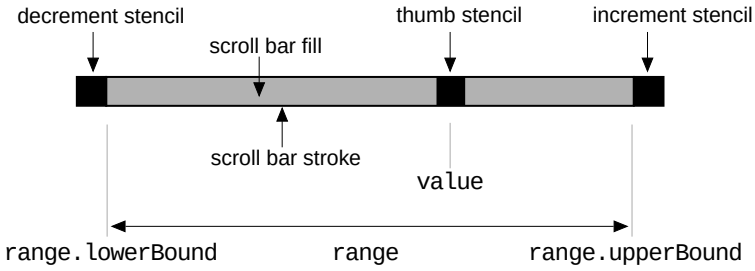


Figure 26: When a scroll bar is enabled, its fill, stroke, and three stencils are visible.

The size of a scroll bar is determined by setting its `height` if its orientation is vertical or its `width` if its orientation is horizontal. (`ScrollBar` objects inherit the instance variables `height` and `width` from `TwoDPresenter`.) The scroll bar sizes itself automatically in the other dimension, so that all three of its stencils are fully displayed.

`ScrollBar` is a `TwoDPresenter` class that incorporates characteristics of a controller as well as a presenter. Other user interface objects (for example, `PushButton`) are associated with a controller. The `ScrollBar` class has its own control mechanisms built in. Note that a scroll bar is really a combination of several individual elements that have button-like behavior. The behavior of each of these elements is conventional in desktop user interface design.

The control mechanisms that manage a scroll bar are similar to those used by other user interface objects. The class defines a number of instance methods that are called automatically on an instance of `ScrollBar`. Although these methods are visible to the scripter, they are not meant to be called directly. An instance or subclass of `ScrollBar` can override these methods. The `processEvent` method plays a central role. This method receives all mouse events that match event interests defined by a scroll bar. It acts as a dispatcher, calling other methods that redraw the scroll bar and update its value.

Creating and Initializing a New Instance

The following script is an example of how to create a new instance of the `ScrollBar` class. Only the first two lines are required; the others are optional. In this example, the variables `rightArrow`, `leftArrow`, and `pushPin` contain `TwoDPresenter` objects that have been instantiated elsewhere.

```
myScrollBar := new ScrollBar \
       orientation:@horizontal \
       incrementStencil:rightArrow \
       decrementStencil:leftArrow \
       thumbStencil:pushPin \
       range:(new NumberRange lowerBound:0 upperBound:300 increment:1)
```

The variable `myScrollBar` contains the initialized scroll bar. Its orientation is horizontal, and it returns values between 0 and 300, as set by its `range`. This scroll bar derives part of its appearance from the three stencils that it defines. To be fully visible, this scroll bar also needs fill and stroke colors.

There are two ways to add the fill and stroke colors to a scroll bar. One way is to assign values to the `fill` and `stroke` instance vaariables as follows:

```
myScrollBar.fill := whiteBrush
myScrollBar.stroke := blackBrush
```

The other way is to instantiate the object using the `object` expression as follows, instead of using `new` as shown above:

```
myScrollBar := object (ScrollBar)
       orientation:@horizontal
       incrementStencil:rightArrow
       decrementStencil:leftArrow
       thumbStencil:pushPin
       range:(new NumberRange lowerBound:0 upperBound:300 increment:1)
       settings
       fill:whiteBrush
       stroke:blackBrush
end
```

The `new` method uses the keywords defined in `init`.

init

`init` *self* `orientation:`*name*
[`incrementStencil:`*stencil*]
[`decrementStencil:`*stencil*]
[`thumbStencil:`*stencil*]
[`range:`*numberRange*] ⇨ *(none)*

self	`ScrollBar` object
`orientation:`	`NameClass` object
`incrementStencil:`	`Stencil` object
`decrementStencil:`	`Stencil` object
`thumbStencil:`	`Stencil` object
`range:`	`NumberRange` object

Superclasses of `ScrollBar` use the following keyword:

`boundary:`	`Stencil` object (defined by `ScrollBar`)
`stationary:`	`Boolean` object

Initializes the ScrollBar object *self*, applying the values supplied with the keywords to the instance variables of the same name. Do not call init directly on an instance — it is automatically called by the new method.

If you omit one of the keyword arguments, the following defaults are used:

```
incrementStencil:undefined
decrementStencil:undefined
thumbStencil:undefined
range:(new NumberRange lowerBound:1 upperBound:100)
boundary:(new Rect x2:0 y2:0)
stationary:false
```

The ScrollBar class inherits the boundary keyword from its superclasses. Since a scroll bar calculates its own boundary using the boundaries of its stencils, the boundary keyword is ignored by the ScrollBar class and is passed on to TwoDPresenter.

If you are using a ScrollBar object in a ScrollingPresenter object, the ScrollingPresenter object automatically sizes the ScrollBar object to the size of the scrolling presenter. If you are not using your ScrollBar object with a ScrollingPresenter object, the size of the scroll bar is determined by setting the value of height or width. By default, the scroll bar sets its height to 100 if its orientation is vertical, and its width to 100 if its orientation is horizontal. A scroll bar automatically lays out its stencils accordingly.

Instance Variables

Inherited from Presenter:

presentedBy	subPresenters	target

Inherited from TwoDPresenter:

bBox	height	transform
boundary	IsImplicitlyDirect	width
clock	isTransparent	window
compositor	isVisible	x
direct	needsTickle	y
eventInterests	position	z
globalBoundary	stationary	
globalTransform	target	

The following instance variables are defined in ScrollBar:

authorData

self.authorData (read-write) *(object)*

Used as the first argument to the function specified by the instance variable valueAction. Can be used by the author to store any data that is relevant to the scroll bar *self*.

clock

self.clock (read-only) Clock

Specifies the clock associated with the scroll bar *self*. Every scroll bar is associated with an instance of Clock, used to trigger repeat actions. When the user presses the mouse button while the pointer is within the boundary of the increment or decrement presenters, the ScrollBar class registers an instance of PeriodicCallback. This callback calls the instance methods incrementStep or decrementStep at regular intervals for as long as the user continues to press the mouse button over the presenter.

Similarly, if the user presses the mouse button while the pointer is within the boundary of the track presenter, the class registers a periodic callback that will call the methods `incrementPage` and `decrementPage`. See also `pauseScale` and `repeatScale`.

decrementStencil

self`.decrementStencil` (read-write) `Stencil`

Specifies the stationary part of the scroll bar *self* that represents a decrement area. When this ares is pressed, the `value` instance variable decreases by `stepAmount`. This stencil can be `undefined`. Setting the decrement stencil triggers the `calculateSize` and `layout` methods. See also `pressDecrementStencil`.

directDrag

self`.directDrag` (read-write) `Boolean`

Specifies how the `value` instance variable is updated for the scroll bar *self*. If `true`, the value is updated as the thumb presenter is dragged; if `false`, the value is updated only when the thumb presenter is dropped. This variable defaults to `false`.

disableBrush

self`.disableBrush` (read-write) `Brush`

Specifies the brush of the entire scrollbar when the scroll bar *self* is disabled — that is, when `enabled` is set to `false`. If the value of `disableBrush` is `undefined`, the user receives no visual clue that the scroll bar is not enabled. The `disableBrush` covers all parts of the scroll bar when the scroll bar is disabled.

dragInterest

self`.dragInterest` (read-only) `MouseMoveEvent`

Specifies the event interest posted by the scroll bar *self* for instances of `MouseMoveEvent`. Do not attempt to set the value of this instance variable.

enabled

self`.enabled` (read-write) `Boolean`

Specifies whether the scroll bar *self* is set to respond to user input events. If `enabled` is set to `false`, the disabled presenter (if defined) is displayed in front of the other subpresenters to indicate that the scroll bar is disabled. The default value of `enabled` is `true`.

fill

self`.fill` (read-write) `Brush`

A scroll bar's appearance is generally determined by the appearance of its four stencils, but the `fill` and `stroke` instance variables can be used separately to provide a fill and stroke for the entire scroll bar. For example, a scroll bar could use the `fill` instance variable to specify a background color.

incrementStencil

self`.incrementStencil` (read-write) `Stencil`

Specifies the stationary part of the scroll bar *self* that represents an increment area. When this area is pressed, the `value` instance variable increases by `stepAmount`. The value of `incrementStencil` can be `undefined`. Setting the increment stencil triggers the `calculateSize` and `layout` methods. See also `pressIncrementStencil`.

orientation

self.orientation (read-only) NameClass

Specifies the orientation of the scroll bar *self*. Possible values are @horizontal and @vertical. The value of orientation must be set on instantiation.

pageAmount

self.pageAmount (read-write) Integer

Specifies the amount to increment or decrement the instance variable value when the user clicks in the track presenter, but not on the thumb presenter, of the scroll bar *self*.

pressDecrementStencil

self.pressDecrementStencil (read-write) TwoDPresenter

Specifies the look of the stationary part of the scroll bar *self* that represents a decrement area when that area receives a mouse press. The value of pressDecrementStencil can be undefined. Setting this stencil triggers the calculateSize and layout methods. See also decrementStencil.

pressIncrementStencil

self.pressIncrementStencil (read-write) TwoDPresenter

Specifies the look of the stationary part of the scroll bar *self* that represents an increment area when that area receives a mouse press. The value of pressIncrementStencil can be undefined. Setting this stencil triggers the calculateSize and layout methods. See also incrementStencil.

pressInterest

self.pressInterest (read-only) MouseDownEvent

Specifies the event interest posted by the scroll bar *self* for instances of MouseDownEvent. Do not attempt to set the value of this instance variable.

range

self.range (read-write) NumberRange

Specifies the range for the instance variable value defined by the scroll bar *self*.

releaseInterest

self.releaseInterest (read-only) MouseUpEvent

Specifies the event interest posted by the scroll bar *self* for instances of MouseUpEvent. Do not attempt to set the value of this instance variable.

stepAmount

self.stepAmount (read-write) Integer

Specifies the amount to increment or decrement the instance variable value when the user clicks on the increment or decrement presenter of the scroll bar *self*.

stroke

self.stroke (read-write) Brush

A scroll bar's appearance is generally determined by the appearance of its four stencils, but the `fill` and `stroke` instance variables can be used separately to provide a fill and stroke for the entire scroll bar.

thumbStencil

self.thumbStencil (read-write) TwoDPresenter

Specifies the moving part of the scroll bar *self*. Setting `thumbStencil` triggers the `calculateSize` and `layout` methods.

value

self.value (read-write) Integer

Specifies the current value of the scroll bar *self*. The `range` instance variable determines possible values. The maximum value occurs when the thumb presenter is at one end of the scroll bar; the minimum value occurs at the other end. Setting `value` updates the position of the thumb presenter, calling the instance method `handleValueChange` if necessary. See the instance variable `directDrag`, which determines how `value` is updated.

valueAction

self.valueAction (read-write) *(function)*

Specifies the default function that is called when the instance variable `value` changes, either by user manipulation or programmatically, in the scroll bar *self*. When `value` changes, the `ScrollBar` class calls the method `handleValueChange` on the scroll bar *self*. The `handleValueChange` method calls the function specified by `valueAction`, passing the arguments `authorData`, `self`, and `value`. Use `valueAction` to respond to the user.

Although any global function, anonymous function, or method can be assigned to `valueAction`, there are important differences in how different classes of functions are dispatched. For information on functions and function dispatch, see the "Object System Kernel" chapter of the *ScriptX Components Guide*.

Instance Methods

Inherited from `TwoDPresenter`:

`adjustClockMaster`	`inside`	`show`
`createInterestList`	`localToSurface`	`surfaceToLocal`
`draw`	`notifyChanged`	`tickle`
`getBoundaryInParent`	`recalcRegion`	
`hide`	`refresh`	

The following instance methods are defined in `ScrollBar`:

calculateSize

`calculateSize` *self* ⇨ Point

Do not call this method directly from the scripter. It is visible to the scripter so that it can be overridden in a subclass of `ScrollBar`. This method sets the boundary of the scroll bar *self* to encompass its subpresenters as follows:

- For horizontal orientation, the height of the scroll bar is set to the height of the tallest presenter. The width of the scroll bar is set to the width of `incrementPresenter` + `decrementPresenter` + `trackPresenter`.
- For vertical orientation, the width of the scroll bar is taken to be the width of the widest presenter. The height of the scroll bar is taken to be the height of `incrementPresenter` + `decrementPresenter` + `trackPresenter`.

decrementPage

`decrementPage` *self point* ⇨*self*

self	`ScrollBar` object
point	`Point` object indicating where the user clicked

Do not call this method directly from the scripter. It is visible to the scripter so that it can be overridden in a subclass of `ScrollBar`. The `trackPress` method calls `decrementPage` on the scroll bar *self* when the user clicks on the decrement area of the track presenter. The location of the decrement area, in relation to the thumb presenter, depends on whether the value of `orientation` for this scroll bar is `@horizontal` or `@vertical`:

- For vertical orientation, click in the area below the thumb presenter.
- For horizontal orientation, click in the area to the left of the thumb presenter.

This method calls the setter method for `value` to decrement the current value of the scroll bar by `pageAmount`. A subclass of `ScrollBar` can use the *point* argument to modify this behavior. For example, to modify or set `value` based on where the user clicked. This argument is expressed in the local coordinates of the scroll bar *self*.

decrementStep

`decrementStep` *self* ⇨*self*

Do not call this method directly from the scripter. It is visible to the scripter so that it can be overridden in a subclass of `ScrollBar`. The `processEvent` method calls `decrementStep` when the user clicks on the decrement presenter of the scroll bar *self*. This method calls the setter method for `value` to decrement the current value of the scroll bar by `stepAmount`. It also registers a `PeriodicCallback` object that will call `decrementStep` repeatedly until the user releases the mouse button. The class `PeriodicCallback` is described in the "Clocks" chapter of the *ScriptX Components Guide*.

handleValueChange

`handleValueChange` *self* ⇨*self*

Do not call this method directly from the scripter. It is visible to the scripter so that it can be overridden in a subclass of `ScrollBar`. This method is called automatically whenever the `value` instance variable is changed. It calls a function specified by the instance variable `valueAction`, passing the arguments `authorData`, *self*, and `value`.

incrementPage

`incrementPage` *self point* ⇨*self*

self	`ScrollBar` object
point	`Point` object indicating where the user clicked

Do not call this method directly from the scripter. It is visible to the scripter so that it can be overridden in a subclass of `ScrollBar`. The `trackPress` method calls `incrementPage` on the scroll bar *self* when the user clicks on the increment area of the

track presenter. The location of the increment area, in relation to the thumb presenter, depends on whether the value of `orientation` for this scroll bar is `@horizontal` or `@vertical`:

- For vertical orientation, click in the area above the thumb presenter.
- For horizontal orientation, click in the area to the right of the thumb presenter.

This method calls the setter method for `value` to increment the current value of the scroll bar by `pageAmount`. A subclass of `ScrollBar` can use the *point* argument to modify this behavior. For example, to increment `value` by some other amount. This argument is expressed in the local coordinates of the scroll bar *self*.

incrementStep

`incrementStep` *self* ⇨*self*

Do not call this method directly from the scripter. It is visible to the scripter so that it can be overridden in a subclass of `ScrollBar`. The `processEvent` method calls `incrementStep` when the user clicks on the increment presenter of the scroll bar *self*. This method calls the setter method for `value` to increment the current value of the scroll bar by `stepAmount`. It also registers a `PeriodicCallback` object that will call `incrementStep` repeatedly until the user releases the mouse button. The class `PeriodicCallback` is described in the "Clocks" chapter of the *ScriptX Components Guide*.

layout

`layout` *self* ⇨*self*

Lays out the various subpresenters of the scroll bar *self*, based on the value of `orientation`. Although `layout` is usually triggered automatically by other methods, it can be called directly from the scripter.

The increment and decrement presenters, if defined, are placed at either end of the scroll bar. Then the track presenter is resized to fill the area between these two presenters. The thumb presenter is placed at the appropriate position on the track presenter, determined by the `value` and `range` instance variables. The disabled presenter, visible only when the value of `enabled` is `false`, is sized so that it covers all of the other presenters. All subpresenters are positioned and aligned, either vertically or horizontally, depending on the orientation of the scroll bar.

Note that `layout` does not affect the z ordering of subpresenters, which determines their layering from front to back.

processEvent

`processEvent` *self interest event* ⇨*self*

self	`ActuatorController` object
interest	`MouseEvent` object, representing an event interest
event	`MouseEvent` object that matches *interest*

Do not call this method directly from the scripter. It is visible to the scripter so that it can be overridden in a subclass of `ScrollBar`. This method is the event receiver for the scroll bar *self*. Its event interests are assigned to the instance variables `pressInterest`, `releaseInterest`, and `dragInterest`. As an event receiver, it receives and processes events that match those interests. In general, a subclass of `ScrollBar` should not override `processEvent`.

thumbDrag

`thumbDrag` *self point* ⇨*self*

self	`ScrollBar` object
point	`Point` object, indicating the current coordinate position of the mouse

Do not call this method directly from the scripter. It is visible to the scripter so that it can be overridden in a subclass of `ScrollBar`. The `processEvent` method calls `thumbDrag` as the user drags the thumb presenter of the scroll bar *self* with the mouse. This method updates the position of the thumb presenter. If the value of `directDrag` is `true`, it also calls the setter method for `value`, updating the value of the scroll bar. The *point* argument is expressed in the local coordinates of the scroll bar *self*.

thumbPress

`thumbPress` *self point* ⇨*self*

self	`ScrollBar` object
point	`Point` object, indicating where the user clicked the mouse

Do not call this method directly from the scripter. It is visible to the scripter so that it can be overridden in a subclass of `ScrollBar`. The `processEvent` method calls `thumbPress` as the user clicks the mouse button over the thumb presenter of the scroll bar *self*. The *point* argument is expressed in the local coordinates of the scroll bar *self*.

thumbRelease

`thumbRelease` *self* ⇨*self*

self	`ScrollBar` object
point	`Point` object, indicating where the user released the mouse

Do not call this method directly from the scripter. It is visible to the scripter so that it can be overridden in a subclass of `ScrollBar`. The `processEvent` method calls `thumbRelease` as the user releases the mouse button over the thumb presenter of the scroll bar *self*. If the value of `directDrag` is `false`, `thumbRelease` also calls the setter method for `value`. The *point* argument is expressed in the local coordinates of the scroll bar *self*.

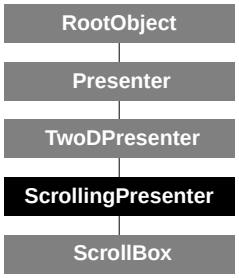
trackPress

`trackPress` *self point* ⇨*self*

self	`ScrollBar` object
point	`Point` *object*

Do not call this method directly from the scripter. It is visible to the scripter so that it can be overridden in a subclass of `ScrollBar`. The `processEvent` method calls `trackPress` when the user clicks within the track stencil of the scroll bar *self*. The *point* argument contains the x, y coordinates where the click occurred on the track stencil, expressed in the local coordinates of the scroll bar *self*. This method calls `incrementPage` or `decrementPage`, as appropriate.

ScrollBox



Class type: Scripted class (abstract)
Resides in: `widgets.sxl`. Works with ScriptX and KMP executables
Inherits from: `ScrollingPresenter`
Component: User Interface

The `ScrollBox` class presents a target presenter or presenters that are potentially larger than the boundary of the `ScrollBox` object itself. A scroll box's `targetPresenter` instance variable contains a collection of one or more instances of `ListBox`. (`ListBox`, also a scripted class in the ScriptX Widget Library, is a subclass of `TextPresenter` that presents a list of strings.)

Creating and Initializing a New Instance

`ScrollBox` is a scripted class, so technically it is not abstract, but it is not meant to be instantiated directly. Subclasses of `ScrollBox`, including `ScrollListBox` and `MultiListBox`, apply the `init` method that is defined by `ScrollBox` as part of their own `init` method.

init

`init` *self* [`font:`*fontContext*] [`hasScrollBar:`*boolean*] [`value:`*integer*] [`fill:`*brush*] [`stroke:`*brush*] [`targetPresenter:`*twoDPresenter*] ⇨ *(none)*

self	`ScrollBox` object
`font:`	`FontContext` object
`hasScrollBar:`	`Boolean` object
`value:`	`Integer` object

Superclasses of `ScrollBox` use the following keywords:

`fill:`	`Brush` object
`stroke:`	`Brush` object
`targetPresenter:`	`TwoDPresenter` object
`vertScrollBar:`	`ScrollBar` object (ignored by `ScrollBox`)
`horizScrollBar:`	`ScrollBar` object (ignored by `ScrollBox`)
`boundary:`	`Rect` object (ignored by `ScrollBox`)
`target:`	Any object (ignored by `ScrollBox`)

Initializes the `ScrollBox` object *self*, applying the values supplied with the keywords to the instance variables of the same name. Do not call `init` directly on an instance—it is called automatically by the `new` method.

If you omit an optional keyword, its default value is used. The defaults are:

```
font:theSystemFont
hasScrollBar:false
value:undefined
fill:undefined
stroke:undefined
```

```
targetPresenter:undefined
```

Instance Variables

Inherited from Presenter:

presentedBy	subPresenters	target

Inherited from TwoDPresenter:

bBox	globalTransform	target
boundary	height	transform
clock	IsImplicitlyDirect	width
compositor	isTransparent	window
direct	isVisible	x
eventInterests	needsTickle	y
globalBoundary	position	z
globalRegion	stationary	

Inherited from ScrollingPresenter:

clippingStencil	subpresenters
fill	targetPresenter
horizScrollBar	vertScrollBar
horizScrollBarDisplayed	vertScrollBarDisplayed
stroke	

The following instance variables are defined in ScrollBox:

authorData

self.authorData (read-write) *(object)*

Specifies an object that is supplied as the first argument when the functions specified by the instance variables selectAction or doubleClickAction on the ScrollBox *self* is called. The value of authorData can be any object.

doubleClickAction

self.doubleClickAction (read-write) *(function)*

Specifies the default function that is called when the user double-clicks on the ScrollBox *self*. Initially, the value of doubleClickAction is undefined. You can write a function to perform any action. This function has two arguments:

```
funcName authorData self
```

authorData	an object stored in the authorData instance variable
selectedLine	Integer object, representing the selected line

Although any global function, anonymous function, or method can be assigned to doubleClickAction, there are differences in how different classes of functions are dispatched. For information on functions and function dispatch, see the "Object System Kernel" chapter of the *ScriptX Architecture and Components Guide*.

doubleClickTime

self.doubleClickTime (read-write) Integer

Specifies the threshold of time within which two mouse clicks must occur if the ScrollBox object self is to interpret them as a double-click gesture.

In contrast with Actuator, which defines multiple clicking as a gesture in a more general sense, ScrollBox defines only a double-click gesture. When a ScrollBox object is instantiated, it queries the system for the double-click time, but doubleClickTime is visible and can be written to directly from the scripter.

fill (TwoDPresenter)

self.font (read-write) FontContext

Specifies the value of fill for the clipping presenter of the ScrollBox *self*. The instance variable clippingPresenter is defined by ScrollingPresenter, a superclass of ScrollBox.

font

self.font (read-write) FontContext

Specifies the FontContext object that is used to render text. The fontSetter method iterates through all ListBox objects that are members of *self*.targetPresenter.

frame

self.frame (read-write) Frame

Specifies the Frame object that is used to render a drop shadow for the ScrollBox *self*.

lastLine

self.lastLine (read-write) Integer

Specifies the previously selected line from the target list box presented by the ScrollBox *self*. The values of lastLine and selectedLine, instance variables defined by ScrollBox, can be different only while the mouse is down. After a mouse-up event is received, the values of lastLine and selectedLine are the same. (Although lastLine is read-write variable, a script should not change the value of lastLine without reason, since a ScrollBox object depends on maintaining its value internally.)

list

self.list (read-write) Sequence

Specifies a sequence of strings, or objects that can be coerced to strings. This sequence is used to create the target text for the ListBox object or objects that are the target of the ScrollBox *self*. Subclasses of ScrollBox must implement a method for listSetter.

numLines

self.numLines (read-only) ImmediateInteger

Specifies the number of lines visible in the targetpresenter of the ScrollBox *self*. For the ScrollBox instance mySB, the value of numLines is equivalent to the following expression:

```
mySB.targetPresenter.height / mySB.stepAmount
```

pageAmount

self.pageAmount (read-write) Integer

Specifies the page amount, the number of pixels to page up or down, when the user clicks in the track area of vertical scrollbar for the ScrollBox instance *self*. The value of pageAmount is set by layout, a method defined by ScrollingPresenter, and specialized by ScrollBox. For the ScrollBox instance mySB, the value of pageAmount is equivalent to the following expression:

```
(mySB.height / mySB.stepAmount - 1) * mySB.stepAmount
```

selectAction

self.selectAction (read-write) *(function)*

Specifies the default function that is called when the user double-clicks on the ScrollBox *self*. Initially, the value of selectAction is undefined. You can write a function to perform any action. This function has two arguments:

```
funcName authorData self
```

authorData	an object stored in the authorData instance variable
selectedLine	Integer object, representing the selected line

Although any global function, anonymous function, or method can be assigned to selectAction, there are differences in how different classes of functions are dispatched. For information on functions and function dispatch, see the "Object System Kernel" chapter of the *ScriptX Architecture and Components Guide*.

selectedLine

self.selectedLine (read-write) Integer

Specifies the currently selected line from target instance of ListBox that is presented by the ScrollBox *self*. Clicking the mouse on a new line changes the value of selectedLine. While the mouse is down, the previous value of selectedLine is stored in lastLine, another instance variable defined by ScrollBox. Once the mouse is released, the values of selectedLine and lastLine are the same.

stepAmount

self.stepAmount (read-write) Integer

Specifies the step amount, equal to the value of *self*.font.leading, when the user scrolls the ScrollBox instance *self*.

targetPresenter (ScrollingPresenter)

self.targetPresenter (read-write) TwoDPresenter

Subclasses of ScrollBox must set a target presenter at initialization time. Since ScrollBox inherits from ScrollingPresenter, the target presenter of the ScrollBox *self* acts as a proxy for the presenter referenced by targetPresenter.

value

self.value (read-write) Integer

Specifies the key of the selected line from the list of the ScrollBox object *self*. Since the list instance variable returns a sequence, this value is an implicit key, the ordinal position of the selected line in that list.

width (TwoDPresenter)

self.width (read-write) Integer

Specifies the width of the ScrollBox *self*, and determines the width of its target presenter.

Instance Methods

Inherited from TwoDPresenter:

adjustClockMaster	hide	refresh
createInterestList	inside	show
draw	localToSurface	surfaceToLocal
getBoundaryInParent	notifyChanged	tickle

Inherited from ScrollingPresenter:

handleHorizScroll	layout	scrollTo
handleVertScroll		

The following instance methods are defined in ScrollBox:

downReceiver

downReceiver *self interest event* ⇨ *self*

self	ScrollingPresenter object
interest	MouseDownEvent object, representing an event interest
event	MouseDownEvent object, representing an event

Do not call this method directly from the scripter. It is visible to the scripter so that it can be overridden in an instance or subclass of ScrollBox. The downReceiver method is the scroll box's event receiver for mousedown events. A ScrollBox object automatically manages interests in mouseup and mousedown events, which it uses to manage selection from a target list. (The target list is implemented by a subclass of ScrollBox.)

repeatScrollAction

repeatScrollAction *self targetPresenter direction* ⇨ *self*

self	ScrollingPresenter object
targetPresenter	MouseDownEvent object, representing an event interest
direction	NameClass object, either @up or @down

Do not call this method directly from the scripter. It is visible to the scripter so that it can be overridden in an instance or subclass of ScrollBox. Depending on the value of *direction*, repeatScrollAction automatically scrolls up or down through the target presenter's list, changing the currently selected line for the ScrollBox object *self*. ScrollBox defines a method for tickle. It is this tickle method that calls repeatScrollAction, once with each tick of the space's clock, to scroll step-by-step through the target presenter list.

selectLine

selectLine *self theLine doAction* ⇨ *self*

self	ScrollingPresenter object
theLine	Integer object, the line in the scroll box's list
doAction	NameClass object, either @dontAction or@doAction

Do not call this method directly from the scripter. It is visible to the scripter so that it can be overridden in an instance or subclass of ScrollBox. The downReceiver and upReceiver methods are the scroll box's event receivers for mousedown events. A ScrollBox object automatically manages interests in mouseup and mousedown events, which determine selection from the target list. (The target list is implemented by a subclass of ScrollBox, through the targetPresenter instance variable.)

When the scroll box receives a mousedown event, it calls `selectLine`, supplying *theLine* and `@dontAction` as the value of the flag *doAction*. When the scroll box receives a mouseup event, it calls `selectLine`, supplying *theLine* and `@doAction` as the value of the flag *doAction*.

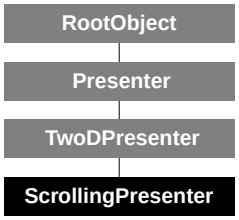
upReceiver

`upReceiver` *self interest event* ⇨ *self*

self	`ScrollingPresenter` object
interest	`MouseUpEvent` object, representing an event interest
event	`MouseUpEvent` object, representing an event

Do not call this method directly from the scripter. It is visible to the scripter so that it can be overridden in an instance or subclass of `ScrollBox`. The `upReceiver` method is the scroll box's event receiver for mouseup events. A `ScrollBox` object automatically manages interests in mouseup and mousedown events, which it uses to manage selection from a target list. (The target list is implemented by a subclass of `ScrollBox`.)

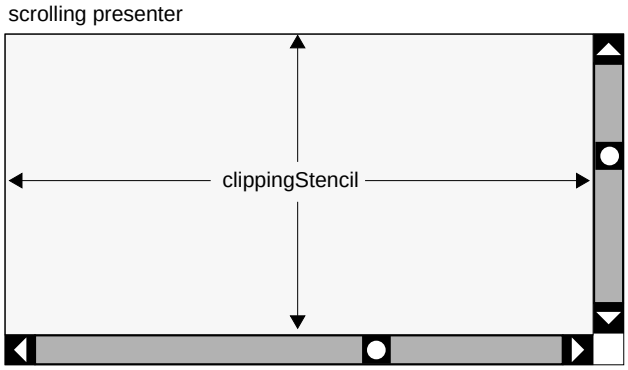
ScrollingPresenter



Class type: Core class (concrete)
Resides in: ScriptX and KMP executables
Inherits from: `TwoDPresenter`
Component: User Interface

The `ScrollingPresenter` class displays a presenter that is potentially larger than the boundary of the scrolling presenter itself. This presenter being scrolled is kept in the `targetPresenter` instance variable. For scrolling text, the target presenter is an instance of `TextPresenter` or `TextEdit`. However, you can scroll any scene by assigning it to the `targetPresenter`.

When you create a scrolling presenter, you can supply one or two scroll bars for scrolling different parts of the target presenter into view. As an alternative, you can manipulate the target presenter using the `scrollTo` method. If scroll bars are present, the `scrollTo` method moves the thumb presenters on each scroll bar as it repositions the target presenter within the scrolling presenter. The `ScrollingPresenter` class lays out the scroll bars automatically by calling layout on itself when `horizScrollBar`, `vertScrollBar`, or `targetPresenter` is modified; they enclose the clipping stencil, and clip the target presenter.



Creating and Initializing a New Instance

The following script is an example of how to create a new instance of the `ScrollingPresenter` class:

```
myScrollingPresenter := new ScrollingPresenter \
       vertScrollBar:(new ScrollBar orientation:@vertical) \
       horizScrollBar:(new ScrollBar orientation:@horizontal) \
       boundary:(new Rect x2:100 y2:100) \
       targetPresenter:(new TextPresenter \
           boundary:(new Rect x2:200 y2:200) \
           target:("Hello" as Text) )
```

The variable `myScrollingPresenter` contains the initialized scrolling presenter. The script creates horizontal and vertical scrollbars, which are automatically sized to fit inside the scrolling presenter's boundary. The target presenter is a text presenter. This text presenter targets the "Hello" instance of `Text`. The `new` method uses the keywords defined in `init`.

init

`init` *self* [`vertScrollBar:`*scrollBar*] [`horizScrollBar:`*scrollBar*] [`fill:`*brush*] [`stroke:`*brush*] [`targetPresenter:`*presenter*] [`boundary:`*rect*] [`target:`*object*] ⇨ *(none)*

self	`ScrollingPresenter` object
`vertScrollBar:`	`ScrollBar` object
`horizScrollBar:`	`ScrollBar` object
`fill:`	`Brush` object
`stroke:`	`Brush` object
`targetPresenter:`	`TwoDPresenter` object

Superclasses of `ScrollingPresenter` use the following keywords:

`boundary:`	`Rect` object
`target:`	(Ignored by `ScrollingPresenter`)

Initializes the `ScrollingPresenter` object *self*, applying the values supplied with the keywords to the instance variables of the same name. Do not call `init` directly on an instance—it is automatically called by the `new` method.

If you omit an optional keyword, its default value is used. The defaults are:

```
vertScrollBar:undefined
horizScrollBar:undefined
fill:undefined
stroke:undefined
targetPresenter:undefined
boundary:(targetPresenter.boundary) y if target presenter is defined,
                                    otherwise (new Rect x2:0 y2:0)
target:undefined
```

The `target` keyword, defined by `Presenter`, is ignored. A subclass of `ScrollingPresenter` could use this keyword to set the target of a subpresenter or target presenter.

Instance Variables

Inherited from `Presenter`:

`presentedBy`	`subPresenters`	`target`

Inherited from `TwoDPresenter`:

`bBox`	`globalTransform`	`target`
`boundary`	`height`	`transform`
`clock`	`IsImplicitlyDirect`	`width`
`compositor`	`isTransparent`	`window`
`direct`	`isVisible`	`x`
`eventInterests`	`needsTickle`	`y`
`globalBoundary`	`position`	`z`
`globalRegion`	`stationary`	

The following instance variables are defined in `ScrollingPresenter`:

clippingPresenter

self.clippingPresenter (read-write) Stencil

Specifies the stencil used to clip the target presenter, which is the presenter being scrolled. The clipped area is positioned within the stencil of the scrolling presenter. Although the value of clippingPresenter is usually a rectangle, it can be any stencil. Note that clippingPresenter replaces clippingStencil, and that despite its name, it is a stencil and not a presenter.

fill (TwoDPresenter)

self.fill (read-write) Brush

Specifies the Brush object for rendering the background of the scrolling presenter *self*. This brush fills the boundary stencil before any other model objects are asked to draw. If the value of fill is undefined, then the background is transparent. This instance variable, as defined for scrolling presenters, is equivalent to the instance variable of the same name, defined by TwoDMultiPresenter. See also stroke.

For more information on stroke and fill, see both the "Spaces and Presenters" and the "2D Graphics" chapters of the *ScriptX Components Guide*.

horizScrollBar

self.horizScrollBar (read-write) ScrollBar

Specifies the horizontal ScrollBar object for the scrolling presenter *self*. The default value is undefined. The author must supply a scroll bar for one to be displayed. The horizontal scroll bar is positioned within the stencil of the scrolling presenter.

horizScrollBarDisplayed

self.horizScrollBarDisplayed (read-write) NameClass

Controls whether or not the ScrollBar object stored in horizScrollBar for the scrolling presenter *self* is displayed. The horizScrollBarDisplayed variable takes on one of the following values—its default value is @always.

@none – indicates that the horizontal scroll bar is never shown.

@asNeeded – indicates that the horizontal scroll bar is displayed only if the target presenter is larger than the scrolling presenter *self* in the vertical direction.

@always – indicates that the horizontal scroll bar is always displayed. If the target presenter is smaller than scrolling presenter *self*, the scroll bar is disabled.

stroke (TwoDPresenter)

self.stroke (read-write) Brush

Specifies the Brush object used for the outline of the scrolling presenter *self*. The stroking of the outline occurs after all contained objects have been rendered. If stroke is undefined, the border of the scrolling presenter *self* is invisible. This instance variable, as defined for scrolling presenters, is equivalent to the instance variable of the same name, defined by TwoDMultiPresenter. See also fill.

For more information on stroke and fill, see both the "Spaces and Presenters" and the "2D Graphics" chapters of the *ScriptX Components Guide*.

subpresenters

self.subpresenters (read-write) Array

Holds the three main parts of the ScrollingPresenter instance *self*: the vertical scroll bar, horizontal scroll bar, and an instance of TwoDMultiPresenter that holds the target presenter. As you assign scroll bars to vertScrollBar and horizScrollBar, these scroll bars are added to the subpresenters array. Note that objects that you add to the target presenter are indirect, rather than direct, subpresenters.

targetPresenter

self.targetPresenter (read-write) TwoDPresenter

Specifies the presenter to be scrolled inside the scrolling presenter *self*. Note that this presenter can be any instance of TwoDPresenter that is not a direct presenter. For information on direct presenters, see the "Spaces and Presenters" chapter of the *ScriptX Components Guide*. Subclasses of ScrollingPresenter often place additional restrictions on the value of targetPresenter.

vertScrollBar

self.vertScrollBar (read-write) ScrollBar

Specifies the vertical ScrollBar object for the scrolling presenter *self*. The default value is undefined. The author must supply a scroll bar for one to be displayed. The vertical scroll bar is positioned within the stencil of the scrolling presenter.

vertScrollBarDisplayed

self.vertScrollBarDisplayed (read-write) NameClass

Controls whether or not the ScrollBar object stored in vertScrollBar is displayed. The vertScrollBarDisplayed variable can take one of the following values—its default value is @always.

@none – indicates that the vertical scroll bar is never shown.

@asNeeded – indicates that the vertical scroll bar is displayed only if the target presenter is larger than the scrolling presenter *self* in the vertical direction.

@always – indicates that the vertical scroll bar is always displayed. If the target presenter is smaller than scrolling presenter *self*, the scroll bar is disabled.

Instance Methods

Inherited from TwoDPresenter:

adjustClockMaster	hide	refresh
createInterestList	inside	show
draw	localToSurface	surfaceToLocal
getBoundaryInParent	notifyChanged	tickle

The following instance methods are defined in ScrollingPresenter:

handleHorizScroll

handleHorizScroll *self scrollBar value* ⇨ *self*

self	ScrollingPresenter object
scrollBar	ScrollBar object, the attached horizontal scroll bar
value	Integer object, the value instance variable on *scrollBar*

Do not call this method directly from the scripter. It is visible to the scripter so that it can be overridden in a subclass of `ScrollingPresenter`. If the horizontal scroll bar *scrollBar* is defined, its `valueAction` instance variable is set to call `handleHorizScroll`. The user triggers a call to `handleHorizScroll` automatically by moving the thumb presenter on the horizontal scroll bar. This method scrolls the target presenter for the scrolling presenter *self*, based on the *value* of this scroll bar.

handleVertScroll

`handleVertScroll` *self scrollBar value* ⇨ *self*

self	`ScrollingPresenter` object
scrollBar	`ScrollBar` object, the attached vertical scroll bar
value	`Integer` object, the `value` instance variable on *scrollBar*

Do not call this method directly from the scripter. It is visible to the scripter so that it can be overridden in a subclass of `ScrollingPresenter`. If the vertical scroll bar *scrollBar* is defined, its `valueAction` instance variable is set to call `handleVertScroll`. The user triggers a call to `handleVertScroll` automatically by moving the thumb presenter on the vertical scroll bar. This method scrolls the target presenter for the scrolling presenter *self*, based on the *value* of this scroll bar.

layout

`layout` *self* ⇨ *self*

The `layout` method is triggered automatically whenever there is a change to the target presenter or properties of the scrolling presenter *self*. It can also be called directly. The `layout` method positions all scroll bars, and positions the `clippingStencil` object.

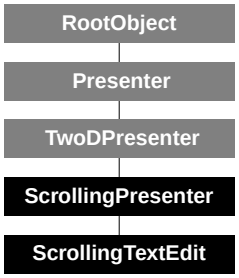
scrollTo

`scrollTo` *self x y* ⇨ *self*

self	`ScrollingPresenter` object
x	`Number` object, the local `x` coordinate of the target presenter
y	`Number` object, the local `y` coordinate of the target presenter

Scrolls the target presenter *self* so that the given point on the target presenter is moved to the top left corner of the box (if possible). This method automatically resets the `value` instance variable on the horizontal and vertical scroll bars, if they are defined, so that they move their thumb presenters and redraw themselves correctly. If either *x* or *y* is out of range, `scrollTo` scrolls the target presenter to the maximum or minimum value that is within range, as appropriate.

ScrollingTextEdit



Class type: Scripted class (concrete)
Resides in: `widgets.sxl`. Works with ScriptX and KMP executables
Inherits from: `ScrollingPresenter`
Component: User Interface

The `ScrollingTextEdit` class defines an editable area of text that can be scrolled. It comes with a vertical scroll bar that automatically adjusts itself properly for a given amount of text. The user can type into the text area, which is an instance of `TextEdit` whose target is the text.

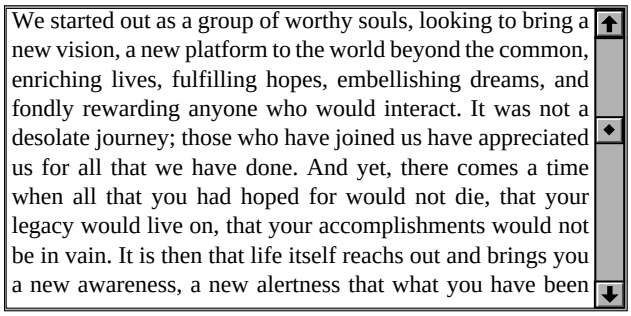


Figure 4-27: A ScrollingTextEdit object.

Known restriction in current version – The scroll bar should, but does not, become active when the typist fills up and then continues to types beyond the visible region. The scroll bar remains dimmed, which means it is not possible to view the bottom-most text. The workaround is to set the height of the `TextEdit` object to the maximum size the typist would encounter.

Creating and Initializing a New Instance

The following script is an example of how to create a new instance of the `ScrollingTextPresenter` class:

```
myScrollingPresenter := new ScrollingPresenter \
       vertScrollBar:(new ScrollBar orientation:@vertical) \
       horizScrollBar:(new ScrollBar orientation:@horizontal) \
       boundary:(new Rect x2:100 y2:100) \
       targetPresenter:(new TextPresenter \
           boundary:(new Rect x2:200 y2:200) \
           target:greatAmericanNovel)
```

The variable `myScrollingPresenter` contains the initialized scrolling presenter. The script creates horizontal and vertical scrollbars, which are automatically sized to fit inside the scrolling presenter's boundary. The target presenter is a text presenter. This

text presenter targets an instance of `String`, stored in the variable `greatAmericanNovel`, that is defined elsewhere. The `new` method uses the keywords defined in `init`.

init

`init` *self* `boundary:`*rect* [`text:`*text*] [`vertScrollBarDisplayed:`*boolean*]

⇨ *(none)*

self	`ScrollingPresenter` object
`text:`	`Text` object to be displayed
`vertScrollBarDisplayed:`	`NameClass` object `@always`, `@never` or `@asNeeded`

Initializes the `ScrollingPresenter` object *self*, applying the values supplied with the keywords to the instance variables of the same name. For simplicity, the inherited keywords `vertScrollBar`, `horizScrollBar`, `fill`, `stroke`, `targetPresenter`, and `target` are not shown here, since this class defines these for you.

Do not call `init` directly on an instance—it is automatically called by the `new` method.

If you omit an optional keyword, its default value is used. The defaults are:

```
text:("" as Text)
vertScrollBar:(new SimpleScrollBar orientation:@vertical \
                        height:boundary.height)
horizScrollBar:undefined
fill:whiteBrush
stroke:undefined
targetPresenter:(an instance of TextEdit that fits in the boundary)
target:undefined (ignored)
```

Instance Variables

Inherited from `Presenter`:

`presentedBy`	`subPresenters`	`target`

Inherited from `TwoDPresenter`:

`bBox`	`globalTransform`	`target`
`boundary`	`height`	`transform`
`clock`	`IsImplicitlyDirect`	`width`
`compositor`	`isTransparent`	`window`
`direct`	`isVisible`	`x`
`eventInterests`	`needsTickle`	`y`
`globalBoundary`	`position`	`z`
`globalRegion`	`stationary`	

Inherited from `ScrollingPresenter`:

`clippingStencil`	`stroke`
`fill`	`targetPresenter`
`horizScrollBar`	`vertScrollBar`
`horizScrollBarDisplayed`	`vertScrollBarDisplayed`

The following instance variables are defined in `ScrollingTextPresenter`:

font

self`.font` (read-write) `FontContext`

Specifies the initial font assigned to the instance of `TextEdit` in the scrolling text presenter *self*.

text

self.`text` (read-write) `Text`

Specifies the text that is scrolled in the scrolling text presenter *self*.

Instance Methods

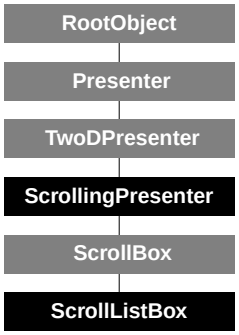
Inherited from `TwoDPresenter`:

`adjustClockMaster`	`hide`	`refresh`
`createInterestList`	`inside`	`show`
`draw`	`localToSurface`	`surfaceToLocal`
`getBoundaryInParent`	`notifyChanged`	`tickle`

Inherited from `ScrollingPresenter`:

`handleHorizScrollBar`	`layout`
`handleVertScrollBar`	`scrollTo`

ScrollListBox



Class type: Scripted class (concrete)
Resides in: `widgets.sxl`. Works with ScriptX and KMP executables
Inherits from: `ScrollBox`
Component: User Interface

The `ScrollListBox` class defines a scrollable list of strings, one of which can be selected at a time, as shown in Figure 4-28. The list can be longer than the visible region, so that you need to scroll the list to bring the hidden part into view. It comes with an optional vertical scroll bar that automatically adjusts itself properly for a given amount of text.

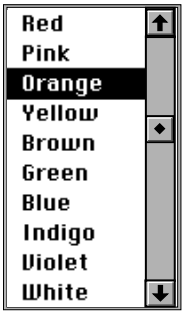


Figure 4-28: A ScrollListBox object

Creating and Initializing a New Instance

The following script is an example of how to create a new instance of the `ScrollingPresenter` class:

```
myScrollListBox := new ScrollListBox \
       list:#("Red", "Pink", "Orange", "Yellow", "Brown", \
              "Green", "Blue", "Indigo", Violet", "White") \
       hasScrollBar:true \
       value:3 \
       boundary:(new Rect x2:100 y2:200)
```

The variable `myScrollListBox` contains the initialized scrolling list box shown in Figure 4-28. The list box contains the strings "Red", "Pink", etc., and has a vertical scrollbar, which is automatically sized to fit inside the scrolling box's boundary of 100 x 200 pixels. The `value` keyword initially selects the 3nd item in the list. The `new` method uses the keywords defined in `init`.

init

`init` *self* [`list:`*sequence*] [`hasScrollBar:`*boolean*] [`value:`*integer*]
[`boundary:`*stencil*] [`font:`*fontContext*]
[`fill:`*brush*] [`stroke:`*brush*] ⇨ *(none)*

self	`ScrollListBox` object
`list:`	`Sequence` object containing items in the list
`font:`	`FontContext` object of items in the list
`hasScrollBar:`	`Boolean` object indicating if scroll bar is present
`value:`	`Integer` object indicating initial item selected

Superclasses of `ScrollListBox` use the following keywords:

`boundary:`	`Rect` object for the overall size of the box
`fill:`	`Brush` object (ignored by `ScrollListBox`)
`stroke:`	`Brush` object (ignored by `ScrollListBox`)
`targetPresenter:`	`TwoDPresenter` object (ignored by `ScrollListBox`)
`vertScrollBar:`	`ScrollBar` object (ignored by `ScrollListBox`)
`horizScrollBar:`	`ScrollBar` object (ignored by `ScrollListBox`)
`target:`	Any object (ignored by `ScrollListBox`)

Initializes the `ScrollListBox` object *self*, applying the values supplied with the keywords to the instance variables of the same name. Do not call `init` directly on an instance—it is called automatically by the `new` method.

If you omit an optional keyword, its default value is used. The defaults are:

```
font:theSystemFont
hasScrollBar:false
list:#()
value:undefined
fill:undefined
stroke:undefined
targetPresenter:undefined
```

Instance Variables

Inherited from `Presenter`:

`presentedBy`	`subPresenters`	`target`

Inherited from `TwoDPresenter`:

`bBox`	`globalTransform`	`target`
`boundary`	`height`	`transform`
`clock`	`IsImplicitlyDirect`	`width`
`compositor`	`isTransparent`	`window`
`direct`	`isVisible`	`x`
`eventInterests`	`needsTickle`	`y`
`globalBoundary`	`position`	`z`
`globalRegion`	`stationary`	

Inherited from `ScrollBox`:

`authorData`	`lastLine`	`stepAmount`
`doubleClickAction`	`list`	`targetPresenter`
`doubleClickTime`	`numLines`	`value`
`fill`	`pageAmount`	`width`
`font`	`selectAction`	
`frame`	`selectedLine`	

Instance Methods

Inherited from TwoDPresenter:

adjustClockMaster	hide	refresh
createInterestList	inside	show
draw	localToSurface	surfaceToLocal
getBoundaryInParent	notifyChanged	tickle

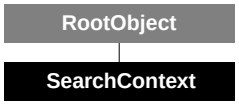
Inherited from ScrollingPresenter:

handleHorizScroll	layout	scrollTo
handleVertScroll		

Inherited from: ScrollBox:

downReceiver
repeatScrollAction
selectLine
upReceiver

SearchContext



Class type: Core class (concrete)
Resides in: ScriptX and KMP executables
Inherits from: `RootObject`
Component: Text and Fonts

The class `SearchContext` is used only in conjunction with instances of the class `StringIndex`. It is used as an argument to the global function `searchIndex` (which is also used only with instances of `StringIndex`) and provides information about where to search. Refer to the entry for `StringIndex` for more information.

Creating and Initializing a New Instance

You never create an instance of `SearchContext` by calling `new`. Instead, you call one of two global functions, `searchIndex` or `initialSearchContext`, each of which returns an instance of `SearchContext`.

- `initialSearchContext` – to create a `SearchContext` object suitable for searching for the first occurrence of a range of characters

The function `initialSearchContext` takes an instance of the class `StringIndex` as its first argument and the cursor position for beginning a search as its second argument. See the entry for `String` for an explanation of cursor position (as opposed to ordinal position).

The following code is an example of creating an instance of `SearchContext` using `initialSearchContext`:

```
global strIndex := new StringIndex string:"This is his test string."
global sc0 := initialSearchContext strIndex 0
```

The variable `sc0` contains a `SearchContext` object suitable for starting a search at the beginning of the string associated with `strIndex` , using the global function `searchIndex`. You simply supply `sc0` as the third argument to `searchIndex` to begin the search.

- `searchIndex` – to create a `SearchContext` object suitable for searching for the second and subsequent occurrences of a range of characters

The function `searchIndex` takes a `SearchContext` object as an argument, and it also returns an instance of `SearchContext`.

The following code is an example of creating an instance of `SearchContext` using `searchIndex`:

```
global sc1:= searchIndex strIndex "his" sc0 false
```

The variable `sc1` contains a `SearchContext` object suitable for a search of the second occurrence of "his" in the string associated with `strIndex` when using the global function `searchIndex`.

If you want to search for multiple occurrences of a string which is being matched, starting from the beginning of a `StringIndex` object's `string`, you use `initialSearchContext` with a *startOffset* of `0` to get the *searchContext* for the first call to

`searchIndex`. Then you use the return value of `searchIndex` as *searchContext* in subsequent calls to `searchIndex`. An example is included after the definition of `initialSearchContext` in the "Instance Methods" section which follows.

There is no `init` method for `SearchContext`.

Instance Variables

string

self`.string` (read-write) `String`

Contains a string which is the `string` instance variable of the `StringIndex` object passed to `initialSearchContext`.

startOffset

self`.startOffset` (read-write) `Integer`

Specifies a cursor position in *self*`.string` that defines the beginning of a character range.

When an instance of `SearchContext` is being used as an argument to the global function `searchIndex`, `startOffset` indicates where to begin searching. When an instance of `SearchContext` is the return value of the global function `searchIndex`, `startOffset` indicates where a successful match begins.

endOffset

self`.endOffset` (read-write) `Integer`

Specifies a cursor position in *self*`.string` that defines the end of a character range.

When an instance of `SearchContext` is the return value of the global function `searchIndex`, `endOffset` indicates the cursor position just after the last character of a successful match. (Unlike `startOffset`, `endOffset` is not used as an input argument; it is used only as an output argument.)

When a `SearchContext` object is returned by a call to `searchIndex`, its `startOffset` and `endOffset` instance variables are cursor positions which define the range of characters which constitute a match.

Example:

```
"This is some sample text."
```

When `startOffset` is 1 and `endOffset` is 4, they define the character set "his" in the sentence above.

Instance Methods

The following global function is defined to create an instance of `SearchContext`:

initialSearchContext

`initialSearchContext` *strIndex startOffset* ⇨ `SearchContext`

strIndex	`StringIndex` object.
startOffset	`Integer` object

(This function is technically a global function, but since its only operation is to create an instance of `SearchContext`, it is included here as if it were an instance method. Unlike an instance method, however, its first argument is not a `SearchContext` object, but rather a `StringIndex` object. This function is also listed under global functions.)

Returns an instance of `SearchContext` which can be used as the third argument to `searchIndex` to start a search of the string associated with *strIndex* at the cursor position designated by *startOffset*.

The following example illustrates using `initialSearchContext` to get the initial search context and then using it in `searchIndex` to find the first occurrence of the string being matched:

```
-- create a StringIndex object initialized with a StringConstant
-- object
global i := new StringIndex string: "This is his best histogram."

-- create a SearchContext object which can be used to start a search at
-- the beginning of the string
global sc := initialSearchContext i 0

-- search for the first occurrence of "his" and store the result in a
-- variable
global sc1 := searchIndex i "his" sc false
```

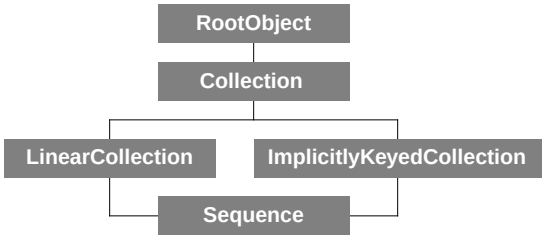
You can now use the variable `sc1` as the search context in the search for the second occurrence of "his", and so on. See `StringIndex` for a more complete example.

You can begin a search anywhere in a string by simply changing the second argument supplied to `initialSearchContext`. The following code demonstrates how to search for the first occurrence of “his”, beginning the search at the fourteenth character of the string associated with `i` (the character immediately following cursor position 13):

```
global scStartAt13 := initialSearchContext i 13
global nextSc := searchIndex i "his" scStartAt13 false
```

The variable *nextSc* can now be used as the *searchContext* argument to `searchIndex` to find the next occurrence of “his” in the string associated with `i`, if there is one.

Sequence



Class type: Core class (abstract)
Resides in: ScriptX and KMP executables
Inherits from: `LinearCollection` and `ImplicitlyKeyedCollection`
Component: Collections

The `Sequence` class is an abstract class for collections in which the key for any given item is its ordinal position in the sequence. Sequences are the collections used most commonly by other classes in the Kaleida Media Player. Through the `IndirectCollection` class, classes throughout the system such as `TitleContainer`, `TwoDMultiPresenter`, `Controller`, and `Window` use the Sequence protocol.

Think of a sequence as a collection in which order is not determined by key, as it is with the explicitly keyed collections. The Sequence protocol is concerned with the ordering of elements. `Sequence` defines methods for adding elements at a particular position, and for setting, sorting, and rearranging elements. Since subclasses of `Sequence` store their elements in a variety of data structures, each concrete subclass must implement certain methods, which are listed on page 665.

Methods defined in `Sequence` can be put into the categories that follow. (Note that the methods for accessing and deleting positional elements are defined in `LinearCollection`.)

Table 4-9: Categories of Sequence methods

ADDING	SETTING	REARRANGING
`addFirst`	`setFirst`	`moveBackward`
`addSecond`	`setSecond`	`moveForward`
`addThird`	`setThird`	`moveToBack`
`addFourth`	`setFourth`	`moveToFront`
`addFifth`	`setFifth`	
`addNth`	`setLast`	SORTING
`append`	`setNth`	`sort`
`appendNew`		
`prepend`		
`prependNew`		

Class Methods

Inherited from `Collection`:

`pipe`

Instance Variables

Inherited from `Collection`:

`bounded`	`maxSize`	`size`
`iteratorClass`	`minSize`	`uniformity`
`keyEqualComparator`	`mutable`	`uniformityClass`
`keyUniformity`	`mutableCopyClass`	`valueEqualComparator`
`keyUniformityClass`	`proprietored`	

Instance Methods

Inherited from `Collection`:

`add`	`forEach`	`iterate`
`addMany`	`forEachBinding`	`localEqual`
`addToContents`	`getAll`	`map`
`chooseAll`	`getAny`	`merge`
`chooseOne`	`getKeyAll`	`pipe`
`chooseOneBinding`	`getKeyOne`	`prin`
`deleteAll`	`getMany`	`removeAll`
`deleteBindingAll`	`getOne`	`removeOne`
`deleteBindingOne`	`hasBinding`	`setAll`
`deleteKeyAll`	`hasKey`	`setOne`
`deleteKeyOne`	`intersects`	`size`
`deleteOne`	`isEmpty`	
`emptyOut`	`isMember`	

Inherited from `LinearCollection`:

`chooseOneBackwards`	`findRange`	`getNthKey`
`chooseOrdOne`	`forEachBackwards`	`getOrdOne`
`deleteFirst`	`getFirst`	`getRange`
`deleteLast`	`getLast`	`localEqual`
`deleteNth`	`getMiddle`	`localLT`
`deleteRange`	`getNth`	`pop`

The following instance methods are defined in `Sequence`:

addFirst

`addFirst` *self value* ⇨ *key*
`addSecond` *self value* ⇨ *key*
`addThird` *self value* ⇨ *key*
`addFourth` *self value* ⇨ *key*
`addFifth` *self value* ⇨ *key*

self	`Sequence` object
value	Any object

Internally, these five methods are C-level macros that expand to:

```
addNth self 1 value
addNth self 2 value
addNth self 3 value
addNth self 4 value
addNth self 5 value
```

addNth

`addNth` *self ordinal value* ⇨ *key*

self	Sequence object
ordinal	`Number` object
value	Any object

Adds the given *value* to the sequence *self* so that it occupies the position specified by *ordinal*. This moves the element which was previously in position *ordinal* to position *ordinal* + 1 and similarly moves all the elements following it by 1. The ordinal position may be between `1` and (*self*`.size) + 1`. Returns the key at which the *value* was inserted. Note that there is no guarantee that the value will actually be placed at the specified ordinal. Some classes, such as `SortedArray`, override the `addNth` method, determining at what position an item can be inserted. However, classes without special positional semantics, such as `LinkedList`, `Array`, and `ArrayList`, behave as expected.

append

`append` *self value* ⇨ *key*

self	Sequence object
value	Any object

Appends the given *value* to the end of the sequence *self*. Returns the key at which the value was inserted. See also the global function `appendReturningSelf`, which calls `append` implicitly, but returns the sequence object rather than the key of the object being added. `appendReturningSelf` is used internally by ScriptX language constructs that append values to a sequence.

appendNew

`appendNew` *self value* ⇨ *key*

self	Sequence object
value	Any object

Is like `append`, except that it adds a new *value* only if it isn't found in the `Sequence` object *self*. If a value is successfully added, the method returns the key for the new value; otherwise, it returns `empty`. See also `prependNew`.

moveBackward

`moveBackward` *self value* ⇨ `Boolean`

self	Sequence object
value	Any object

Moves the given *value*, if found, backward in the sequence *self* by swapping the value with its neighbor. This method does nothing if the value is not found or is already at the end of the sequence. It returns `true` if the value was found, and `false` if it was not found.

moveForward

`moveForward` *self value* ⇨ `Boolean`

self	Sequence object
value	Any object

Moves the given *value*, if found, forward in the sequence *self* by swapping the value with its neighbor. This method does nothing if the value is not found or is already at the beginning of the sequence. It returns `true` if the value was found, and `false` if it was not found.

moveToBack

`moveToBack` *self value* ⇨ `Boolean`

self	Sequence object
value	Any object

Moves the given *value*, if found, to the back of the sequence *self* and shifts all the succeeding items forward. This method does nothing if the value is not found or is already at the end of the sequence. It returns `true` if the value was found, and `false` if it was not found.

moveToFront

`moveToFront` *self value* ⇨ `Boolean`

self	Sequence object
value	Any object

Moves the given *value*, if found, to the front of the sequence *self* and shifts all the preceding items backward. This method does nothing if the value is not found or is already at the beginning of the sequence. It returns `true` if the value was found, and `false` if it was not found.

prepend

`prepend` *self value* ⇨ *key*

self	Sequence object
value	Any object

Prepends the given *value* to the sequence *self*. This has the same effect as `addNth` *self* `1` *value*. Returns the key at which the *value* was inserted.

prependNew

`prependNew` *self value* ⇨ *key*

self	Sequence object
value	Any object

This method is like `prepend`, except that it adds *value* only if it isn't found in the sequence *self*. If the value is successfully added, `prependNew` returns the collection's key for the value; otherwise, it returns `empty`.

setFirst

`setFirst` *self value* ⇨ *(none)*
`setSecond` *self value* ⇨ *(none)*
`setThird` *self value* ⇨ *(none)*
`setFourth` *self value* ⇨ *(none)*
`setFifth` *self value* ⇨ *(none)*

self	Sequence object
value	Any object

These five methods are really macros that expand to

```
setNth self 1 value
setNth self 2 value
setNth self 3 value
setNth self 4 value
setNth self 5 value
```

setLast

`setLast` *self value* ⇨ *(none)*

self	`Sequence` object
value	Any object

Sets the last value in the sequence *self* to the given *value*. In other words, it replaces the last value. If the sequence is empty, it reports an error.

setNth

`setNth` *self ordinal value* ⇨ *(none)*

self	`Sequence` object
ordinal	`Integer` object
value	`Number` object

Sets the value at the *ordinal* position in the given sequence *self* to the given *value*. The *ordinal* may be between `1` and `(`*self*`.size) + 1`.

sort

`sort` *self ltFunction* ⇨ *(none)*

self	`Sequence` object
ltFunction	`Function` object

Sorts the values in the sequence *self*, using the given comparison function *ltFunction* (less-than function) on the values. The function should be written to take two values as arguments, for example `compfunc` *value1 value2*, and return `true` if *value1* should appear before *value2*. See the section "Comparison Functions" in the "Collections" chapter of the *ScriptX Components Guide* for a list of the built-in functions you can use for *ltFunction*. Since sorted collections are already sorted, the `sort` method has no effect on them unless *ltFunction* changes.

Subclasses Must Implement

`Collection` defines several methods that a sequence must implement. For more information on these methods in the Collection protocol, see page 176.

Subclasses of `Sequence` must implement the following methods. These methods report an exception if called but not implemented.

```
addNth
deleteNth
getNth
```

A more efficient subclass should implement:

```
append
isMember
```

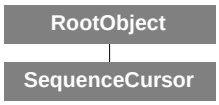
Subclasses whose size may be `@unknown` should implement:

```
append
hasBinding
hasKey
```

Subclasses that need to use a comparator function other than `ueq` should implement:

```
valueEqualComparatorGetter
keyEqualComparatorGetter
```

SequenceCursor



Class type: Core class (abstract)
Resides in: ScriptX and KMP executables
Inherits from: `RootObject`
Component: Collections

The `SequenceCursor` is an abstract mixin class. It is designed to be mixed in to a subclass of `Sequence` to give the user a "cursor" into the sequence and control over that cursor. In a sense, it is an iterator, but the `SequenceCursor` class keeps the cursor pointing to the same item even when items are deleted or added (and does the right thing when the current item is deleted).

For example, if the cursor is pointing to item 15 and a previous item is deleted (such as item 10), then the cursor continues to point to the same item, which is now item 14. Conversely, if the cursor is pointing to item 15 and a value is inserted at any previous item, then the cursor will continue pointing to the same item, which is item 16. If the cursor is pointing to item 15, and that item is deleted, the cursor will point to the new item 15.

You can navigate through the sequence with the `goTo` method. The `goTo` method takes as an argument an ordinal position in the sequence; when called, `goTo` points the cursor at that position. The `goTo` method contains the fundamental implementation for `SequenceCursor`. The instance variable `cursor` and the methods `forward` and `backward` all rely on the `goTo` method.

One of the benefits of `SequenceCursor` is that the `goTo` method gets called every time an item is deleted or its value is changed. Subclasses need only override `goTo` to gain control of a `SequenceCursor` object's behavior. You would typically override `goTo` as follows:

```
method goTo self ordinal -> (
    apply nextMethod goTo self ordinal
    -- your own title's implementation that happens every "goTo"
)
```

`SequenceCursor` is used by `OneOfNPresenter`. It allows you to go forward, go backward, and go directly to any presenter.

Another benefit of `SequenceCursor` is that it allows you to use any of the many collection methods directly on the collection (such as `deleteNth`), rather than limiting you to the smaller set of iterator operations.

Creating and Initializing a New Instance

`SequenceCursor` is not meant to be instantiated directly—it is a true mixin class, in that it provides all the functionality you need for adding a cursor to a sequence—you don't have to implement the functionality.

When you create a new class and mix `SequenceCursor` with a `Sequence` subclass (using the `class` keyword), you should mix `SequenceCursor` in *ahead* of the `Sequence` subclass. This arrangement gives priority to the `SequenceCursor` methods such as `deleteNth` over those in `Array`.

The following script mixes `SequenceCursor` with `Array` to create a new class called `ArrayCursor`, and then creates an instance of that class:

```
class ArrayCursor(SequenceCursor, Array)
end
seqObj := new ArrayCursor initialSize:10 growable:true
```

The `seqObj` variable holds a new instance of the `ArrayCursor` class. It has an initial size of 10 and can grow beyond that range. Notice that this new class takes the same keywords that `Array` takes.

Instance Variables

cursor

self`.cursor` (read-write) `Integer`

Specifies the current cursor location in the sequence *self*. Setting the `cursor` instance variable calls the `goTo` method:

```
goTo self (self.cursor)
```

Instance Methods

backward

`backward` *self* ⇨ *(object)*

Moves the cursor back one position in the sequence *self*, and returns the object at this new cursor position. This method calls the `goTo` method:

```
goTo self (self.cursor - 1)
```

forward

`forward` *self* ⇨ *(object)*

Moves the cursor forward one position in the sequence *self*, and returns the object at this new cursor position. This method calls:

```
goTo self (self.cursor + 1)
```

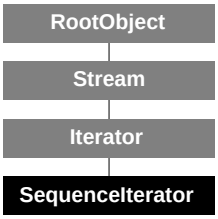
goTo

`goTo` *self ordinal* ⇨ *(object)*

self	`SequenceCursor` object
ordinal	`Integer` object

Moves the cursor to the position specified by *ordinal* in the sequence *self*, and returns the object at this new cursor position. (Because ScriptX is not sensitive to uppercase or lowercase letters, you can type this `goto` if you prefer.)

SequenceIterator



Class type: Core class (concrete)
Resides in: ScriptX and KMP executables
Inherits from: `Iterator`
Component: Collections

A `SequenceIterator` object iterates over `Sequence` objects.

Creating and Initializing a New Instance

You create a new instance of `SequenceIterator` by calling `iterate` on a sequence.

For any given collection, the Kaleida Media Player uses the value of the virtual instance variable `iteratorClass` to determine which iterator to create. Every subclass of `Collection` must implement the `iteratorClassGetter` method. For most subclasses of `Sequence`, the value of `iteratorClass` is `SequenceIterator`. (An exception is `LinkedList`, which has its own iterator class.)

Instance Variables

Inherited from `Iterator`:

`key`	`source`	`value`

Instance Methods

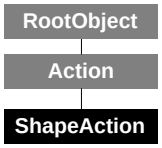
Inherited from `Stream`:

`cursor`	`next`	`seekFromStart`
`flush`	`plug`	`setStreamLength`
`isAtFront`	`previous`	`streamLength`
`isPastEnd`	`read`	`write`
`isReadable`	`readReady`	`writeReady`
`isSeekable`	`seekFromCursor`	
`isWritable`	`seekFromEnd`	

Inherited from `Iterator`:

`excise`	`seekKey`	`seekValue`
`remainder`		

ShapeAction



Class type: Core class (concrete)
Resides in: ScriptX and KMP executables
Inherits from: `Action`
Component: Animation

The `ShapeAction` class represents an action that will change the shape that a `TwoDShape` object displays at a specified time, by changing its stencil, under the control of an action list player. To be triggered, an instance of `ShapeAction` needs to be added to the action list of an action list player, then the player needs to be played. Stencils include `Rect`, `Oval`, `Line`, `Bitmap`, and other classes.

Creating and Initializing a New Instance

The following script is an example of how to create a new instance of the `ShapeAction` class:

```
myAction := new ShapeAction \
              shape:(new Oval x2:50 y2:50) \
              targetNum:3
              time:15
```

The variable `myAction` holds an initialized instance of `ShapeAction`. This instance specifies that, at the player's time of 15 ticks, the player's third target (`targetNum:3`), which is an instance of `TwoDShape`, changes its shape to an oval. The `new` method uses the keywords defined in `init`.

init

`init` *self* [`shape:`*stencil*] [`targetNum:`*integer*] [`time:`*integer*] ⇨ *(none)*

self	`ShapeAction` object
`shape:`	`Stencil` object indicating the final shape for the target object

Superclass `Action` uses the following keywords:

`targetNum:`	`Integer` object indicating which 2D shape in the target list of the player to change
`time:`	`Integer` representing the time in ticks to trigger the action

Initializes the `ShapeAction` object *self*, applying the values supplied with the keywords to the instance variables of the same name. At the time specified by `time`, this action causes the 2D shape specified by `targetNum` to change to the stencil specified by `shape`. If `shape` is not a `Stencil` object, this method reports an exception. Do not call `init` directly on an instance—it is automatically called by the `new` method.

If you omit an optional keyword, its default value is used. The defaults are:

```
targetNum:0
time:0
shape:undefined
```

Instance Variables

Inherited from `Action`:

`authorData` `targetNum` `time`
`playOnly`

The following instance variables are defined in `ShapeAction`:

shape

self`.shape` (read-write) `Stencil`

Specifies the new stencil that the target `TwoDShape` object should display when the shape action *self* is triggered.

Instance Methods

Inherited from `Action`:

`trigger`

The following instance method is defined in `ShapeAction`:

trigger (Action)

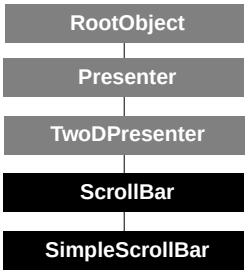
`trigger` *self target player* ⇨ *target*

self	`ScriptAction` object
target	Any object
player	`ActionListPlayer` object

Causes the shape action *self* to replace the stencil in the 2D shape *target* with the stencil specified in the `shape` instance variable. This method is called by an `ActionListPlayer` object at the time specified by the `time` instance variable. This action list player is passed in as the third argument *player*.

When the action list player calls trigger, the value for *target* is automatically determined by taking the `targetNum` instance variable and finding the object in the corresponding slot in the player's target list. If the `targetNum` instance variable is out of range (that is, less than 1 or greater than the size of the target list), the value of the *target* is `empty`. Also, if there is no *target* in a certain slot in the target list yet, the value of *target* is `undefined`.

SimpleScrollBar



Class type: Scripted class (concrete)
Resides in: `widgets.sxl`. Works with ScriptX and KMP executables
Inherits from: `ScrollBar`
Component: User Interface

`SimpleScrollBar`, from the ScriptX widget library, specializes `ScrollBar` to create a scrollbar that shares the look and feel of other widgets. `SimpleScrollBar` simplifies the initialization syntax of a standard scrollbar, supplying standard bitmaps for use in creating the stencils that comprise the graphical elements of scroll bars. (These class variables are not documented here, but are easily modified or specialized. See the source code for the ScriptX widget library.)

Creating and Initializing a New Instance

The following script creates a new instance of the `SimpleScrollBar` class:

```
mySimpleScrollBar := new SimpleScrollBar \
       height:100 \
       width:20
```

The variable `mySimpleScrollBar` contains the initialized scrollbar. In contrast with `ScrollBar`, the `orientation` keyword is not required. A `SimpleScrollBar` object determines its own orientation based on values supplied for the keywords `height` and `width`. Since the height of `mySimpleScrollBar` is greater than the width, the scrollbar's orientation will be vertical. (`SimpleScrollBar` uses the `orientation` instance variable, defined by `ScrollBar`—possible values for `orientation` are `@vertical` and `@horizontal`.)

As an alternative, you can set the `orientation` of the `SimpleScrollBar` object at initialization, and supply the major dimension (`height` for a vertical scrollbar; `width` for a horizontal scrollbar). The `SimpleScrollBar` class defines a class variable, `minSpan`, that is used to set the value of the other dimension to the minimum span.

```
anotherSimpleScrollBar := new SimpleScrollBar \
       height:100 \
       orientation:@vertical
```

The variable `anotherSimpleScrollBar` contains a vertical scrollbar whose width is set to the default value of 15, which is determined by the `minSpan` class variable.

The `new` method uses the keywords defined in `init`.

init

`init` *self* [`height:`*integer*] [`width:`*integer*] [`orientation:`*name*] ⇨ *(none)*

self	`SimpleScrollBar` object
`height:`	`Integer` object
`width:`	`Integer` object

Superclasses of SimpleScrollBar use the following keyword:

orientation:	NameClass object, @vertical or @horizontal
incrementStencil:	ignored by SimpleScrollBar
decrementStencil:	ignored by SimpleScrollBar
thumbStencil:	ignored by SimpleScrollBar
range:	ignored by SimpleScrollBar
boundary:	ignored by SimpleScrollBar

Initializes the SimpleScrollBar object *self*, applying the values supplied for width, height, and orientation to determine the boundary and orientation of the scrollbar. The values of incrementStencil, decrementStencil, thumbStencil, and range are set automatically. SimpleScrollBar maintains a set of bitmaps in its class variables that it uses to create stencils. Do not call init directly on an instance—it is automatically called by the new method.

If you omit one of the keyword arguments, the following defaults are used:

height:0
width:0
orientation:(@vertical or @horizontal, based on height and width)

Instance Variables

Inherited from Presenter:

presentedBy	subPresenters	target

Inherited from TwoDPresenter:

bBox	globalTransform	target
boundary	height	transform
clock	IsImplicitlyDirect	width
compositor	isTransparent	window
direct	isVisible	x
eventInterests	needsTickle	y
globalBoundary	position	z
globalRegion	stationary	

Inherited from ScrollBar:

authorData	fill	range
clock	incrementStencil	releaseInterest
decrementStencil	orientation	stepAmount
directDrag	pageAmount	stroke
disableBrush	pressDecrementStencil	thumbStencil
dragInterest	pressIncrementStencil	value
enabled	pressInterest	valueAction

Instance Methods

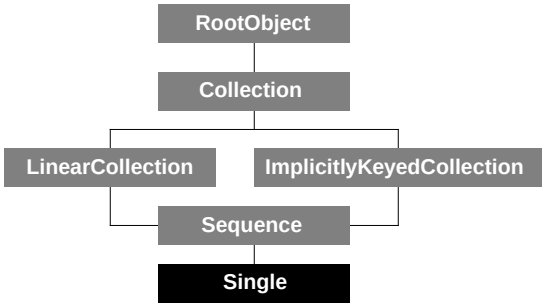
Inherited from TwoDPresenter:

adjustClockMaster	hide	refresh
createInterestList	inside	show
draw	localToSurface	surfaceToLocal
getBoundaryInParent	notifyChanged	tickle

Inherited from ScrollBar:

calculateSize	incrementPage	thumbDrag
decrementPage	incrementStep	thumbPress
decrementStep	layout	thumbRelease
handleValueChange	processEvent	trackPress

Single



Class type: Core class (concrete)
Resides in: ScriptX and KMP executables
Inherits from: `Sequence`
Component: Collections

The `Single` class represents an array with one value. A `Single` object is created with a fixed size of 1. You cannot add (`addNth`) or delete (`deleteNth`) any item, but you can set its value (`setNth`).

See also the `Pair`, `Triple`, and `Quad` classes.

Creating and Initializing a New Instance

To create an instance of `Single`, call the `new` method specifying the `Single` class as the first parameter. The following script is an example:

```
mySingle := new Single values:#(100)
```

The variable `mySingle` contains the initialized instance of `Single` with the value 100.

init

`init` *self* ⇨ *(none)*

self `Single` object

Initializes the `Single` object *self*, setting the one item in the list to `undefined`. Do not call `init` directly on an instance—it is automatically called by the `new` method.

Class Methods

Inherited from `Collection`:

`pipe`

Instance Variables

Inherited from `Collection`:

`bounded`	`maxSize`	`size`
`iteratorClass`	`minSize`	`uniformity`
`keyEqualComparator`	`mutable`	`uniformityClass`
`keyUniformity`	`mutableCopyClass`	`valueEqualComparator`
`keyUniformityClass`	`proprietored`	

Instance Methods

Inherited from `Collection`:

`add`	`forEach`	`iterate`
`addMany`	`forEachBinding`	`localEqual`
`addToContents`	`getAll`	`map`
`chooseAll`	`getAny`	`merge`
`chooseOne`	`getKeyAll`	`pipe`
`chooseOneBinding`	`getKeyOne`	`prin`
`deleteAll`	`getMany`	`removeAll`
`deleteBindingAll`	`getOne`	`removeOne`
`deleteBindingOne`	`hasBinding`	`setAll`
`deleteKeyAll`	`hasKey`	`setOne`
`deleteKeyOne`	`intersects`	`size`
`deleteOne`	`isEmpty`	
`emptyOut`	`isMember`	

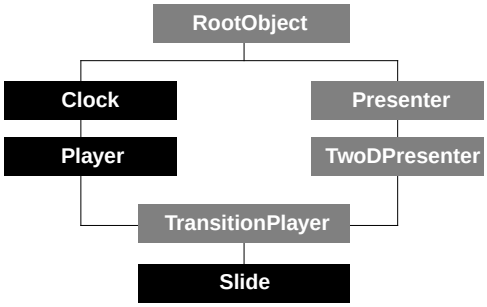
Inherited from `LinearCollection`:

`chooseOneBackwards`	`findRange`	`getNthKey`
`chooseOrdOne`	`forEachBackwards`	`getOrdOne`
`deleteFirst`	`getFirst`	`getRange`
`deleteLast`	`getLast`	`localEqual`
`deleteNth`	`getMiddle`	`localLT`
`deleteRange`	`getNth`	`pop`

Inherited from `Sequence`:

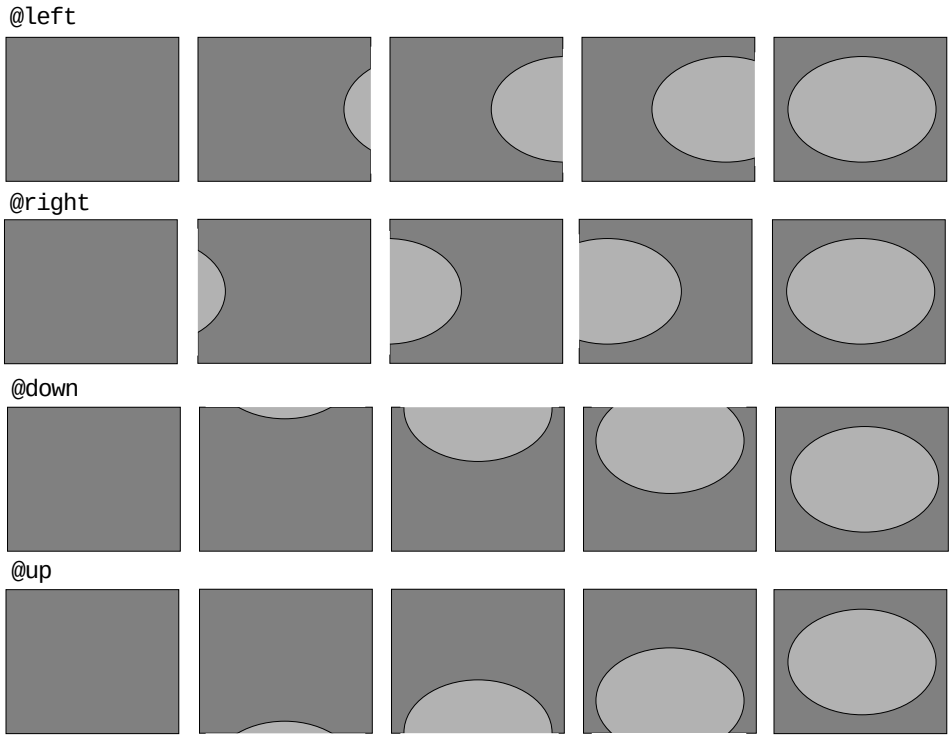
`addFifth`	`moveBackward`	`setFourth`
`addFirst`	`moveForward`	`setLast`
`addFourth`	`moveToBack`	`setNth`
`addNth`	`moveToFront`	`setSecond`
`addSecond`	`prepend`	`setThird`
`addThird`	`prependNew`	`sort`
`append`	`setFifth`	
`appendNew`	`setFirst`	

Slide



Class type: Core class (concrete)
Resides in: ScriptX and KMP executables
Inherits from: `TransitionPlayer`
Component: Transitions

The `Slide` class creates the visual effect of the target presenter sliding onto the screen. You specify the direction in which the target slides by setting the `direction` instance variable, which is defined by `TransitionPlayer`. Possible values include `@left`, `@right`, `@up`, and `@down`, as shown below. For example, `@left` means the target image slides onto the screen moving to the left.



Creating and Initializing a New Instance

The following script is an example of how to create a new instance of the `Slide` class:

```
myTransition := new Slide \
                duration:60 \
                direction:@left \
```

```
target:myShape
useOffscreen:true
```

The variable myTransition contains the initialized transition. When you play myTransition, the presenter myShape slides onto the screen from the right to the left over a period of 60 ticks. The player is set to use an offscreen bitmap for efficiency.

You determine in which space the transition will take effect in by adding this instance to that space. Then, when you play the transition player, myShape is "transitioned" into that space.

The new method uses the keywords defined in init.

init

init *self* [duration:*integer*] [direction:*name*]
[movingTarget:*boolean*] [useOffscreen:*boolean*] [target:*twoDPresenter*]
[boundary:*stencil*] [masterClock:*clock*] [scale: *integer*] ⇨ *(none)*

This method is inherited from TransitionPlayer with no change in keywords—refer to that class for details. Do not call init directly on an instance—it is automatically called by the new method.

Instance Variables

Inherited from Presenter:

presentedBy	subPresenters	target

Inherited from TwoDPresenter:

bBox	height	transform
boundary	IsImplicitlyDirect	width
clock	isTransparent	window
compositor	isVisible	x
direct	needsTickle	y
eventInterests	position	z
globalBoundary	stationary	
globalTransform	target	

Inherited from Clock:

callbacks	rate	ticks
effectiveRate	resolution	time
masterClock	scale	title
offset	slaveClocks	

Inherited from Player:

audioMuted	globalContrast	globalVolumeOffset
dataRate	globalHue	markerList
duration	globalPanOffset	status
globalBrightness	globalSaturation	videoBlanked

Inherited from TransitionPlayer:

autoSplice	direction	movingTarget
backgroundBrush	duration	target
cachedTarget	frame	useOffscreen

The following instance variables are defined in Slide:

direction (TransitionPlayer)

self.direction (read-write) NameClass

Specifies the direction in which the slide transition *self* should be applied. Possible values are @left, @right, @up, @down, @southeast, @northeast, @southwest or @northwest.

Instance Methods

Inherited from TwoDPresenter:

adjustClockMaster	inside	show
createInterestList	localToSurface	surfaceToLocal
draw	notifyChanged	tickle
getBoundaryInParent	recalcRegion	
hide	refresh	

Inherited from Clock:

addPeriodicCallback	clockAdded	pause
addRateCallback	clockRemoved	resume
addScaleCallback	effectiveRateChanged	timeJumped
addTimeCallback	forEachSlave	waitTime
addTimeJumpCallback	isAppropriateClock	waitUntil

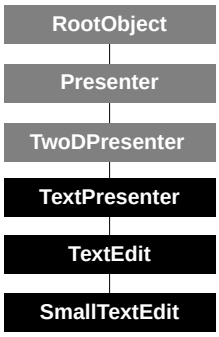
Inherited from Player:

addMarker	goToBegin	playPrepare
eject	goToEnd	playUnprepare
fastForward	goToMarkerFinish	playUntil
getMarker	goToMarkerStart	resume
getNextMarker	pause	rewind
getPreviousMarker	play	stop

Inherited from TransitionPlayer:

playPrepare

SmallTextEdit



Class type: Scripted class (concrete)
Resides in: `widgets.sxl`. Works with ScriptX and KMP executables
Inherits from: `TextEdit`
Component: User Interface

The `SmallTextEdit` class is a presenter for editable text that creates a simple field for text entry. Like the other objects in the ScriptX Widget Library, it is rendered with a simple box and drop shadow.

Creating and Initializing a New Instance

The following script creates a new instance of `SmallTextEdit`:

```
myEditField := new SmallTextEdit \
               boundary:(new Rect x2:150 y2:24) \
               font:myFontContext\
               text:"enter here"\
               fill:ltGrayBrush
```

The variable `myEditField` contains an initialized `SmallTextEdit` instance with a rectangular boundary of 150 x 24 pixels. Initially, this object presents the text "`enter here`." Its attributes are determined by `myFontContext`, and `ltGrayBrush`, a global instance of `Brush`, is used to render the fill. The `new` method uses keywords that are defined by `init`.

init

`init` *self* [`font:`*font*] [`text:`*string*] [`boundary:`*rect*] [`stationary:`*boolean*] [`fill:`*brush*] [`stroke:`*brush*] ⇨ *(none)*

self	`ListBox` object
`font:`	`FontContext` object
`text:`	`String` object

Superclasses of `SmallTextEdit` use these keywords.

`boundary:`	`Rect` object
`target:`	`String` object, ignored by `ListBox`
`stationary:`	`Boolean` object
`fill:`	`Brush` object
`stroke:`	`Brush` object

Initializes the `SmallTextEdit` object *self*, applying the arguments as follows: The `font` keyword takes a `FontContext` object, which it uses to set the attributes of the `SmallTextEdit` presenter. The `text` keyword takes a string, or any item that can be

freely coerced to a Text object, which it uses to set the presenter's target. The boundary sets the vertical and horizontal boundaries of the presenter. The target keyword is ignored, since the presenter's target is set at initialization, based on the value supplied for text. A script can set the values of stationary, fill, and stroke at initialization; they are applied to instance variables of the same name, defined by TwoDPresenter.Do not call init directly on an instance—it is called automatically by the new method.

If you omit an optional keyword, its default value is used. The defaults are:

```
font:theSystemFont
width:50
text:("")
stationary:true
fill:whiteBrush
stroke:blackBrush
```

Instance Variables

Inherited from Presenter:

presentedBy	subPresenters	target

Inherited from TwoDPresenter:

bBox	height	transform
boundary	IsImplicitlyDirect	width
clock	isTransparent	window
compositor	isVisible	x
direct	needsTickle	y
eventInterests	position	z
globalBoundary	stationary	
globalTransform	target	

Inherited from TextPresenter:

attributes	fill	selectionForeground
cursor	inset	stroke
cursorBrush	offset	
enabled	selectionBackground	

The following instance variables are defined in SmallTextEdit:

font

self.font (read-write) FontContext

Specifies the FontContext object that is used to determine the text attributes of the SmallTextEdit object *self*.

text

self.text (read-write) Sequence

Specifies the Text object that is used to set the target of the SmallTextEdit object *self*.

Instance Methods

Inherited from TwoDPresenter:

adjustClockMaster	hide	refresh
createInterestList	inside	show
draw	localToSurface	surfaceToLocal
getBoundaryInParent	notifyChanged	tickle

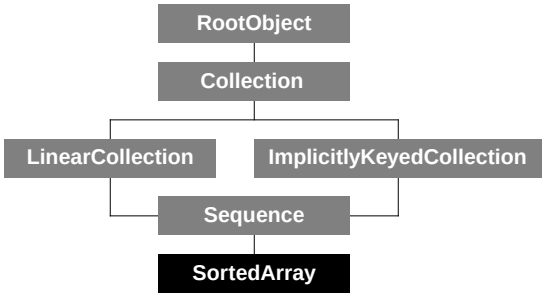
Inherited from `TextPresenter`:

`calculate`	`getOffsetForXY`	`processMouseDown`
`copySelection`	`getPointForOffset`	

Inherited from `TextEdit`:

`clearSelection`	`processFocus`	`processMouseMove`
`cutSelection`	`processKeydown`	`processMouseUp`
`pasteToSelection`	`processMouseDown`	

SortedArray



Class type: Core class (concrete)
Resides in: ScriptX and KMP executables
Inherits from: `Sequence`
Component: Collections

A `SortedArray` object is a list of values maintained in sorted order. New elements are not appended to the end of the array, but inserted into a position depending on value.

Objects that are added to a sorted array must implement the generic function `localLT` to allow for comparison with other objects. See the discussions of comparison in both the "Collections" and the "Object System Kernel" chapters of the *ScriptX Components Guide*.

Note – A `SortedArray` object does not automatically sort itself if you modify one of its member items. To modify an item in a sorted array, you should explicitly remove it, make a change, and add it back to the array.

Creating and Initializing a New Instance

The following script is an example of how to create a new instance of the `SortedArray` class:

```
myArray := new SortedArray \
          initialSize: 100 \
          growable: true \
          ltFunction: (x y -> return ((x as String)<(y as String)))
```

The variable `myArray` contains the initialized sorted array. The array's initial size is 100, it can grow, and its "less-than" function for sorting items is as specified by the given function. The `new` method uses the keywords defined in `init`.

init

`init` *self* [`initialSize:`*integer*] [`growable:`*boolean*] [`ltFunction:`*function*]
⇨ *(none)*

self	`SortedArray` object
`initialSize:`	`Integer` object
`growable:`	`Boolean` object
`ltFunction:`	A function

Initializes the `SortedArray` object *self*, applying the arguments as follows: `initialSize` is the amount of space to reserve for the initial set of items. `initialSize` must be 1 or larger. If `growable` is `true`, the collection cannot be expanded beyond its `initialSize`. If `growable` is `false`, it grows in chunks. The `ltFunction` parameter allows you to

specify the comparison function used in ordering the elements of the array. This function should take two arguments and return `true` when the first argument is less than the second, and return `false` otherwise.

If it is not growable, then the instance variable `maxSize` becomes the value that was passed in as `initialSize`. Otherwise, `initialSize` is used as a hint for the correct amount of memory to allocate. Do not call `init` directly on an instance—it is automatically called by the `new` method.

If you omit an optional keyword, its default value is used. The defaults are:

```
initialSize:20
growable:true
ltFunction:ult
```

ScriptX may in fact create a larger `SortedArray` object (but never smaller) than you specify with `initialSize`, due to its interaction with memory management at initialization time.

If you create a subclass of `SortedArray` that overrides the `init` method, be sure to invoke its superclass's `init` method using the `nextMethod` call.

Class Methods

Inherited from `Collection`:

`pipe`

Instance Variables

Inherited from `Collection`:

`bounded`	`maxSize`	`size`
`iteratorClass`	`minSize`	`uniformity`
`keyEqualComparator`	`mutable`	`uniformityClass`
`keyUniformity`	`mutableCopyClass`	`valueEqualComparator`
`keyUniformityClass`	`proprietored`	

Instance Methods

Inherited from `Collection`:

`add`	`forEach`	`iterate`
`addMany`	`forEachBinding`	`localEqual`
`addToContents`	`getAll`	`map`
`chooseAll`	`getAny`	`merge`
`chooseOne`	`getKeyAll`	`pipe`
`chooseOneBinding`	`getKeyOne`	`prin`
`deleteAll`	`getMany`	`removeAll`
`deleteBindingAll`	`getOne`	`removeOne`
`deleteBindingOne`	`hasBinding`	`setAll`
`deleteKeyAll`	`hasKey`	`setOne`
`deleteKeyOne`	`intersects`	`size`
`deleteOne`	`isEmpty`	
`emptyOut`	`isMember`	

Inherited from `LinearCollection`:

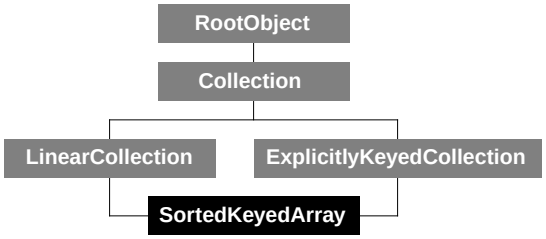
`chooseOneBackwards`	`findRange`	`getNthKey`
`chooseOrdOne`	`forEachBackwards`	`getOrdOne`
`deleteFirst`	`getFirst`	`getRange`
`deleteLast`	`getLast`	`localEqual`
`deleteNth`	`getMiddle`	`localLT`

deleteRange	getNth	pop

Inherited from Sequence:

addFifth	moveBackward	setFourth
addFirst	moveForward	setLast
addFourth	moveToBack	setNth
addNth	moveToFront	setSecond
addSecond	prepend	setThird
addThird	prependNew	sort
append	setFifth	
appendNew	setFirst	

SortedKeyedArray



Class type: Core class (concrete)
Resides in: ScriptX and KMP executables
Inherits from: `LinearCollection` and `ExplicitlyKeyedCollection`
Component: Collections

A `SortedKeyedArray` object is a list of key-value bindings, maintained in key-sorted order. Each binding that is appended to the collection is not stored at the end, but placed in order according to the key.

Keys that are added to a sorted keyed array must implement the generic function `localLT` to allow for comparison with other objects. See the discussions of comparison in both the "Collections" and the "Object System Kernel" chapters of the *ScriptX Components Guide*.

Note – A `SortedKeyedArray` does not automatically sort itself if you modify one of its keys. To modify a key in a sorted keyed array, you should explicitly remove it, make a change, and add it back to the array.

Creating and Initializing a New Instance

The following script is an example of how to create a new instance of the `SortedKeyedArray` class:

```
myArray := new SortedKeyedArray \
           initialSize:100 \
           growable:true \
           ltFunction:(x y -> return ((x as String)<(y as String)))
```

The variable `myArray` contains the initialized sorted keyed array. The array's initial size is 100, it can grow, and its "less-than" function for sorting items is as specified by the given function. The `new` method uses the keywords defined in `init`.

init

`init` *self* [`initialSize:`*integer*] [`growable:`*boolean*] [`ltFunction:`*function*]
⇨ *(none)*

self	`SortedKeyedArray` object
`initialSize:`	`Integer` object
`growable:`	`Boolean` object
`ltFunction:`	`Function` object

Initializes the `SortedKeyedArray` object *self*, applying the arguments as follows: `initialSize` is the amount of space to reserve for the initial set of items. `initialSize` must be 1 or larger. If `growable` is `true`, the collection cannot be expanded beyond its `initialSize`. If `growable` is `false`, it grows in chunks. The `ltFunction` parameter

allows you to specify the comparison function used in ordering the keys of the array. This function should take two arguments and return `true` when the first argument is less than the second, and return `false` otherwise.

If it is not growable, then the instance variable `maxSize` becomes the value that was passed in as `initialSize`. Otherwise, `initialSize` is used as a hint for the correct amount of memory to allocate. Do not call `init` directly on an instance—it is automatically called by the `new` method.

If you omit an optional keyword, its default value is used. The defaults are:

```
initialSize:20
growable:true
ltFunction:ult
```

ScriptX may in fact create a larger `SortedKeyedArray` object (but never smaller) than you specify with `initialSize`, due to its interaction with memory management at initialization time.

If you create a subclass of `SortedKeyedArray` that overrides the `init` method, be sure to invoke its superclass's `init` method using the `nextMethod` call.

Class Methods

Inherited from `Collection`:

`pipe`

Instance Variables

Inherited from `Collection`:

`bounded`	`maxSize`	`size`
`iteratorClass`	`minSize`	`uniformity`
`keyEqualComparator`	`mutable`	`uniformityClass`
`keyUniformity`	`mutableCopyClass`	`valueEqualComparator`
`keyUniformityClass`	`proprietored`	

Instance Methods

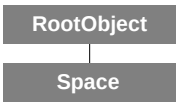
Inherited from `Collection`:

`add`	`forEach`	`iterate`
`addMany`	`forEachBinding`	`localEqual`
`addToContents`	`getAll`	`map`
`chooseAll`	`getAny`	`merge`
`chooseOne`	`getKeyAll`	`pipe`
`chooseOneBinding`	`getKeyOne`	`prin`
`deleteAll`	`getMany`	`removeAll`
`deleteBindingAll`	`getOne`	`removeOne`
`deleteBindingOne`	`hasBinding`	`setAll`
`deleteKeyAll`	`hasKey`	`setOne`
`deleteKeyOne`	`intersects`	`size`
`deleteOne`	`isEmpty`	
`emptyOut`	`isMember`	

Inherited from `LinearCollection`:

`chooseOneBackwards`	`findRange`	`getNthKey`
`chooseOrdOne`	`forEachBackwards`	`getOrdOne`
`deleteFirst`	`getFirst`	`getRange`
`deleteLast`	`getLast`	`localEqual`
`deleteNth`	`getMiddle`	`localLT`
`deleteRange`	`getNth`	`pop`

Space



Class type: Core class (abstract)
Resides in: ScriptX and KMP executables
Inherits from: `RootObject`
Component: Spaces and Presenters

The `Space` class is fundamental to the modeling and simulation capabilities of ScriptX. `Space` provides an environment to hold objects, a means for determining which objects are allowed into the space, a clock for timing, and a mechanism that allows controllers to modify the objects. Being the root abstract class for all spaces, `Space` provides a small, general base of functionality common to the other spaces, including `TwoDSpace`, `Window`, `GroupSpace`, and `PageLayer`.

Subclasses of `Space` must mix in `IndirectCollection` or one of its subclasses. This allows each space to hold a collection of objects. The `targetCollection` instance variable defines which collection is used, in order to allow each concrete subclass of `Space` to store its model objects in the most appropriate way. In this way, spaces provide the `Collection` protocol over the objects they contain. Subclasses of `Space` that can display objects mix in `TwoDMultiPresenter`, a subclass of `IndirectCollection`.

When creating a subclass of `Space`, specify `Space` first in the list of superclasses:

```
class mySpace (Space, IndirectCollection) end
```

When you create an instance of a subclass of `Space`, the `Space` class does the following:

- Creates an array to hold the list of protocols the subclass provides
- Creates an array to hold the controllers that will be added to the space
- Creates an array to hold the enabled tickled controllers that will be tickled in the space
- Creates a clock with the scale you specify
- Adds a periodic callback to the clock to tickle the controllers at every tick

Creating and Initializing a New Instance

Because `Space` is an abstract class, you cannot create an instance of `Space`. However, you can subclass `Space`, being sure to mix in `IndirectCollection`. To create an instance of a subclass of `Space`, use the `init` keywords as defined by `Space` and `IndirectCollection`.

The following is an example of creating an instance from a subclass of `Space` called `ModelSpace`:

```
class ModelSpace (Space, IndirectCollection) end
mySpace := new ModelSpace \
           targetCollection:(new LinkedList) \
           scale:20
```

This creates an instance of `ModelSpace` as a `LinkedList` object, with a clock whose `scale` is 20.

For performance reasons, when creating an instance of `Space` that is also a presenter (that is, inherits from `TwoDMultiPresenter`), you should omit the `targetCollection` keyword and allow it to create its default collection. Such presenters require collections that can be traversed quickly when drawing and handling events. Performance could suffer if you change the `targetCollection` to something other than the default.

However, when creating an instance of a subclass of `Space` that is strictly a model and not a presentation space (that is, it does not inherit from `Presenter`), choose whatever value for `targetCollection` is most appropriate for the model you are `creating`. This collection does not need to be traversed for drawing or user events.

init

`init` *self* [`scale:`*integer*] ⇨ *(none)*

self	`Space` object
`scale:`	`Integer` object

Initializes the `Space` object *self*, creates a clock and applies the value supplied with `scale` to this clock. Do not call `init` directly on an instance—it is automatically called by the `new` method.

If you omit an optional keyword, its default is used. The defaults are:

```
scale:1
```

Instance Variables

clock

self`.clock` (read-write) `Clock`

Specifies the clock belonging to the space *self*. Whenever you create an instance from the `Space` family of classes, a clock is automatically created and assigned to this instance variable. The clock is turned on by default (its rate is 1).

This clock is used by any `Controller` objects to determine time-dependent behavior of model objects, such as when `tickle` is called on controllers and presenters. With every tick of this clock, the `tickle` method is called. Model objects and controllers can make inquiries directly to this clock, set up callbacks, or make “slaved” clocks or players.

For `Window` objects, this clock also determines the frame rate for the compositor. Thus, for a window, if this clock’s scale is 24 and rate is 1, it composites 24 frames per second.

When a space is added to a presentation hierarchy, its clock is automatically slaved to the next clock up the presentation hierarchy (using `adjustClockMaster`). In other words, this clock’s master is the clock of the space’s parent space, if any. Slaving clocks together makes it easy to pause and resume all clocks in that space.

This means the absolute rate of this clock is driven by the product of its rate and its master’s rate. These two clocks can have independent scales, meaning this clock could run at a slower or faster pace than its master’s clock.

All top clocks for a given title are kept in the title container’s `topClocks` instance variable.

controllers

self`.controllers` (read-only) `Array`

Specifies the list of `Controller` objects that control the space *self*.

protocols

self.protocols (read-only) Array

Specifies an array of classes associated with the space *self*. These classes form the necessary protocol for objects added to the space. When an object is added to the space, the space checks to see if the object has all of the protocols before adding the object to the space. It does this in the `isAppropriateObject` method by using `isAKindOf` to test if the object has all protocol classes among its superclasses.

The `protocols` instance variable is read-only and points to an array of classes. The attribute "read-only" means that you cannot make the `protocols` instance variable point to a different array—"read-only" does not stop you from adding or removing items from the array. You can add classes to this array as you would with any array, as in the following example:

```
append mySpace.protocols Projectile
```

tickleList

self.tickleList (read-only) Array

Specifies the list of enabled `Controller` objects that implement the `tickle` method and are being tickled on the space *self*. The order of the controllers in this list is the order in which they are being tickled during each compositor cycle. All controllers on this list are enabled and are actively being tickled; a controller that becomes disabled is removed from this list.

Instance Methods

isAppropriateObject (IndirectCollection)

`isAppropriateObject` *self addedObject* ⇨ `Boolean`

self	`Space` object that object is being added to
addedObject	The object added to the space's list; can be any object

This method is automatically called when an object is added to any space. This method looks at the object being added to the parent space, using `isAKindOf`, and compares it to the list supplied by protocols—if the object has all of the listed protocols, the object is added to the space's list, and returns `true`.

The `isAppropriateObject` method is called by any method that adds an object to the space *self*, such as `setOne` or `setNth`. These add methods are defined in `IndirectCollection`.

objectAdded (IndirectCollection)

`objectAdded` *self addedObject* ⇨ *(none)*

self	`Space` object
addedObject	Any object

This method gets called automatically when an object that has been accepted by `isAppropriateObject` is added to any space. This method iterates over all controllers listed in the `controllers` instance variable, adding *addedObject* to only those controllers whose `wholespace` is `true`. You can specialize this method in a subclass of `Space` to perform any action you want to occur every time an object is added.

objectRemoved (IndirectCollection)

`objectRemoved` *self removedObject* ⇨ *(none)*

self	Space object
removedObject	Any object

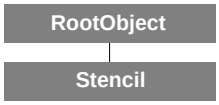
This method gets called automatically when any object is removed from the space *self*. This method iterates over all controllers listed in the `controllers` instance variable, removing *removedObject* from each of them. You can specialize this method in a subclass of Space to perform any action you want to occur every time an object is removed.

Subclasses Commonly Implement

Subclasses of the Space class should implement the following methods if they want special functionality when a model is added to or removed from a space:

```
objectAdded
objectRemoved
```

Stencil



Class type: Core class (abstract)
Resides in: ScriptX and KMP executables
Inherits from: `RootObject`
Component: 2D Graphics

The `Stencil` class and its subclasses define images to be rendered—filled or stroked—onto a surface. Two areas of interest within a stencil are its *bounding box* and *image area*. The image area is the area occupied by the image; this can be thought of as the area contained within the outline or shape of the image. The bounding box is the smallest rectangle that completely contains the image area. In addition to rendering, the image area of one stencil can be used to clip the image rendered by another stencil.

Subclasses of `Stencil` represent particular types of images. For example, `Oval`, `Rect`, and `Line` provide stencils for commonly used shapes. `Path` can represent general polygons while the `Bitmap` class handles bitmapped image data. The `TextStencil` class provides control of text and fonts within graphics.

Several methods defined by `Stencil` are arithmetic operations that accept a second stencil and return a third stencil that is the result of the operation. This new stencil may be of a different class than either of the original stencils. In particular, instances of the `Region` class are used by ScriptX to represent the result of operations that can't logically be represented by any other subclass of `Stencil`. Instances of `Region` respond to all the methods defined by `Stencil` and can be used in all operations that accept stencils—for example, in the rendering operations defined on `Surface`.

Instance Variables

bBox

self`.bBox` (read-only) `Rect`

Specifies the smallest rectangle that completely contains the image represented by the stencil *self*. In `Stencil` subclasses, the value in this variable changes as the overall height or width of the image area changes ("bBox" means "bounding box").

Instance Methods

inside

`inside` *self point* ⇨ `Boolean`

self	`Stencil` object
point	`Point` object

Tests whether the `Point` object *point* is inside the image area of the stencil *self*. The `inside` method first performs a less costly test on the stencil's bounding box. If that test succeeds, it then tests the stencil's real boundary. If the stencil's real boundary is not rectangular, and the point is directly on the edge of the image area, the value returned by this method is subject to rounding error.

intersect

`intersect` *self otherStencil mutateOrCreate* ⇨ `Stencil`

self	`Stencil` object
otherStencil	`Stencil` object
mutateOrCreate	`NameClass` object representing the desired result: `@mutate` or `@create`

Creates and returns a stencil that is the geometric intersection of the two stencils, *self* and *otherStencil*, representing the overlap between the image areas of these stencils. The value returned depends on the value of *mutateOrCreate*:

`@mutate` – place the resulting union directly in *self*

`@create` – create a copy of *self* and return the copy

The intersection of certain stencils may be best represented by an instance of `Region`; in such cases the *mutateOrCreate* argument is ignored and a copy is returned.

onBoundary

`onBoundary` *self x y proximity* ⇨ `Boolean`

self	`Stencil` object
x	`Number` object
y	`Number` object
proximity	`Number` object

Returns `true` if the point with coordinates specified by *x* and *y* is on the boundary of the image area represented by the stencil *self*. The value of *proximity* determines how close to this boundary (inside or outside) the point must be for this method to return `true`. A `0` value for *proximity* doesn't work for detecting point values on the boundary; use `1` instead.

subtract

`subtract` *self otherStencil mutateOrCreate* ⇨ `Stencil`

self	`Stencil` object
otherStencil	`Stencil` object
mutateOrCreate	`NameClass` object representing the desired result: `@mutate` or `@create`

Creates and returns a stencil containing the image that remains after the image area of the stencil *otherStencil* has been removed from the image area of the stencil *self*. The value returned depends on the value of *mutateOrCreate*:

`@mutate` – place the resulting union directly in *self*

`@create` – create a copy of *self* and return the copy

The difference of certain stencils may be best represented by an instance of `Region`; in such cases the *mutateOrCreate* argument is ignored and a region is returned.

transform

`transform` *self matrix mutateOrCreate* ⇨ `Stencil`

self	`Stencil` to transform
matrix	`TwoDMatrix` object representing the transformation
mutateOrCreate	`NameClass` object specifying the desired result: `@mutate` or `@create`.

Transforms the stencil *self* by the `TwoDMatrix` object *matrix*. The value returned depends on the value of *mutateOrCreate*: `@mutate` transforms *self* directly; `@create` creates a copy of *self*, then transforms and returns the copy. To transform a line, you first call the appropriate transformation methods on *matrix*, then call this method on the instance of the `Stencil` subclass.

In many cases, when you use the `transform` method on a stencil, such as a `Rect`, the resulting object will be a new `Path` or `Region` object, regardless of the value of *mutateOrCreate*.

union

`union` *self otherStencil mutateOrCreate* ⇨ `Stencil`

self	`Stencil` object
otherStencil	`Stencil` object
mutateOrCreate	`NameClass` object representing the desired result: `@mutate` or `@create`

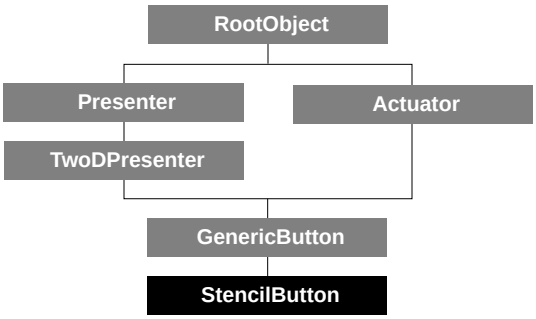
Creates and returns a stencil that is the geometric union of the stencils *self* and *otherStencil*. The value returned depends on the value of *mutateOrCreate*:

`@mutate` – place the resulting union directly in *self*

`@create` – create a copy of *self* and return the copy

The union of certain stencils may be best represented by an instance of `Region`; in such cases the *mutateOrCreate* argument is ignored and a copy is returned.

StencilButton



Class type: Scripted class (concrete)
Resides in: widgets.sxl. Works with ScriptX and KMP executables
Inherits from: GenericButton
Component: User Interface

StencilButton is a user interface Widget Kit class that provides a framed button whose appearance is defined by a stencil such as a bitmap and a Frame object that gives the button a three dimensional look.

The clipping boundary of a StencilButton object is calculated automatically as the smallest rectangle that encloses the stencil and the button frame.

Creating and Initializing a New Instance

The following script is an example of how to create a new instance of the StencilButton class:

```
bitmapButton := new StencilButton \
       stencil:(importDIB "media/kIcon.bmp")
```

The variable bitmapButton contains an initialized StencilButton object. The new method uses the keywords defined in init.

init

`init` *self* [`stencil:`*stencil*] [`boundary:`*stencil*] ⇨ *(none)*

self	StencilButton object
stencil:	Stencil object
boundary:	Stencil object

Initializes the StencilButton object *self*, applying the values supplied with the keywords to the instance variables of the same name. Automatically supplies a frame for the button and calculates a boundary that encompasses the stencil and the button frame. Do not call init directly on an instance — it is automatically called by the new method.

If you omit one of the keyword arguments, the following defaults are used:

```
stencil:(new Rect x2:10 y2:10)
boundary:unsupplied
stationary:false
```

You should not provide a value for boundary. If you do not provide a value for boundary, then the boundary of the new StencilButton object is calculated automatically as the smallest rectangle that encloses the stencil and the button frame.

Class Variables

borderWidth

self.borderWidth (read-write) Integer

Used to calculate the width of the frame or border around the stencil of the StencilButton. The default value is 2.

Instance Variables

Inherited from Actuator:

enabled	pressed	toggledOn
menu		

Inherited from Presenter:

presentedBy	subPresenters	target

Inherited from TwoDPresenter:

bBox	height	transform
boundary	IsImplicitlyDirect	width
clock	isTransparent	window
compositor	isVisible	x
direct	needsTickle	y
eventInterests	position	z
globalBoundary	stationary	
globalTransform	target	

Inherited from GenericButton:

activateAction	pressAction	releaseAction
authorData		

The following instance variables are defined in StencilButton:

border

self.border (read-write) Rect

Specifies the rectangle that encloses the stencil and frame of the StencilButton object *self*. This is calculated automatically by the recalcRegion method using the StencilButton object's stencil, frame, boundary, and border width. The following example shows the relationship among these variables:

```
StencilButton.borderWidth
⇨ 2
bitmapButton.stencil.bBox
⇨ [0, 0, 30, 30] as Rect
bitmapButton.border
⇨ [2, 2, 40, 40] as Rect
bitmapButton.boundary
⇨ [0, 0, 42, 42] as Rect
```

buttonFrame

self.buttonFrame (read-write) Frame

Specifies the Frame object that is displayed when the StencilButton object *self* is not pressed or toggled on. A default Frame object is supplied that gives the stencil button a raised appearance.

recessedFrame

self.recessedFrame (read-write) Frame

Specifies the Frame object that is displayed when the StencilButton object *self* is pressed or toggled on. A default Frame object is supplied that gives the stencil button a lowered (or recessed) appearance.

stencil

self.stencil (read-write) Stencil

Specifies the Stencil object that is displayed inside the frame of the StencilButton object *self*. The default value of stencil is a Rect object as shown in the init method above.

Instance Methods

Inherited from Actuator:

activate	press	toggleOff
multiActivate	release	toggleOn

Inherited from TwoDPresenter:

adjustClockMaster	inside	show
createInterestList	localToSurface	surfaceToLocal
draw	notifyChanged	tickle
getBoundaryInParent	recalcRegion	
hide	refresh	

Inherited from GenericButton:

activate	press	release
multiActivate		

The following instance methods are defined in StencilButton:

draw (TwoDPresenter)

draw *self surface clip* ➪ *(none)*

self	StencilButton object to draw
surface	DisplaySurface or BitmapSurface object to draw to
clip	Stencil object for parent's clip area

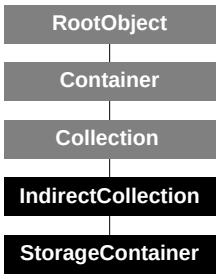
Tells the stencil button *self* to render its image onto the display surface *surface*, with clipping defined by the stencil *clip*. The draw method is automatically called by the compositor to render the stencil button to the compositor's display surface. If the stencil button *self* is pressed or toggled on, then the recessedFrame appearance of the button is drawn; otherwise, the buttonFrame appearance of the stencil button is drawn.

recalcRegion

`recalcRegion` *self* ⇨ `TwoDMatrix`

Calculates the correct bounding box of the `StencilButton` object *self*. By default, the size of the button is the smallest width and height that encloses the stencil and the button frame. This method returns a `TwoDMatrix` object that contains the new width and height. This method is called automatically each time either the `stencil` or the button frame (the `border`, `buttonFrame`, or `recessedFrame`) changes. If the width or height of the button is changed after this method is called, then the stencil and button frame are clipped to the new width and height.

StorageContainer



Class type: Core class (concrete)
Resides in: ScriptX and KMP executables
Inherits from: `IndirectCollection`
Component: Title Management

The `StorageContainer` class provides general purpose storage of objects. Objects of all ScriptX classes can be saved in storage containers, including core class instances, scripted classes, instances of scripted classes, modules, and global variables. Each storage container represents a file.

Do not attempt to subclass the `StorageContainer` class.

Typically, you will use library containers (instances of `LibraryContainer`, `TitleContainer`, and `AccessoryContainer`), which are storage containers, and you will not instantiate `StorageContainer` directly.

Creating and Initializing New Instances

The following sample script shows how to create a new instance of the `StorageContainer` class:

```
myCon := new StorageContainer \
          path:"con.sxk"
```

The variable `myCon` contains the initialized `StorageContainer` instance. The new container represents the file `con.sxk` in the directory represented by the global constant `theStartDir`. This file is created as a new container to be stored and retrieved from. The `new` method uses the keywords defined in `init`.

Note – The convention for naming library container files, valid across all platforms, is to use the `.sxk` extension, as shown above (meaning "ScriptX Storage Container").

init

`init` *self* [`dir:`*dirRep*] `path:`*stringOrSequence*
[`targetCollection:`*collection*] ⇨ *(none)*

self	`StorageContainer` object
`dir:`	`DirRep` or `Sequence` object representing a path to the container
`path:`	`String` object or `Sequence` of strings

The superclass `IndirectCollection` uses the following keyword:

`targetCollection:`	`Collection` object

Creates a new `StorageContainer` object *self*, applying the keyword arguments as follows:

- The keyword path specifies either a String object or a Sequence of strings that represent a file where the container resides. This argument represents the subpath, including filename, where the container resides. This subpath representation should be in the same form as the *path* argument to DirRep methods.
- The keyword dir specifies the DirRep or Sequence object representing the path to the container. The subpath specified by the argument to path is interpreted relative to this argument.

If a file with this dir and path already exists but is not open, that file is overwritten. If a file with this dir and path is already open in this ScriptX session, an error is reported, and the existing file is not overwritten.

The storage container's collection is specified by the targetCollection keyword. It is useful to make the target collection an explicitly keyed collection, such as an instance of HashTable or KeyedLinkedList, so that each item in the storage container can be identified by a name or string constant key.

Do not call init directly on an instance—it is automatically called by the new method.

If you omit an optional keyword, its default value is used. The defaults are:

```
dir:theStartDir
targetCollection:(new Array initialSize:14 growable:true)
```

Class Methods

Inherited from Collection:

```
pipe
```

open

open *self* [dir:*dirRep*] path:*stringOrSequence* [mode:*name*] ⇨ StorageContainer

self	StorageContainer class
dir:	DirRep object
path:	String object or Sequence of strings
mode:	NameClass object: @update or @readonly

Opens the storage container *self* with the filename given by path in the directory specified by *dirRep*. If path is a sequence of strings, the last string is the filename and the preceding strings are the subdirectories off *dirRep*. The keyword mode specifies the access mode in which to open the container, and can be either @readonly or @update.

If you omit an optional keyword, its default value is used. The defaults are:

```
dir:theStartDir
mode:@readonly
```

Instance Variables

Inherited from Collection:

```
bounded                 maxSize                 size
iteratorClass           minSize                 uniformity
keyEqualComparator      mutable                 uniformityClass
keyUniformity           mutableCopyClass        valueEqualComparator
keyUniformityClass      proprietored
```

Inherited from IndirectCollection:

```
targetCollection
```

Instance Methods

Inherited from Collection:

add	forEach	iterate
addMany	forEachBinding	localEqual
addToContents	getAll	map
chooseAll	getAny	merge
chooseOne	getKeyAll	pipe
chooseOneBinding	getKeyOne	prin
deleteAll	getMany	removeAll
deleteBindingAll	getOne	removeOne
deleteBindingOne	hasBinding	setAll
deleteKeyAll	hasKey	setOne
deleteKeyOne	intersects	size
deleteOne	isEmpty	
emptyOut	isMember	

Inherited from IndirectCollection:

isAppropriateObject	objectAdded	objectRemoved

Since a StorageContainer object is an indirect collection, you can also use any methods defined in the class specified by targetCollection. The target collection is by default an instance of Array, which inherits from Sequence; in this case, the following instance methods are accessible:

Accessible from LinearCollection:

chooseOneBackwards	findRange	getNthKey
chooseOrdOne	forEachBackwards	getOrdOne
deleteFirst	getFirst	getRange
deleteLast	getLast	localEqual
deleteNth	getMiddle	localLT
deleteRange	getNth	pop

Accessible from Sequence by redirection:

addFifth	moveBackward	setFourth
addFirst	moveForward	setLast
addFourth	moveToBack	setNth
addNth	moveToFront	setSecond
addSecond	prepend	setThird
addThird	prependNew	sort
append	setFifth	
appendNew	setFirst	

The following instance methods are defined in StorageContainer:

close

close *self* ⇨ Boolean

Saves and closes the storage container *self*, and makes the container and all of its contained objects purgeable.

requestPurgeForAllObjects

requestPurgeForAllObjects *self* ⇨ *(none)*

Marks as purgeable all objects that currently reside in memory and are associated with the storage container *self*. Note that objects in a container cannot be purged from memory until they have been stored, until the container has been updated.

update (RootObject)

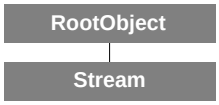
`update` *self* ⇨ *(none)*

Causes all objects added to the storage container *self* to be written to disk. Saves the storage container *self* along with all top-level objects and subobjects contained by *self* to the file that *self* represents, if the `mode` of *self* is `@update` and if the underlying file is writable. If the `mode` of *self* is not `@update`, or if the underlying file is not writable, then this method fails.

This method does not purge objects from memory, and it keeps the storage container open so that you can continue working with it.

See also the `update` method in the `RootObject` class to update a single object already added to a storage container.

Stream



Class type: Core class (abstract)
Resides in: ScriptX and KMP executables
Inherits from: `RootObject`
Component: Streams

The `Stream` class represents linear sequences of data elements in a source-independent way. A `Stream` object can be readable, writable, or seekable, depending on the type of data elements it represents and how it was created.

If a stream is seekable, its data can be accessed from different points in the stream. The "cursor" indicates the current point of focus in a seekable stream. There are methods to move the cursor to different parts of a stream. For example, if a stream is seekable, you could move the cursor to the midpoint of the stream, read the next block of data, and then move the cursor to ten characters from the beginning of the stream and read the next word of data. For a more complete discussion of `Stream` and its subclasses, see the "Streams" chapter in the *ScriptX ScriptX Components Guide*.

Instance Methods

The following methods can be used on instances of `Stream` (or its subclasses). They can only be used if they are appropriate to the type of stream. For example, you cannot write to stream that was created read-only.

To implement a subclass of stream, you should override all these methods.

cursor

`cursor` *self* ⇨ `Integer`

Returns the current position of the cursor in the stream *self*. The value 0 represents the point just before the first item in the stream, 1 represents the point between the first and second items, and so on.

flush

`flush` *self* ⇨ *(none)*

Flushes any pending output from the stream *self*.

isAtFront

`isAtFront` *self* ⇨ `Boolean`

Returns `true` if the cursor is at position 0, that is, at the start of the stream *self*. Returns `false` if the cursor is at any other position or the stream is non-seekable.

isPastEnd

`isPastEnd` *self* ⇨ `Boolean`

Returns `true` if the cursor is past the end of the data in the stream *self*; otherwise, it returns `false`. For some subclasses of `Stream`, this method always returns `false` (serial port streams, for example).

isReadable

`isReadable` *self* ⇨ `Boolean`

Returns `true` if the stream *self* is readable; otherwise, it returns `false`.

isSeekable

`isSeekable` *self* ⇨ `Boolean`

Returns `true` if the stream *self* is seekable; otherwise, it returns `false`.

isWritable

`isWritable` *self* ⇨ `Boolean`

Returns `true` if the stream *self* is writable; otherwise, it returns `false`.

next

`next` *self* ⇨ `Boolean`

Returns `false` if the cursor is at the end of the stream *self*. Otherwise, it moves the cursor to the next position and returns `true`. Note that `isSeekable` doesn't have to return `true` for `next` to work.

plug

`plug` *self* ⇨ *(none)*

Closes the stream *self* after releasing any memory acquired for its data. It invalidates the object.

previous

`previous` *self* ⇨ `Boolean`

Returns `false` if the cursor is at the beginning of the stream *self*. Otherwise, it moves the cursor to the previous position in the stream and returns `true`. This method is only implemented by seekable streams. If a stream isn't seekable, this method reports an exception.

read

`read` *self* ⇨ *(object)*

Reads one data element from the stream *self*.

readReady

`readReady` *self* ⇨ `Integer`

Returns the number of items in the stream *self* that can be read without blocking.

seekFromCursor

`seekFromCursor` *self offset* ⇨ *(none)*

self	`Stream` object
offset	`Integer`

Moves the cursor the number of elements indicated by *offset* from the current cursor position in the stream *self*. Otherwise, this method is identical to `seekFromStart`—see that method for more on seeking.

seekFromEnd

`seekFromEnd` *self offset* ⇨ *(none)*

self	`Stream` object
offset	`Integer` object

Seeks (moves the cursor) to the data element at the position indicated by *offset*, starting from the end of the stream *self*. A positive value for *offset* moves the cursor past the end of the stream, and a negative value for *offset* moves the cursor toward the start of the stream. This method is otherwise identical to `seekFromStart`—see that method for more on seeking. The following example moves the cursor in a position to read the last item in the stream:

```
seekFromEnd myStream -1
```

seekFromStart

`seekFromStart` *self offset* ⇨ *(none)*

self	`Stream` object
offset	`Integer` object

Seeks (moves the cursor) to the data element at the position indicated by *offset* from the beginning of the stream *self*. A positive value for *offset* moves the cursor toward the end of the stream, and a negative value for *offset* moves the cursor toward the start of the stream. The exception `seekBounds` is reported if you try to seek past the start of the stream. However, it is legal to seek past the end of the stream. With `ByteStream` and its subclasses, any gaps between the end of the stream and the cursor may be automatically filled with zeros when the stream is written. The following example moves the cursor in a position to read the first item in the stream:

```
seekFromStart myStream 0
```

setStreamLength

`setStreamLength` *self length* ⇨ *(none)*

self	`Stream` object
length	`Integer` object

Sets the number of data elements in the stream *self* to the value of *length*. If *length* is larger than the old size, the stream is padded at the end; if smaller, the data at the end is discarded. The stream *self* must be writable (otherwise, it reports an exception).

streamLength

`streamLength` *self* ⇨ `Integer`

Returns the number of data elements in the stream *self*.

write

`write` *self value* ⇨ *(none)*

self	`Stream` object
value	`Object`

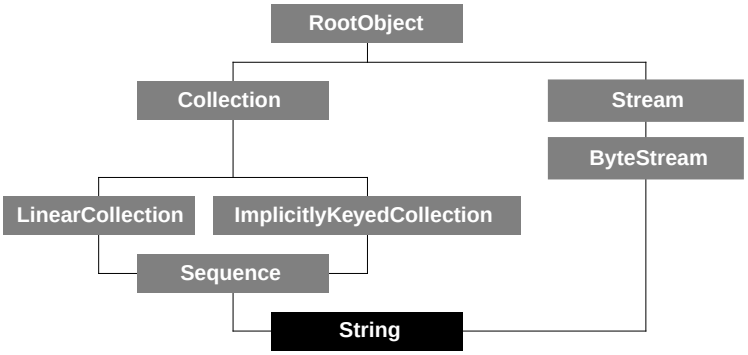
Writes one data element to the stream *self*.

writeReady

`writeReady` *self* ⇨ `Integer`

Returns the number data elements of the stream *self* that can be written without blocking. For file streams (returned by the `DirRep` method `getStream`), no guarantee can be made about blocking, so this method usually returns 0.

String



Class type: Core class (concrete)
Resides in: ScriptX and KMP executables
Inherits from: `Sequence` and `ByteStream`
Component: Text and Fonts

The `String` class stores a sequence of unsigned integer values that represent Unicode/ISO 10646 character data. Internally, those characters are stored in a variable-length format to allow for representation of and ordering of thousands of international characters and symbols. Each element of a string may occupy from one to six bytes. From a script, each element of a string object is seen as a single Unicode/ISO 10646 character. For an explanation of variable-length encoding, see the "Text and Fonts" chapter of the *ScriptX Components Guide*.

Note – The `String` class is not equivalent to string data types defined by languages such as C, C++, or Pascal. The `ByteString` class, which also inherits from `Sequence` and `ByteStream` can be used to store uninterpreted character data.

The `String` and `StringConstant` classes are closely related (`StringConstant` inherits from `String`). Instances of `StringConstant` cannot be modified; to change them, those instances must be coerced into instances of `String`. String literals (a sequence of characters surrounded by double-quotes, as typed in scripts) create instances of the `StringConstant` class. See `StringConstant` for more information on string literals.

Instances of `String` and `StringConstant` are printed differently to a stream. An instance of `StringConstant` prints as its literal representation. A `String` object prints with additional information that identifies it as a full-blown `String` object.

```
"fooConst" -- a StringConstant object
"fooString" as String -- a String object
```

The `String` class inherits from the abstract classes `Collection`, `Sequence`, and `Stream`. This complex inheritance provides the ability to treat `String` objects in many different ways.

When viewed as collections, each character in a string object represents an element in a collection that can be accessed using that character's ordinal position (strings also inherit from `Sequence`). Most of the methods that can be used on sequences such as arrays can also be used on `String` objects.

When viewed as streams, strings are considered write-only (and non-seekable) streams. Many of the methods available to streams that seek or read from a stream may report an exception when used with a `String` object. By creating an iterator on a String object, however, you can perform stream operations.

In addition to collection and stream behaviors, `String` objects can also be added and subtracted using the addition (+) and subtraction (-) ScriptX language operators:

```
amphibian := "toad" as String
amphibian + "stool"
⇨  "toadstool"
fruit := "banana" as String
fruit - "an"
⇨  "bana" -- deletes first instance in target string
```

String arithmetic is described in more detail in the *ScriptX Components Guide*.

Many of the methods defined on the `String` class operate on ranges of characters in the string. Those ranges are expressed as cursor positions, which occur between characters, with position 0 indicating the beginning of the string (before the first ordinal character). This is different from the *ordinal position* of each character, which occurs directly on the character itself. Note that a single character is also a range of two cursor positions. Figure 29 illustrates the difference between cursor and ordinal positions.

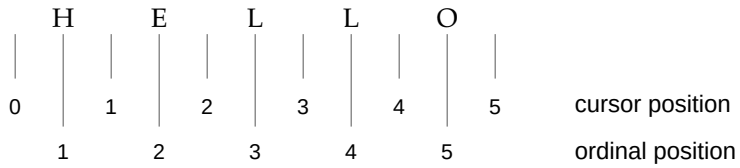


Figure 29: Cursor And Ordinal Positions

Creating and Initializing a New Instance

Instances of the `String` class can be created in two ways:

- by coercing a string literal (an instance of the `StringConstant` class) into a `String` object:

```
myString := "Hello world!" as String
```

- by using the `new` method:

```
myString := new String string:"Hello world!"
```

Each of these examples assigns a new instance of `String` to the variable `myString`. The `new` method in the second example uses the `string` keyword defined by the `init` method.

init

`init` *self* [`string:`*string*] ⇨ *(none)*

self	`String` object
`string:`	A string literal

Initializes the `String` object *self*, where `string` is a string literal, enclosed in quotes. Do not call `init` directly on an instance—it is automatically called by the `new` method.

If you omit an optional keyword, its default value is used. The defaults are:

`string:""` (a string with zero length)

Class Methods

Inherited from `Collection`:

`pipe`

Instance Variables

Inherited from `Collection`:

`bounded`	`maxSize`	`size`
`iteratorClass`	`minSize`	`uniformity`
`keyEqualComparator`	`mutable`	`uniformityClass`
`keyUniformity`	`mutableCopyClass`	`valueEqualComparator`
`keyUniformityClass`	`proprietored`	

Instance Methods

Inherited from `Stream`:

`cursor`	`next`	`seekFromStart`
`flush`	`plug`	`setStreamLength`
`isAtFront`	`previous`	`streamLength`
`isPastEnd`	`read`	`write`
`isReadable`	`readReady`	`writeReady`
`isSeekable`	`seekFromCursor`	
`isWritable`	`seekFromEnd`	

Note – `Stream` methods such as `read`, `previous` , `next`, `seekFromCursor`, and the other seek methods, work with `String` objects only if you create an iterator on the `String` object. (An iterator is readable, writable, and seekable, whereas a `String` object is only writable.)

Inherited from `ByteStream`:

`fileIn`	`readByte`	`writeString`
`pipe`	`readReady`	
`pipePartial`	`writeByte`	

Inherited from `Collection`:

`add`	`forEach`	`iterate`
`addMany`	`forEachBinding`	`localEqual`
`addToContents`	`getAll`	`map`
`chooseAll`	`getAny`	`merge`
`chooseOne`	`getKeyAll`	`pipe`
`chooseOneBinding`	`getKeyOne`	`prin`
`deleteAll`	`getMany`	`removeAll`
`deleteBindingAll`	`getOne`	`removeOne`
`deleteBindingOne`	`hasBinding`	`setAll`
`deleteKeyAll`	`hasKey`	`setOne`
`deleteKeyOne`	`intersects`	`size`
`deleteOne`	`isEmpty`	
`emptyOut`	`isMember`	

Inherited from `LinearCollection`:

`chooseOneBackwards`	`findRange`	`getNthKey`
`chooseOrdOne`	`forEachBackwards`	`getOrdOne`
`deleteFirst`	`getFirst`	`getRange`
`deleteLast`	`getLast`	`localEqual`
`deleteNth`	`getMiddle`	`localLT`

deleteRange	getNth	pop

Inherited from Sequence:

addFifth	moveBackward	setFourth
addFirst	moveForward	setLast
addFourth	moveToBack	setNth
addNth	moveToFront	setSecond
addSecond	prepend	setThird
addThird	prependNew	sort
append	setFifth	
appendNew	setFirst	

The following instance methods are defined in String:

copyFromTo

copyFromTo *self start end* ⇨ String

self	String object
start	Number object, the beginning cursor position
end	Number object, the ending cursor position

Copies and returns the substring located between the cursor positions *start* and *end* in the string *self*. The cursor is positioned between characters, as shown in Figure 29, "Cursor And Ordinal Positions," on page 706. The copyFromTo method does not change the original string (*self*).

```
sourceString := "Nixon in China" as String
copyFromTo sourceString 9 13
⇨ "Chin"
print sourceString
⇨ "Nixon in China" -- the source string was not affected
```

cutFromTo

cutFromTo *self start end* ⇨ String

self	String object
start	Number object, the beginning cursor position
end	Number object, the ending cursor position

Cuts values between the cursor positions *start* and *end* from the string *self*, and returns the substring that was cut. The cutFromTo method modifies the original string (*self*).

```
sourceString := "Nixon in China" as String
cutFromTo sourceString 9 13
⇨ "Chin"
print sourceString
⇨ "Nixon in a" -- the original string is shortened by the cut
```

deleteFromTo

deleteFromTo *self start end* ⇨ String

self	String object
start	Number object, the beginning cursor position
end	Number object, the ending cursor position

Deletes values between the cursor positions *start* and *end* from the string *self*, and returns the new version of the string. The `deleteFromTo` method modifies the original string (*self*).

```
holyBagel := "Bagels and lox" as String
deleteFromTo holyBagel 1 12
⇨  "Box"
```

getLowercase

`getLowercase` *self* ⇨ `String`

Returns a lowercase version of the string *self*.

```
SX := "ScriptX Version 1.0" as String
getLowercase SX
⇨  "scriptx version 1.0"
```

hashOf

`hashOf` *self* ⇨ `ImmediateInteger`

Returns a positive `ImmediateInteger` representing a 29-bit hash value for the string *self*. The `hashOf` generic function is most commonly used to hash keys for items to be stored in a hash table.

insertAt

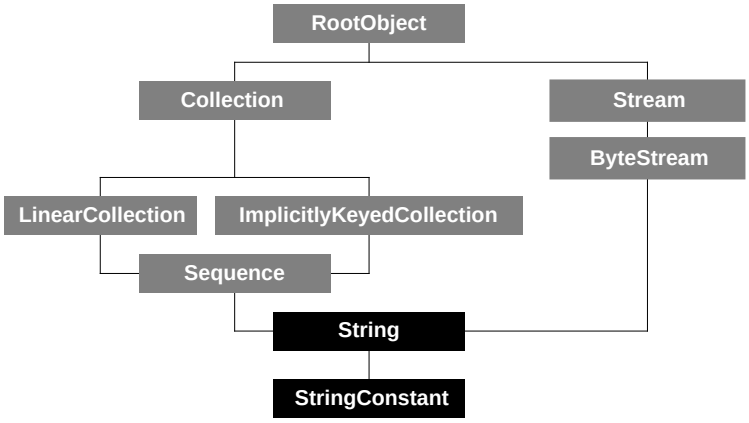
`insertAt` *self addedString position* ⇨ `String`

self	`String` object
addedString	`String` object to be inserted
position	`Number` object, a cursor position

Inserts the string *addedString* into the string *self* at the cursor position *position*, and returns the changed string (*self*).

```
oldStr := new String string:"Mason"
insertAt oldStr "di" 2
⇨  "Madison"
```

StringConstant



Class type: Core class (concrete, sealed)
Resides in: ScriptX and KMP executables
Inherits from: `String`
Component: Text and Fonts

The `StringConstant` class represents a string whose contents cannot be changed. `StringConstant` objects are closely related to `String` objects (which represent mutable strings), and instances of each can be freely coerced into the class of the other.

String literals, that is, characters surrounded by double quotes as typed in scripts, evaluate to `StringConstant` objects.

The `StringConstant` class inherits all the methods defined by the `String` class. However, methods that modify the string (for example, `addMany`, `insertAt`, or `emptyOut`) report the `immutable` exception.

The addition (+) and subtraction (–) language operators can be used on `StringConstant` objects. However, the result of string addition and subtraction is a `String` object.

```
getClass ("fluffy" + " bunny")
⇨ String
```

Creating and Initializing a New Instance

`StringConstant` objects are created automatically by string literals, which are represented in scripts by a set of characters surrounded by double quotes:

```
"yoyoma" -- creates a StringConstant object
```

You can also create a `StringConstant` object using the `new` method. `StringConstant` does not define its own `init` method. Instead, it uses the `init` method defined by `String`, its superclass.

```
yourString := new StringConstant string:"foo" + "ban"
Class Methods
```

Inherited from `Collection`:

`pipe`

Instance Variables

Inherited from Collection:

bounded	maxSize	size
iteratorClass	minSize	uniformity
keyEqualComparator	mutable	uniformityClass
keyUniformity	mutableCopyClass	valueEqualComparator
keyUniformityClass	proprietored	

Instance Methods

Inherited from Stream:

cursor	next	seekFromStart
flush	plug	setStreamLength
isAtFront	previous	streamLength
isPastEnd	read	write
isReadable	readReady	writeReady
isSeekable	seekFromCursor	
isWritable	seekFromEnd	

Note – Since StringConstant objects are not readable, writable, or seekable, Stream methods will not operate on them. Stream methods will work, however, if you coerce a StringConstant object to a String object and then create an iterator on that String object. (Iterators are readable, writable, and seekable, whereas String objects are only writable.)

Inherited from ByteStream:

fileIn	readByte	writeString
pipe	readReady	
pipePartial	writeByte	

Inherited from Collection:

add	forEach	iterate
addMany	forEachBinding	localEqual
addToContents	getAll	map
chooseAll	getAny	merge
chooseOne	getKeyAll	pipe
chooseOneBinding	getKeyOne	prin
deleteAll	getMany	removeAll
deleteBindingAll	getOne	removeOne
deleteBindingOne	hasBinding	setAll
deleteKeyAll	hasKey	setOne
deleteKeyOne	intersects	size
deleteOne	isEmpty	
emptyOut	isMember	

Inherited from LinearCollection:

chooseOneBackwards	findRange	getNthKey
chooseOrdOne	forEachBackwards	getOrdOne
deleteFirst	getFirst	getRange
deleteLast	getLast	localEqual
deleteNth	getMiddle	localLT
deleteRange	getNth	pop

Inherited from Sequence:

addFifth	moveBackward	setFourth
addFirst	moveForward	setLast

addFourth	moveToBack	setNth
addNth	moveToFront	setSecond
addSecond	prepend	setThird
addThird	prependNew	sort
append	setFifth	
appendNew	setFirst	

Inherited from `String`:

copyFromTo	deleteFromTo	hashOf
cutFromTo	getLowercase	insertAt

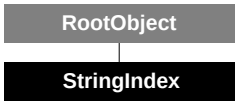
The following instance method is defined in `StringConstant`:

copy

`copy` *self* ⇨ `String`

Creates a copy of the `StringConstant` object *self*, coerces it to a `String` object, and returns that object. Note that this is a specialization of `copy`, since `copy` normally returns another object of the same class as itself.

StringIndex



Class type: Core class (concrete)
Resides in: ScriptX and KMP executables
Inherits from: `RootObject`
Component: Text and Fonts

The `StringIndex` class makes it possible to search text quickly. It operates on a string of any length and any type, which includes `StringConstant`, `String`, and `Text` objects. When you create an instance of `StringIndex`, that object automatically builds a signature index for its `string` instance variable. This signature index is completely transparent to the user. For fine tuning performance, you can set the instance variable `bitsPerWord` to a value from 0 to 64, with a larger number increasing the speed of the search but also the size of the index.

The global function `searchIndex` searches the signature index and locates the first match for a given set of characters. It returns a `SearchContext` object which describes the range of the matched character set in the string being searched. Memory usage is minimized because the whole string being searched need not be brought into memory. Initially, only the signature index is in memory; then if possible matches are found, only the corresponding objects are brought into memory to check for a definite match.

Creating and Initializing a New Instance

The following script is an example of how to create a new instance of the `StringIndex` class:

```
myStringIndex := new StringIndex \
                     string:"This string will be indexed." \
                     bitsPerWord:6
```

The variable `myStringIndex` is an instance of `StringIndex` which contains a string literal and 6 bits per word as a resolution parameter for building its signature index. You must supply a value for the `string` keyword when you create an instance of `StringIndex`. The `new` method uses the `string` and `bitsPerWord` keywords defined by the `init` method.

init

`init` *self* `string:`*string* [`bitsPerWord:`*integer*] ⇨ *(none)*

self	`StringIndex` object
`string:`	`String`, `StringConstant`, or `Text` object
`bitsPerWord:`	`Integer` object

Initializes the `StringIndex` object *self*, and applies the arguments to the corresponding instance variables, where *string* can be a string literal, a `String` object, or a `Text` object, and `bitsPerWord` is an optional keyword argument used only for fine tuning performance.

If you omit an optional keyword, its default value is used. The default values are:

```
bitsPerWord:8
```

Do not call `init` directly on an instance—it is automatically called by the `new` method.

Instance Variables

string

self`.string` (read-write) `String`

Specifies the string upon which the signature index will be built by the `StringIndex` object *self*. This instance variable must be set at initialization.

bitsPerWord

self`.bitsPerWord` (read-write) `Integer`

Specifies the resolution to be used in the construction of the signature index by the `StringIndex` object *self*. Possible values are from 0 to 64. The larger the number, the faster the search and the larger the index. This value is used only for fine tuning performance.

Instance Methods

The following functions are global functions defined to work with `StringIndex` objects:

searchIndex

`searchIndex` *strIndex match searchContext whole* ⇨ `SearchContext`

strIndex	`StringIndex` object
match	`String` object
searchContext	`SearchContext` object
whole	`Boolean` object

(Although this functions is technically a global function, it is included here because it operates only on instances of `StringIndex` and therefore works like an instance method. It is also included in the section on global functions.)

The global function `searchIndex` searches *string* for *match* using *searchContext* to tell it where to start or resume searching and using *whole* to tell it whether or not a match must be a whole word. It returns `empty` if *match* is not found; otherwise, it returns a `SearchContext` object.

The instance variable *strIndex* is the string which will be searched; *match* is the set of characters to search for and must be a string which has at least three characters and contains no spaces. If *whole* is `true`, it indicates that *match* must be matched by a whole word; `false` indicates that *match* may be embedded within a larger word and still be considered a match.

The global function `initialSearchContext` (documented in `SearchContext`) returns an instance of the class `SearchContext`, which supplies values indicating where to begin searching. Refer to `SearchContext` for more information. The following code creates a `StringIndex` object and then an initial *searchContext* to use with `searchIndex`:

```
-- create a StringIndex object initialized with a StringConstant object
-- and create a searchContext parameter such that the search starts at
-- the beginning
global i := new StringIndex \
    string:"His research in thistles secures his place in history,\
    but he can't play whist worth a darn."
```

```
global sc := initialSearchContext i 0
```

The variable sc contains the initial search context to begin the search. You can now use it as *searchContext* when invoking searchIndex to find the first occurrence of the string being matched:

```
-- search the string for the first occurrence of "his" and store the
-- result in a variable
global sc1 := searchIndex i "his" sc false

-- use the result as the search context for locating the second
-- occurrence of "his". Store the result in another variable, which can
-- be used as the search context for finding the third occurrence of
-- "his", and so on
global sc2 := searchIndex i "his" sc1 false
```

You can write a function such as the following to search for a specified number of occurrences in one string or in multiple strings. In this function definition example, indexes is an array of one or more StringIndex objects, match is the string to match, limit is the number of occurrences to search for, and whole specifies whether match must match a whole word or not.

```
function searchIndexes indexes match limit whole -> (
    local hits := #()
    for nextIndex in indexes do (
        local searchContext := initialSearchContext(nextIndex, 0)
        repeat while (searchContext != empty) do (
            searchContext := searchIndex nextIndex match searchContext\
                whole
            if (searchContext != empty) do (
                append hits searchContext)
            if (size(hits) >= limit) do return(hits)
        )
    )
    hits
)
```

The following example searches for the first four occurrences of “his”. The first argument must be an array so that collection methods can be used on it:

```
global matches := searchIndexes #(i) "his" 4 false
```

To verify that the matches found by searchIndexes are correct , you can define a function such as the following to print them out:

```
function printHits matches -> (
    for c := 1 to matches.size do (
        print(copyFromTo(matches[c].string, matches[c].startOffset \
        matches[c].endOffset))
    )
)

--Find the first five occurrences of "his", looking for either whole
--words or embedded matches, and verify that they are correct
global h := searchIndexes #(i) "his" 5 false
printHits h
```

You can begin a search at any position you want by simply giving that position as the second argument to `initialSearchContext`. For example, the following code creates a *searchContext* argument for `searchIndex` such that the search for the first occurrence of "his" starts at cursor position 17 in the string associated with the `StringIndex` object `i`. Note that using cursor position 17 means that the search will start with the 18th character.

```
global scStartAt17 := initialSearchContext i 17
global scReturn := searchIndex i "his" scStartAt17 false
```

updateIndex

`updateIndex` *strIndex* ⇨*self*

strIndex `StringIndex` object

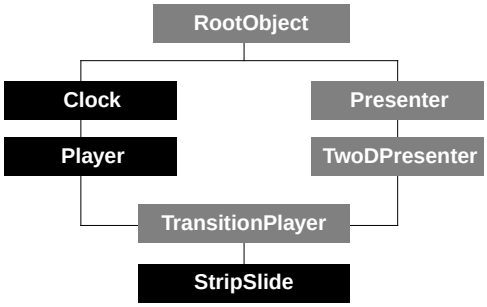
(Although this functions is technically a global function, it is included here because it operates only on instances of `StringIndex` and therefore works like an instance method. It is also included in the section on global functions.)

The global function `updateIndex` rebuilds *strIndex*'s signature index based on *strIndex*'s `string` instance variable. In order to keep the index valid, `updateIndex` should be used any time *strIndex*'s `string` is modified.

The following code changes the `string` instance variable of the `StringIndex` object `strIndex` and then updates the signature index of `strIndex` based on its new modified string:

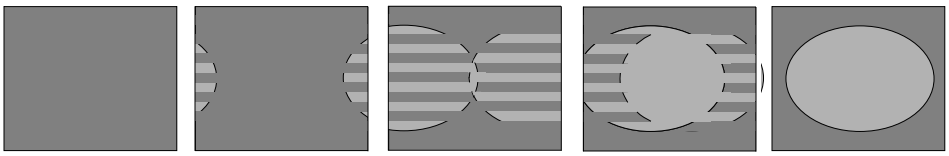
```
global strIndex := new StringIndex \
    string:"This is the original string."
strIndex.string := "This is a new string."
updateIndex strIndex
```

StripSlide



Class type: Loadable class (concrete)
Resides in: `ltrans.lib`. Works with ScriptX and KMP executables
Inherits from: `TransitionPlayer`
Component: Transitions

The `StripSlide` transition player provides a visual effect of where two parts of the target gradually slide together toward the center from opposite sides. The `@horizontal` direction playing forward is shown below. The target appears when the transition is played forward, and disappears when played backward (transition's rate set to -1).



Directions: `@horizontal`, `@vertical`

Rate: Can play forward or backward.

For a side-by-side illustrations of all transitions, see the Transitions chapter in the *ScriptX Components Guide*.

Creating and Initializing a New Instance

The following script is an example of how to create a new instance of the `StripSlide` class:

```
myTransition := new StripSlide \
                    duration:60 \
                    direction:@horizontal \
                    target:myShape
```

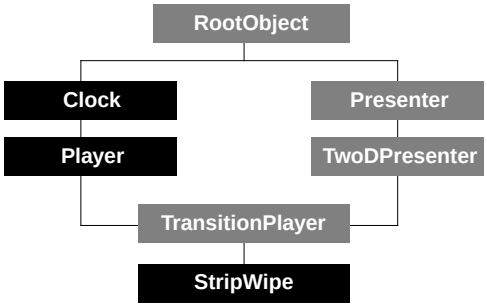
The variable `myTransition` contains the initialized transition. The transition reveals the image `myShape` by starting with it broken into two parts that gradually slide together horizontally toward the center from opposite sides, and has a duration of 60 ticks.

You determine which space the transition will take effect in by adding this instance into that space. Then, when you play the transition player, `myShape` is transitioned into that space.

The `new` method uses the keywords defined in `init`.

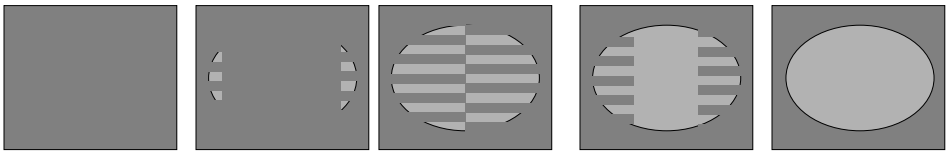
NOTE – For the instance variables and methods, see the `BarnDoor` class.

StripWipe



Class type: Loadable class (concrete)
Resides in: `ltrans.lib`. Works with ScriptX and KMP executables
Inherits from: `TransitionPlayer`
Component: Transitions

The `StripWipe` transition player provides a visual effect where the target gradually appears, wiped into view with either horizontal or vertical strips. The `@horizontal` direction playing forward is shown below. The target appears when the transition is played forward, and disappears when played backward (transition's rate set to -1).



Directions: `@horizontal`, `@vertical`

Rate: Can play forward or backward.

For a side-by-side illustrations of all transitions, see the Transitions chapter in the *ScriptX Components Guide*.

Creating and Initializing a New Instance

The following script is an example of how to create a new instance of the `StripWipe` class:

```
myTransition := new StripWipe \
                  duration:60 \
                  direction:@horizontal \
                  target:myShape
```

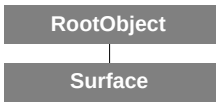
The variable `myTransition` contains the initialized transition. The transition reveals the image `myShape` wiped into view with either horizontal strips, and has a duration of 60 ticks.

You determine which space the transition will take effect in by adding this instance into that space. Then, when you play the transition player, `myShape` is transitioned into that space.

The `new` method uses the keywords defined in `init`.

NOTE – For the instance variables and methods, see the `BarnDoor` class.

Surface



Class type: Core class (abstract)
Resides in: ScriptX and KMP executables
Inherits from: `RootObject`
Component: 2D Graphics

The `Surface` class represents a general 2D rendering plane and defines the methods for the standard rendering operations `fill`, `stroke`, and `transfer`.

The `Surface` subclasses can be used directly to render stencils. They are also used in the context of the `draw` method of a presenter. (See the chapters "2D Graphics" and "Spaces and Presenters" in the *ScriptX Components Guide* for more information.)

Subclasses of `Surface` provide various ways to render stencils. For example, the `BitmapSurface` class provides a pixel-based area of memory in which to perform rendering operations. The `BitmapSurface` subclass `DisplaySurface` provides system-independent access to the windows of the underlying window system.

The units of the coordinate space provided by the surface depends on the subclass being used. For example, units provided by a `BitmapSurface` or `DisplaySurface` object correspond to pixels.

The following example creates a display surface, an oval stencil, and then strokes the oval onto the display surface:

```
global ds := new DisplaySurface
global ov := new Oval x2:100 y2:200
stroke ds ov ov identityMatrix blackBrush
```

Even though a display surface looks like a window, it is much more primitive. Notice that if you occlude the oval with any other window (such as the Listener window), and then move the window away, the oval does *not* redraw itself. Surfaces have no clocks or draw methods to refresh them after they are drawn, like windows do.

Instance Variables

boundary

self`.boundary` (read-write) `Rect`

Specifies the boundary of the surface *self* as a `Rect` object. Subclasses may prevent setting this instance variable when the drawing surface represented cannot be resized. A fill of the boundary affects all pixels on the surface.

Instance Methods

fill

`fill` *self source clip position brush* ⇨ *self*

self	`Surface`
source	`Stencil`
clip	`Stencil`
position	`TwoDMatrix`
brush	`Brush`

Renders the image of the *source* stencil onto the surface *self* after applying the *position* matrix and the *clip* stencil. Fills the image area of *source* with the brush specified by *brush*. Brush properties that apply when stroking a stencil are `color`, `pattern`, and `inkMode` defined by the `Brush` class. Returns *self*.

Use the `fill` method with `defaultBrush` as *brush* to imprint the image in a bitmap on a surface.

stroke

`stroke` *self source clip position brush* ⇨ *self*

self	`Surface` object
source	`Stencil` object
clip	`Stencil` object
position	`TwoDMatrix` object
brush	`Brush` object

Renders the image of the *source* stencil onto the surface *self* after applying the *position* matrix and the *clip* stencil. This method outlines the image area with the brush specified by *brush;* all brush properties apply when stroking a stencil, including `color`, `pattern`, `inkMode`, `lineWidth`, `lineCap`, `lineJoin`, `miterLimit`, and `dashPattern` defined on the `Brush` class. Returns *self*.

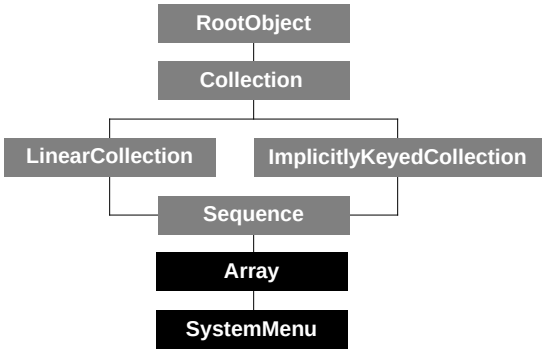
transfer

`transfer` *self source clip position* ⇨ *self*

self	`Surface` object
source	`Surface` object
clip	`Stencil` object
position	`TwoDMatrix` object

Transfers the contents of the *source* surface onto the surface *self*, positioning the source image with the *position* matrix and limiting the resulting image to the image area of the *clip* stencil. Returns *self*.

SystemMenu



Class type: Core Class (concrete)
Resides in: ScriptX and KMP executables
Inherits from: `Array`
Component: Title Management

The `SystemMenu` class provides a container for `SystemMenuItem` objects. A `SystemMenuBar` object acts as a container for `SystemMenu` objects. The standard menus that appear on all platforms, such as the **File** and **Edit** menus, are implemented as `SystemMenu` objects by ScriptX.

`SystemMenu` inherits from `Array`. The class specializes those methods that add, set, or delete the value of any element in the array, such as `addNth`, `setNth`, and `deleteNth`, to insure that elements are instances of `SystemMenuItem`, and that the menu bar is updated when any changes are made.

Creating and Initializing a New Instance

The following script creates a new instance of `SystemMenu`:

```
myMenuChoices := new SystemMenu \
       name:"Edit"
```

The variable `myMenuChoices` contains the initialized system menu. When `myMenuChoices` is instantiated, ScriptX applies the default values of `growable` and `initialSize`, keywords defined by the parent `Array` class.

init

`init` *self* [`name:` *string*] ⇨ *self*

self	`SystemMenu` object
`name:`	`String` object, used to set the name of the menu

Superclasses of `SystemMenu` apply the following keywords:

`initialSize:`	`Integer` object
`growable:`	`Boolean` object

Initializes the `SystemMenu` object *self*, setting the name that appears at the top of the menu to the value supplied for `name`. See the `Array` class for information on how `initialSize` and `growable` are applied.

Instance Variables

authorData

self.`authorData` (read-write) *(object)*

Specifies an object that is supplied as an argument when the system menu's preselect action or select action is called.

name

self.`name` (read-write) `String`

Specifies the name of the system menu *self*. The value of name is stored internally as a string, such as `"Edit"` or `"File"`. Any object that can be coerced to a string can be used to set the value of name.

preSelectAction

self.`preSelectAction` (read-write) *(function)*

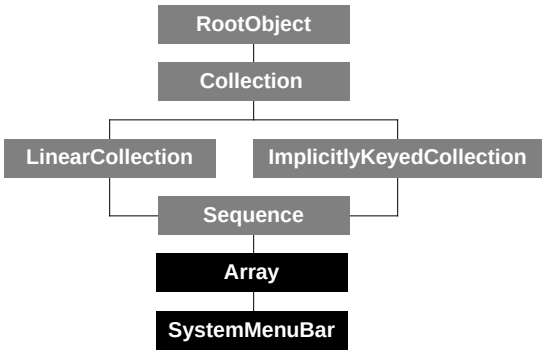
Specifies a function that is called automatically when the user clicks on the menu in the menu bar, but before a menu item is selected. This function is called with the one argument, for which the value of `authorData` is supplied.

selectAction

self.`selectAction` (read-write) *(function)*

Specifies a function that is called automatically when a menu item is selected. This function is called with the one argument, for which the value of `authorData` is supplied.

SystemMenuBar



Class type: Core class (concrete)
Resides in: ScriptX and KMP executables
Inherits from: `Array`
Component: Title Management

The `SystemMenuBar` class gives ScriptX the ability to control the appearance of the menu bar that is presented by the native operating system. Each title has its own system menu bar, as specified by the `systemMenuBar` instance variable defined by `TitleContainer`. You can hide or show the menu bar and enable or disable its menu items. One active title can show the menu bar, while another hides it. The menu bar automatically changes depending on which title has user focus.

The location and appearance of the system menu bar is platform-dependent, as shown in the following figure. With MacOS, the menu bar is always located at the top of the screen. In Windows and OS/2, the menu bar is located below the ScriptX title bar, which appears at the top of the screen only when the ScriptX window is "maximized."

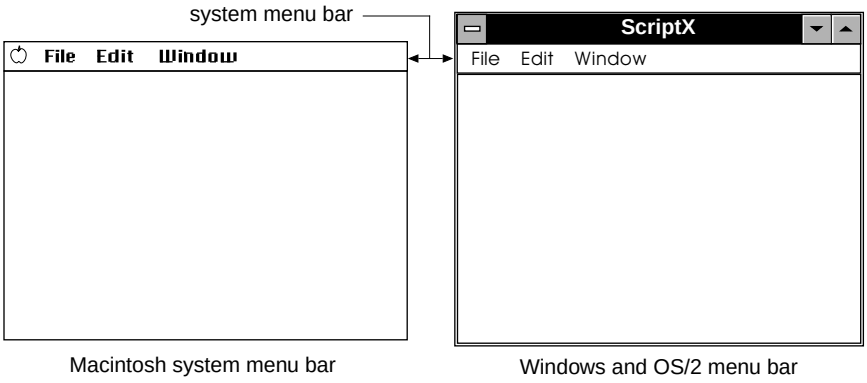


Figure 4-30: The menu bar's appearance is determined by the underlying platform.

A system menu bar is a container for system menus, and a system menu is a container for system menu items. `SystemMenuBar` inherits from `Array`, which it specializes to store only instances of `SystemMenu`. The class specializes those methods that add, set, or delete the value of any element in the array, such as `addNth`, `setNth`, and `deleteNth`, to insure that elements are instances of `SystemMenu`, and that the menu bar is updated when any changes are made.

The `SystemMenuBar` class provides access, via class variables, to internal functions that open and close titles, open accessories, bring up a new listener window, and bring up a page setup dialog box. This allows a programmer to invoke these functions as the select action of a menu item. Each of these functions takes one argument, which is ignored, for signature compatibility.

Creating and Initializing a New Instance

The following script creates a new instance of `SystemMenuBar`:

```
myMenus := new SystemMenuBar
```

The variable `myMenus` contains the initialized system menu bar. When `myMenus` is instantiated, ScriptX applies the default values of `growable` and `initialSize`, keywords defined by the parent `Array` class.

init

`init` *self* ⇨ *self*

self `SystemMenuBar` object

Superclasses of `SystemMenuBar` apply the following keywords:

`initialSize:`	`Integer` object
`growable:`	`Boolean` object

Initializes the `SystemMenuBar` object *self*. See the `Array` class for information on how `initialSize` and `growable` are applied.

Class Variables

closeTitleAction (SystemMenuBar)

self`.closeTitleAction` (read-only) *(function)*

Returns the internal function that closes a title. The associated function takes one argument, which is ignored.

newListenerAction (SystemMenuBar)

self`.newListenerAction` (read-only) *(function)*

Returns the internal function that opens a new Listener window. The associated function takes one argument, which is ignored.

openAccessoryAction (SystemMenuBar)

self`.openAccessoryAction` (read-only) *(function)*

Returns the internal function that opens an accessory. The associated function takes one argument, which is ignored.

openTitleAction (SystemMenuBar)

self`.openTitleAction` (read-only) *(function)*

Returns the internal function that opens a title. The associated function takes one argument, which is ignored.

pageSetupAction (SystemMenuBar)

self`.pageSetupAction` (read-only) *(function)*

Returns the internal function that opens a page setup dialog. The associated function takes one argument, which is ignored.

Instance Variables

hasUserFocus

self.hasUserFocus (read-only) Boolean

When true, the appearance of the system menu bar *self* responds to changes in its instance variables and methods; when false, it does not respond to changes. In the current release, since each title has exactly one system menu bar, this variable is set to true whenever a window in the title has user focus.

A title container manages focus on its system menu bar automatically—it stores a reference to its system menu bar in its systemMenuBar instance variable. It is possible for no window in a title to have focus while its system menu bar has focus. This can happen if two titles share the same menu bar—the background title does not have focus, but its menu bar has focus because it is used by the foreground title.

isVisible

self.isVisible (read-write) Boolean

When true, the system menu bar *self* is visible; when false, the system menu bar is hidden. Setting isVisible has the same effect as calling the methods show and hide.

Instance Methods

hide

hide *self* ⇨ *self*

Hides the system menu bar *self*. Has the same effect as setting isVisible to false.

setUserFocus

setUserFocus *self hasUserFocus* ⇨ Boolean

self	SystemMenuBar object
hasUserFocus	Boolean object

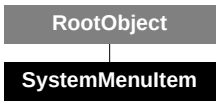
Sets user focus for the system menu bar *self*. This method is not usually called from the scripter, since the title container normally manages focus. It is visible to the scripter so that it can be specialized.

show

show *self* ⇨ *self*

Shows the system menu bar *self*. Has the same effect as setting isVisible to true.

SystemMenuItem



Class type: Core Class (concrete)
Resides in: ScriptX and KMP executables
Inherits from: `RootObject`
Component: Title Management

The `SystemMenuItem` class implements the behavior of individual menu items on a system menu

Creating and Initializing a New Instance

The following script creates a new instance of `SystemMenuItem`:

```
myItem := new SystemMenuItem \
       name: "Print List"
```

The variable `myItem` contains the initialized system menu item, which can be added to a system menu. It is displayed in a menu as a choice or menu command with the name "Print List." The `new` method uses keyword arguments defined by the `init` method.

init

`init` *self* ⇨ *self*

self	`SystemMenuItem` object
`name:`	`String` object

Initializes the `SystemMenuItem` object *self*, applying the keyword `name` to the instance variable of the same name.

If you omit an optional keyword, its default value is used:

`name:undefined`

Instance Variables

authorData

self`.authorData` (read-write) *(object)*

Specifies an object that is supplied as an argument when the system menu item's select action is called.

checked

self`.checked` (read-write) `Boolean`

Specifies whether the menu item *self* is checked.

enabled

self`.enabled` (read-write) `Boolean`

Specifies whether the menu item *self* is enabled.

name

self.name (read-write) String

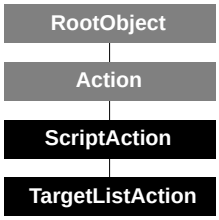
Specifies the name of the system menu item *self*. The value of name is stored internally as a string, such as "Copy" or "Paste". Any object that can be coerced to a string can be used to set the value of name.

selectAction

self.selectAction (read-write) *(function)*

Specifies a function that is called automatically when a menu item is selected. This function is called with the one argument, for which the value of authorData is supplied.

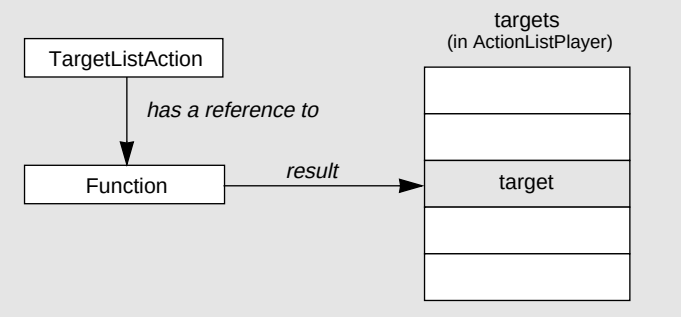
TargetListAction



Class type: Core class (concrete)
Resides in: ScriptX and KMP executables
Inherits from: `ScriptAction`
Component: Animation

The `TargetListAction` class represents an action that will execute a specified function at a certain time and replace a specified position in the target list with the return value from the function. The function does not change the number of items in the target list. This all happens under the control of an action list player. To be triggered, an instance of `TargetListAction` needs to be added to the action list of an action list player, then the player needs to be played.

The following figure shows how a target list action works. First you save a function to its `script` instance variable and a time to it `time` instance variable. When the specified time arrives, the function is evaluated, and its result is placed in the player's target list at the position specified by the `targetNum` instance variable (inherited from the `Action` class). The previous occupant of that slot in the target list is removed (unless that previous occupant is identical to the new occupant). See the `ScriptAction` class, from which `TargetListAction` inherits, for more details about the script.



To cause a target list action to remove a target object from the `targets` and not replace it with another object, the function should return `undefined`. This does not necessarily dispose the object; you must do that separately.

Use `TargetListAction` if you want ScriptX to automatically dispose of objects as you rewind an `ActionListPlayer` object. Fast forward simply plays all actions quickly up to the specified time. However, rewinding is not so simple. Rewinding is accomplished by rewinding to the very beginning of the action list and fast forwarding to the specified time. If there is a possibility that you might rewind to a point in time where a target object might not exist, you must make sure that object is disposed of during the rewind. You can do this by adding objects to the target list using a `TargetListAction` object and specifying a dispose function for `rewindScript`. Then, during any rewind, all the objects on the target list are automatically disposed of before playing the action list player from the beginning of the action list.

Creating and Initializing a New Instance

The following script is an example of how to create a new instance of the TargetListAction class. The two functions create an oval and dispose of the oval:

```
function createOval action target player ->
    (new TwoDShape target:(new Oval x2:80 y2:80) fill:blackBrush)

function disposeOval action target player ->
    (deleteOne target.presentedBy target; undefined)

myAction := new TargetListAction \
              script:createOval \
              rewindScript:disposeOval \
              targetNum:4 \
                  time:20
```

The variable myAction holds an initialized instance of ScriptAction. This instance executes the function createOval at the action list player's time of 20 ticks. The result of the function (an oval) is placed at position 4 of the list specified by the player's targets instance variable. When rewind is called on the action list player to a point earlier than time 20, the rewind script is run. The new method uses the keywords defined in init.

init

init *self* [rewindScript:*function*] [script:*function*]
[targetNum:*integer*] [time:*integer*] ⇨ *(none)*

self	TargetListAction object
rewindScript:	Function object to be executed when rewind occurs

Superclass ScriptAction uses the following argument:

script:	Function object to be executed when the TargetListAction is triggered

Superclass Action uses the following keywords:

targetNum:	Integer indicating which object in the target list of the player for this TargetListAction to apply the action to
time:	Integer object representing the time in ticks to trigger the action

Initializes the TargetListAction object *self*, applying the arguments to the instance variables of the same name. At the time specified by time, this target list action will cause the function specified by script to execute and place the object it returns into the player's target list at the position specified by targetNum. The script function must be in the form described in ScriptAction. (See ScriptAction for more details.) Do not call init directly on an instance—it is automatically called by the new method.

If you omit an optional keyword, its default value is used. The defaults are:

```
rewindScript:undefined
targetNum:0
time:0
script:undefined
```

Instance Variables

Inherited from Action:

authorData targetNum time

`playOnly`

Inherited from `ScriptAction`:

`script`

The following instance variable is defined in `TargetListAction`:

rewindScript

self`.rewindScript` (read-write) *(function)*

Specifies the function that runs when `rewind` is called on the action list player that controls the target list action *self*. When you create an instance of `TargetListAction`, you should provide both a script to create the object, and the rewind script for destroying the object. You should define the rewind script with the same arguments as the `ScriptAction` script:

```
function myFunc action target player -> ( -- body of function )
```

If you want the target to be removed from the target list without adding another object, the function should return `undefined`; for example:

```
myAction.rewindScript := (action target player ->
       deleteOne target.presentedBy target
       undefined)
```

When the rewind script is called, the first argument `action` is passed in as `undefined` because the target list action is not being triggered. As a convenience, the format of this function is the same as for the `script` instance variable. This makes it possible for the target list action to use the same function to remove an object when playing forward (`script`), as to dispose of the object when rewinding (`rewindScript`)—as long as neither script uses the first argument.

Although any global function, anonymous function, or method can be assigned to `rewindScript`, there are differences in how different classes of functions are dispatched. For information on functions and function dispatch, see the "Object System Kernel" chapter of the *ScriptX Components Guide*.

Instance Methods

Inherited from `Action`:

`trigger`

Inherited from `ScriptAction`:

`trigger`

The following instance method is defined in `TargetListAction`:

trigger (ScriptAction)

`trigger` *self target player* ⇨ *(object)*

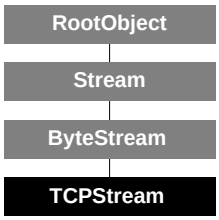
self	`TargetListAction` object
target	Any object
player	`ActionListPlayer` object

Causes the target list action *self* to execute the function specified by the `script` instance variable and replace a specified position in the target list with the return value from the function. The position is specified by the `targetNum` instance variable. This method returns the object returned by evaluating the function.

This method is called by an `ActionListPlayer` object at the time specified by the `time` instance variable. This action list player is passed in as the third argument *player*.

When the action list player calls `trigger`, the value for *target* is automatically determined by taking the `targetNum` instance variable and finding the object in the corresponding slot in the action list player's target list. If the `targetNum` instance variable is out of range (that is, less than 1 or greater than the size of the target list), the value of the *target* is `empty`. Also, if there is no *target* in a certain slot in the target list yet, the value of *target* is `undefined`.

TCPStream



Class type: Loadable class (concrete)
Resides in: `web.lib`. Works with ScriptX and KMP executables
Inherits from: `Stream`
Component: Streams

The `TCPStream` class creates a readable and writable, but unseekable, stream that allows a ScriptX program to receive data through Transmission Control Protocol (TCP), the standard transport layer protocol on the internet.

ScriptX developers should be aware that internet clients configure their systems independently with a TCP driver. Kaleida labs has not tested `TCPStream` with every possible system/driver configuration. Although any machine that is already connected to the internet must be configured to use a TCP driver, users may choose a TCP driver from a variety of vendors or service providers, and different versions of the popular TCP drivers may be in use. For the latest information on using the Kaleida Media Player with different TCP drivers, see Kaleida's World Wide Web site (`http://www.kaleida.com/`).

Macintosh users generally connect to the internet using MacTCP, which is supplied by Apple Computer. Windows systems connect with one of several implementations of Winsock, a dynamic link library. Windows 3.1 users must use a Winsock implementation that has a 16-bit stack, while Windows 95 users may use either a 16-bit or 32-bit version. OS/2 users can connect to the internet using either as a DOS or OS/2 client, using BSD or, as a Windows client, using Winsock.

Output to an instance of `TCPStream` is buffered. Data is not sent until one of the following conditions is met:

- the buffer is full
- the program calls `flush` (an instance method defined by `Stream`) on the TCP stream
- the program attempts to read from the TCP stream

Creating and Initializing a New Instance

The following script creates an instance of `TCPStream` connected to the specified server port:

```
clientStream := new TCPStream host:"www.kaleida.com" port:80
```

The variable clientStream contains a reference to an instance of `TCPStream` whose host is `www.kaleida.com`, connected through port 80.

The following script creates an instance of `TCPStream` that accepts connections from other programs:

```
serverStream := new TCPStream port:1099 listen:true
listeningToClientStream := accept serverStream
```

The variable `serverStream` contains a reference to an instance of `TCPStream` that listens for connections from clients. When a client connects to the server stream, the client should generally spawn a new thread in which to process the stream and continue listening for more connections. The server thread can block while waiting for a connection by calling `accept`. When a client connects, the program assigns the client stream to another process and resumes listening for more connections.

The `new` method uses the keyword arguments defined by `init`.

init

`init` *self* `port`:*integer* [`host`:*string*] [`listen`:*Boolean*] ⇨ *(none)*

self	`TCPStream` object
`port:`	`ImmediateInteger` object
`host:`	`String` object, an internet name or IP address
`listen:`	`Boolean` object
`oOBDCallback:`	function, used to create a callback

Initializes the `TCPStream` object *self*, applying the keyword arguments as follows: `port` takes an integer, representing the port number that specifies the application socket to the TCP layer, `host` takes a `String` object representing either the full internet name of the host ("www.kaleida.com") or its IP address ("198.186.9.22"), and is used to set the values of the instance variables `host` and `ipAddress`. `listen` is a state variable that specifies whether the new stream should wait for a connection. The `oOBDCallback` keyword is used to specify a function that will be called when the stream receives out-of-band data, such as interrupts.

If you omit an optional keyword, its default value is used. The defaults are:

```
host:undefined
listen:false
oOBDCallback:undefined
```

Instance Variables

host

self`.host` (read-only) `StringConstant`

Specifies the internet name of the remote system.

ipAddress

self`.ipAddress` (read-only) `StringConstant` or `Sequence`

Specifies the internet IP address of the remote system. IP addresses can be specified using either as a string constant or a sequence that stores four integer values, such as `Array` or `Quad`.

```
myTCPStream.ipAddress := "198.186.9.22" -- string constant
myTCPStream.ipAddress := #(198, 186, 9, 22) -- quad
```

port

self`.port` (read-only) `ImmediateInteger`

Specifies the application port through which the `TCPStream` object *self* is connected with the remote host.

Instance Methods

Inherited from `Stream`:

`cursor`	`next`	`seekFromStart`
`flush`	`plug`	`setStreamLength`
`isAtFront`	`previous`	`streamLength`
`isPastEnd`	`read`	`write`
`isReadable`	`readReady`	`writeReady`
`isSeekable`	`seekFromCursor`	
`isWritable`	`seekFromEnd`	

Inherited from `ByteStream`:

`fileIn`	`read`	`write`
`pipe`	`readByte`	`writeByte`
`pipePartial`	`readReady`	`writeString`

The following instance methods are defined in `TCPStream`:

accept

`accept` *self* ⇨ `TCPStream`

Causes the current thread to block while waiting for a connection from a client. When a connection is established, `accept` returns a `TCPStream` object that is connected to the client.

Note – Certain ScriptX operations should not call a function or method, such as `accept`, that can cause a thread to block. For more information, see the discussion of blocking in the "Threads" chapter of the *ScriptX Components Guide*.

getHostName

`getHostName` *self* ⇨ `String`

Returns the value of the instance variable `host` as a `String`.

getIPAddress

`getIPAddress` *self* ⇨ `Quad`

Returns the value of the instance variable `ipAddress` as a `Quad`.

isReadable (Stream)

`isReadable` *self* ⇨ `Boolean`

Returns `true`, because a `TCPStream` object is readable.

isReadable (Stream)

`isSeekable` *self* ⇨ `Boolean`

Returns `false`, because a `TCPStream` object is not seekable.

isWritable (Stream)

`isWritable` *self* ⇨ `Boolean`

Returns `true`, because a `TCPStream` object is writable.

readOOBData

`readOOBData` *self* ⇨ `Array`

Returns out of band data to the `TCPStream` object *self*, if any has been received. If no out of band data has been received, `readOOBData` returns `undefined`.

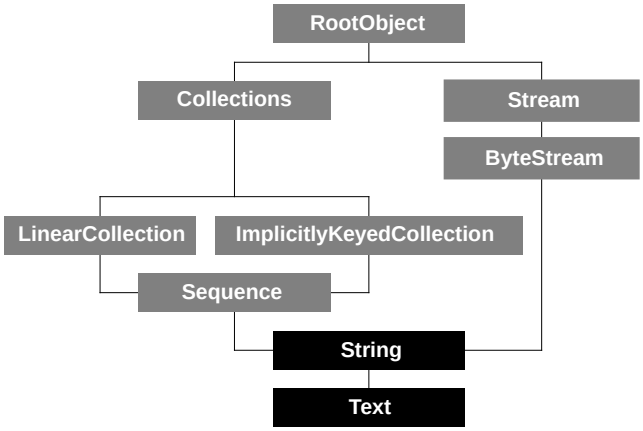
writeOOBData

`writeOOBData` *self array* ⇨ *(none)*

self	`TCPStream` object
array	`Array` object containing out of band data

Writes out of band data to the `TCPStream` object *self*. Out of band data is used to send data, such as interrupts, outside of the normal packet mechanism. For example, `writeOOBData` could be used to interrupt an FTP transmission from a host computer in mid stream.

Text



Class type: Core class (concrete)
Resides in: ScriptX and KMP executables
Inherits from: `String`
Component: Text and Fonts

The `Text` class represents text strings. `Text` objects differ from `String` objects in that they include stylistic attributes such as typeface, point size, and leading.

The `Text` class is not a presenter. To present text, you must use an instance of `TextPresenter` or `TextEdit`, with a `Text` object as its target.

Ranges of characters in `Text` objects, as used in selections and attributes, are expressed using cursor positions. Cursor positions occur between characters, with position 0 indicating the beginning of the string (before the first character). This differs from the *ordinal position* of each character, which occurs directly on the character itself. Figure 31 illustrates the difference between cursor and ordinal positions. Note that a single character is also a range of two cursor positions.

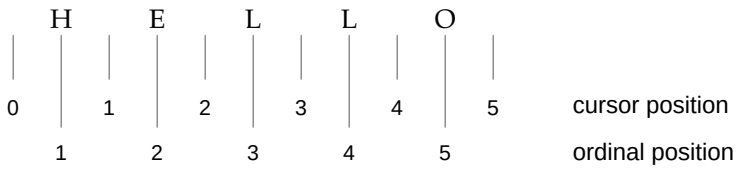


Figure 31: Cursor And Ordinal Positions

Ranges of characters in a `Text` object can be assigned *attributes*, which define how that range of text is displayed and formatted when it is the target of a presenter. By default, a `Text` object uses the attributes defined by the `TextPresenter` or `TextEdit` object that displays it. You can specify attributes different from the default for a given range of characters using the `setAttr` and `setAttrFromTo` generic functions, and query the attributes by using the `getAttr` and `getAttrs` generic functions.

Attributes are stored as a keyed linked list, where the key is the name of an attribute, such as `@font` or `@size`, and the value is the value of that attribute. Table 10 summarizes the available text attributes. For more information, see the "Text and Fonts" chapter of the *ScriptX Components Guide*.

Table 10: Text Attributes

Attribute	Possible values	Values Represent
`@action`	A function or generic function	An action to perform when this range of characters is selected on a text presenter. For more information, see the "Text and Fonts" chapter of the *ScriptX Components Guide*.
`@brush`	`Brush` object	The color, pattern, and so on, of the text.
`@font`	`Font` object	The font family (typeface)
`@size`	`Number` object	The point size of the text
`@weight`	`@extraLight`, `@light`, `@regular`, `@medium`, `@demiBold`, `@bold`, `@extraBold`, `@heavy`	The boldness of the text (as defined by variations in the font family). The `@regular` value is the standard weight, available for most textbook fonts. The `@medium` value is less common.
`@width`	`@condensed`, `@normal`, `@expanded`	The horizontal spacing of the text (as defined by variations in the font family).
`@style`	`@roman`, `@italic`, `@oblique`	The obliqueness of the text (as defined by variations in the font family).
`@underline`	`0`, `1`	`0` represents no underlining, `1` represents underlining (as defined by the font). The `@underline` attribute in future versions of ScriptX will use numeric values to indicate the space between the under line and the text.
`@leading`	`Number` object	The spacing, in points, between lines of text.
`@paraLeading`	`Number` object	The spacing, in points, between paragraphs.
`@firstLineLeading`	`Number` object	The spacing, in points, between this line of text and the top of the text's presenter
`@alignment`	`@flush`, `@flushLeft`, `@flushToEnd`, `@flushRight`, `@fill`, `@center`, `@tty`	The alignment of the text within the presenter.

Table 10: Text Attributes *(Continued)*

Attribute	Possible values	Values Represent
`@paraIndent`	`Number` object	Number of points to indent the first line of a paragraph.
`@indent`	`Number` object	Number of points to indent the left margin of the text. This applies to lines other than the first line of a paragraph
`@indentFromEnd`	`Number` object	Number of points to indent the right margin of the text

Creating and Initializing a New Instance

Instances of the `Text` class can be created in two ways:

- By coercing a string literal (an instance of the `StringConstant` class) into a `Text` object:

```
myText := "Hello, world!" as Text
```

- By using the `new` method:

```
myText := new Text string:"Hello, world!"
```

Each of these examples assigns a new instance of `Text` to the variable `myText`. The `new` method in the second example uses the `string` keyword defined by the `init` method.

init

`init` *self* [`string:`*string*] ⇨ *(none)*

self	`Text` object
`string:`	`String` or `Text` object

Initializes the `Text` object *self*, where `string:` provides the string of characters. Do not call `init` directly on an instance—it is automatically called by the `new` method.

If you omit the keyword `string`, a string of zero length is supplied by default:

```
string:""
```

Class Variables

defaultAttributes

self`.defaultAttributes` (read-write) `KeyedLinkedList`

Specifies the default values for the `attributes` instance variable in new instances of the class *self*. The attributes in `defaultAttributes` are initialized to `undefined`.

Note – It is safest never to set the `deaultAttributes` class variable for `Text` objects. Setting `defaultAttributes` affects all new `Text` objects, those outside of your title as well as those within it. This means that if another title is running, you have changed the default attributes for any new `Text` objects in that title in addition to those in your own title.

Class Methods

Inherited from `Collection`:

`pipe`

Instance Variables

Inherited from `Collection`:

`bounded`	`maxSize`	`size`
`iteratorClass`	`minSize`	`uniformity`
`keyEqualComparator`	`mutable`	`uniformityClass`
`keyUniformity`	`mutableCopyClass`	`valueEqualComparator`
`keyUniformityClass`	`proprietored`	

The following instance variable is defined in `Text`:

selection

self`.selection` (read-write) `Integer` or `IntegerRange`

Specifies the range of text that is selected within the text *self*, measured using cursor position. The value of `selection` can be one of three values:

- `empty`, representing no selection
- an `Integer` object, representing a single cursor position within the text
- an `IntegerRange` object, representing a range of characters within the text

The instance variable `selection` is used by the `TextEdit` methods `clearSelection`, `cutSelection`, `copySelection`, and `pasteToSelection`. One way to create a selection is by dragging the mouse over the characters you want to be affected by one of the `TextEdit` methods. The second way is to set the `selection` instance variable to the range of characters to be included. The following code selects the characters between cursor positions 7and 11 (the word `want`):

```
myText := "Text I want to edit" as Text
myText.selection := 7 to 11
```

Instance Methods

Inherited from `Collection`:

`add`	`forEach`	`iterate`
`addMany`	`forEachBinding`	`localEqual`
`addToContents`	`getAll`	`map`
`chooseAll`	`getAny`	`merge`
`chooseOne`	`getKeyAll`	`pipe`
`chooseOneBinding`	`getKeyOne`	`prin`
`deleteAll`	`getMany`	`removeAll`
`deleteBindingAll`	`getOne`	`removeOne`
`deleteBindingOne`	`hasBinding`	`setAll`
`deleteKeyAll`	`hasKey`	`setOne`
`deleteKeyOne`	`intersects`	`size`
`deleteOne`	`isEmpty`	
`emptyOut`	`isMember`	

Inherited from `LinearCollection`:

`chooseOneBackwards`	`findRange`	`getNthKey`

chooseOrdOne	forEachBackwards	getOrdOne
deleteFirst	getFirst	getRange
deleteLast	getLast	localEqual
deleteNth	getMiddle	localLT
deleteRange	getNth	pop

Inherited from Sequence:

addFifth	moveBackward	setFourth
addFirst	moveForward	setLast
addFourth	moveToBack	setNth
addNth	moveToFront	setSecond
addSecond	prepend	setThird
addThird	prependNew	sort
append	setFifth	
appendNew	setFirst	

Inherited from Stream:

cursor	next	seekFromStart
flush	plug	setStreamLength
isAtFront	previous	streamLength
isPastEnd	read	write
isReadable	readReady	writeReady
isSeekable	seekFromCursor	
isWritable	seekFromEnd	

Inherited from ByteStream:

fileIn	readByte	writeString
pipe	readReady	
pipePartial	writeByte	

Inherited from String:

copyFromTo	deleteFromTo	hashOf
cutFromTo	getLowercase	insertAt

Note – Only the method insertAt will operate on Text objects and preserve the attributes. Both strings must be Text objects in order to preserve attributes. Methods like copyFromTo and concatenation operations will copy or concatenate the characters in a Text object, but not the attributes associated with those characters.

The following instance methods are defined in Text:

getAttr

getAttr *self key cursorPosition* ⇨ (*object*)

self	Text object
key	NameClass object
cursorPosition	Integer object, representing a cursor position

Returns the value of the attribute specified by *key* of the text at the given *cursorPosition* within the text *self*, or undefined if the given text at that *cursorPosition* does not have a value for that attribute. Text attributes are summarized in Table 10, "Text Attributes," on page 737 and in greater detail in the *ScriptX Components Guide*.

getAttrRange

`getAttrRange` *self key cursorPosition* ⇨ `NumberRange`

self	`Text` object
key	`NameClass` object
cursorPosition	`Integer` object, representing a cursor position

If the given attribute specified by *key* exists at the *cursorPosition* within the text *self*, it returns the range of cursor positions over which that attribute is set.

getAttrs

`getAttrs` *self cursorPosition* ⇨ `KeyedLinkedList`

self	`Text` object
cursorPosition	`Integer` object, representing a cursor position

Returns a keyed linked list representing all the attributes of the text *self* at the given *cursorPosition*, or an empty keyed linked list if the text at the given `cursorPosition` has no attributes. Text attributes are summarized in Table 10, "Text Attributes," on page 737 and in greater detail in the *ScriptX Components Guide*.

setAttr

`setAttr` *self key cursorPosition newValue* ⇨ *(none)*

self	`Text` object
key	`NameClass` object
cursorPosition	`Integer` object, representing a cursor position
newValue	`NameClass` or `Number` object

Applies the attribute set by *key* to the text beginning after the given *cursorPostion* and continuing to the end of the string in the `Text` object *self*. If the attribute has previously been set for a character between `cursorPosition` and the end, the change will be applied only up to that character. The attribute is set to *newValue*.

Suppose you have a `TextEdit` object that presents a target `Text` object whose target is the string `"1234567890"`, as shown in the first line below. The first `setAttr` expression changes the value of the attribute `@size` from position 8 to the end of the target string. The second `setAttr` expression changes `@size` from position 5 through 7 (it stops at 7 because of the `@size` setting already there).

```
txt := new Text string:"1234567890"
setAttr txt @size 7 24 -- sets "890" to point size 24
setAttr txt @size 4 36 -- sets "567" to point size 36
setAttr txt @underline 0 1 -- underlines all the text
```

It is important to remember that `cursorPosition` is indeed a cursor positon, and not an ordinal position. Cursor position 7 lies between ordinal positions 7 and 8, and in the example above, it affects the characters from ordinal position 8 to the end of the string. For a summary of text attributes, see Table 10, "Text Attributes," on page 737 and the "Text and Fonts" chapter of the *ScriptX Components Guide*.

setAttrFromTo

`setAttrFromTo` *self key startPosition endPosition newValue* ⇨ *(none)*

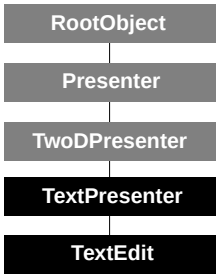
self	`Text` object
key	`NameClass` object
startPosition	`Integer` object, representing a cursor position
endPosition	`Integer` object, representing a cursor position
newValue	`NameClass` or `Number` object

Applies the attribute *key* to the range of text identified by *startPostion* and *endPostion* in the text *self*. The attribute is set to *newValue*. It is important to remember that startPosition and endPosition are cursor positions and tht the characters which will be affected by setAttrFromTo are those that lie between startPosition and endPosition. For example, the following code sets the weight of "345678" to boldface type:

```
setAttrFromTo txt @weight 2 8 @bold --sets "345678" to bold
```

Text attributes are summarized in Table 10, “Text Attributes,” on page 737 and in greater detail in the *ScriptX Components Guide*.

TextEdit



Class type: Core class (concrete)
Resides in: ScriptX and KMP executables
Inherits from: `TextPresenter`
Component: Text and Fonts

The `TextEdit` class is a presenter for text that allows selections, insertions, and other text editing through mouse and keyboard events.

Creating and Initializing a New Instance

The following script is an example of how to create a new instance of the `TextEdit` class:

```
myMessage := new TextEdit \
             boundary:(new Rect x2:300 y2:200) \
             target:("Hello, World!" as Text) -- cannot be a string
```

The variable `myMessage` contains an initialized `TextEdit` instance with a rectangular boundary of 300 x 200 pixels. This object presents the text "Hello, World!" Its attributes are the default attributes inherited from `TextPresenter`, which in turn serve as the default attributes for the `Text` object which is its `target` instance variable.

init

`init` *self* `boundary:`*region* `target:`*text* [`fill:`*brush*] [`stroke:`*brush*] ⇨ *(none)*

The `init` method for `TextEdit` is inherited from `TextPresenter` with no change in keywords. However, the value for the `target` keyword must be an object of class `Text` (unlike a `TextPresenter` object, which allows any `String` object). Refer to the `TextPresenter` class for further details. Do not call `init` directly on an instance—it is automatically called by the `new` method.

Instance Variables

Inherited from `Presenter`:

`presentedBy`	`subPresenters`	`target`

Inherited from `TwoDPresenter`:

`bBox`	`height`	`transform`
`boundary`	`IsImplicitlyDirect`	`width`
`clock`	`isTransparent`	`window`
`compositor`	`isVisible`	`x`
`direct`	`needsTickle`	`y`
`eventInterests`	`position`	`z`
`globalBoundary`	`stationary`	
`globalTransform`	`target`	

Inherited from TextPresenter:

attributes	fill	selectionForeground
cursor	inset	stroke
cursorBrush	offset	
enabled	selectionBackground	

Instance Methods

Inherited from TwoDPresenter:

adjustClockMaster	inside	show
createInterestList	localToSurface	surfaceToLocal
draw	notifyChanged	tickle
getBoundaryInParent	recalcRegion	
hide	refresh	

Inherited from TextPresenter:

calculate	getOffsetForXY	processMouseDown
copySelection	getPointForOffset	

The following instance methods are defined in TextEdit:

clearSelection

`clearSelection` *self* ⇨ `Text`

Deletes the selected text from *self*'s target `Text` object.

cutSelection

`cutSelection` *self* ⇨ `Text`

Removes the selected text from *self*'s target `Text` object and stores it on the clipboard (as specified by the global variable `theClipboard`).

pasteToSelection

`pasteToSelection` *self* ⇨ `Text`

Inserts the text currently stored on the clipboard (as specified the global variable `theClipboard`) into *self*'s target `Text` object at the current insertion point. If the current selection is a range of characters, that selection is replaced with the text on the clipboard.

processFocus

`processFocus` *self interest focusEvent* ⇨ `Boolean`

self	`TextEdit` object
interest	`MouseEvent` object
focusEvent	`FocusEvent` object

This method is automatically called when the user clicks on *self* with the mouse.

This method causes the `TextEdit` object *self* to gain focus and all other `TextEdit` objects to lose focus. It does this by forcing focus of *self* to the state given by *focusEvent*, and adds interests in a keydown event and a global mouse-down event. It currently always returns `true`. The given *interest* is ignored in this method.

processKeyDown

`processKeyDown` *self interest event* ⇨ `Boolean`

self	`TextEdit` object
interest	`KeyboardDownEvent` object
event	`KeyboardDownEvent` object

This method is automatically called on the `TextEdit` object *self* when that object has focus and the user presses a key.

This method processes the key-down *event* applied to the `TextEdit` object *self*, by getting the key from the event's `keyCode` instance variable and inserting it as a character, replacing any selected text. It currently always returns `true`.

processMouseDown

`processMouseDown` *self interest event* ⇨ `Boolean`

self	`TextEdit` object
interest	`MouseDownEvent` object
event	`MouseDownEvent` object

This method is automatically called on the `TextEdit` object *self* when the user presses a mouse button over *self*.

This method processes the given mouse-down *event* applied to the `TextEdit` object *self* by releasing the highlight in the text, forcing the keyboard focus (insertion point) to the location of the mouse down event, and setting the `selection` instance variable to the cursor position. In this case it returns `true`. It returns `false` if a global mouse-down interest exists, preventing the `TextEdit` object from getting focus.

It also adds a mouse move interest so that any subsequent mouse moves will be processed. It does not call the function specified by the `@action` attribute (as `processMouseDown` does in `TextPresenter`).

processMouseMove

`processMouseMove` *self interest event* ⇨ `Boolean`

self	`TextEdit` object
interest	`MouseMoveEvent` object
event	`MouseMoveEvent` object

This method is automatically called on the `TextEdit` object *self* when the user moves the mouse over *self*.

This method processes the given mouse-move *event* applied to the `TextEdit` object *self* by highlighting the text and updating the `selection` instance variable to the range of cursor positions from the previous mouse-down to the *event*. Always returns `true`.

It also adds a mouse up interest so that any subsequent mouse up will be processed. The given *interest* is ignored in this method.

processMouseUp

`processMouseUp` *self interest event* ⇨ `Boolean`

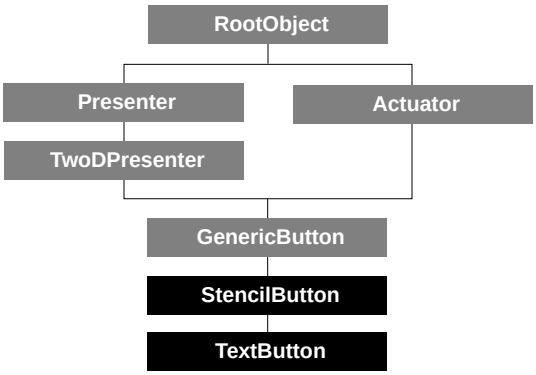
self	`TextEdit` object
interest	`MouseUpEvent` object
event	`MouseUpEvent` object

This method is automatically called on the `TextEdit` object *self* when the user releases a mouse button.

This method processes the given mouse-up *event* applied to the `TextEdit` object *self* by highlighting the text and updating the `selection` instance variable. Always returns `true`.

The given *interest* is ignored in this method. This method actually calls `processMouseMove` to perform the updating.

TextButton



Class type: Scripted class (concrete)
Resides in: widgets.sxl. Works with ScriptX and KMP executables
Inherits from: StencilButton
Component: User Interface

TextButton is a user interface Widget Kit class that provides a framed button whose appearance is defined by a text object and a Frame object that gives the button a three dimensional look. The only difference between a TextButton object and a StencilButton object is that a TextButton is specialized to display a string instead of a stencil. The TextButton class provides text and font parameters that StencilButton does not have.

The clipping boundary of a TextButton object is calculated automatically as the smallest rectangle that encloses the string and the button frame.

Creating and Initializing a New Instance

The following script is an example of how to create a new instance of the TextButton class:

```
myTextButton := new TextButton \
       text:"Kaleida!"
```

The variable myTextButton contains an initialized TextButton object. The new method uses the keywords defined in init.

init

`init` *self* [`text:`*string*] [`font:`*font*] [`stencil:`*stencil*]
[`boundary:`*stencil*] ⇨ *(none)*

self	TextButton object
text:	String object
font:	Font object
stencil:	Stencil object
boundary:	Stencil object

Initializes the TextButton object *self*, applying the values supplied with the keywords to the instance variables of the same name. Coerces the value of text to a String object, creates a new TextStencil object to display the specified text in the specified font, supplies a frame for the button, and calculates a boundary that encompasses the text stencil and the button frame. Do not call init directly on an instance — it is automatically called by the new method.

If you omit one of the keyword arguments, the following defaults are used:

```
text:"Hello"
font:theSystemFont
stencil:undefined
stationary:false
```

The TextButton class inherits the stencil keyword from StencilButton. The stencil keyword is not used by the TextButton class. If you assign a Stencil object to the stencil keyword or instance variable, you will not get the effect you want, so do not use the stencil keyword in TextButton. If you want the button to use a stencil instead of a string or other text object, use the StencilButton class.

You should not provide a value for boundary. If you do not provide a value for boundary, then the boundary of the new TextButton object is calculated automatically as the smallest rectangle that encloses the text stencil and the button frame.

Instance Variables

Inherited from Actuator:

enabled	pressed	toggledOn
menu		

Inherited from Presenter:

presentedBy	subPresenters	target

Inherited from TwoDPresenter:

bBox	height	transform
boundary	IsImplicitlyDirect	width
clock	isTransparent	window
compositor	isVisible	x
direct	needsTickle	y
eventInterests	position	z
globalBoundary	stationary	
globalTransform	target	

Inherited from GenericButton:

activateAction	pressAction	releaseAction
authorData		

Inherited from StencilButton:

border	recessedFrame	stencil
buttonFrame		

The following instance variables are defined in TextButton:

font

self.font (read-write) Font

Specifies the font in which to render the text of the TextButton object *self*.

text

self.text (read-write) String

Specifies the string of characters to be displayed on the TextButton object *self*.

Instance Methods

Inherited from Actuator:

activate	press	toggleOff
multiActivate	release	toggleOn

Inherited from TwoDPresenter:

adjustClockMaster	inside	show
createInterestList	localToSurface	surfaceToLocal
draw	notifyChanged	tickle
getBoundaryInParent	recalcRegion	
hide	refresh	

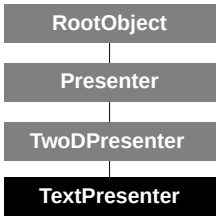
Inherited from GenericButton:

activate	press	release
multiActivate		

Inherited from StencilButton:

draw	recalcRegion

TextPresenter



Class type: Core class (concrete)
Resides in: ScriptX and KMP executables
Inherits from: `TwoDPresenter`
Component: Text and Fonts

The `TextPresenter` class provides simple presenter behavior for text. A text presenter uses an instance of the `String` class (a `Text`, `String`, or `StringConstant` object) to determine the actual text it is presenting (the presenter's *target*). Generally you use an instance of the class `Text` as the target string for a text presenter because that allows you to change the attributes of the text. If the text to be displayed will have uniform attributes (all the characters have the same appearance), for example, to display a title or a label for some other object, you can use an instance of `String` or `StringConstant`.

The `TextPresenter` class is intended for static display, as opposed to the `TextEdit` class, a subclass of `TextPresenter` which is designed for making changes to the text. Nevertheless, `TextPresenter` includes much of the behavior for displaying selections, editing its target text and handling changes in attributes. However, access to those actions is provided only through the use of methods and instance variables on the `TextPresenter` and `Text` classes, and not through mouse events or keyboard input. `TextEdit` includes behavior for direct text selection using mouse events and input of text from the keyboard, as well as cut/copy/paste actions.

The `selection` instance variable in the `Text` class determines the location of the insertion point or highlighted text. To highlight a range of text from code, set the `selection` instance variable of the text (the text presenter's `target`), and set the brush values in the `selectionBackground` and `selectionForeground` instance variables of the text presenter.

A `TextPresenter` object has a set of *attributes*, which determines the overall appearance of its target text: the font, size, color, and so on. These attributes serve as default attributes for the presenter's target. `String` and `StringConstant` objects always use the attributes of the presenter which presents them. A `Text` object, however, is capable of defining attributes for some or all of its characters , and if that is done, the `Text` attributes override the attributes of the `TextPresenter`. For example, if the text presenter's `@size` attribute is `12`, then the `@size` attribute of its target text will be `12` unless the target is a `Text` object which has had its `@size` attribute set to something other than `12`. It is interesting to note that `TextPresenter` attributes are never applied to `TextPresenter` objects; they are only used as the default attributes for the target text which is being presented. This functionality is reflected in the names of the methods for getting and setting `TextPresenter` attributes (`setDefaultAttr`, `getDefaultAttr`, and `getDefaultAttrs`).

Attributes are stored as a keyed linked list, where the key is the name of an attribute, such as `@font` or `@size`, and the value is the value of that attribute. Table 11 summarizes the text attributes which can be set for `TextPresenter` objects. The only difference between attributes for `Text` and `TextPresenter` is that `TextPresenter` objects do not recognize `@action` attributes. For more information, see the "Text and Fonts" chapter of the *ScriptX Components Guide*.

Table 11: TextPresenter Attributes

Attribute	Possible values	Values Represent
`@brush`	`Brush` object	The color, pattern, and so on of the text.
`@font`	`Font` object	The font family
`@size`	`Number` object	The point size of the text
`@weight`	`@extraLight`, `@light`, `@regular`, `@medium`, `@demiBold`, `@bold`, `@extraBold`, `@heavy`	The "boldness" of the text (as defined by variations in the font family).
`@width`	`@condensed`, `@normal`, `@expanded`	The horizontal spacing of the text (as defined by variations in the font family).
`@style`	`@roman`, `@italic`, `@oblique`	The obliqueness of the text (as defined by variations in the font family).
`@underline`	`0`, `1`	0 represents no underlining, 1 represents underlining (as defined by the font). The `@underline` attribute in future versions of ScriptX will use numeric values to indicate the space between the under line and the text.
`@leading`	`Number` object	The spacing, in points, between lines of text.
`@paraLeading`	`Number` object	The spacing, in points, between paragraphs.
`@firstLineLeading`	`Number` object	The spacing, in points, between this line of text and the top of the text's presenter
`@alignment`	`@flush`, `@flushLeft`, `@flushToEnd`, `@flushRight`, `@fill`, `@center`, `@tty`	The alignment of the text within the presenter.
`@paraIndent`	`Number` object	Number of points to indent the first line of a paragraph.
`@indent`	`Number` object	Number of points to indent the left margin of text. This applies only to lines other than the first line of a paragraph.
`@indentFromEnd`	`Number` object	Number of points to indent the right margin of the text

Creating and Initializing a New Instance

The following script is an example of how to create a new instance of the `TextPresenter` class:

```
myMessage := new TextPresenter \
             boundary:(new Rect x2:300 y2:200) \
             target:"Hello, World!" \
             fill:whiteBrush \
             stroke:(new Brush color:redColor)
```

The variable `myMessage` contains an initialized instance of `TextPresenter` with the specified rectangular boundary. This object presents the text, "Hello, World!" in red on a white background. Its attributes are set to the default values listed below. The `new` method uses the keywords defined in `init`.

init

`init` *self* `boundary:`*stencil* `target:`*string* [`fill:`*brush*] [`stroke:`*brush*] ⇨ *(none)*

self	`TextPresenter` object
`boundary:`	`Rect` object
`target:`	`String` object (typically a `Text` object, but it may also be a `String` or `StringConstant` object)
`fill:`	`Brush` object
`stroke:`	`Brush` object

Initializes the `TextPresenter` object *self*, applying the arguments as follows: `boundary` is the stencil in which the `TextPresenter` object is rendered (you can only specify rectangular boundaries); `target` is the `Text`, `String`, or `StringConstant` object that will be drawn; `fill` is the brush that defines the color and pattern of the text presenter's background, if any; `stroke` is the brush that defines the appearance of the text presenter's boundary, if any. Do not call `init` directly on an instance—it is automatically called by the `new` method.

If you omit an optional keyword, its default value is used. The defaults are:

```
fill:undefined
stroke:undefined
stationary:false
```

The `attributes` instance variable in every new instance of `TextPresenter` is automatically initialized with default values which are listed here and described in further detail in the *ScriptX Components Guide*.

Attribute	Default Value
`@font`	The default font as defined by the system. In the current release, the default fonts are Helvetica on the Macintosh and Arial on Windows.
`@size`	`12`
`@weight`	`@regular`
`@width`	`@normal`
`@style`	`@roman`
`@leading`	`13`
`@paraLeading`	`empty`
`@firstLineLeading`	`empty`
`@alignment`	`@fill`
`@indent`	`0`
`@indentFromEnd`	`0`
`@paraIndent`	`0`
`@brush`	`blackBrush`
`@underline`	`0`

Instance Variables

Inherited from `Presenter`:

`presentedBy`	`subPresenters`	`target`

Inherited from `TwoDPresenter`:

`bBox`	`height`	`transform`
`boundary`	`IsImplicitlyDirect`	`width`
`clock`	`isTransparent`	`window`
`compositor`	`isVisible`	`x`
`direct`	`needsTickle`	`y`
`eventInterests`	`position`	`z`
`globalBoundary`	`stationary`	
`globalTransform`	`target`	

The following instance variables are defined in `TextPresenter`:

attributes

self`.attributes` (read-write) `KeyedLinkedList`

Specifies a list of key-value pairs that define the overall attributes of the target text displayed by the `TextPresenter` *self*. Attributes of individual characters or ranges of characters (as defined by a `Text` object in *self*`.target`) override these attributes. A list of the attributes is contained in Table 11 and described in further detail in the *ScriptX Components Guide*.

Each instance of `TextPresenter` is automatically assigned default values for `attributes` at initialization. These default values are listed above in the table included in the description of the `init` method for `TextPresenter`.

Note – The instance variable `attributes` is included for informational purposes only; it is not accessible to the user.

cursor

self`.cursor` (read-write) `Stencil`

Specifies the `Stencil` object that is used to render the outline of the cursor for the `TextPresenter` *self*. The origin of the cursor's stencil has its `x` midway between the two characters and its `y` at the baseline of the line. If `cursor` is `undefined`, no cursor is displayed.

Each instance of `TextPresenter` is initialized with the default value for `cursor`. The default is `undefined`.

cursorBrush

self`.cursorBrush` (read-write) `Brush`

Specifies the `Brush` object that is used to render the `cursor` object of the text presenter *self*. .

Each instance of `TextPresenter` is initialized with the default value for `cursorBrush`. The default is `blackBrush`.

enabled

self.enabled (read-write) Boolean

Allows this TextPresenter object to accept mouse down events that activate hypertext-like actions. If enabled is true, mouse clicks on characters within in the target text (which must be a Text object) activate the function defined by an @action attribute in that Text object. The *ScriptX Components Guide* describes the interaction between mouse events and actions on Text and TextPresenter objects.

fill

self.fill (read-write) Brush

Specifies the brush, if any, used to fill the text presenter object (that is, the background of the text being displayed). If fill is undefined, the text presenter is transparent.

inset

self.inset (read-write) Point

Specifies a Point object representing the space between the boundary of the text presenter and the margins of the text. The x value of the Point object (*self*.inset.x) specifies the left and right margins, and the y value (*self*.inset.y) specifies the top and bottom margins.

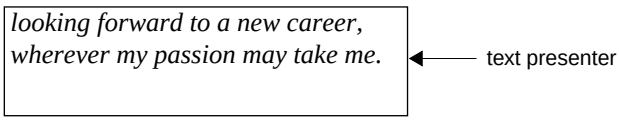
Note that if the x and y of the inset are 0, the stroke of the text presenter may obscure the edges of the text.

offset

self.offset (read-write) Integer

Specifies the first visible character in the target text of the text presenter *self*, displayed at the upper-left corner of the text presenter. The offset value is the cursor position in the text of the character that is first rendered.

For example, in the following figure, the full text starts with "I am looking forward to a new..." Setting the offset to 5 causes the first 5 characters to be visibly truncated at the start of the text presenter, as shown.



Setting this value has the effect of causing the text to shift in the text presenter, visibly (but not internally) truncating all characters ahead of the specified offset. This positions the character whose cursor position in the text is given by offset to appear at the upper-left corner of the text presenter.

selectionBackground

self.selectionBackground (read-write) Brush

Specifies the Brush object that is used to render the background (highlight) of any selected text in the text presenter. Selected text is specified by mouse events (when the user drags the mouse across text) or by the text presenter's target's selection instance variable (defined in Text). If selectionBackground is undefined, the selection is transparent.

Each instance of TextPresenter is initialized with the default value for selectionBackground. The default is blackBrush.

selectionForeground

self.selectionForeground (read-write) Brush

Specifies the Brush object that is used to render the foreground of any selected text in the text presenter; that is, the text itself. Text selections are specified by mouse events or by the selection instance variable on the Text object serving as the target.

Each instance of TextPresenter is initialized with the default value for selectionForeground. The default is a Brush object with color: redColor.

stroke

self.stroke (read-write) Brush

Specifies the brush, if any, used to stroke the boundary of the text presenter. If stroke is undefined, the boundary is transparent.

The stroke of the text presenter may overlap the edges of the text, particularly in cases where the *self*.stroke.linewidth is greater than 1. Use the inset instance variable to indent the text from the boundary.

Instance Methods

Inherited from TwoDPresenter:

adjustClockMaster	hide	refresh
createInterestList	inside	show
draw	localToSurface	surfaceToLocal
getBoundaryInParent	notifyChangeds	tickle

The following instance methods are defined in TextPresenter:

calculate

calculate *self constraint value* ⇨ Integer

self	TextPresenter object
constraint	NameClass object: @width
value	Number object

Given *constraint*, which is a direction, and a number of pixels, calculate returns the number of pixels required in the other direction to display *self*'s target in its entirety. The value returned by calculate is dependent on the length and attributes of the text this text presenter is presenting.

The *constraint* argument can currently be only @width and indicates the direction of the *value* argument. The *value* argument indicates the number of pixels available in that direction. So, for example, the following call to calculate returns the height in pixels required to present the text in the TextPresenter object myTP:

```
myTP.height := calculate myTP @width (myTP.width)
```

Note – In the current release, only the @width constraint is available. Any other value given for the *constraint* argument returns undefined.

Note – In the current release, the last line of any text which is used as the target of a text presenter should end with a carriage return. The `calculate` method works only on complete paragraphs, which by definition end with a carriage return. Incomplete paragraphs (paragraphs which have no terminating carriage return) are displayed and are valid for use with `getPointForOffset` and `getOffsetForXY`; however, `calculate` does not work with incomplete paragraphs. If you try to use `calculate` on a text presenter whose target text does not end with a carriage return, you will get a return value of 0.

The following code demonstrates correct termination of text which is used as the target of a text presenter:

```
"This is a<CR>
great piece of<CR>
software."<CR> -- this <CR> must be there

"This is a great piece of software."<CR> -- this <CR> must be there
```

copySelection

`copySelection` *self* ⇨ `Text`

Copies the current selection for this text presenter's target to the system clipboard (as specified by the global variable `theClipboard`), and returns that selection. Note that this is a specialization of the default `copy` method, which simply returns a copy of *self*.

getDefaultAttr

`getDefaultAttr` *self key* ⇨ *(object)*

self	`TextPresenter` object
key	`NameClass` object

Returns the value of the attribute specified by *key* in *self*, which will be a `NameClass` object or a `Number` object. The following code will return the type size to which the `TextPresenter` object `tp` is set:

```
global myNum := getDefaultAttr tp @size
```

getDefaultAttrs

`getDefaultAttrs` *self* ⇨ `KeyedLinkedList`

Returns a keyed linked list representing all the (default) attributes of the `TextPresenter` object *self*. The following code will return all the attributes to which the `TextPresenter` object `tp` is set:

```
global myAttributes := getDefaultAttrs tp
```

getLastVisibleOffset

`getLastVisibleOffset` *self* ⇨ `Integer`

Returns the offset of the last character displayed in the `TextPresenter` object's `bitmapSurface`. This offset can then be passed to the method `getPointForOffset` to determine the character's `x` and `y` coordinate values. The method is used to determine the amount of text visible to the user, which is needed for operations such as printing.

getOffsetForXY

`getOffsetForXY` *self x y* ⇨ `Integer`

self	`TextPresenter` object
x	`Number` object
y	`Number` object

Returns a positive integer that represents a cursor position in the target text of the text presenter *self*. This cursor position corresponds to the insertion point nearest to the coordinate position *x*, *y* in the local coordinates of the text presenter *self*. Positions are measured from the top left corner of the text presenter.

getPointForOffset

`getPointForOffset` *self offset* ⇨ `Point`

self	`TextPresenter` object
offset	`Number` object

Returns a `Point` object representing the `x` and `y` coordinates within a text presenter of the given cursor position *offset* in the text. The value of `getPointForOffset` varies based on the current offset of the text itself (as changing the offset shifts the text within the text presenter and changes the position). Positions are measured from the top left corner of the text presenter.

processMouseDown

`processMouseDown` *self interest event* ⇨ `Boolean`

self	`TextPresenter` object
interest	`MouseDownEvent` object
event	`MouseDownEvent` object

Tests whether the location of the mouse-down *event* applied to the text presenter *self* is within a range of text that has an `@action` attribute assigned to it. If so, it activates the function specified by that `@action` attribute. The `@action` attribute is specific to `Text` objects and is not an attribute on `TextPresenter` objects. If the action is called, it returns `true`; otherwise it returns `false`.

The function specified by the `@action` attribute on the text presenter's target text must have been defined to take three arguments:

`myFunc` *textPresenter range offset*

These arguments are this text presenter, the range of characters within the `Text` object that had been defined to have this `@action` attribute, and the number of cursor positions specified by the location of the mouse down event from the beginning of the range (rather than from the beginning of the text). The *ScriptX Components Guide* describes the interaction between mouse events and actions on `Text` and `TextPresenter` objects.

The `processMouseDown` method can also be overridden in subclasses to handle other mouse behaviors. For example, the `TextEdit` class uses `processMouseDown` to handle insertions and selections within the text.

setContext

`setContext` *self surface clip* ⇨ *self*

self	`TextPresenter` object
surface	`Surface` object
clip	`Region` object

Provides the `TextPresenter` *self* with a surface and clip (collectively called the context) for laying out the target text. This information must be explicitly specified when you call the methods `calculate`, `getPointForOffset`, or `getOffsetForXY` and the context is not readily available to the text presenter.

In the following cases, a `TextPresenter` object has the context available, and `setContext` need *not* be used:

a. The text has already been displayed on a surface. In this case, the `TextPresenter` object caches the surface and clip information.

b. The text has not yet been displayed, but the `TextPresenter` object is part of a presentation hierarch attached to a `Window` object. In this case, the presenter can find the display surface and clip associated with the compositor.

In the following cases, the `TextPresenter` object does not have access to the context, and `setContext` *should* be called:

a. The text will be (but has not yet been) drawn to an offscreen bitmap surface.

b. The text will be (but has not yet been) drawn to a printer or printer surface. In both of these cases, there is no compositor available from which to obtain the context.

setDefaultAttr

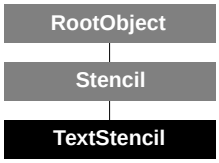
`setDefaultAttr` *self key newValue* ⇨ *(object)*

self	`TextPresenter` object
key	`NameClass` object
newValue	`NameClass` or `Number` object

Sets *self*'s default attribute, represented by *key*, to *newValue* and returns the new value. The following code sets the `@style` attribute of the `TextPresenter` object `tp` to `@italic`:

```
setDefaultAttr tp @style @italic
```

TextStencil



Class type: Core class (concrete)
Resides in: ScriptX and KMP executables
Inherits from: `Stencil`
Component: 2D Graphics

The `TextStencil` class defines an object whose image area is made up of a string of characters rendered in a specific font. This class is provided for completeness of the graphics layer, by supporting text drawing capabilities provided by several graphic user interface systems. Instances of `TextStencil` can be used to put labels and callouts on illustrations and user interface controls.

Note that `TextStencil` objects are not presenters—to display text as an image, create an instance of `TwoDShape` using a `TextStencil` object as the *stencil* argument.

Just as with other kinds of stencils, text stencils can be translated, scaled and rotated by using the `transform` method in conjunction with `TwoDMatrix` objects. See the discussion of the `TwoDMatrix` class in this book, or the chapter on Two D Graphics in the *ScriptX Components Guide* for details.

Creating and Initializing a New Instance

The following script is an example of how to create a new instance of the `TextStencil` class:

```
myText := new TextStencil \
    font:(new platformFont name:"Arial" \
    macintoshName:"Helvetica" \
    string:"Hello, world!"
```

The variable `myText` contains the initialized `TextStencil` instance. The `new` method uses the keywords defined in `init`.

init

`init` *self* `font:`*font* `string:`*string* ⇨ *(none)*

self	`TextStencil` object
`font:`	`Font` object
`size`:	Integer object indicating the point size of the text.
`string:`	`String` or `Text` object

Initializes the `TextStencil` object *self*, applying the arguments as follows: `font` sets the font instance to use, `size` indicates the size of the font, and `string` sets the string to display with the text stencil. If the string is a `Text` object, it creates a text stencil with the font and size of the `Text` object. If the `font` and `size` keywords are explicitly stated, however, they override the previous font and size of the text. Note that unlike a `Text` object, a `TextStencil` cannot change the values of its attributes for different characters.

Do not call `init` directly on an instance—it is automatically called by the `new` method.

Instance Variables

Inherited from `Stencil`:

`bBox`

The following instance variables are defined in `TextStencil`:

font

self`.font` (read-write) `Font`

Specifies the font in which to render the text stencil *self*.

size

self`.size` (read-write) `Font`

Specifies the point size in which to render the font of the text stencil *self*.

string

self`.string` (read-write) `String`

Specifies the string of characters to be rendered by the text stencil *self*.

Instance Methods

Inherited from `Stencil`:

`inside`	`onBoundary`	`transform`
`intersect`	`subtract`	`union`

The following instance methods are defined in `TextStencil`:

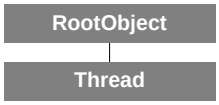
moveToZero

`moveToZero` *self* ⇨ (*object*)

self `Text` object

Brings the beginning of the baseline to (0, 0) which is the default position for `TextStencil`.

Thread



Class type: Core class (abstract, sealed)
Resides in: ScriptX and KMP executables
Inherits from: `RootObject`
Component: Threads

The `Thread` class represents separate processes that run on their own. A thread represents a true flow of control. Each thread has its own stack, and each thread runs a control function that is called when the thread begins running. This function may be a simple expression or a complex sequence of expressions, and is often in the form of an infinite loop. See the discussion of thread functions in the "Threads" chapter of the *ScriptX Components Guide*.

Scripts that operate on threads frequently refer to the global variable `theRunningThread`. A title begins running in a thread, which runs its core function. This thread may spawn new threads, which in turn may spin off other threads. Although threads can create other threads, they do not actually own these threads as is true in many other thread systems. A thread represents an independent flow of control. The global variable `theRunningThread` stores the thread that is currently in control.

ScriptX provides a thread operator, the ampersand (`&`). This operator gives developers a convenient syntax for creating a separate thread of execution

```
function printList n -> for x =: 1 to n do print x
-- explicitly create a new thread
myThread := new Thread func:printList arg:10000
-- create a thread using the thread operator
printList 10000 & -- the '&' starts up a separate thread
```

Each of these techniques creates a separate thread that starts in the active state, runs at normal priority, is fully preemptible, and runs with a stack of the default size for the given operating system. The second form is useful if the default values for the keywords are acceptable.

Creating and Initializing a New Instance

The global function `callInThread` provides a simple technique for creating a `Thread` object, with default behavior. Use the `new` method when it is necessary to set initial values for many of the parameters.

The following script creates a new instance of the `Thread` class:

```
myThread := new Thread\
              func:myFunc \
              arg:myInt \
              stackSize:128 \
              priority:@user \
              preemptibility:@fullyPreemptible \
              startInactive:true \
              label:debuggerLabel
```

The variable `myThread` contains the initialized thread. This thread starts up in the inactive status, with a stack that contains 128 bensons. Other keyword arguments, which can be entered in any order, correspond with instance variables defined on the class `Thread`. The `new` method uses keywords defined in `init`.

init

`init` *self* `func:`*function* [`arg:`*object*] [`stackSize:`*integer*]
[`priority:`*name*] [`preemptibility:`*name*] [`startInactive:`*boolean*]
[`label:`*object*] ⇨ *(none)*

self	`Thread` object
`func:`	`Function` object
`arg:`	An object that is the argument of the thread's function
`stackSize:`	`Integer` object representing size of stack in bensons
`priority:`	`NameClass` object representing priority with the scheduler
`preemptibility:`	`NameClass` object
`startInactive:`	`Boolean` object
`label:`	Any object

Initializes the `Thread` object *self*, applying the arguments as follows: The keyword `func` specifies the function to run in the thread, and `arg` specifies that function's one argument. The function must accept one argument; if you do not want to pass in a value, then pass this argument as `undefined`. The `stackSize` keyword represents the size in bensons of the thread's stack. It may be passed as `undefined` to use the default stack size. The `priority` keyword specifies a `NameClass` object, either `@user` or `@system`, that will be used to set the priority of the new thread. The `preemptibility` keyword specifies a `NameClass` object that indicates an initial value for the instance variable of the same name. The `startInactive` keyword specifies a Boolean value that is used to set the initial value of the thread's `status`. A thread's `label` keyword can specify any object. This label is displayed when you print, which is useful for debugging. Do not call `init` directly on an instance—it is automatically called by the `new` method.

If you omit an optional keyword, its default value is used. The defaults are:

```
arg:undefined
stackSize:(not specified, but greater than 100)
priority:@user
preemptibility:@fullyPreemptible
startInactive:false
label:undefined
```

Stack sizes are specified in bensons, a platform-independent unit that represents one bytecode frame. Each benson allows for one level of recursion on a stack. No default stack size is currently specified—developers can assume that the default size will always provide at least 100 bensons. If a negative number is supplied, it is interpreted as the actual number of bytes to allocate for the stack.

Instance Variables

arg

self`.arg` (read-write) *(object)*

Specifies the argument to the function specified by the `func` instance variable for the thread *self*. Its value may be any object, or it may be `undefined`.

func

self`.func` (read-write) *(function)*

Specifies the control function that is called when the thread *self* runs. This function is always called with the `arg` instance variable as its argument.

Although any global function, anonymous function, or method can be assigned to `func`, there are differences in how different classes of functions are dispatched. For information on functions and function dispatch, see the "Object System Kernel" chapter of the *ScriptX Components Guide*.

label

self`.label` (read-write) *(object)*

Specifies a label for the thread *self*, which can be any object. This label is displayed when you print the results of many thread operations. It is useful primarily for debugging.

pendingAction

self`.pendingAction` (read-only) `NameClass`

Specifies an action that is pending on the thread *self*. A pending action, such as killing the thread, has been deferred because the thread is protected. (See the `protection` instance variable on `Thread`.) The following are possible values that correspond to those operations that are deferred on a protected thread.

`@killed`	An attempt to kill the thread has been deferred
`@restart`	An attempt to restart the thread has been deferred
`@inactive`	An attempt to deactivate the thread has been deferred
`@done`	An attempt to return from the thread has been deferred

Use `pendingAction` only in ScriptX code that is used for development and debugging. In future releases of the Kaleida Media Player, it is likely to change.

priority

self`.priority` (read-write) `NameClass`

Specifies the priority of the thread *self*. There are two priority levels, `@user` and `@system`. The high priority level is meant to be used only for threads that perform critical tasks, and that block immediately or run quickly to completion. For more information, see the section on "Priority" in the "Threads" chapter of the *ScriptX Components Guide*.

protection

self`.protection` (read-only) `Integer`

Specifies the protection level of the thread *self*. Possible values are the non-negative integers, where a zero value means the thread is not protected.

To set the `protection` instance variable, use `threadProtect` and `threadUnprotect`. Calls to `threadProtect` and `threadUnprotect` must be balanced; call `threadProtect` twice, and you must subsequently call `threadUnprotect` twice to remove the thread's protection.

Use `protection` only in ScriptX code that is used for development and debugging. In future releases of the Kaleida Media Player, it is likely to change.

result

self.`result` (read-only) *(object)*

Specifies the result from the execution of the thread *self*. This variable is set when the thread exits or returns. A thread may return by calling the ScriptX expression `return` inside its control function, or it may use the global function `threadExit` or the method `threadReturn`.

Normally, a thread will return only while it is executing, but the method `threadReturn` can be called on a thread that is not running. If a thread has never returned a value, the value of `result` will be `undefined`. If a thread is restarted, the old value persists until it is explicitly changed when the thread exits or returns again.

status

self.`status` (read-only) `NameClass`

Specifies the status of the thread *self*, which can take on eight possible values. The status of a thread is not set directly by script. The scheduler itself is responsible for running a thread that is active. Otherwise, the value of `status` for a thread changes while the title is running, reflecting the logical flow of control in the program. Some methods, such as `threadActivate`, directly change the status of a thread, but only from one given state to another. For a discussion of `status`, see the "Threads" chapter of the *ScriptX Components Guide*.

The following are the eight possible values for `status`:

`runnning`	The thread is executing (it is the thread currently stored in `theRunningThread`)
`@active`	The thread is runnable, but not currently running; it will begin running when it comes its turn, as determined by its priority
`@inactive`	The thread is not runnable, but it isn't waiting on anything; the only way to start it back up is with a call to `threadActivate`
`@waiting`	The thread is not runnable; it is waiting on a gate or pipe (see the `waitingOn` instance variable)
`@done`	The thread is done executing; it may have a return value in the `result` instance variable
`@killed`	The thread was killed
`@restart`	The thread is in the process of restarting (a momentary status that is rarely observed in practice)
`@starting`	The thread is in the process of starting for the first time (a momentary status that is rarely observed in practice)

Use `status` only in ScriptX code that is used for development and debugging. In future releases of the Kaleida Media Player, it is likely to change.

waitingOn

self.`waitingOn` (read-only) `Gate` or `Pipe`

Specifies the gate or pipe that the thread *self* is currently waiting on. If the thread is not waiting on anything, the value of `waitingOn` is `undefined`.

Use `waitingOn` only in ScriptX code that is used for development and debugging. In future releases of the Kaleida Media Player, it is likely to change.

Instance Methods

threadActivate

`threadActivate` *self* ⇨ *(none)*

Makes the thread *self* active. This method does nothing if the thread is already active or running. If the thread is waiting on a gate, `threadActivate` is not allowed and will report the `threadProhibited` exception.

threadDeactivate

`threadDeactivate` *self* ⇨ *(none)*

Makes the thread *self* inactive. This method does nothing if the thread is already inactive. If the thread is waiting on a gate, `threadDeactivate` is not allowed and will report the `threadProhibited` exception.

threadInterrupt

`threadInterrupt` *self exception arg* ⇨ *(none)*

self	`Thread` object
exception	`Exception` object
arg	Any object

Interrupts the thread *self* with the *exception* and argument *arg*. When the thread next gets a chance to run, it will immediately report the given exception with the given argument. The method `threadInterrupt` can cause serious problems and should be used with great care. A script cannot call `threadInterrupt` on another thread; doing so will report the `threadProhibited` exception.

threadKill

`threadKill` *self* ⇨*(none)*

Makes the thread *self* terminate and discard its stack.

This method should only be called on the running thread. Killing a thread that has not returned may leave data in an inconsistent state. Other techniques for managing and synchronizing threads have far less destructive potential.

threadProtect

`threadProtect` *self* ⇨ *(none)*

Increases the protection on the given thread *self*, which should be currently running. When a thread is protected, the following operations are deferred: `threadReturn`, `threadKill`, `threadRestart`, `threadInterrupt`, and `threadDeactivate`. Changing the value of `protection` for a thread that is not currently running is permitted, although it is a potential source of deadlock and conflict. (See also `threadUnprotect`.)

threadRestart

`threadRestart` *self* ⇨ *(none)*

Makes the thread *self* restart. This throws away its stack and starts its function again from the beginning. A thread that has been restarted gives up its time slice. It becomes active, and will run the next time its turn comes with the scheduler. Restarting does not throw away the old return value, which remains in place until the thread returns a new value.

This method should be called only on the currently running thread. If the thread *self* is not currently running, calling `threadRestart` is a potential source of deadlock or conflict, but it is permitted. This method is prohibited on the main thread in a title.

threadReturn

`threadReturn` *self result* ⇨ *(none)*

self	`Thread` object
result	Any object

Makes the thread *self* return with the value specified in *result*. If the thread *self* is not in fact the currently running thread, using `threadReturn` is a potential source of deadlock or conflict, but it is permitted. Generally, the best way to return from a thread is with the global function `threadExit()`, which is equivalent to this anonymous function:

```
(x -> threadReturn(theRunningThread, x)).
```

threadUnprotect

`threadUnprotect` *self* ⇨ *(none)*

Decreases the protection on the thread *self*. If the thread becomes unprotected and there is a deferred action, the action is executed. Changing the value of `protection` for a thread that is not currently running is permitted, although it is a potential source of deadlock and conflict. (See also `threadProtect`.)

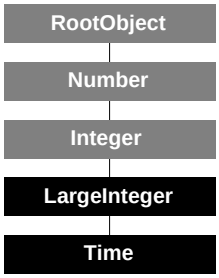
threadWait

`threadWait` *self gate* ⇨ *(none)*

self	`Thread` object
gate	`Gate` object

Makes the thread *self* wait on *gate*. If the thread is not currently running, calling this method has a high potential for deadlock and conflict, but it is permitted. Generally, the best way to wait on a gate is with the `acquire` method (defined by `Gate`).

Time



Class type: Core class (concrete, sealed)
Resides in: ScriptX and KMP executables
Inherits from: `LargeInteger`
Component: Numerics

The `Time` class represents a fixed time; an instance of time can be set to any value of hours, minutes, seconds and ticks. Given an instance of `Time`, its value does not change with the passing of time. However, the `time` instance variable of a `Clock` contains a `Time` instance that is updated regularly to reflect the clock's current time.

Instances of `Time` are stored internally as integer numbers representing ticks but are displayed as hours, minutes, seconds and ticks. The `scale` instance variable sets the number of ticks in a second. Here is how a `Time` instance might display:

Format: *hours*`:`*minutes*`:`*seconds*`:`*ticks*
Example: `1:30:15:25`

Any numeric operations that are performed on an instance of `Time` are performed on the underlying integer number of ticks. For example, you can perform integer addition and subtraction on a time object, and you can multiply or divide time by integers.

When you create a new instance of `Time`, you can optionally specify a time value for it with the `timeStamp` keyword, and specify a scale:

```
t1 := new Time timeStamp:162475 scale:30 -- returns 1:30:15:25
```

To convert an instance of `Time` to its underlying integer, coerce it to `LargeInteger`:

```
t1 as LargeInteger      -- returns 162475 ticks
```

You can coerce any integer to time, in seconds, by using the `as` construct. For example, "`2 as Time`" creates a `Time` object `0:0:2:0` with a time stamp of 2 seconds.

Note that two-operand arithmetic operations on `Time` objects will return a `Time` object with a scale equal to that of the first operand. For example, given `Time` objects `a` and `b`, the operation `a + b` will return a result in the scale of `a`. When you add times together, the result does not roll over at 12 hours or 24 hours—it keeps going up past 24 hours. `Time` objects cannot be multiplied or divided by each other.

For example, let's suppose we have two `Time` objects, such as `t1` and `t2`, that are initialized as follows:

```
t1 := new Time timeStamp:30 scale:30 -- 0:0:1:0 as Time
t2 := new Time timeStamp:1 scale:1 -- 0:0:1:0 as Time
```

Adding times is a commutative operation; `t1 + t2` is the same as `t2 + t1`. Therefore, the following expressions imply that `t3 = t4`.

```
t3 := t1 + t2 -- 0:0:2:0 as Time
```

```
t4 := t2 + t1 -- 0:0:2:0 as Time
```

But, the scales of `t3` and `t4` are not the same. The result of an operation, such as addition, takes the scale of the left-hand operand.

```
t3.scale     -- returns 30 (the scale of t1)
t4.scale     -- returns 1 (the scale of t2)
```

Creating and Initializing a New Instance

The following script is an example of how to create a new instance of the `Time` class:

```
t1 := new Time \
       timeStamp:7200 \
       scale:2
```

The variable `t1` contains an initialized instance of `Time`, which displays as 1:0:0:0, since 7200 ticks at a scale of 2 is 1 hour, calculated from 7200/(60*60*2). The `new` method uses the keywords defined in `init`.

init

`init` *self* [`timeStamp:`*integer*] [`scale:`*integer*] ⇨ *(none)*

self	`Time` object
`timeStamp:`	`Integer` indicating the time in ticks.
`scale:`	`Integer` that indicates the scale of the time.

Initializes the `Time` object *self*, converting its `timeStamp` value to hours, minutes, seconds, and ticks, and assigning those values to the corresponding instance variables. Also assigns the value of `scale` to its corresponding instance variable. Do not call `init` directly on an instance—it is automatically called by the `new` method.

If you omit an optional keyword, its default value is used. The defaults are:

```
timeStamp:0
scale:1
```

Instance Variables

hours

self`.hours` (read-write) `Integer`

Specifies the number of hours in the time *self*.

minutes

self`.minutes` (read-write) `Integer`

Specifies the number of minutes in the time *self*.

scale

self`.scale` (read-write) `Integer`

Specifies the number of ticks in a second for the time *self*.

seconds

self`.seconds` (read-write) `Integer`

Specifies the number of seconds in the time *self*.

ticks

self.ticks (read-write) Integer

Specifies the number of ticks in the time *self*. There are scale number of ticks in a second.

Instance Methods

Inherited from Number:

abs	floor	radToDeg
acos	frac	random
asin	inverse	rem
atan	ln	round
atan2	log	sin
ceiling	max	sinh
coerce	min	sqrt
cos	mod	tan
cosh	morph	tanh
degTorad	negate	trunc
exp	power	

Inherited from Integer:

length	logicalOp	lshift
logicalAnd	logicalOr	rshift
logicalNot	logicalXor	

The following instance methods are defined in Time:

addHours

addHours *self hours* ⇨ Time

self	Time object
hours	Integer object

Adds *hours* to the given time *self*, and returns a new Time object representing the result. Use positive or negative values for *hours* (using the + or – operator) to add or subtract the corresponding number of hours.

addMinutes

addMinutes *self minutes* ⇨ Time

self	Time object
minutes	Integer object

Adds *minutes* to the time *self*, and returns a new Time object representing the result. Use positive or negative values for *minutes* (using the + or – operator) to add or subtract the corresponding number of minutes.

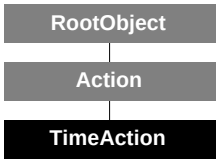
addSeconds

addSeconds *self seconds* ⇨ Time

self	Time object
seconds	Integer object

Adds *seconds* to the time *self*, and returns a new Time object representing the result. Use positive or negative values for *seconds* (using the + or – operator) to add or subtract the corresponding number of seconds.

TimeAction



Class type: Core class (concrete)
Resides in: ScriptX and KMP executables
Inherits from: `Action`
Component: Animation

`TimeAction` class represents an action that makes its action list player go to a specified time. To be triggered, an instance of `TimeAction` needs to be added to the action list of an action list player, then the player needs to be played.

Creating and Initializing a New Instance

The following script is an example of how to create a new instance of the `TimeAction` class:

```
myAction := new TimeAction \
              destTime:20 \
              time:200
```

This makes the action list player jump back to the time 20 ticks when it reaches the time 200 ticks. The `new` method uses the keywords defined in `init`.

init

`init` *self* [`destTime:`*integer*] [`time:`*integer*] [`targetNum:`*integer*] ⇨ *(none)*

self	`TimeAction` object
`destTime:`	`Integer` object representing the time the function is to jump to

Superclass `Action` uses the following keywords:

`time:`	`Integer` object representing the time in ticks to trigger the time jump
`targetNum:`	`Integer` object indicating which object in the target list of the player to affect

Initializes the `TimeAction` object *self*, applying the values supplied with the keywords to the instance variables of the same name.

For the most part, the `targetNum` keyword is ignored by `TimeAction`, because a time action affects the action list player and not just an object on its target list.

However, `targetNum` has one use—when you supply a value for `targetNum`, and if that channel is muted (using `setMuteChannel` defined in `ActionListPlayer`), then the time action will not occur. See `setMuteChannel` for more details. Do not call `init` directly on an instance—it is automatically called by the `new` method.

If you omit an optional keyword, its default is used. The defaults are:

```
destTime:undefined
targetNum:0
time:0
```

Instance Variables

Inherited from `Action`:

`authorData` `targetNum` `time`
`playOnly`

The following instance variable is defined in `TimeAction`:

destTime

self`.destTime` (read-write) `Integer`

Specifies the destination time that the action list player jumps to when the time action *self* is triggered.

Instance Methods

Inherited from `Action`:

`trigger`

The following instance method is defined in `TimeAction`:

trigger (Action)

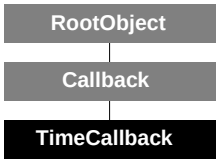
`trigger` *self target player* ⇨ *self*

self	`TimeAction` object
target	Any object (ignored by trigger)
player	`ActionListPlayer` object

Causes the time action *self* to make the action list player *player* jump to the time specified by the `destTime` instance variable. This method is called by the action list player at the time specified by the `time` instance variable.

Because this action affects the action list player and not an object on its target list, *target* is ignored.

TimeCallback



Class type: Core class (concrete)
Resides in: ScriptX and KMP executables
Inherits from: `Callback`
Component: Clocks

The `TimeCallback` class is used to perform actions when the clock reaches a certain time.

You never create an instance of `TimeCallback` directly. Instead, request a callback from a clock using the `addTimeCallback` method defined in `Clock`.

Instance Variables

Inherited from `Callback`:

`authorData`	`onceOnly`	`script`
`condition`	`order`	`target`
`label`	`priority`	

The following instance variables are defined in `TimeCallback`:

condition (Callback)

self`.condition` (read-write) `NameClass`

Determines the condition under which the callback will be activated. Valid values for `TimeCallback` objects are `@forward`, `@backward`, and `@either`. The default value is `@either`.

timeIsTicks

self`.timeIsTicks` (read-write) `Boolean`

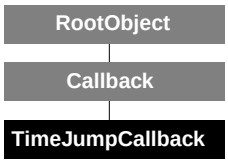
Determines whether or not the `TimeCallback` instance *self* is rescheduled based on changes to the scale of its clock. When this value is `true`, the callback will be rescheduled when the scale changes; when `false`, the callback ignores scale changes. By default, the value of this instance variable is `true`.

Instance Methods

Inherited from `Callback`:

`cancel`

TimeJumpCallback



Class type: Core class (concrete)
Resides in: ScriptX and KMP executables
Inherits from: `Callback`
Component: Clocks

The `TimeJumpCallback` class is used to perform actions when the clock's time is reset by a particular amount.

You never create an instance of `TimeJumpCallback` directly. Instead, you request a callback from a clock using the `addTimeJumpCallback` method defined by the `Clock` class.

The `order` instance variable, inherited from `Callback`, has no effect with a `TimeJumpCallback` instance.

Instance Variables

Inherited from `Callback`:

`authorData`	`onceOnly`	`script`
`condition`	`order`	`target`
`label`	`priority`	

The following instance variables are defined in `TimeJumpCallback`:

condition

self`.condition` (read-write) `NameClass`

Determines the condition under which the callback will be activated. The following values are valid for `TimeJumpCallback` objects:

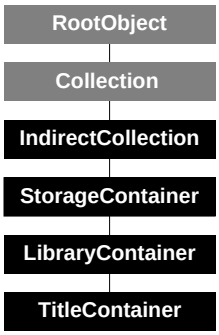
```
@lessThan
@greaterThan
@equal
@notEqual
@lessThanOrEqual
@greaterThanOrEqual
@change (default)
```

Instance Methods

Inherited from `Callback`:

`cancel`

TitleContainer



Class type: Core class (concrete)
Resides in: ScriptX and KMP executables
Inherits from: `LibraryContainer`
Component: Title Management

The `TitleContainer` class represents a file on disk that contains a ScriptX interactive multimedia title or tool. The user can open and close a title container independent of other ScriptX titles. (This contrasts with library and accessory containers, which are not stand-alone and are intended to be used by titles.)

A title container is a collection, by way of inheriting from `IndirectCollection`; its default target collection is the `Array` class. You should add to a title container any objects that you want saved to the title. You can add objects either explicitly (by adding them directly to the title container), or implicitly (by reference through an instance variable). No objects are automatically loaded when a title is opened—the startup script should load whatever objects are initially needed.

A title container contains a list of its top clocks and top players so that the title as a whole can be paused, resumed, and muted. A title container has a list of windows so they can be opened and closed with the title. It also has a list of open accessories that it uses. These lists are all maintained automatically by the title container as clocks, players, and windows are added to the title—in turn, these objects have `title` instance variables that are set to the title. (See those classes for further details.)

In addition, a title container provides an interface to two system resources: the ScriptX menu bar and the clipboard. Creating a title container automatically creates its own ScriptX menu bar, known as the system menu bar. See the `SystemMenuBar` class for more details.

A title container also provides the interface to the clipboard with the methods `cutSelection`, `copySelection`, and `pasteToSelection`. which in turn call the same methods on the currently active window.

The following global variables can hold title containers: `theScratchTitle`, `theOpenContainers`, and `theTitleContainer`. These variables are described in the chapter “Global Functions and Variables.”

Calling `open` on a title loads all libraries in its `libraries` instance variable, then runs the script in the `preStartupAction` instance variable, where it can exit if the proper versions of the libraries have not been loaded or if the environment does not meet the minimum requirements. If a user was supplied, `open` then calls `addUser`. Finally, it prepends the new title to `theOpenContainers`, and runs the script in `startupAction`.

Creating and Initializing a New Instance

The following script is an example of how to create a new instance of the `TitleContainer` class:

```
myTitle:= new TitleContainer \
          path:"MyTitle.sxt" \
          name:"My Title" \
          targetCollection:(new HashTable)
```

The variable `myTitle` contains the initialized title container, which is stored in a new file called `MyTitle.sxt` in the directory given by `theStartDir` (the default). Its name when asked to print is `My Title`.

Note – The convention for naming title container files, valid across all platforms, is to use the `.sxt` extension, as shown above (meaning "ScriptX Title").

This title container's contents are a hash table, so its objects can be identified using names or string constants as keys. The title container has no users. The `new` method uses the keywords defined in `init`.

init

`init` *self* [`dir:`*dirRep*] `path:`*collectionOrString* [`name:`*string*]
[`user:`*libraryContainer*] [`targetCollection:`*collection*] ⇨ *(none)*

This method is inherited from `LibraryContainer` with no change in keywords and defaults—refer to that class for details. Do not call `init` directly on an instance—it is automatically called by the `new` method.

Class Methods

Inherited from `Collection`:

`pipe`

Inherited from `StorageContainer`:

`open`

Inherited from `LibraryContainer`:

`open`

The following class methods are defined in `TitleContainer`:

open (StorageContainer)

`open` *self* [`dir:`*dirRep*] `path:`*collection* [`mode:`*name*]
[`user:`*libraryContainer*] ⇨ `TitleContainer`

Similar to the `open` method in `LibraryContainer`, except it opens a title container. For example:

```
open TitleContainer path:"MYTITLE.SXT" name:"My Title"
```

To summarize, this method opens the title container, calls the function in `preStartupAction`, loads its `libraries` instance variable, and makes itself a user of each library. If the `user` keyword is supplied, this method then calls `addUser` on the supplied library container with *self* as its new user. This method then adds the title to `theOpenContainers` global variable, and finally calls the function in `startupAction`.

If the title is already loaded when `open` is called, all this method does is call `bringToFront` on the title, to give it user focus.

Most of the implementation is inherited from `LibraryContainer` with no change in keywords or defaults. Refer to `open` in that class for more details.

Instance Variables

Inherited from `Collection`:

`bounded`	`maxSize`	`size`
`iteratorClass`	`minSize`	`uniformity`
`keyEqualComparator`	`mutable`	`uniformityClass`
`keyUniformity`	`mutableCopyClass`	`valueEqualComparator`
`keyUniformityClass`	`proprietored`	

Inherited from `IndirectCollection`:

`targetCollection`

Inherited from `LibraryContainer`:

`copyright`	`preStartupAction`	`users`
`directory`	`startupAction`	`version`
`libraries`	`terminateAction`	
`name`	`type`	

The following instance variables are defined in `TitleContainer`:

accessories

self`.accessories` (read-only) `Array`

Specifies a list of accessories that the title container *self* is currently using. Do not directly add or remove items from this list—calling the `addAccessory` and `removeAccessory` methods automatically maintains this list.

This instance variable is transient—it is not saved with the title. It is used to keep track of the accessories so that when the title closes, it can automatically call `removeAccessory` on its accessories.

audioMuted

self`.audioMuted` (read-write) `Boolean`

When set to `true`, this instance variable mutes all players in the `topPlayers` list of the title container *self*. When set to `false`, it unmutes all players.

hasUserFocus

self`.hasUserFocus` (read-only) `Boolean`

Specifies whether or not the title container *self* is the frontmost of all open titles. This flag is read-only, and is automatically set to `true` in three cases: when the title is first created or opened, when any of its windows gets a `mouseDown` event, and when `bringToFront` is called on the title container. Conversely, when any other title gets user focus, the `hasUserFocus` flag of the title losing focus is set to `false`.

Notice that a title can have user focus even if all its windows and its system menu bar are hidden. It manifests this by all background titles having `hasUserFocus` set to `false`. However, even though the frontmost window of a background title has its `hasUserFocus` set to `false`, its title bar still appears as if its `hasUserFocus` were `true` (and clicking on that window sets it true).

Notice that library containers and accessory containers cannot have user focus.

systemMenuBar

self.systemMenuBar (read-only) SystemMenuBar

Specifies the ScriptX menu bar that belongs to the title container *self*. This menu bar is automatically created when the title container is created. Each title container has its own menu bar—this menu bar appears when the title has user focus. In this case, you can also choose to hide the menu bar. See the SystemMenuBar class for more details.

topClocks

self.topClocks (read-only) Array

Specifies the list of all top clocks and top players currently in memory that belong to the title container *self*. A top clock belongs to a title container if its title instance variable is set to that title container. (It is not necessary for the top clock to have been added to that title container using a collection method such as add, prepend, or append). A top clock is a clock or player at the top of a timing hierarchy.

This list is used by the pause and resume methods to ensure all clocks in a title are stopped and started. This list is very dynamic; top clocks are added to and removed from this list as they are moved into and out of memory.

This list is transient and is automatically maintained by the system—you should not add or delete clocks or players from this array. This list keeps a "weak" hold on its clocks, meaning that when all normal (non-weak) references to a top clock are removed, the weak references are all automatically dropped and the top clock is available to be garbage collected.

topPlayers

self.topPlayers (read-only) Array

Specifies a list of all currently open top players that belong to the title container *self*. A player belongs to a title container if its title instance variable is set to that title container. (It is not necessary for the player to have been added to that title container using a collection method such as add, prepend, or append).

A top player is a player at the top of a timing hierarchy. For instance, if a MoviePlayer object were a top player, it would appear in this list, but its slaves, AudioPlayer and VideoPlayer, would not appear here.

This list is used by the audioMuted instance variable to ensure all players in a title are silenced. This list is very dynamic; top players are added to and removed from this list as they are moved into and out of memory.

This list is transient and is automatically maintained by the system—you should not add or delete players from this array. This list keeps a "weak" hold on its players, meaning that when all normal (non-weak) references to a top player are removed, the weak references are all automatically dropped and the top player is available to be garbage collected.

windows

self.windows (read-only) Array

Specifies a list of all currently loaded windows that are managed by the title container *self*. A window is managed by a title container if and only if its title instance variable is set to that title container. (It is not necessary for the window to have been added to that title container using a collection method such as add, prepend, or append). This list includes any kind of window, including dialog boxes, floating palettes, and windows.

A title manages its windows by making sure the windows are closed when the title is closed, the windows share user focus with the title, and the windows' compositors are paused when the title is paused. A window is automatically removed from this list when it is garbage collected.

This list is sorted by focus priority—the same order the user would see the windows on-screen. Therefore, to get the frontmost window of the title container that has user focus, you can do the following:

```
theTitleContainer.windows[1]
```

This list is transient and is automatically maintained by the system—you should not add or delete windows from this array. This list keeps a "weak" hold on its windows, meaning that when all normal (non-weak) references to a window are removed, the weak references are all automatically dropped and the window is available to be garbage collected.

Instance Methods

Inherited from `Collection`:

`add`	`forEach`	`iterate`
`addMany`	`forEachBinding`	`localEqual`
`addToContents`	`getAll`	`map`
`chooseAll`	`getAny`	`merge`
`chooseOne`	`getKeyAll`	`pipe`
`chooseOneBinding`	`getKeyOne`	`prin`
`deleteAll`	`getMany`	`removeAll`
`deleteBindingAll`	`getOne`	`removeOne`
`deleteBindingOne`	`hasBinding`	`setAll`
`deleteKeyAll`	`hasKey`	`setOne`
`deleteKeyOne`	`intersects`	`size`
`deleteOne`	`isEmpty`	
`emptyOut`	`isMember`	

Inherited from `IndirectCollection`:

`isAppropriateObject`	`objectAdded`	`objectRemoved`

Inherited from `StorageContainer`:

`close`	`update`
`requestPurgeForAllObjects`	

Inherited from `LibraryContainer`:

`addUser`	`objectAdded`	`terminate`
`close`	`recurPrin`	
`isAppropriateObject`	`removeUser`	

Since a `TitleContainer` object is an indirect collection, you can also use any methods defined in the class specified by `targetCollection`. The target collection is by default an instance of `Array`, which inherits from `Sequence`. In this case, the following instance methods are accessible:

Accessible from `LinearCollection`:

`chooseOneBackwards`	`findRange`	`getNthKey`
`chooseOrdOne`	`forEachBackwards`	`getOrdOne`
`deleteFirst`	`getFirst`	`getRange`
`deleteLast`	`getLast`	`localEqual`
`deleteNth`	`getMiddle`	`localLT`
`deleteRange`	`getNth`	`pop`

Accessible from Sequence by redirection:

addFifth	moveBackward	setFourth
addFirst	moveForward	setLast
addFourth	moveToBack	setNth
addNth	moveToFront	setSecond
addSecond	prepend	setThird
addThird	prependNew	sort
append	setFifth	
appendNew	setFirst	

The following instance methods are defined in TitleContainer:

addAccessory

addAccessory *self accessoryContainer* ⇨ Collection

self	TitleContainer object
accessoryContainer	AccessoryContainer object

Use this method to add the given *accessoryContainer* to the title container *self*.

This method calls getAccessory on the accessory, which returns to the title a collection of accessory objects and classes, then prepends the accessory to the title's accessories list.

You should override this method in a subclass to properly incorporate the accessory objects and classes into the title. You might instantiate its classes, or add its presenter objects to a window—whatever is appropriate for that title. The convention is for this method to return the collection of objects and classes passed from the accessory to the title.

bringToFront

bringToFront *self* ⇨ Boolean

Brings the title forward on the screen in front of all other titles, sets hasUserFocus for that title to true, and then returns true.

Clicking on a window calls bringToFront on the window and on its title.

clearSelection

clearSelection *self* ⇨ *(object)*

When the user chooses **Clear** from the ScriptX menu, this clearSelection method is automatically called on the title container *self* that currently has user focus. You can also call this method from a script directly on a title container. In either case, the title container then calls clearSelection on the frontmost window in that title. It is up to the clearSelection method in the window to delete the selected objects. By convention, this method should return the object being cleared.

The clearSelection method has no effect on the clipboard.

close (StorageContainer)

close *self* [user:*libraryContainer*] ⇨ Boolean

self	StorageContainer object to close
libraryContainer	StorageContainer object currently using *self*

Closes the title container *self*, making the title and its contained objects purgeable. Most of the implementation is inherited from `StorageContainer` with no change in keywords or defaults. Refer to `close` in that class for more details. In addition, see the `terminate` method in this class (which `close` calls) for other operations that occur, such as hiding the windows, pausing the clocks, and closing the accessories.

This method is called on the current title container by choosing **Close Title** from the File menu.

copySelection

`copySelection` *self* ⇨ *(object)*

When the user chooses **Copy** from the ScriptX menu, this `copySelection` method is automatically called on the title container *self* that currently has user focus. You can also call this method from a script directly on a title container. In either case, the title container then calls `copySelection` on the frontmost window in that title. It is up to the `copySelection` method in the window to call `setClipboard` to do the copy. By convention, this method should return the object being copied.

See the `copySelection` method in the `Window` class for more details.

cutSelection

`cutSelection` *self* ⇨ *(object)*

When the user chooses **Cut** from the ScriptX menu, this `cutSelection` method is automatically called on the title container *self* that currently has user focus. You can also call this method from a script directly on a title container. In either case, the title container then calls `cutSelection` on the frontmost window in that title. It is up to the `cutSelection` method in the window to call `setClipboard` to do the cut. By convention, this method should return the object being cut.

See the `cutSelection` method in the `Window` class for more details.

isAppropriateAccessory

`isAppropriateAccessory` *self accessoryContainer* ⇨ `Boolean`

self	`TitleContainer` object
accessoryContainer	`AccessoryContainer` object

This method checks that the title *self* can successfully use the given *accessoryContainer*. If this method returns `false`, then the accessory will not be added to this title; if it returns true, the accessory is added. This method is automatically called when the user opens a title container using the **Open** menu command, not when opening the title container using the `open` method from a script.

This method is implemented in this class to unconditionally return `true`. You should override this method to perform whatever test you want to ensure the accessory will work with this title. You might test the accessory's version number, or `type` instance variable (both inherited from `LibraryContainer`).

pasteToSelection

`pasteToSelection` *self* ⇨ *(none)*

When the user chooses **Paste** from the ScriptX menu, this `pasteToSelection` method is automatically called on the title container *self* that currently has user focus. You can also call this method from a script directly on a title container. In either case, the title container then calls `pasteToSelection` on the frontmost window in that title. It is up to the `pasteToSelection` method in the window to call `getClipboard` to do the paste. By convention, this method should return the object being pasted.

See the `pasteToSelection` method in the `Window` class for more details.

pause

`pause` *self* ⇨ `Boolean`

Calls `pause` on all clocks in the `topClocks` list. To actually resume the clocks, you must call `resume` once for every time `pause` has been called. This method returns `true`.

printTitle

`printTitle` *self* ⇨ *(object)*

Called when the **Print** menu option is chosen when the title *self* has user focus. The default implementation calls the `PrintWindow` method on the topmost `Window` of the title. This method should be overridden by a subclass for a different implementation.

removeAccessory

`removeAccessory` *self accessory* ⇨ `Boolean`

Removes the given *accessory* from the title *self*. This takes the accessory out of the `accessories` list, and calls `close` on the accessory container. The `close` method closes the accessory only if this title was its only user. (See the `close` method in `LibraryContainer` for an explanation.) Either the title that uses the accessory or the accessory itself can call this method.

You should override this method in a subclass to properly remove the accessory objects from the title—whatever is appropriate for that title. The convention is for this method to return a boolean which indicates if it succeeded.

resume

`resume` *self* ⇨ `Boolean`

Calls `resume` on all clocks in the `topClocks` list. To actually resume the clocks, you must call `resume` once for every time `pause` has been called. Returns `true` if, when done, all clocks are resumed. Returns `false` if at least one clock is still paused, or if there are no clocks in the `topClocks` list.

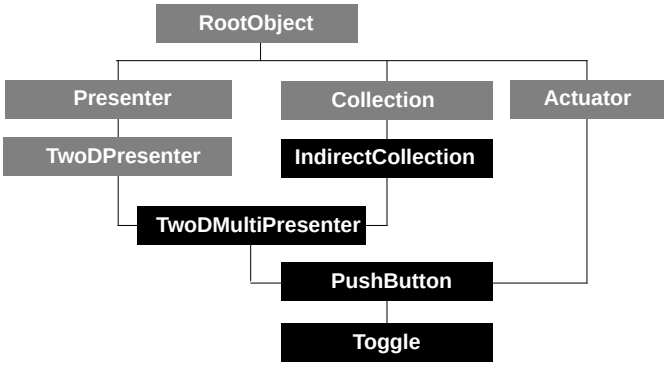
terminate (LibraryContainer)

`terminate` *self* ⇨ *self*

Refer to `terminate` in `LibraryContainer` for a general description.

In addition to what is described in the `terminate` method of `LibraryContainer`, this method calls `hide` on all windows listed in *self*`.windows`, calls `playUnprepare` on all players in *self*`.topPlayers`, calls `pause` on all clocks in *self*`.topClocks`, and calls `close` on each accessory in its `accessories` instance variable.

Toggle



Class type: Core class (concrete)
Resides in: ScriptX and KMP executables
Inherits from: `PushButton`
Component: User Interface

The `Toggle` class provides the functionality for presenting check boxes or radio buttons. Such buttons are actuators that can be switched between two checked states. A `Toggle` object defines two additional presenters, stored in the instance variables `toggledOnPresenter` and `toggledOffPresenter`, that represent its on and off status.

`Toggle` redefines the `activate` method to call either `toggleOn` or `toggleOff`, depending on the current state of the toggle. It also redefines the `handleActivate`, `handlePress`, and `handleRelease` methods, allowing them to call scripted functions stored in `activateAction`, `pressAction`, and `releaseAction` with a third argument that indicates the current state of the toggle.

See the "User Interface" chapter of the *ScriptX Components Guide* for more information on the `Toggle` class, including a diagram that depicts z ordering on a `Toggle` object.

Creating and Initializing a New Instance

The following script is an example of how to create a new instance of the `Toggle` class. The rectangle `myBox` forms the shape of the toggle.

```
myBox := new Rect x1:50 y1:50 x2:60 y2:60
turnOn := new TwoDShape boundary:(new Line x2:10 y2:10)
pressBox := new TwoDShape boundary:myBox fill:blackBrush
releaseBox := new TwoDShape boundary:myBox fill:whiteBrush \
              stroke:blackBrush
myToggle := new Toggle \
              toggledOnPresenter:turnOn \
              pressedPresenter:pressBox \
              releasedPresenter:releaseBox
```

The variable `myToggle` contains the initialized toggle, which defines three subpresenters that will determine its appearance. Notice that `toggledOffPresenter` and `disabledPresenter` are not passed in as arguments, and are therefore `undefined`, which means they have no effect. This example shows an empty square when it is toggled off, a square with a diagonal line through it when it is toggled on, and a black square when it is pressed. The `new` method on `Toggle` uses keywords defined in `init`.

init

`init` *self* [`toggledOnPresenter:`*twoDPresenter*]
[`toggledOffPresenter:`*twoDPresenter*]
[`pressedPresenter:`*twoDPresenter*]
[`releasedPresenter:`*twoDPresenter*]
[`disabledPresenter:`*twoDPresenter*] ⇨ *(none)*

self	`Toggle` object
`toggledOnPresenter:`	`TwoDPresenter` object
`toggledOffPresenter:`	`TwoDPresenter` object

The superclass `PushButton` uses these keywords:

`pressedPresenter:`	`TwoDPresenter` object
`releasedPresenter:`	`TwoDPresenter` object
`disabledPresenter:`	`TwoDPresenter` object

Superclasses of `Toggle` and `PushButton` use these keywords:

`fill:`	`Brush` object
`stroke:`	`Brush` object
`boundary:`	`Stencil` object (ignored by `Toggle`)
`target:`	Any object (ignored by `Toggle`)
`targetCollection:`	`Sequence` object (use carefully)
`stationary:`	`Boolean` object

Initializes the `PushButton` object *self*, applying the values supplied with the keywords to the instance variables of the same name. Do not call `init` directly on an instance—it is automatically called by the `new` method.

If you omit an optional keyword, its default value is used. The defaults are:

```
toggledOnPresenter:undefined
toggledOffPresenter:undefined
pressedPresenter:undefined
releasedPresenter:undefined
disabledPresenter:undefined
stationary:false
```

The `Toggle` class inherits a number of keywords from superclasses of both `PushButton` and `Toggle`. These keywords are generally not used. For more information, see the class definition of `PushButton`.

Class Methods

Inherited from `Collection`:

`pipe`

Instance Variables

Inherited from `Actuator`:

`enabled`	`pressed`	`toggledOn`
`menu`		

Inherited from `Collection`:

`bounded`	`maxSize`	`size`
`iteratorClass`	`minSize`	`uniformity`
`keyEqualComparator`	`mutable`	`uniformityClass`
`keyUniformity`	`mutableCopyClass`	`valueEqualComparator`
`keyUniformityClass`	`proprietored`	

Inherited from `IndirectCollection`:

`targetCollection`

Inherited from `Presenter`:

`presentedBy` `subPresenters` `target`

Inherited from `TwoDPresenter`:

`bBox`	`height`	`transform`
`boundary`	`IsImplicitlyDirect`	`width`
`clock`	`isTransparent`	`window`
`compositor`	`isVisible`	`x`
`direct`	`needsTickle`	`y`
`eventInterests`	`position`	`z`
`globalBoundary`	`stationary`	
`globalTransform`	`target`	

Inherited from `TwoDMultiPresenter`:

`clock` `fill` `stroke`

Inherited from `PushButton`:

`activateAction`	`multiActivateAction`	`releaseAction`
`authorData`	`pressAction`	`releasedPresenter`
`disabledPresenter`	`pressedPresenter`	

The following instance variables are defined in `Toggle`:

activateAction (PushButton)

self`.activateAction` (read-write) *(function)*

Specifies the default function that is called when the button is activated by being pressed and released. Initially this instance variable is `undefined`. You can write a function to perform any action. `Toggle` specializes this instance variable to store a function with three arguments:

`funcName` *authorData self toggledOn*

authorData	data in the `authorData` instance variable
self	the `Toggle` object *self* to which the action is attached
toggledOn	a `Boolean` object indicating the state of *self*.

Although any global function, anonymous function, or method can be assigned to `activateAction`, there are differences in how different classes of functions are dispatched. For information on functions and function dispatch, see the "Object System Kernel" chapter of the *ScriptX Components Guide*.

pressAction (PushButton)

self`.pressAction` (read-write) *(function)*

Specifies the default function that is called when the button is pressed. The function has three arguments, as shown in `activateAction`.

releaseAction (PushButton)

self`.releaseAction` (read-write) *(function)*

Specifies the default function to call when the button has been pressed and is released without being activated. Typically, this happens when the user has moved the mouse pointer away from the button before releasing it. The function has three arguments, as shown in `activateAction`.

toggledOffPresenter

self.toggledOffPresenter (read-write) TwoDPresenter

Specifies the presenter for the toggle *self* when it is in the off state. This presenter is layered in front of the presenter specified by the releasedPresenter or pressedPresenter, depending on the state of the button. If the toggledOffPresenter is undefined, then nothing is drawn over the presenter specified by releasedPresenter or pressedPresenter when the toggle is off. Do not define toggledOffPresenter to be the same as the pressedPresenter, releasedPresenter, or disabledPresenter defined by PushButton.

toggledOnPresenter

self.toggledOnPresenter (read-write) TwoDPresenter

Specifies the presenter for the toggle *self* when it is in the on state. This presenter is layered in front of the presenter specified by the releasedPresenter or pressedPresenter, depending on the state of the button. If the toggledOnPresenter is undefined, then nothing is drawn over the presenter specified by releasedPresenter or pressedPresenter when the toggle is on. Do not define toggledOffPresenter to be the same as the pressedPresenter, releasedPresenter, or disabledPresenter defined by PushButton.

Instance Methods

Inherited from Actuator:

activate	press	toggleOff
multiActivate	release	toggleOn

Inherited from Collection:

add	forEach	iterate
addMany	forEachBinding	localEqual
addToContents	getAll	map
chooseAll	getAny	merge
chooseOne	getKeyAll	pipe
chooseOneBinding	getKeyOne	prin
deleteAll	getMany	removeAll
deleteBindingAll	getOne	removeOne
deleteBindingOne	hasBinding	setAll
deleteKeyAll	hasKey	setOne
deleteKeyOne	intersects	size
deleteOne	isEmpty	
emptyOut	isMember	

Inherited from IndirectCollection:

isAppropriateObject	objectAdded	objectRemoved

Inherited from TwoDPresenter:

adjustClockMaster	inside	show
createInterestList	localToSurface	surfaceToLocal
draw	notifyChanged	tickle
getBoundaryInParent	recalcRegion	
hide	refresh	

Inherited from TwoDMultiPresenter:

draw	findFirstInStencil	moveToBack
findAllAtPoint	isAppropriateObject	moveToFront
findAllInStencil	moveBackward	objectAdded
findFirstAtPoint	moveForward	objectRemoved

Inherited from PushButton:

activate	handleMultiActivate	layout
calculateSize	handlePress	press
handleActivate	handleRelease	release

Since a Toggle object is an indirect collection, you can also use any methods defined in the class specified by targetCollection. The target collection is typically an instance of Array, which inherits from Sequence, so the following instance methods are redirected to a toggle.

Accessible from LinearCollection:

chooseOneBackwards	findRange	getNthKey
chooseOrdOne	forEachBackwards	getOrdOne
deleteFirst	getFirst	getRange
deleteLast	getLast	localEqual
deleteNth	getMiddle	localLT
deleteRange	getNth	pop

Accessible from Sequence:

addFifth	moveBackward	setFourth
addFirst	moveForward	setLast
addFourth	moveToBack	setNth
addNth	moveToFront	setSecond
addSecond	prepend	setThird
addThird	prependNew	sort
append	setFifth	
appendNew	setFirst	

The following instance methods are defined in Toggle:

activate (Actuator)

`activate` *self* ⇨*self*

Calls toggleOn or toggleOff as appropriate on the Toggle object *self*. Then the method calls activate on its superclass PushButton.

calculateSize

`calculateSize` *self* ⇨ `Point`

Do not call this method directly from the scripter. It is normally called by the actuator. It is visible so that it can be overridden in a scripted subclass. This method calculates the correct bounding box of the toggle *self*. By default, the size of the toggle is the largest width and height of all its contained presenters. This method returns a Point object containing the new width and height. This method is automatically triggered each time any of the contained presenters is changed. If the width or height of the button is changed after this method is called, then the subpresenters will be clipped by the new width and height.

handleActivate (PushButton)

`handleActivate` *self* ⇨*self*

Do not call this method directly from the scripter. It is visible so that it can be overridden in a scripted subclass. This method calls the function specified by activateAction, an instance variable defined on PushButton. The class Toggle redefines handleActivate to call this function with three arguments: *authorData*, *self*, and *toggledOn*.

handlePress (PushButton)

`handlePress` *self* ⇨*self*

Do not call this method directly from the scripter. It is visible so that it can be overridden in a scripted subclass. This method calls the function specified by `pressAction`, an instance variable defined on `PushButton`. The class `Toggle` redefines `handlePress` to call this function with three arguments: *authorData*, *self*, and *toggledOn*.

handleRelease (PushButton)

`handleRelease` *self* ⇨*self*

Do not call this method directly from the scripter. It is visible so that it can be overridden in a scripted subclass. This method calls the function specified by `releaseAction`, an instance variable defined on `PushButton`. The class `Toggle` redefines `handleRelease` to call this function with three arguments: *authorData*, *self*, and *toggledOn*.

layout

`layout` *self* ⇨*self*

Sets the correct locations of all contained presenters for the toggle *self*. This method is automatically triggered each time any presenters contained by the `Toggle` class change, but it can also be called directly.

toggleOff

`toggleOff` *self* ⇨*self*

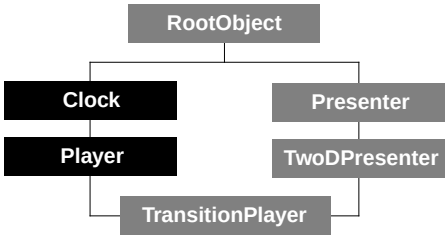
Sets the value of `toggledOn` to `false` for the toggle *self*, and displays the presenter specified by `toggledOffPresenter`, if it is not `undefined`. The `toggledOn` instance variable is defined by `Actuator`.

toggleOn

`toggleOn` *self* ⇨*self*

Sets the value of `toggledOn` to `true` for the toggle *self*, and displays the presenter specified by `toggledOnPresenter`, if it is not `undefined`. The `toggledOn` instance variable is defined by `Actuator`.

TransitionPlayer



Class type: Core class (abstract)
Resides in: ScriptX and KMP executables
Inherits from: `Player` and `TwoDPresenter`
Component: Transitions

The `TransitionPlayer` class defines a protocol for creating visual transition effects when a new presenter is added to a space. Subclasses of `TransitionPlayer` provide the concrete transition effects that can be instantiated. Subclasses of `TransitionPlayer` that are built into the Kaleida Media Player are listed here:

- `Slide` – causes the new image to slide onto the screen
- `Wipe` – causes the new image to be wiped onto the screen
- `Iris` – causes the new image to appear through an opening or closing aperture
- `BarnDoor` – causes the new image to appear by opening or closing doors

Other ScriptX transitions can be loaded. Loadable units are code segments, stored in a directory that is accessible to the Kaleida Media Player, that load dynamically at runtime. Among the loadable transition classes that ship with the ScriptX Language and Class Library are `Blinds`, `Checkerboard`, `DiamondIris`, `Dissolve`, `Fan`, `GarageDoor`, `Push`, `RandomChunks`, `RectIris`, `RectWipe`, `StripSlide`, and `StripWipe`.

Technically, a transition is a visual effect that occurs when *adding* an image to a 2D space. The image is a 2D presenter specified by the `target` instance variable, and can be either a static (still) image, or a dynamic image, such as a video or animation. The transition effect is applied only within the boundary of the target presenter.

By adding an instance of `TransitionPlayer` to a space, you determine that the transition takes effect in that space. When you call `play` on the transition player, its target presenter "transitions" into that space.

When a transition is played, the compositor draws the target presenter in incremental steps in front of other presenters at that location. As the transition is applied, the original image is covered up gradually. Note that the presenters that comprise the original image still exist in the space, even when they have been obscured completely. A transition player does not remove other presenters from the space. If you want other presenters removed, you must explicitly remove them.

To remove an image from a space is to add it in reverse. You can specify a target presenter that you want to remove by setting the transition's rate to a negative value, which plays the transition backwards. (Some of the loadable transitions that ship with ScriptX, such as `Dissolve` and `RandomChunks`, are direct presenters and do not work in reverse.)

You can increase the efficiency with which the compositor draws the target presenter in a new space by setting the value of `useOffscreen` to `true`. Setting `movingTarget` to `false` also increases performance, for certain transitions. The price of this increase in performance is that the target presenter must be static or "frozen" while it is being drawn into the space. For example, if the value of `movingTarget` is `false`, then a movie player must display the same frame until the transition is complete.

Note that TransitionPlayer establishes a complete protocol for transitions, defining properties such as direction and duration for all transitions. Individual subclasses of TransitionPlayer do not necessarily implement all these properties, or respond to all possible settings. For example, an Iris player recognizes @open and @close as possible values for direction, while a Wipe player responds to other values such as @up, @down, @left, and @right, and a Dissolve player ignores direction altogether.

Creating and Initializing a New Instance

TransitionPlayer is an abstract class that cannot be instantiated. However, all concrete subclasses of TransitionPlayer use the same syntax for creating a new instance. To create a new instance of a subclass, use the new method, as shown in the class definitions of Slide, Wipe, Iris, and BarnDoor.

TransitionPlayer defines an init method that is inherited by its concrete subclasses.

init

init *self* [duration:*integer*] [direction:*name*]
[movingTarget:*boolean*] [useOffscreen:*boolean*] [target:*twoDPresenter*]
[boundary:*stencil*] [masterClock:*clock*] [scale: *integer*] ⇨ *(none)*

self	TransitionPlayer object
duration:	Integer object representing the length of the transition
direction:	NameClass object that sets the direction of the transition
movingTarget:	Boolean object
useOffscreen:	Boolean object

Superclass TwoDPresenter uses the following keywords:

target:	TwoDPresenter object representing the presenter to be added to the space using the transition
boundary:	Stencil object (ignored by TransitionPlayer)
stationary:	Boolean object

Superclass Clock uses the following keywords:

masterClock:	Clock object to use as the player's master clock
scale:	Integer object to use as the scale for the player.

Initializes the TransitionPlayer object *self*, applying the values supplied with the keywords to the instance variables of the same names. Do not call init directly on an instance—it is called automatically by the new method.

If you omit an optional keyword, its default value is used. The defaults are:

```
duration:99
direction:undefined
movingTarget:false
useOffscreen:false
target:(new TwoDShape)
boundary:(new Rect)
masterClock:undefined
scale:1
stationary:false
```

Instance Variables

Inherited from Presenter:

presentedBy	subPresenters	target

Inherited from TwoDPresenter:

bBox	height	transform
boundary	IsImplicitlyDirect	width

clock	isTransparent	window
compositor	isVisible	x
direct	needsTickle	y
eventInterests	position	z
globalBoundary	stationary	
globalTransform	target	

Inherited from Clock:

callbacks	rate	ticks
effectiveRate	resolution	time
masterClock	scale	title
offset	slaveClocks	

Inherited from Player:

audioMuted	globalContrast	globalVolumeOffset
dataRate	globalHue	markerList
duration	globalPanOffset	status
globalBrightness	globalSaturation	videoBlanked

The following instance variables are defined in TransitionPlayer:

autoSplice

self.autoSplice (read-write) Boolean

If the value of autoSplice is true, when the transition player *self* finishes, it removes the transition player from its parent space and puts the target presenter in its place in that space. That is, it automatically splices the target into the parent space. The default is false.

backgroundBrush

self.backgroundBrush (read-only) Brush

Specifies the brush used for the background of the cached target (a bitmap surface) of the transition player *self*. See the cachedTarget instance variable for more details. Set this brush when transitioning a white target to something other than white, and use its color for the invisible color. (Note that backgroundBrush is a brush, while invisibleColor is a color.) For targets that aren't white, background brush doesn't need to be set, but can be set to produce interesting effects.

cachedTarget

self.cachedTarget (read-only) BitmapSurface

Specifies an offscreen bitmap into which the target of the transition player *self* is rendered when the value of useOffscreen is true. This bitmap surface is created when a program calls playPrepare. The cached target appears identical to the actual target, except for transparency and matte effects. It has a white rectangle enclosing the image. A white rectangle works well for transitions over a whole space. If a white rectangle is unsuitable, you can specify that the color white be invisible or transparent on the cached target by setting the values of invisibleColor and matteColor, both of which are defined by the Bitmap class and inherited by BitmapSurface.

direction

self.direction (read-write) NameClass

Specifies the direction in which the transition player *self* should be applied. Possible values include @left, @right, @up, @down, @open, or @close. Loadable transitions can accept other values for direction. Individual transitions respond to some directions and not to others. For example, an iris responds only to @open and @close.

duration (Player)

self.duration (read-write) Integer

Specifies the number of ticks that the transition player *self* should take.

The length of time that a transition takes is equal to the duration divided by the scale of the transition player. (If the transition has no master clock, this time is in seconds.) For example, a transition with a value of 60 for `duration` and a value of 30 for `scale` will take 2 seconds.

The "smoothness" of a transition is determined by the number of frames over which it occurs. This is equal to the transition player's scale. For example, a wipe transition with a length of 2 seconds and a scale of 30 ticks per second comprises 60 overall steps from start to finish. The more steps to the transition, the smoother the transition will appear.

Duration must have a value greater than 0. Attempting to set a non-positive value for `duration` causes ScriptX to report an exception.

frame

self.frame (read-only) Integer

Specifies the current frame of the transition player *self*. The value of `frame` is always an integer bounded by 0 and the value of `duration`.

movingTarget

self.movingTarget (read-write) Boolean

Specifies whether the entire region that has been redrawn since the first frame of the transition is updated with each frame. If the value of `movingTarget` is `true`, then a target presenter that is changing is updated with each frame. For example, if a transition player is being used to add a movie player to a space, and the value of `movingTarget` is `true`, then the movie is updated with each frame.

By default, the value of `movingTarget` is `false`. When `movingTarget` is set to `false`, only the part of the target presenter that actually changes with each frame is updated. For example, if the target presenter is drawn in with a wipe, the `Wipe` transition player updates only the new strips that are shown.

target (Presenter)

self.target (read-write) TwoDPresenter

Specifies the 2D presenter for the transition player *self*. If there is no target presenter, ScriptX applies the transition to a transparent `BitmapSurface` object. The boundary of the target determines the area over which the transition has an effect.

useOffscreen

self.useOffscreen (read-write) Boolean

If the value of `useOffscreen` is `false`, as in the default state, the transition player *self* transitions in the target presenter with all of its dynamics. If the target presenter presents any moving objects, these objects do not freeze during the transition. For example, an animation can be playing while a transition wipes it onto the screen.

Transition players with `useOffscreen` set to `false` do not work well when the target presenter is a direct presenter with a dynamic image, such as a digital video player or movie player. For a player that is a direct presenter, you should either set the value of `direct` to `false`, or freeze the player's motion by setting `useOffscreen` to `true`.

When the value of `useOffscreen` is `true`, a transition player creates a static cache of the target in `cachedTarget`, and uses this cached target in the transition. A cache is a bitmap snapshot of the image. If the original image is composed of many objects, or is dynamic in nature (displays moving objects, animation, or video), then the cached target allows the transition to run faster. However, any motion by the target presenter is frozen during the transition. If memory is tight and the target is large, you may want to set `useOffscreen` to `false` so that it does not create a cached target.

If the transition is a direct presenter (the value of `direct` is `true`) and the value of `useOffscreen` is also `true`, then ScriptX uses the compositor's offscreen buffer to apply the transition instead of creating a separate one for the transition. To use the compositor's offscreen buffer, the transition player must also be in the presentation hierarchy before the program calls `play` or `playPrepare`. It must be in the presentation hierarchy so that the transition can find the compositor whose buffer it is going to use. Using the compositor's offscreen buffer can generate a major savings in memory.

If the transition is not a direct presenter, or if it is not in the presentation hierarchy, and the value of `useOffscreen` is `true`, then ScriptX allocates an offscreen buffer for the transition. If warnings are turned on, ScriptX prints a warning to the `debug` stream. (For more information about warnings, see the definition of the global function `warnings` in Chapter 2, "Global Functions.")

Instance Methods

Inherited from `TwoDPresenter`:

`adjustClockMaster`	`inside`	`show`
`createInterestList`	`localToSurface`	`surfaceToLocal`
`draw`	`notifyChanged`	`tickle`
`getBoundaryInParent`	`recalcRegion`	
`hide`	`refresh`	

Inherited from `Clock`:

`addPeriodicCallback`	`clockAdded`	`pause`
`addRateCallback`	`clockRemoved`	`resume`
`addScaleCallback`	`effectiveRateChanged`	`timeJumped`
`addTimeCallback`	`forEachSlave`	`waitTime`
`addTimeJumpCallback`	`isAppropriateClock`	`waitUntil`

Inherited from `Player`:

`addMarker`	`goToBegin`	`playPrepare`
`eject`	`goToEnd`	`playUnprepare`
`fastForward`	`goToMarkerFinish`	`playUntil`
`getMarker`	`goToMarkerStart`	`resume`
`getNextMarker`	`pause`	`rewind`
`getPreviousMarker`	`play`	`stop`

The following instance method is defined in `TransitionPlayer`:

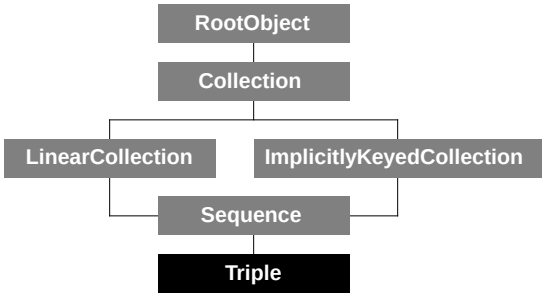
playPrepare (Player)

`playPrepare` *self nextRate* ⇨ `Boolean`

self	`TransitionPlayer` object to prepare for playing
nextRate	`Number` representing the next rate for the player

Prepares the transition player *self* to play its transition by creating the offscreen bitmap specified by `cachedTarget` (if `useOffscreen` is `true`). Once `playPrepare` is invoked, `play` and `stop` can be called at will. The value of `rate` for playing forward is 1.

Triple



Class type: Core class (concrete)
Resides in: ScriptX and KMP executables
Inherits from: `Sequence`
Component: Collections

The `Triple` class represents arrays of three values. A `Triple` object is created with a fixed size of 3. You cannot add (`addNth`) or delete (`deleteNth`) items, but you can set their values (`setNth`).

See also the `Single`, `Pair`, and `Quad` classes.

Creating and Initializing a New Instance

The following script is an example of how to create a new instance of the `Triple` class:

```
myT := new Triple values:#(2,4,6)
```

The variable `myT` contains the initialized triple, which contains the numbers 2, 4 and 6.

init

`init` *self* ⇨ *(none)*

self `Triple` object

Initializes the `Triple` object *self*, setting each of the three items in the list to `undefined`. Do not call `init` directly on an instance—it is automatically called by the `new` method.

Class Methods

Inherited from `Collection`:

`pipe`

Instance Variables

Inherited from `Collection`:

`bounded`	`maxSize`	`size`
`iteratorClass`	`minSize`	`uniformity`
`keyEqualComparator`	`mutable`	`uniformityClass`
`keyUniformity`	`mutableCopyClass`	`valueEqualComparator`
`keyUniformityClass`	`proprietored`	

Instance Methods

Inherited from `Collection`:

`add`	`forEach`	`iterate`
`addMany`	`forEachBinding`	`localEqual`
`addToContents`	`getAll`	`map`
`chooseAll`	`getAny`	`merge`
`chooseOne`	`getKeyAll`	`pipe`
`chooseOneBinding`	`getKeyOne`	`prin`
`deleteAll`	`getMany`	`removeAll`
`deleteBindingAll`	`getOne`	`removeOne`
`deleteBindingOne`	`hasBinding`	`setAll`
`deleteKeyAll`	`hasKey`	`setOne`
`deleteKeyOne`	`intersects`	`size`
`deleteOne`	`isEmpty`	
`emptyOut`	`isMember`	

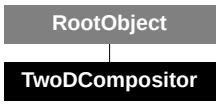
Inherited from `LinearCollection`:

`chooseOneBackwards`	`findRange`	`getNthKey`
`chooseOrdOne`	`forEachBackwards`	`getOrdOne`
`deleteFirst`	`getFirst`	`getRange`
`deleteLast`	`getLast`	`localEqual`
`deleteNth`	`getMiddle`	`localLT`
`deleteRange`	`getNth`	`pop`

Inherited from `Sequence`:

`addFifth`	`moveBackward`	`setFourth`
`addFirst`	`moveForward`	`setLast`
`addFourth`	`moveToBack`	`setNth`
`addNth`	`moveToFront`	`setSecond`
`addSecond`	`prepend`	`setThird`
`addThird`	`prependNew`	`sort`
`append`	`setFifth`	
`appendNew`	`setFirst`	

TwoDCompositor



Class type: Core class (concrete)
Resides in: ScriptX and KMP executables
Inherits from: RootObject
Component: Spaces and Presenters

The TwoDCompositor class manages the drawing of presenters to a window's display surface. A title can have more than one window; each window has its own display surface, presentation hierarchy, and 2D compositor. Each presentation hierarchy has a top presenter that the 2D compositor calls methods on.

The ScriptX compositor is interrupt-driven. The compositor maintains a private record of which parts of the display surface have changed, its refresh region. Each time a presenter changes, it calculates its own refresh region and then passes a reference to this region to the compositor. The compositor accumulates changes to the display surface in its refresh region. Normally, these changes are accumulated in an offscreen frame buffer, which the compositor then uses to refresh the display surface.

Direct presenters are handled as a special case. Since direct presenters are not drawn to the offscreen frame buffer, the compositor must poll direct presenters once with each cycle. Any changes to direct presenters are drawn directly to the display surface.

During the composite phase of the modeling/presentation cycle, the compositor draws its refresh region to the frame buffer or display surface, depending on the value of useOffscreen. The refresh region is restored to an empty region, and the composite cycle begins again. In this way, the compositor redraws only those portions of its display surface that change.

Although you do not create an instance of TwoDCompositor directly from the scripter, you have access to each window's compositor through the instance variable compositor, defined by Window. As a program runs, it is commonplace to disable the compositor during a long series of operations or transformations on a presenter, so that it does not redraw with each step. Other properties and methods are visible to allow for more advanced optimization techniques.

See the warnings function in Chapter 2, "Global Functions" for diagnotics related to the compositor.

Creating and Initializing a New Instance

Do not create an instance of TwoDCompositor directly. An instance of TwoDCompositor is created automatically when you create an instance of Window.

Instance Variables

displaySurface

self.displaySurface (read-write) DisplaySurface

Specifies the display surface to which the 2D compositor *self* is drawing.

drawCallback

self.drawCallback (read-only) PeriodicCallback

Specifies a callback that calls composite on the 2D compositor *self*. This instance variable is set up in response to creating a window, after the 2D compositor and display surface are created.

enabled

self.enabled (read-write) Boolean

Turns the 2D compositor *self* on (true) or off (false). The default for new compositors is true. When enabled is false, the composite method will do nothing even if it is being called. This allows you to temporarily turn off the compositor while preparing a complex scene change involving several presenters.

offscreen

self.offscreen (read-only) BitmapSurface

Specifies the BitmapSurface object used for offscreen rendering by the TwoDCompositor object *self*. If useOffscreen is set to false, then the value of offscreen is undefined.

showChangedRegion

self.showChangedRegion (read-write) Boolean

Available only in the ScriptX Language and Class Library, and not with the Kaleida Media Player, showChangedRegion is meant to be used as a diagnostic tool. It is a flag that directs the TwoDCompositor object *self* to mark a red outline around any regions in the window that subsequently change. (It considers movement of the Listener window to be a changed region if the window is behind it.) Regions with red outlines accumulate until the window is refreshed (which you can do by hiding and showing the window).

Set showChangedRegion to true as a diagnostic tool so that you can optimize your title to do the least amount of updating possible. When showChangedRegion is set to true, it may be apparent that some areas of the screen are being updated, even though the underlying image has not changed. You can improve the performance of the compositor by reducing the area of the surface that is updated with each cycle. For example, suppose that an animation works by switching between several large bitmaps. You could redesign the animation to present a large bitmap and several smaller ones that layer on top of it successively.

topPresenter

self.topPresenter (read-only) TwoDPresenter

Specifies the top-level presenter, always a window, that is associated with the 2D compositor *self*. This top presenter and its subpresenters comprise the presentation hierarchy that is drawn by the compositor.

useOffScreen

self.useOffScreen (read-write) Boolean

Specifies whether the compositor should use an offscreen bitmap for compositing. By default, the value of useOffScreen is true, meaning that changed regions are drawn to an offscreen frame buffer before they are drawn to the screen.

When `useOffScreen` is set to `false`, the compositor saves memory and drawing time by drawing the entire contents of the window directly to the screen. In effect, this makes every presenter in the window a direct presenter. Setting `useOffScreen` to `false` saves memory and increases performance, but it can result in flashing where presenters overlap on a display surface.

Instance Methods

composite

`composite` *self* ⇨ *self*

Starts the composite cycle of the 2D compositor *self*. While the compositor is running, the thread system cannot switch threads.

The compositor maintains a private list of presenters that need to be tickled, those for which the value of `needsTickle` is `true`. It calls `tickle` on each presenter in its tickle list. Although the compositor iterates through its tickle list before it suspends the thread switcher, the `tickle` method should not be allowed to block. (For information on blocking, see the "Threads" chapter of the *ScriptX Components Guide*.)

If the value of `useOffscreen` is `true`, the changed region of the compositors's offscreen bitmap is drawn to the display surface. A `TwoDCompositor` object maintains a private record of this changed region, which is updated incrementally each time a program calls `notifyChanged` on any presenter in the presentation hierarchy.

If the value of `useOffscreen` is `false`, presenters draw directly to the display surface. When an offscreen bitmap is not in use, all presenters are in effect direct presenters. The compositor iterates through the presentation hierarchy, calling `draw` on each 2D presenter, If presenters overlap, this may cause a "flashing" effect.

Next, the compositor updates any direct presenters. The compositor maintains a private list of direct presenters. Direct presenters that need to be refreshed are drawn directly to the display surface.

Do not call `composite` directly from the scripter; it is invoked from the callback specified by the instance variable `drawCallback`.

processPaletteChange

`processPaletteChange` *self interest event* ⇨ *self*

self	`TwoDCompositor` object
interest	`PaletteChangedEvent` object, used as an interest
event	`PaletteChangedEvent` object, used as an event

Do not call `processPaletteChange` directly from the scripter. A `TwoDCompositor` object maintains an interest in palette changed events, for which the instance method `processPaletteChange` is registered as the event receiver. The arguments *self*, *interest*, and *event* give `processPaletteChange` the standard signature of an event receiver function.

When the compositor receives a palette changed event, `processPaletteChange` runs, creating a new bitmap surface of the proper depth for both the display surface and the frame buffer.

recalcDirect

`recalcDirect` *self* ⇨ `Stencil`

Causes the 2D compositor *self* to recalculate the region of the display surface that is controlled by direct presenters.

refreshRegion

`refreshRegion` *self stencil* ⇨ *(none)*

self	`TwoDCompositor` object
stencil	`Stencil` object, the boundary of the region to refresh

Causes the `TwoDCompositor` object *self* to refresh a portion of its display surface whose boundary is defined by *stencil*.

If the value of `useOffscreen` is `true`, `refreshRegion` uses the offscreen bitmap (a `BitmapSurface` object referenced by *self*`.offScreen`) to refresh the region of the display surface that is encompassed by *stencil*.

If the value of `useOffscreen` is `false`, `refreshRegion` iterates through the presentation hierarchy in depth-first order, calling `draw` on each presenters in the affected region, with *stencil* as the clipping stencil.

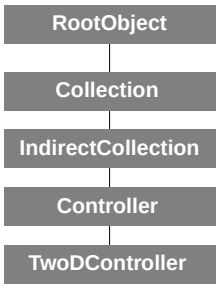
resize

`resize` *self* ⇨ *self*

self	`TwoDCompositor` object

Causes the 2D compositor *self* to recalculate the width, height, and pixel depth of its display surface, and of its offscreen bitmap, if one is in use. A call to `resize` is triggered automatically when the user resizes the window or sets the color depth manually.

TwoDController



Class type: Core class (abstract)
Resides in: ScriptX and KMP executables
Inherits from: `Controller`
Component: Controllers

The `TwoDController` class is the abstract class for all controller objects in a 2D space. This class inherits from `Controller` and has the same features as that class with one difference: it ensures that objects being controlled are in the same front-to-back order as they appear in the 2D space.

A controller object directs the layout or behavior of objects contained in a space, interprets user input events, or performs other control functions. Built-in 2D controller subclasses include `Interpolator`, `Bounce`, `Gravity`, `Movement` (described in the "Controllers" chapter of the *ScriptX Components Guide*) and `ActuatorController`, `RowColumnController`, and others (described in the "User Interface" chapter of the *ScriptX Components Guide*).

Note – For performance reasons, when you create an instance of `TwoDController` or any of its subclasses, if you omit the `targetCollection` instance variable, the class defines an optimized default target collection. This collection is optimized to be traversed quickly when the controller calls `tickle` on all its controlled objects. You should use discretion in changing the target collection to something other than what is supplied by default so that performance does not suffer.

Creating and Initializing a New Instance

Because `TwoDController` is an abstract class, you cannot create an instance of it, nor is it useful to call the `new` method on `TwoDController` directly. However, you should call `init` from subclasses of `TwoDController` that override `init`, as described below, to properly initialize instances of the subclass.

init

`init` *self* [`targetCollection:`*sequence*] [`space:`*space*]
[`wholespace:`*boolean*] [`enabled:`*boolean*] ⇨ *(none)*

self `Controller` object

Superclasses `Controller` and `IndirectCollection` use the following keywords:

`targetCollection:`	`Sequence` object (use with caution)
`space:`	`Space` object
`wholespace:`	`Boolean` object
`enabled:`	`Boolean` object

Initializes the `TwoDController` object *self*, applying the values supplied with the keywords to instance variables of the same name. See the previous note about `targetCollection`. Do not call `init` directly on an instance—it is automatically called by the `new` method.

If you omit an optional keyword, its default value is used. The defaults are:

```
targetCollection:(new Array initialSize:14 growable:true)
space:undefined
wholeSpace:false
enabled:true
```

Class Methods

Inherited from `Collection`:

`pipe`

Instance Variables

Inherited from `Collection`:

`bounded`	`maxSize`	`size`
`iteratorClass`	`minSize`	`uniformity`
`keyEqualComparator`	`mutable`	`uniformityClass`
`keyUniformity`	`mutableCopyClass`	`valueEqualComparator`
`keyUniformityClass`	`proprietored`	

Inherited from `IndirectCollection`:

`targetCollection`

Inherited from `Controller`:

`enabled`	`space`	`wholeSpace`
`protocols`		

Instance Methods

Inherited from `Collection`:

`add`	`forEach`	`iterate`
`addMany`	`forEachBinding`	`localEqual`
`addToContents`	`getAll`	`map`
`chooseAll`	`getAny`	`merge`
`chooseOne`	`getKeyAll`	`pipe`
`chooseOneBinding`	`getKeyOne`	`prin`
`deleteAll`	`getMany`	`removeAll`
`deleteBindingAll`	`getOne`	`removeOne`
`deleteBindingOne`	`hasBinding`	`setAll`
`deleteKeyAll`	`hasKey`	`setOne`
`deleteKeyOne`	`intersects`	`size`
`deleteOne`	`isEmpty`	
`emptyOut`	`isMember`	

Inherited from `IndirectCollection`:

`isAppropriateObject`	`objectAdded`	`objectRemoved`

Inherited from `Controller`:

`isAppropriateObject`	`tickle`

Since a TwoDController object is an indirect collection, you can also use any methods defined in the class specified by targetCollection. The target collection is typically an instance of Array, which inherits from Sequence, so the following instance methods are redirected to a controller:

Accessible from LinearCollection:

chooseOneBackwards	findRange	getNthKey
chooseOrdOne	forEachBackwards	getOrdOne
deleteFirst	getFirst	getRange
deleteLast	getLast	localEqual
deleteNth	getMiddle	localLT
deleteRange	getNth	pop

Accessible from Sequence:

addFifth	moveBackward	setFourth
addFirst	moveForward	setLast
addFourth	moveToBack	setNth
addNth	moveToFront	setSecond
addSecond	prepend	setThird
addThird	prependNew	sort
append	setFifth	
appendNew	setFirst	

Subclasses Commonly Implement

TwoDController subclasses commonly implement the following methods:

```
enabledGetter
enabledSetter
isAppropriateObject
modelAdded
modelRemoved
objectAdded
objectRemoved
spaceGetter
spaceSetter
tickle
wholeSpaceGetter
wholeSpaceSetter
```

Accepting and Rejecting Events

If the TwoDController subclass can receive events, then it's important to implement the following so it can accept or reject events (accept events when enabled is set to true and reject events when enabled is set to false):

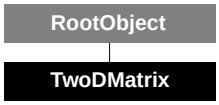
```
enabledGetter
enabledSetter
```

Control Space Dependencies

If the TwoDController subclass has anything dependent on a control space, such as an event interest, it's important to implement the following:

```
spaceGetter
spaceSetter
```

TwoDMatrix



Class type: Core class (concrete, sealed)
Resides in: ScriptX and KMP executables
Inherits from: `RootObject`
Component: 2D Graphics

The `TwoDMatrix` class provides normalized, affine matrices that can be used for linear transformations on two-dimensional coordinate systems. The `TwoDMatrix` class is used to define coordinate spaces for rendering operations. An introduction to transforming with matrices is provided in the "2D Graphics" chapter in the *ScriptX Components Guide.* This discussion explains some internal details of matrices.

There are three types of transformations you can perform in ScriptX through matrix multiplication: translation, scaling, and rotation.

- Translation moves a point to a new position in its coordinate system.
- Scaling expands or shrinks the points within a coordinate system relative to one another.
- Rotation causes a point to move to a new position, at a fixed distance from the axis of its current coordinate system but with new coordinates.

Matrix instances are implemented as arrays of six values. The contents of these arrays are represented by the `TwoDMatrix` instance variables: `a`, `b`, `c`, `d`, `tx`, `ty`. These variables define a normalized matrix M that can be represented as follows:

$$M = \begin{pmatrix} a & b & 0 \\ c & d & 0 \\ tx & ty & 1 \end{pmatrix}$$

If a normalized point P1 is defined as P1 = (x y 1), that point can be transformed to a new point P2 by the operation P1 x M. Another way of expressing this operation is:

P2 = (ax + cy + tx, bx + dy + ty, 1)

In this formula, x and y are the coordinates of point P1, and `a`, `b`, `c`, `d`, `tx`, and `ty` are the values in the matrix M. These values determine the type of operation that the matrix performs.

Translation is performed by adding the coordinates of the translation to the current coordinates of a point. X-axis translation is represented by the value of tx; y-axis translation is represented by the value of `ty`.

Scaling is performed by multiplying the current coordinates of a point. X-axis scaling is represented by the value of `a`, and y-axis scaling by the value of `d`.

Rotation is performed by applying the appropriate trigonometric functions to the coordinates of the point; the precise formulas are beyond the scope of this text.

All of these transformations can be performed through instance methods defined by the `TwoDMatrix` class. To perform a transformation, you usually begin with an identity matrix, available by calling `mutableCopy` on the global constant `identityMatrix`. You then use the appropriate methods on the new mutable matrix to perform the transformation. The order of the operations affects the outcome; a rotation followed by a translation produces a different effect than a translation followed by a rotation.

If you have the conceptual grounding to do so, you can perform other types of transformations, such as shear and non-affine transformations, directly on the elements in the array representing a matrix.

Once you have transformed the matrix, you can use it to affect a rendering operation. The `Stencil` method `transform` lets you apply a matrix directly to a stencil, while the `Surface` methods `fill` and `stroke` let you apply a matrix to position a stencil on the rendering surface.

Creating and Initializing a New Instance

The following script is an example of how to create a new instance of the `TwoDMatrix` class. It creates a matrix that translates all coordinates by 10 in both the x and y axes:

```
myMatrix := new TwoDMatrix \
              a:1 \
              b:0 \
              c:0 \
              d:1 \
              tx:10 \
              ty:10
```

The variable `myMatrix` contains a matrix. Its form is:

$$\begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 10 & 10 & 0 \end{pmatrix}$$

init

`init` *self* [`a:`*number*] [`b:`*number*] [`c:`*number*] [`d:`*number*] [`tx:`*number*] [`ty:`*number*] ⇨ *(none)*

self	`TwoDMatrix` object
`a:`	`Number` object
`b:`	`Number` object
`c:`	`Number` object
`d:`	`Number` object
`tx:`	`Number` object
`ty:`	`Number` object

Initializes the `TwoDMatrix` object *self*, applying the arguments as described in the introduction to this class description.

With none of the specified keyword arguments, the `new` method returns the global constant `identityMatrix` as an immutable matrix. However, if you specify any of the keywords, the resulting matrix will be mutable. Do not call `init` directly on an instance—it is automatically called by the `new` method.

Instance Variables

a

self.a (read-write) Number

Specifies the number in the first row, first column in the matrix represented by *self*. This value represents x axis scaling.

b

self.b (read-write) Number

Specifies the number in the first row, second column in the 2D matrix represented by *self*.

c

self.c (read-write) Number

Specifies the number in the second row, first column in the 2D matrix represented by *self*.

d

self.d (read-write) Number

Specifies the number in the second row, second column in the 2D matrix represented by *self*. This value represents y axis scaling.

tx

self.tx (read-write) Number

Specifies the number in the third row, first column in the 2D matrix represented by *self*. This value represents x axis translation.

ty

self.ty (read-write) Number

Specifies the number in the third row, second column in the 2D matrix represented by *self*. This value represents y axis translation.

Instance Methods

copy

copy *self* ⇨ TwoDMatrix

Creates and returns a matrix that is an immutable copy of the 2D matrix *self*.

invert

invert *self* ⇨ TwoDMatrix

Returns the inverse of the original 2D matrix *self*. Raises an exception undefinedResult and returns identityMatrix if the matrix cannot be inverted. Matrices constructed starting with the global constant identityMatrix and applying the standard TwoDMatrix transformation methods can always be inverted. Matrices constructed through the parameters of the new method may not be invertible.

mul

`mul` *self otherMatrix* ⇨ `TwoDMatrix`

self	`TwoDMatrix` object
otherMatrix	`TwoDMatrix` object

Returns the matrix that is the result of multiplying the 2D matrices *self* and *otherMatrix*. Matrix multiplication is performed with *self* on the left and *otherMatrix* on the right. If *self* is an immutable matrix (as is the global constant `identityMatrix`), this method creates a new instance and returns the results in that matrix. If *self* is mutable, the results are returned in *self*.

The `precat` instance method performs matrix multiplication in the opposite order of `mul`.

mutableCopy

`mutableCopy` *self* ⇨ `TwoDMatrix`

Creates and returns a matrix that is a mutable copy of the 2D matrix *self*.

precat

`precat` *self otherMatrix* ⇨ `TwoDMatrix`

self	`TwoDMatrix` object
otherMatrix	`TwoDMatrix` object

Returns the matrix that is the result of multiplying 2D matrices *self* and *otherMatrix*. Matrix multiplication is performed with *self* on the right and *otherMatrix* on the left. If *self* is an immutable matrix (as is the global constant `identityMatrix`), this method creates a new instance and returns the results in that matrix. If *self* is mutable, the results are returned in *self*.

The `mul` instance method performs matrix multiplication in the opposite order of `precat`.

reset

`reset` *self* ⇨ `TwoDMatrix`

Resets the 2D matrix *self* and returns an identity matrix. If the matrix is mutable, this method resets its elements to that of the identity matrix. If it isn't mutable, this method leaves *self* as is and returns the global constant `identityMatrix`.

rotate

`rotate` *self angle units* ⇨ `TwoDMatrix`

self	`TwoDMatrix` object
angle	`Number` object
units	`NameClass` object

Modifies the 2D matrix *self* so that points transformed by the matrix are rotated around the origin by *angle*. The interpretation of *angle* is controlled by the value of *units*, which may be `@degrees` or `@radians`. If *self* is an immutable matrix (as is the global constant `identityMatrix`), this method creates a new instance and returns the results in that matrix. If *self* is mutable, the results are returned in *self*.

scale

`scale` *self sx sy* ⇨ `TwoDMatrix`

self	`TwoDMatrix` object
sx	`Number` object
sy	`Number` object

Modifies the 2D matrix *self* so that points transformed by the matrix are scaled by the value of *sx* horizontally and *sy* vertically. If *self* is an immutable matrix (as is the global constant `identityMatrix`), this method creates a new instance and returns the results in that matrix. If *self* is mutable, the results are returned in *self*.

setTo

`setTo` *self otherMatrix* ⇨ `TwoDMatrix`

self	`TwoDMatrix` object
otherMatrix	`TwoDMatrix` object

If the 2D matrix *self* is mutable, this method does a matrix assignment and sets the instance variables of *self* to those of *otherMatrix*. If *self* isn't mutable, this method returns a mutable copy of *otherMatrix*.

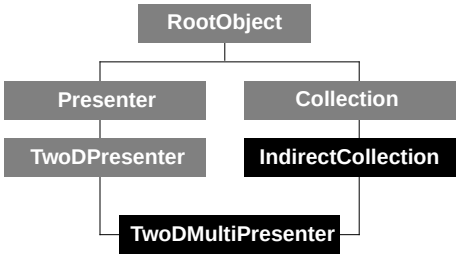
translate

`translate` *self x y* ⇨ `TwoDMatrix`

self	`TwoDMatrix` object
x	`Number` object
y	`Number` object

Modifies the 2D matrix *self* so that points transformed by the matrix are translated by *x* horizontally and *y* vertically. If *self* is an immutable matrix (as is the global constant `identityMatrix`), this method creates a new instance and returns the results in that matrix. If *self* is mutable, the results are returned in *self*.

TwoDMultiPresenter



Class type: Core class (concrete)
Resides in: ScriptX and KMP executables
Inherits from: `TwoDPresenter` and `IndirectCollection`
Component: Spaces and Presenters

The `TwoDMultiPresenter` class provides a way of presenting a collection of `TwoDPresenter` objects. Like many 2D presenters, an instance of `TwoDMultiPresenter` has a boundary with a fill and stroke that are displayed. The boundary forms a clipping region; it clips any subpresenters outside the boundary (Use `GroupPresenter` for a presenter that grows to include added presenters).

In addition, objects can be added to it, since it is a collection, and all of these objects are displayed. Members of the collection are presented because the `subpresenters` instance variable (inherited from `Presenter`) points to them, as shown in the following illustration. To simplify the diagram, not all instance variables are shown.

TwoDMultiPresenter
and its subclasses

Instance Variables:

height	20
width	50
position	100, 100
fill	whitePaint
stroke	blackPaint
presentedBy	undefined
subpresenters	●

Members:

myMulti[1]	car1
myMulti[2]	car2
myMulti[3]	tree
myMulti[4]	myMovie

The mechanism for maintaining subpresenters is straightforward. The `subpresenter` instance variable points to the members of the `TwoDMultiPresenter` object itself, which is some kind of collection. Therefore, any object added to the presenter is automatically in its list of subpresenters, where it becomes part of the presentation hierarchy, and subsequently drawn. In contrast, `OneOfNPresenter` keeps a list of potentially displayable presenters—only one presenter at a time is held in `subpresenter`.

`TwoDMultiPresenter` is useful when you need to manage multiple `TwoDPresenter` objects, but you don't want the overhead of the controllers and a clock that `TwoDSpace` provides.

For example, an instance of `TwoDMultiPresenter` might contain a collection of images of trees arranged as a forest, with larger trees in front and smaller trees receding in the back.

In general, presenters that inherit from TwoDMultiPresenter can have multiple subpresenters, while presenters that don't inherit from TwoDMultiPresenter do not. Because of this, presenters that inherit from TwoDMultiPresenter are called *containers*.

TwoDMultiPresenter provides the following management of its subpresenters:

- It sorts the subpresenters according to their z value, sorting larger z values to the front of the screen (the first item in the subpresenters list is frontmost on the screen). Notice that the z value actually rearranges the subpresenters within the list pointed to by the subpresenters. Although the z instance variable is defined in TwoDPresenter, only TwoDMultiPresenter sorts subpresenters by z value—TwoDPresenter does not.
- It checks to see if any presenter being added to its subpresenters is already being presented by another TwoDSpace object, if so, it removes that presenter from that other space. This prevents a presenter from being in two spaces at once.

The isAppropriateObject method ensures that any object added to the TwoDMultiPresenter is a kind of TwoDPresenter.

In the new and init methods, the target keyword is ignored—it is available for you to use in any subclass you create, as a target for a presenter. For example, you might save a series of numbers to the target, and use them to determine the heights of bars in a bar chart.

Creating and Initializing a New Instance

The following script is an example of how to create a new instance of the TwoDMultiPresenter class:

```
myMulti := new TwoDMultiPresenter \
              fill: whiteBrush \
              stroke: blackBrush \
              boundary: (new Rect x2:20 y2:100)
```

The variable myMulti contains the initialized 2D multipresenter of size 20 x 100 pixels, with a white fill and a black outline. The new method uses the keywords defined in init.

For performance reasons, when creating an instance of TwoDMultiPresenter or any of its subclasses (such as TwoDSpace, GroupSpace, Window, or PageLayer), you should not specify the targetCollection keyword, so it can create its default collection. Such presenters require collections that are fast to traverse when drawing and handing events. Performance could suffer if you change the value of targetcollection to something other than the default.

Important – It is best for the boundary of a 2D multipresenter to be either an instance of Rect or Region. If it is any other kind of stencil, then there will be garbage generated during each draw of the 2D multipresenter which needs to be garbage collected, causing slower performance.

---

init

`init` *self* [`boundary:`*stencil*] [`fill:`*brush*] [`stroke:`*brush*] [`target:`*object*]
[`targetCollection:`*sequence*] ⇨ *(none)*

self	TwoDMultiPresenter object
fill:	Brush object
stroke:	Brush object

Superclasses of TwoDMultiPresenter use the following keywords:

boundary:	Stencil object
target:	(Ignored by TwoDMultiPresenter)
stationary:	Boolean object
targetCollection:	Sequence object

Initializes the TwoDMultiPresenter object *self*, applying the values supplied with the keywords to instance variables of the same name. The target keyword is ignored by TwoDMultiPresenter, but you can use it in a subclass. Do not call init directly on an instance—it is automatically called by the new method.

If you omit an optional keyword, its default value is used. The defaults are:

```
fill:undefined
stroke:undefined
target:undefined
stationary:false
boundary:(new Rect x1:0 y1:0 x2:0 y2:0)
targetCollection:(new Array initialSize:14 growable:true)
```

Class Methods

Inherited from Collection:

pipe

You can also use any class methods defined in the class specified by targetCollection.

Instance Variables

Inherited from Collection:

bounded	maxSize	size
iteratorClass	minSize	uniformity
keyEqualComparator	mutable	uniformityClass
keyUniformity	mutableCopyClass	valueEqualComparator
keyUniformityClass	proprietored	

Inherited from IndirectCollection:

Inherited from Presenter:

presentedBy	subPresenters	target

Inherited from TwoDPresenter:

bBox	height	transform
boundary	IsImplicitlyDirect	width
clock	isTransparent	window
compositor	isVisible	x
direct	needsTickle	y
eventInterests	position	z
globalBoundary	stationary	
globalTransform	target	

The following instance variables are defined in TwoDMultiPresenter:

fill

self.fill (read-write) Brush

Specifies the Brush object for rendering the background of the 2D multipresenter *self*. This brush fills the boundary stencil before any contained objects are asked to draw. If the value of fill is undefined, the background is transparent. Its default value for TwoDMultiPresenter objects is undefined.

stroke

self.stroke (read-write) Brush

Specifies the Brush object used for the outline of the 2D multipresenter *self*. The stroking of the outline occurs after it has been filled, but before all contained objects have been rendered. If subpresenters overlap the outline, that part of the outline is overwritten by subpresenters. If the value of stroke is undefined, the border of the TwoDMultiPresenter object is invisible.

Instance Methods

Inherited from Collection:

add	forEach	iterate
addMany	forEachBinding	localEqual
addToContents	getAll	map
chooseAll	getAny	merge
chooseOne	getKeyAll	pipe
chooseOneBinding	getKeyOne	prin
deleteAll	getMany	removeAll
deleteBindingAll	getOne	removeOne
deleteBindingOne	hasBinding	setAll
deleteKeyAll	hasKey	setOne
deleteKeyOne	intersects	size
deleteOne	isEmpty	
emptyOut	isMember	

Inherited from IndirectCollection:

isAppropriateObject	objectAdded	objectRemoved

Inherited from TwoDPresenter:

adjustClockMaster	inside	show
createInterestList	localToSurface	surfaceToLocal
draw	notifyChanged	tickle
getBoundaryInParent	recalcRegion	
hide	refresh	

Since a TwoDMultiPresenter object is an indirect collection, you can also use any methods defined in the class specified by targetCollection. The target collection is typically an instance of Array, which inherits from Sequence, so the following instance methods are redirected to an instance of TwoDMultiPresenter:

Accessible from LinearCollection:

chooseOneBackwards	findRange	getNthKey
chooseOrdOne	forEachBackwards	getOrdOne
deleteFirst	getFirst	getRange
deleteLast	getLast	localEqual
deleteNth	getMiddle	localLT
deleteRange	getNth	pop

Accessible from Sequence:

addFifth	moveBackward	setFourth
addFirst	moveForward	setLast

addFourth	moveToBack	setNth
addNth	moveToFront	setSecond
addSecond	prepend	setThird
addThird	prependNew	sort
append	setFifth	
appendNew	setFirst	

The following instance methods are defined in `TwoDMultiPresenter`:

draw (TwoDPresenter)

`draw` *self surface clip* ⇨ *(none)*

self	`TwoDMultiPresenter` object
surface	`DisplaySurface` object
clip	`Region` object for parent's clip area

Tells the 2D multipresenter *self* to render its image onto the display *surface*, with clipping defined by the stencil *clip*. The `draw` method first fills the presenter, using the brush defined by `fill`. Then it strokes the presenter's outline, using the brush defined by `stroke`. Subpresenters are rendered last, in a higher layer than the 2D multipresenter's own fill and stroke. This means that a subpresenter that overlaps the outline overwrites that part of the outline.

Although `draw` can be called directly, it is usually called by the 2D compositor, which knows what surface the presenter should be rendering onto. It does not draw deep. Each draw method that is implemented by subclasses of `TwoDMultiPresenter` draws the presenter itself. The compositor iterates through all presenters in a presentation hierarchy in an orderly manner, calling `draw` on each contained presenter.

When drawing, with any rendering commands (`fill`, `stroke`, or `draw`), you should use `globalBoundary` as their clip argument.

findAllAtPoint

`findAllAtPoint` *self point* ⇨ `Array`

self	`TwoDMultiPresenter` object
point	`Point` object

Returns a list of objects within the 2D multipresenter *self* that intersect the given point *whereAt*. Specify the point in the local coordinates of the 2D multipresenter.

findAllInStencil

`findAllInStencil` *self point* ⇨ `Array`

self	`TwoDMultiPresenter` object
point	`Stencil` object

Returns a list of objects within the 2D multipresenter *self* that intersect the area defined by *stencil*.

findFirstAtPoint

`findFirstAtPoint` *self point* ⇨ *(object)*

self	`TwoDMultiPresenter` object
point	`Point` object

Returns the frontmost object within the 2D multipresenter *self* that intersects the given point *whereAt*. Specify the point in the local coordinates of the 2D multipresenter.

findFirstInStencil

`findFirstInStencil` *self stencil* ⇨ *(object)*

self	`TwoDMultiPresenter` object
stencil	`Stencil` object

Returns the front-most object within the 2D multipresenter *self* that intersects the area defined by *stencil*.

isAppropriateObject (IndirectCollection)

`isAppropriateObject` *self addedObject* ⇨ `Boolean`

self	`TwoDMultiPresenter` object
addedObject	Any object

This method is automatically called when the object *addedObject* is added to the 2D multipresenter *self*. It performs any checking that needs to be done before an object is actually added to the `TwoDMultiPresenter` object. If it returns `false` or throws an exception, the object is not added.

This method performs the following test: if the object being added is already a member of the `TwoDMultiPresenter` object, it returns `false`, preventing the object from being added; otherwise, it returns `true`, allowing the object to be added.

Incidentally, `isAppropriateObject` is called by the method you use to add an object to the `TwoDMultiPresenter` object *self*, such as `setOne` or `setNth`. These add methods are defined by `IndirectCollection`.

moveBackward (Sequence)

`moveBackward` *self objectToMove* ⇨ `Boolean`

self	`TwoDMultiPresenter` object
objectToMove	`TwoDPresenter` object

Moves the object *objectToMove* backward one position in the subpresenters list of the 2D multipresenter *self*, by swapping with its neighbor. This method will change the value of `z`, if necessary, to move the object. It returns `false` if the 2D presenter is already at the back.

moveForward (Sequence)

`moveForward` *self objectToMove* ⇨ `Boolean`

self	`TwoDMultiPresenter` object
objectToMove	`TwoDPresenter` object

Moves the object *objectToMove* forward one position in the subpresenters list of the 2D multipresenter *self*, by swapping with its neighbor. This method will change the value of `z`, if necessary, to move the object. It returns `false` if the 2D presenter is already at the front.

moveToBack (Sequence)

`moveToBack` *self objectToMove* ⇨ `Boolean`

self	`TwoDMultiPresenter` object
objectToMove	Any object

Moves the object *objectToMove* to the back of the subpresenters list of the 2D multipresenter *self*. This method will change the value of `z`, if necessary, to move the object. It returns `false` if the 2D presenter is already at the back.

moveToFront (Sequence)

`moveToFront` *self objectToMove* ⇨ `Boolean`

self	`TwoDMultiPresenter` object
objectToMove	Any object

Moves the object *objectToMove* to the front of the subpresenters list of the 2D multipresenter *self*. This method will change the value of `z`, only if necessary, to move the object. It returns `false` if the 2D presenter is already at the front.

objectAdded (IndirectCollection)

`objectAdded` *self addedObject* ⇨ *(none)*

self	`TwoDMultiPresenter` object
addedObject	Any object

This method automatically gets called when the object *addedObject* has been accepted by `isAppropriateObject` and is added to the 2D multipresenter *self*. This method removes *addedObject* from any other subpresenters list it might be in, ensuring that the object appears only once in the presentation hierarchy. The `objectAdded` method also sets the `presentBy` instance variable of the newly added object to the 2D multipresenter *self*.

In addition, the `objectAdded` method makes sure the newly added item is inserted in its proper place in the subpresenters list, as sorted by its z value.

Incidentally, the chain of calling `isAppropriateObject` and `objectAdded` begins when you call a method to add an object to the 2D multipresenter *self*, such as `setOne` or `setNth`. These collection methods are defined by `IndirectCollection`, which redirects them to the target collection.

You can specialize this method in a subclass of `TwoDMultiPresenter` to perform any action you want to occur every time an object is added.

objectRemoved (IndirectCollection)

`objectRemoved` *self removedObject* ⇨ *(none)*

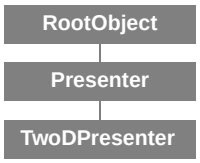
self	`TwoDMultiPresenter` object
removedObject	Any object

This method is called automatically when the object *removedObject* is removed from 2D multipresenter *self*. `TwoDMultiPresenter` specializes `objectRemoved` to call `refreshRegion` on the 2D compositor associated with the 2D multipresenter *self*. This method also sets the `presentBy` instance variable of the removed object to `undefined`.

Incidentally, `objectRemoved` is called when you call a method to remove an object from the 2D multipresenter *self*, such as `deleteOne` or `deleteNth`. These collection methods are defined by `IndirectCollection`, which redirects them to the target collection.

You can specialize this method in a subclass of `TwoDMultiPresenter` to perform any action you want to occur every time an object is removed.

TwoDPresenter



Class type: Core class (abstract)
Resides in: ScriptX and KMP executables
Inherits from: `Presenter`
Component: Spaces and Presenters

The `TwoDPresenter` class represents two-dimensional graphic presenters that can draw themselves onto display surfaces. Every `TwoDPresenter` object implements the `draw` method. A 2D compositor calls the presenter's `draw` method to render the image on the display surface. `TwoDPresenter` objects use the stencil and brush graphic primitives provided in the 2D Graphics component for drawing geometric or bitmap images.

Many classes in the core classes inherit from `TwoDPresenter`, and thus are capable of graphic presentation.

- `Document`, `Page`, `PageTemplate`, `PageElement`, and `PageLayer` (Document Templates component)
- `Window`, `TwoDSpace`, `TwoDShape`, `GroupPresenter`, `GroupSpace`, `CostumedPresenter`, and `OneOfNPresenter` (Spaces and Presenters component)
- `PushButton`, `Toggle`, `Menu`, `ScrollBar`, and `ScrollingPresenter` (User Interface component)
- `TextPresenter` and `TextEdit` (Text and Fonts component)
- `MoviePlayer`, `DigitalVideoPlayer`, and `InterleavedStreamPlayer` (Media Players component)
- `TransitionPlayer`, `BarnDoor`, `Iris`, `Slide`, `Wipe` (Transitions component)

These presenters come in two varieties: "containers" which can hold and display other presenters, and "simple" presenters which display only one presenter. Examples of container presenters include `TwoDMultiPresenter`, `GroupPresenter`, `TwoDSpace` and `Window`. Single presenters include `TwoDShape`, `OneOfNPresenter`, `TransitionPlayer`, and `Page`. The container presenters have a stroke and background fill. The `subpresenters` instance variable determines which additional objects are to be presented, and in what order.

The `TwoDPresenter` class is abstract—the `draw` method defined by `TwoDPresenter` is only a placeholder. It does not draw subpresenters, nor does it sort subpresenters. This behavior is implemented by `TwoDMultiPresenter`. Each concrete subclass of `TwoDPresenter` should implement a method for `draw` to draw any subpresenters it might have.

Each concrete subclass of `TwoDPresenter` manages how objects are stored in its subpresenters list. For example, for `TwoDMultiPresenter` and its subclasses, all of the objects it contains are subpresenters, and are displayed together. However, for `OneOfNPresenter`, only one object is moved into the subpresenters list at a time, and only that object is displayed.

If you create a subclass of `TwoDPresenter` rather than `TwoDMultiPresenter`, your program must create an array and set the value of `subPresenters` to reference this array. The relationship between this array and the instance variable `presentedBy` must be managed explicitly by your program. As presenters are added to the array, set the value of `presentedBy` to point to the parent presenter.

For further description of subpresenters, see the "Spaces and Presenters" chapter in the *ScriptX Components Guide*.

Creating and Initializing a New Instance

Because `TwoDPresenter` is an abstract class, you cannot create an instance of `TwoDPresenter`, nor is it useful to call the `init` method on `TwoDPresenter` directly. However, you should call `nextMethod` from any subclass of `TwoDPresenter` that overrides `init`, to initialize instances of the subclass.

init

`init` *self* [`target:`*object*] [`boundary:`*stencil*] [`stationary:`*boolean*] ⇨ *(none)*

self	`TwoDPresenter` object
`target:`	Any object the presenter displays
`boundary:`	`Stencil` object that determines the shape of the presenter
`stationary:`	`Boolean` object

Initializes the `TwoDPresenter` object *self*, applying the keywords to instance variables of the same name. Different subclasses of `TwoDPresenter` use `target` and `boundary` for different purposes. Some subclasses of `TwoDPresenter` ignore these keywords. Do not call `init` directly on an instance—it is automatically called by the `new` method.

If you omit an optional keyword, its default value is used. The defaults are:

```
target:undefined
boundary:(new Rect x1:0 y1:0 x2:0 y2:0)
stationary:false
```

Instance Variables

Inherited from `Presenter`:

`presentedBy` `subPresenters` `target`

The following instance variables are defined in `TwoDPresenter`:

bBox

self`.bBox` (read-only) `Rect`

Specifies the smallest rectangle that fully encloses the 2D presenter *self*. The term `bBox` is an abbreviation for "bounding box."

boundary

self`.boundary` (read-write) `Stencil`

A stencil that describes the boundary of the 2D presenter *self*. If the presenter is a `TwoDMultiPresenter` or `TwoDSpace` object, this boundary is also used for clipping. As a shortcut, the `height` and `width` instance variables allow you to change the size of the boundary. The `boundary` instance variable is read-only for certain subclasses, for example where the boundary is intrinsic to the `TwoDPresenter` object, such as with movies, and for `GroupSpace` and `GroupPresenter`, whose boundaries are the unions of their subpresenters.

clock

self.clock (read-only) Clock

Specifies the clock which refreshes the 2D presenter *self*. The value of this instance variable is undefined for most 2D presenters—only presenters inheriting from the Space class automatically have clocks, including TwoDSpace, PageLayer, GroupSpace, Window and the subclasses of Window.

When a presenter has a clock, that clock is automatically slaved to the next clock up the presentation hierarchy (using adjustClockMaster). Slaving clocks together makes it easy to pause and resume all clocks in that space.

See the description of clock in the Space class for further details.

compositor

self.compositor (read-only) TwoDCompositor

Specifies the compositor used to draw the 2D presenter *self*. This instance variable is implemented as a real (slot-based) instance variable, and its value is set automatically when the presenter *self* is added to the presentation space.

direct

self.direct (read-write) Boolean

When true, specifies that the 2D presenter *self* render directly to a rectangular area of the screen, rather than through the off-screen frame buffer. This allows performance improvements for presenters such as video that have quickly changing images. The default value for direct for new presenters is false. This instance variable can be changed at any time.

A presenter whose direct instance variable is true is called a direct presenter. A direct presenter takes over a rectangular region of the display surface and can draw as little or as much of the area as is needed, with the assurance that the image hasn't changed since the last time it drew. For example, a digital video presenter or transition can do "incremental" updates at each frame, just changing the bits of the surface that are old.

A direct presenter's rectangle is visually in front of all other presenters on that display surface. This occurs because direct presenters are drawn to the display surface after all non-direct presenters. Note that subpresenters of a direct presenter are implicitly direct.

Since it is not drawn to the offscreen buffer, the direct presenter has control of the region within its bounding box. Two direct presenters should not overlap on the display surface, because that can cause flickering as both presenters struggle to update the overlapped area. A direct presenter skips the offscreen composition phase of the compositor's cycle entirely. Even if a direct presenter has other presenters overlapping in front of it in the presentation hierarchy, it is still drawn in front.

eventInterests

self.eventInterests (read-only) KeyedLinkedList

Specifies event interests in the 2D presenter *self*. For example, it can contain an interest in a mouse button being pressed down (MouseDownEvent) while the mouse pointer is within the boundary of the 2D presenter. This instance variable returns undefined until an interest has been added to it.

The advantage of storing interests in this list is that they travel with the presenter wherever it goes, in and out of spaces, and the presenter can respond to events only within its boundary. You cannot directly add or remove items from the list—use addEventInterests, defined in the Events class, for this.

This list has exactly the same features as the `interests` instance variable in the `Event` class. That is, interests are arranged by priority, and within a priority, most-recent interests are accessed first. (The details of how interests are stored within this keyed linked list are determined by efficiency and memory considerations.)

Interests other than mouse event interests can be stored here also. For example, if you created a new kind of event to support the pressure from a pen device, you could store interests in those events in this list.

globalBoundary

self`.globalBoundary` (read-only) `Rect`

Specifies the clip rectangle used to define the clipping area when rendering the 2D presenter *self*.

This rectangle is calculated when `notifyChanged` is called, and is used by the compositor to determine which parts of the window should be redrawn—only presenters that intersect the changed region are redrawn. It can be out of date if something on the presenter (such as its location) has changed but `notifyChanged` has not yet run to update the `globalBoundary`.

Do not use this instance variable as the *clip* argument to fill, stroke or draw any subpresenters of *self*. Instead you should use the clip that is passed in to the `draw` method, and you can additionally reduce this clipping area if desired.

globalTransform

self`.globalTransform` (read-only) `TwoDMatrix`

Specifies the fully-concatenated transformation matrix for the 2D presenter *self*. This matrix determines where the 2D presenter is located in the "global" coordinates of the window that the presenter is displayed in.

You cannot reliably use the `TwoDMatrix` referenced by `globalTransform` to transform points or stencils to other presenter coordinate systems. You must use `surfaceToLocal` and `localToSurface`, instance methods defined by `TwoDPresenter`.

height

self`.height` (read-write) `Float`

Changes the height of the boundary of the presenter *self*. This is a shortcut for setting a new value for `boundary`.

isImplicitlyDirect

self`.isImplicitlyDirect` (read-only) `Boolean`

Indicates that the `TwoDPresenter` instance *self* is drawn, either explicitly or implicitly, as a direct presenter. When a presenter is a direct presenter (the value of `direct` is `true`), all its subpresenters are drawn as if they are direct presenters too. This property returns `true` if a presenter or any of its superpresenters is a direct presenter.

isTransparent

self`.isTransparent` (read-write) `Boolean`

Specifies whether the `TwoDPresenter` instance *self* is transparent. A presenter is transparent if it does not render all of the pixels encompassed by its boundary.

Use the `isTransparent` property, together with `stationary`, as a hinting mechanism to speed drawing by the compositor. It is used to speed drawing only when the `TwoDPresenter` instance *self* is a stationary presenter (the value of `stationary` is `true`). When the presenter *self* is stationary, the compositor uses `isTransparent` to determine whether or not to render objects that are occluded by the presenter.

The default value for `isTransparent` is `false`.

isVisible

self`.isVisible` (read-only) `Boolean`

When `true`, the 2D presenter *self* is shown; when `false`, it is hidden. Although `isVisible` is a read-only instance variable, you can set it indirectly by calling the methods `show` and `hide`.

Note that hiding an object stops drawing and composition, but does not stop time. If the presenter is a window, its clock continues running; however, the window's compositor is disabled when `isVisible` is set to `false`.

needsTickle

self`.needsTickle` (read-write) `Boolean`

Indicates whether the 2D presenter *self* needs to have its `tickle` method called with each composite cycle.

The ScriptX 2D graphics compositor does not automatically poll presenters in the space. Presenters are "tickled" only if they register with the compositor. The compositor maintains a private list of presenters that need to be tickled, and any presenter can be added to this tickle list by setting the value of `needsTickle` to `true`. See also `tickle`.

position

self`.position` (read-write) `Point`

Specifies the x-y location of the 2D presenter *self* in its parent space, or, if it is the top presenter, in the display surface. The `position` variable is merely another way of specifying the `x` and `y` instance variables. You would specify `position`, rather than `x` and `y`, in a script when changing both values, to be sure both `x` and `y` take effect at the same compositor cycle. If `x` and `y` were set separately, the 2D presenter could move in two steps, as a shift in `x` and a shift in `y`.

You can also extract `position` from the `TwoDMatrix` object specified by `transform`.

stationary

self`.stationary` (read-write) `Boolean`

When set to `true`, specifies that the `TwoDPresenter` instance *self* is large, that it doesn't move much, or both. By default the value of `stationary` is `false`.

Use the `stationary` property as a hinting mechanism to speed drawing by the compositor. When the value of stationary is `true`, presenters that lie beneath the `TwoDPresenter` instance *self*, and are completely occluded by it, are not redrawn with each compositor cycle.

A space within another space, such as a `TwoDSpace` object with its own controllers and model objects, is a good candidate for presentation as a stationary presenter. The stationary presenter controls the area within its bounding box, just as the direct presenter does, saving compositor time by preventing the compositor from rendering objects that it occludes (provided that the value of `isTransparent` is `false`). Since the space within a space still needs to use offscreen drawing for its own model objects, use of a direct presenter is not appropriate.

Subpresenters of a stationary presenter are not stationary. In this respect, a stationary presenter differs from a direct presenter. Subpresenters of a stationary presenter can be drawn to an offscreen frame buffer, for smooth animation performance. By contrast, subpresenters of a direct presenter are implicitly direct, and cannot use the compositor's off-screen buffer.

target (Presenter)

self.`target` (read-write) *(object)*

Specifies the object that the 2D presenter *self* is presenting. The target need not be a presenter. When a `target` is not needed, its value should be `undefined`. The implementation for `target` is inherited from `Presenter` and specialized with one difference: when you set this value, the `targetSetter` method defined by `TwoDPresenter` calls `notifyChanged`.

For example, an instance of `String`, `StringConstant`, or `Text` is the target for a `TextPresenter` object. By itself, an instance of `Text` cannot draw—it requires a presenter such as `TextPresenter` to display it.

transform

self.`transform` (read-write) `TwoDMatrix`

Specifies the `TwoDMatrix` object that determines the x,y location of 2D presenter *self*, as well as its scaling and rotation, if any. To interpret this `TwoDMatrix` object, see the `TwoDMatrix` class.

Note for Current Release – Rotation is not supported for `TwoDPresenter` in this release. You can perform the same visual effect, rotating a `TwoDShape` object, by applying `rotate` to its 2D matrix.

width

self.`width` (read-write) `Float`

Changes the width of the boundary of the presenter *self*. This is a shortcut for setting a new value for `boundary`.

window

self.`window` (read-only) `Window`

Specifies the window that the 2D presenter *self* is in. If the presenter is not in a window, `window` returns `undefined`.

x

self.`x` (read-write) `Float`

Specifies the x value in the transformation matrix, and, thus, the x position of the 2D presenter *self* in coordinates relative to its most immediate container. Thus, if the presenter is in a 2D space within a window, its x-value is relative to the 2D space. This value can also be extracted from the `TwoDMatrix` object specified by `transform`.

y

self.`y` (read-write) `Float`

Specifies the y value in the transformation matrix, and, thus, the y position of the 2D presenter *self* in coordinates relative to its most immediate container. This value can also be extracted from the `TwoDMatrix` object specified by `transform`.

z

self.z (read-write) Integer

Specifies the front-to-back display position of the 2D presenter *self*, relative to other overlapping presenters within their parent TwoDMultiPresenter container. If the 2D presenter *self* is not contained in an instance of TwoDMultiPresenter or one of its subclasses, z is ignored.

The value of z can be any integer from –2,147,483,647 to +2,147,483,648. The default is 0. A higher positive z value corresponds to a position towards the front of the display.

To understand how setting z changes the display order of presenters, you need to understand subpresenters lists. All presenters within a TwoDMultiPresenter container appear in that container's subpresenters list. A presenter at the front of the list (that is, with a smaller index number) displays in front of a presenter at the back of the list. This is because within a subpresenters list, the presenter at the bottom of the list (the largest index number) draws first, and subsequent presenters draw on top of it.

To change the position of a presenter in the front-to-back order, you can specify a different value for the z instance variable. Setting this value actually moves the presenter in the subpresenters list, so that once again the order of presenters in the list determines their draw order. A higher, more positive z value corresponds to a position in front of other presenters.

The following table shows the relationship between index numbers in a subpresenters list, z-numbers, and visual ordering:

Table 12: Index numbers, z-numbers, and visual ordering

Index in subpresenters lists:	1	2	3...	n
z-number:	high	medium		low
Visual ordering:	front	middle		back

For example, setting the z-value to 1 moves a presenter in the list ahead of all presenters with a z-value of 0, and consequently, it displays in front of them.

Unless you specify otherwise, all presenters have a z-value of 0. If you do not explicitly specify a z-value, the front-to-back order of subpresenters is determined by their order in the list of subpresenters. To add a new presenter in front of other subpresenters, you call prepend; to add it behind other subpresenters, you call append.

To learn more about subpresenters, see the chapter "Spaces and Presenters" in the *ScriptX Components Guide*.

Instance Methods

adjustClockMaster

adjustClockMaster *self oldMasterClock newMasterClock* ⇨ *(none)*

self	TwoDPresenter object
oldMasterClock	Clock object that *self* might have been slaved to
newMasterClock	Clock object to become slaved to

If the clock of the 2D presenter self was slaved off *oldMasterClock*, then this method causes it to become slaved off *newMasterClock*; otherwise, no change is made to this clock. If this clock is a top clock, no change is made. In any case, this same method is called recursively on all subpresenters of *self* to adjust their master clocks.

When you change a presenter's position in a presentation hierarchy, if the presenter has a clock, this method changes the clock's position in its timing hierarchy. This ensures that all clocks in the presentation hierarchy are slaved together. When a presenter's position in a presentation hierarchy is changed, the `presentedBySetter` method is called, which automatically calls `adjustClockMaster`. The `presentedBySetter` method is the setter method for the `presentedBy` instance variable.

A presenter is considered to have a clock if it has a `clock` instance variable that holds a clock. Any subclass of the `Space` class can have a clock.

For example, say you have a 2D space, `mySpace`, that you want to move from one window to another. When you move it, the `adjustClockMaster` method is automatically called; if `mySpace.clock` was slaved off the clock of the old window, then this method causes it to be slaved off the clock of the new window. However, if the clock was originally not slaved off the clock of the old window, then it is not disturbed. This latter condition occurs if you manually set the clock's master to other than its default.

createInterestList

`createInterestList` *self* ⇨ `KeyedLinkedList`

This method should be used only when creating new kinds of event interests, such as those for a new kind of device, like a pen device. You call this method on the 2D presenter *self* only if `eventInterests` is `undefined` on that 2D presenter. Calling `createInterestList` on the presenter creates the `eventInterest` instance variable in the presenter. This gives the presenter a place to hold the event interests. (Note that `eventInterests` is a read-only instance variable.)

The `eventInterests` instance variable is normally created by invoking the method `addEventInterest` in a subclass of `Event`, such as `MouseDownEvent`, that stores event interests on presenters.

draw

`draw` *self surface clip* ⇨ *self*

self	`TwoDPresenter` object to draw
surface	`DisplaySurface` or `BitmapSurface` object to draw to
clip	`Stencil` object for parent's clip area

Tells the 2D presenter *self* to render its image onto the display *surface*, with clipping defined by the stencil *clip*. The compositor calls `draw` automatically on each presenter, telling the presenter to render itself to the compositor's display surface. For special features, you can explicitly call this method on either a display surface, for on-screen drawing, or on a bitmap surface, for off-screen drawing. The generic function `draw` has no implementation in `TwoDPresenter`; therefore, a draw method must be implemented by its subclasses.

As implemented on subclasses of `TwoDPresenter`, draw methods do not draw deep. Invoking `draw` causes a presenter to draw itself; it does not call `draw` on its subpresenters.

To draw a presenter, you can use the value of `globalBoundary` as the clip argument for any rendering commands (`fill`, `stroke`, or `draw`).

getBoundaryInParent

`getBoundaryInParent` *self* ⇨ `Stencil`

Returns the boundary of the 2D presenter *self* in its parent's coordinates. The parent is specified by *self*`.presentedBy`. The parent is the container holding the presenter, typically an instance of `Window`, `TwoDSpace`, `TwoDMultiPresenter`, `GroupPresenter`, or `GroupSpace`.

For example, if the 2D presenter is a `Rect` object with coordinates 0,0,10,10 and located in a parent 2D space at 20,20, this method would return a `Rect` object with coordinates 20,20,30,30.

hide

`hide` *self* ⇨ *self*

Hides the 2D presenter *self*. This sets `isVisible` to `false`. When you hide a 2D presenter that has been added to a window, the object maintains its place in the window's presentation hierarchy—the object is just not drawn. Because the object is still a member of the window, the garbage collecter will not remove it from memory, as long as the window is still showing. However, hiding a window is necessary, but not sufficient, for any presenter to be garbage collected.

When hiding a window, its clock continues running; however, `enabled` is set to `false` in the window's compositor. Calling `hide` on a window or other 2D multipresenter causes it to call `hide` on all of its subpresenters.

inside

`inside` *self point* ⇨ `Boolean`

self	`TwoDPresenter` object
point	`Point` object

Tests whether the `Point` object *point* is inside the image area of the presenter *self*, returning `true` if the point is inside *self* and `false` otherwise. The `inside` method first performs a less costly test on the presenter's bounding box. If that test succeeds, it then tests the presenter's real boundary. The coordinate system of the point is that of the presenter *self*. If the presenter is a rectangle with dimensions h and w, the point at 0,0 would return `true`, and the point at h,w would return `false`.

localToSurface

`localToSurface` *self point mode* ⇨ `Point`

self	`TwoDPresenter` object
point	`Point` or `Rect` object
mode	`NameClass` object, either `@mutate` or `@create`

Transforms a point from local to display coordinates. More specifically, given a point or a rectangle measured in the presenter's own coordinate system, this method transforms that object to the coordinate system of the display surface that the presenter *self* is displayed on. This display surface is accessible, on the presenter *self*, through the presenter's compositor (*self*`.compositor.displaySurface`). The presenter's own coordinate system has its origin at the upper left corner of its bounding box.

The object returned depends on the value of *mode*. If that value is `@create`, then `localToSurface` creates a copy of *point*, and then transforms and returns the copy. If that value is `@mutate`, `surfaceToLocal` transforms and returns the original `Point` object.

For an explanation of coordinate systems in ScriptX, see the "Spaces and Presenters" chapter of the *ScriptX Components Guide*. Note that the class `FullScreenWindow` differs from other windows in that display and window coordinates do not coincide.

Note – In version 1.5 of ScriptX, `surfaceToLocal` and `localToSurface` replace `windowToLocal` and `localToWindow`, which have been eliminated.

notifyChanged

`notifyChanged` *self positionChanged* ⇨ *self*

self	`TwoDPresenter` object
positionChanged	`Boolean` object

Causes the `TwoDPresenter` object *self* to compute its changed region and pass this region on to the compositor by calling `refreshRegion`, an instance method defined by `TwoDCompositor`.

The second argument, *positionChanged*, is an optimization hint for the compositor. If the presenter's position or boundary has changed, pass the value `true`. If its position or boundary has not changed and only the image has changed, then pass `false` as the second argument. If only the image has changed, and not the boundary or position, then the compositor can skip several steps and render the presenter more efficiently.

Since `notifyChanged` is a costly operation, it should be called only as needed, ideally just once in each composite cycle. Calls to `notifyChanged` are less costly if the second argument is `false`.

recalcRegion

`recalcRegion` *self* ⇨ `Region`

Recalculates the values of `globalTransform` and `globalBoundary`. Do not call `recalcRegion` from the scripter. It is called automatically when `notifyChanged` is called with its second argument set to `true`. The second argument to `notifyChanged` is a flag that indicates whether the position or boundary has changed. Specialize `recalcRegion` when you want to trap for or be notified of changes to the presenter's position or boundary.

refresh

`refresh` *self surface clip* ⇨ *(none)*

self	`TwoDPresenter` object that is a direct presenter
surface	`DisplaySurface` object to draw to
clip	`Stencil` object for parent's clip area

Redraws the direct presenter *self*. The compositor calls this method on direct presenters only when there is some area of the display surface that has been erased (for example, by another window moving in front and then moving away). The direct presenter should respond to this method by redrawing its entire image area—that is, by doing not just an incremental update, but a complete update. The default behavior of `refresh` is to call `draw` on *self*.

show

`show` *self* ⇨ *self*

Shows the 2D presenter *self*. Has the same effect as setting `isVisible` to `true`. Calling `show` on a window or other 2D multipresenter causes it to call `show` on all of its subpresenters.

surfaceToLocal

`surfaceToLocal` *self point mode* ⇨ `Point`

self	`TwoDPresenter` object
point	`Point` or `Rect` object
mode	`NameClass` object, either `@mutate` or `@create`

Transforms a point from display to local coordinates. More specifically, given a point or rectangle measured in the coordinate system of the presenter's display surface, this method transforms that object to the presenter's own coordinate system. The presenter's coordinate system has its origin at the upper left corner of its bounding box.

The object returned depends on the value of *mode*. If that value is `@create`, then `surfaceToLocal` creates a copy of *point*, and then transforms and returns the copy. If that value is `@mutate`, `surfaceToLocal` transforms and returns the original `Point` object.

For an explanation of coordinate systems in ScriptX, see the "Spaces and Presenters" chapter of the *ScriptX Components Guide*. Note that the class `FullScreenWindow` differs from other windows in that display and window coordinates do not coincide.

Note – In the current version of ScriptX, `surfaceToLocal` and `localToSurface` replace `windowToLocal` and `localToWindow`, which have been eliminated.

---

tickle

`tickle` *self clock* ⇨ *self*

Do not call `tickle` directly from the scripter. It is invoked automatically by the compositor, through a callback on the space's clock.

Causes the `TwoDPresenter` object *self* to perform some action each time the compositor runs. The `TwoDPresenter` class does not actually define a method for `tickle`—`tickle` is a generic function that can be specialized by any subclass of `TwoDPresenter` that needs to perform an update action with each tick or frame. Since `tickle` is called with every tick of the clock just prior to drawing the screen, it is an ideal way to synchronize animations. Subclasses of `TwoDPresenter` that implement a method for `tickle` include `TextEdit` and the `TransitionPlayer` family of classes.

In this respect, `tickle`, together with the accessor functions that implement the instance variable `needsTickle`, defines a protocol that `TwoDPresenter` objects can implement. This protocol is called the Ticklish protocol.

Note that controllers also implement the Ticklish protocol. The Ticklish protocol is implemented on both presenters and controllers so that programmers can maintain the separation of modeling, presentation, and control that is a paradigm of the Model-Presenter-Controller (MPC) model. Time-based behaviors that involve real interaction among objects should be implemented using controllers. For information on the MPC model, see the "Spaces and Presenters" chapter of the *ScriptX Components Guide*.

The `TwoDPresenter` implementation of the Ticklish protocol is interrupt-driven. Presenters are polled, and `tickle` is called on the presenter, only if the presenter is registered with the compositor. To make a presenter respond to `tickle`, first set the value of `needsTickle` to `true`. Setting `needsTickle` adds the presenter to the "tickle list," a private structure that is maintained by the 2D graphics compositor. If a presenter is ticklish, the compositor calls `tickle` automatically on that presenter once during each modeling/presentation cycle. (Note that this mechanism differs from how the Ticklish protocol is implemented by controllers.)

A `tickle` method must call `notifyChanged`, either directly or indirectly, if changes that it makes in an object are to have an effect on presentation. It is usually not necessary to call `notifyChanged` directly; subclasses of `TwoDPresenter` that are defined in the core classes do so automatically. For example, the setter methods for `x`, `y`, `width`, `height`, `stroke`, `fill`, and other `TwoDPresenter` properties implicitly call `notifyChanged` each time a new value is set.

Since `tickle` methods are called with every tick of the clock, they should run briefly and return a value. `tickle` is called in a callback thread, so a `tickle` method should never be allowed to block. If a tickle method needs to run some process that could block or take a long time to run, it should activate that process in another thread.

For information on compositor hints that can be used to optimize the performance of the 2D graphics compositor, see the "Spaces and Presenters" chapter of the *ScriptX Components Guide*.

Subclasses Commonly Implement

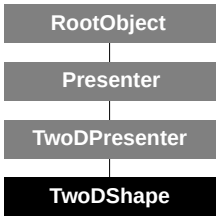
Subclasses of `TwoDPresenter` should implement the following method if they want objects to draw:

```
draw
```

Subclases of `TwoDPresenter` that update their appearance or perform some action with each tick of the top presenter's click should implement the following method.

```
tickle
```

TwoDShape



Class type: Core class (concrete)
Resides in: ScriptX and KMP executables
Inherits from: `TwoDPresenter`
Component: Spaces and Presenters

The `TwoDShape` class is used for drawing with stencil and brush. A `TwoDShape` object can use any stencil, a `Brush` object for filling, another `Brush` object for stroking (outlining), and a position for the stencil (or a matrix used as a position).

Because the `TwoDShape` class inherits from `TwoDPresenter`, a `TwoDShape` object is capable of imaging itself in collaboration with a 2D graphic compositor.

Figure 32 shows the objects that can be created from the `TwoDShape` class given the stencils available in the 2D Graphics component.

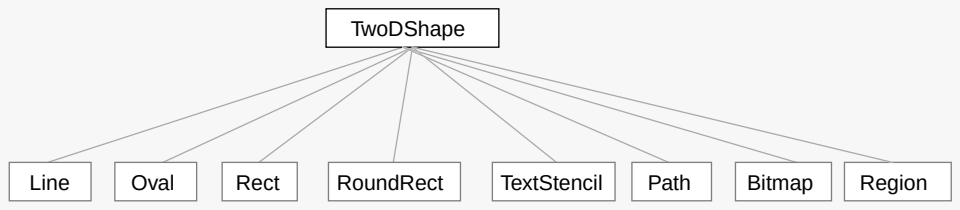


Figure 32: Objects that can be created from TwoDShape.

Creating and Initializing a New Instance

The following script is an example of how to create a new instance of the `TwoDShape` class. For most subclasses of `TwoDPresenter` the `target` and `boundary` keywords specify different objects; however, `TwoDShape` is a degenerate case where they must both specify the same object. Only one of `boundary` or `target` may be passed in (or the values supplied for both must be equal); `TwoDShape` ensures that both instance variables point to the same object.

```
myShape := new TwoDShape \
              boundary:(new Rect x2:80 y2:80)\
              fill:whiteBrush \
              stroke:blackBrush
```

The variable `myShape` contains the initialized 2D shape. The shape is a rectangle with an area of 80 x 80 pixels, has a white fill, and is outlined in black. The `new` method uses the keywords defined in `init`.

init

`init` *self* [`fill:`*brush*] [`stroke:`*brush*] [`target:`*stencil*]
[`boundary:`*stencil*] ⇨ *(none)*

self	`TwoDShape` object
`fill:`	`Brush` object
`stroke:`	`Brush` object

The following keywords are used by superclass `TwoDPresenter`:

`target:`	`Stencil` object
`boundary:`	`Stencil` object
`stationary:`	`Boolean` object

Initializes the `TwoDShape` object *self*, applying the arguments to the instance variables of the same name. The `target` keyword can be used by a subclass. Do not call `init` directly on an instance—it is automatically called by the `new` method.

If you omit an optional keyword, its default is used. The defaults are:

```
fill:undefined
stroke:undefined
target:undefined
boundary:(new Rect x2:0 y2:0)
stationary:false
```

Instance Variables

Inherited from `Presenter`:

`presentedBy`	`subPresenters`	`target`

Inherited from `TwoDPresenter`:

`bBox`	`height`	`transform`
`boundary`	`IsImplicitlyDirect`	`width`
`clock`	`isTransparent`	`window`
`compositor`	`isVisible`	`x`
`direct`	`needsTickle`	`y`
`eventInterests`	`position`	`z`
`globalBoundary`	`stationary`	
`globalTransform`	`target`	

The following instance variables are defined in `TwoDShape`:

fill

self`.fill` (read-write) `Brush`

Specifies the brush, if any, used to fill the boundary of the 2D shape *self*. If this variable is set to `undefined`, the fill is transparent. Its default is `whiteBrush` if its target is a bitmap; otherwise, it defaults to `undefined`.

stroke

self`.stroke` (read-write) `Brush`

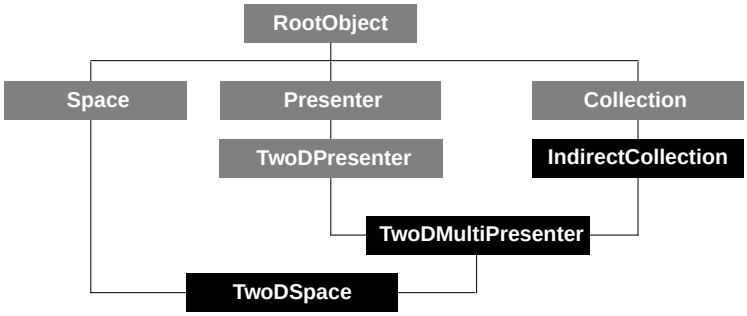
Specifies the brush, if any, used to stroke the outline of the 2D shape *self*. If this variable is set to `undefined`, the outline is transparent.

Instance Methods

Inherited from `TwoDPresenter`:

`adjustClockMaster`	`inside`	`show`
`createInterestList`	`localToSurface`	`surfaceToLocal`
`draw`	`notifyChanged`	`tickle`
`getBoundaryInParent`	`recalcRegion`	
`hide`	`refresh`	

TwoDSpace



Class type: Core class (concrete)
Resides in: ScriptX and KMP executables
Inherits from: `Space` and `TwoDMultiPresenter`
Component: Spaces and Presenters

The `TwoDSpace` class is the prime 2D graphics space in ScriptX. This class combines the presentation features of `TwoDMultiPresenter` with the modeling features of `Space`. An instance of `TwoDSpace` is an environment where 2D presenter objects live—bitmaps, shapes, video images, animations, and other media. The environment can be interactive, with any object being clickable or draggable, including windows, toggles, pushbuttons, scrolling lists, and so forth. As the user controls objects in the 2D space, those objects can control objects in that or any other space—or objects not in any space.

The `TwoDSpace` class gets its presentation features from `TwoDMultiPresenter`. Any object successfully added to a `TwoDSpace` automatically becomes a subpresenter and a part of the presentation hierarchy. The order of these subpresenters can be determined by setting their `z` values. `TwoDSpace` has a boundary that is fixed and clips its subpresenters. The `draw` method first fills the space, forming a background, then iterates `draw` over each subpresenter, and finally strokes the space's boundary.

`TwoDSpace` class gets its modeling and simulation features from `Space`. `TwoDSpace` has a mechanism for determining which objects are allowed into the space, a clock for timing, and controllers to modify the objects.

The prime model objects of `TwoDSpace` are instances of the `TwoDPresenter` class and instances of its subclasses, such as User Interface objects and 2D shapes. All `TwoDPresenter` objects are designed to live in a 2D space.

A `TwoDSpace` is a kind of collection, since it inherits from `IndirectionCollection`. Presenters in a `TwoDSpace` object are rendered in the order in which they appear in this collection, with the last item rendered first. Thus, presenters at the front of the collection appear in front of presenters at the end of the collection. Thus, objects within a `TwoDSpace` object present themselves in a manner similar to most object-based drawing programs, with some objects occluding others.

The visible shape of a `TwoDSpace` object within its parent space or display surface is defined by the stencil supplied by the `boundary` instance variable (defined in `TwoDPresenter`). The parent space or display surface defines the origin and coordinates. All presenters within the space are clipped to the space's boundary. Similarly, the location of a `TwoDSpace` object within its parent space or surface is given by the matrix supplied by the `transform` instance variable, also defined in the coordinates of the parent space or surface.

Note – If the boundary of the space is something other than a `Rect` or a `Region`, then garbage will be generated when the space is drawn or updated, and generating and collecting garbage adversely affects performance. Therefore, to improve your space's performance, use an instance of either `Rect` or `Region` for its boundary. For example, if

the boundary is an instance of `Oval`, then garbage is generated. However, you can easily make an oval-shaped region and use that instead, for better performance. If you must use a boundary that is not a `Rect` or a `Region`, then try to minimize the number of times you draw or update the space.

A `TwoDSpace` object is itself a `TwoDPresenter` object, and so can be contained in other `TwoDSpace` objects. As a presenter, a 2D presenter presents its contained `Presenter` objects. In this way, `TwoDSpace` objects can be arranged in a hierarchy. You can iterate down through these hierarchies by means of the `subPresenters` instance variable of each `Presenter` object, or up by means of the `presentedBy` instance variable.

A `TwoDSpace` object has a clock (or player) as does any other space. This clock (or player) is held by `topClocks` or `topPlayers` in the title container. (See the `TitleContainer` class for more information.)

In the `new` and `init` methods, the `target` keyword is ignored—it is available for you to use in any subclass you create, as a target for a presenter. For example, you might save a series of numbers to the target, and use them to determine the heights of bars in a bar chart.

Creating and Initializing a New Instance

The following script is an example of how to create a new instance of the `TwoDSpace` class: (Notice `target` is omitted because it is ignored.)

```
mySpace := new TwoDSpace \
              boundary:(new Rect x2:640 y2:480) \
              fill:whiteBrush \
              stroke:blackBrush \
              scale:20
```

This script creates an instance of `TwoDSpace` with a size of 640 by 480 pixels. The `new` method uses the keyword arguments defined by the `init` method.

For performance reasons, when creating an instance of `TwoDSpace` (actually, any instance of `TwoDMultiPresenter` or its subclasses), you should not specify the `targetCollection` keyword, so it can create its default collection. Such presenters require collections that can be traversed quickly when drawing and handling events. Performance could suffer if you change the `targetcollection` to something other than the default.

init

`init` *self* [`boundary:`*stencil*] [`fill:`*brush*] [`stroke:`*brush*]
[`scale:`*integer*] [`target:`*object*] [`targetCollection:`*sequence*] ⇨ *(none)*

self `TwoDSpace` object

Superclasses of `TwoDSpace` use the following keywords:

`boundary:`	`Stencil` object
`fill:`	`Brush` object
`stroke:`	`Brush` object
`scale:`	`Integer` object
`target:`	Ignored by `TwoDSpace`
`targetCollection:`	`Sequence` object
`stationary:`	`Boolean` object

Initializes the TwoDSpace object *self*, applying the arguments to instance variables of the same name. The targetCollection is the kind of collection to create. The target is ignored by TwoDSpace, but you are free to use it in a subclass. Do not call init directly on an instance—it is automatically called by the new method.

If you omit an optional keyword, its default value is used. The defaults are:

```
boundary:(new Rect x1:0 y1:0 x2:1 y2:1)
fill:undefined
stroke:undefined
scale:24
target:undefined
targetCollection:(new Array initialSize:14 growable:true)
stationary:false
```

Class Methods

Inherited from Collection:

pipe

Instance Variables

Inherited from Collection:

bounded	maxSize	size
iteratorClass	minSize	uniformity
keyEqualComparator	mutable	uniformityClass
keyUniformity	mutableCopyClass	valueEqualComparator
keyUniformityClass	proprietored	

Inherited from IndirectCollection:

targetCollection

Inherited from Presenter:

presentedBy	subPresenters	target

Inherited from TwoDPresenter:

bBox	height	transform
boundary	IsImplicitlyDirect	width
clock	isTransparent	window
compositor	isVisible	x
direct	needsTickle	y
eventInterests	position	z
globalBoundary	stationary	
globalTransform	target	

Inherited from TwoDMultiPresenter:

clock	fill	stroke

Inherited from Space:

clock	protocols	tickleList
controllers		

The following instance variable is redefined in TwoDSpace:

protocols (Space)

self.protocols (read-only) Array

The protocols array for instances of TwoDSpace is initialized to contain the class TwoDPresenter. This means that any object added to an instance of TwoDSpace must have TwoDPresenter as one of its superclasses. See the Space class for further description about this instance variable.

The attribute "read-only" means that you cannot make the protocols instance variable point to a different array—"read-only" does not stop you from adding or removing items from the array.

Instance Methods

Inherited from Collection:

add	forEach	iterate
addMany	forEachBinding	localEqual
addToContents	getAll	map
chooseAll	getAny	merge
chooseOne	getKeyAll	pipe
chooseOneBinding	getKeyOne	prin
deleteAll	getMany	removeAll
deleteBindingAll	getOne	removeOne
deleteBindingOne	hasBinding	setAll
deleteKeyAll	hasKey	setOne
deleteKeyOne	intersects	size
deleteOne	isEmpty	
emptyOut	isMember	

Inherited from IndirectCollection:

isAppropriateObject	objectAdded	objectRemoved

Inherited from TwoDPresenter:

adjustClockMaster	inside	show
createInterestList	localToSurface	surfaceToLocal
draw	notifyChanged	tickle
getBoundaryInParent	recalcRegion	
hide	refresh	

Inherited from TwoDMultiPresenter:

draw	findFirstInStencil	moveToBack
findAllAtPoint	isAppropriateObject	moveToFront
findAllInStencil	moveBackward	objectAdded
findFirstAtPoint	moveForward	objectRemoved

Inherited from Space:

isAppropriateObject	objectAdded	objectRemoved

Since a TwoDSpace object is an indirect collection, you can also use any methods defined in the class specified by targetCollection. The target collection is typically an instance of Array, which inherits from Sequence, so the following instance methods are redirected to a 2D space:

Accessible from LinearCollection:

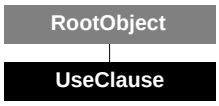
chooseOneBackwards	deleteSecond	getMiddle
chooseOrdOne	deleteThird	getNth
deleteFifth	findRange	getNthKey
deleteFirst	forEachBackwards	getOrdOne
deleteFourth	getFifth	getRange
deleteLast	getFirst	getSecond

deleteNth	getFourth	getThird
deleteRange	getLast	pop

Accessible from Sequence:

addFifth	moveBackward	setFourth
addFirst	moveForward	setLast
addFourth	moveToBack	setNth
addNth	moveToFront	setSecond
addSecond	prepend	setThird
addThird	prependNew	sort
append	setFifth	
appendNew	setFirst	

UseClause



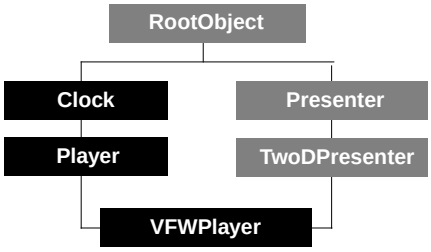
Class type: Core class (concrete, sealed)
Resides in: ScriptX and KMP executables
Inherits from: `RootObject`
Component: Object System Kernel

The `UseClause` class represents a“uses” relationship between two modules. In the ScriptX module definition expression, a module can be defined so that it uses other modules.

Creating and Initializing a New Instance

An instance of `UseClause` is created syntactically through the ScriptX language, in the module defintion expression.

VFWPlayer



Class type: Loadable class (concrete)
Resides in: `vfwplyr.lib`. Works with ScriptX and KMP executables
Inherits from: `Player` and `TwoDPresenter`
Component: Media Players

Kaleida Labs provides two loadable classes that can be used to play movies directly from hard disk without importing them into ScriptX. These classes are the `VFWPlayer` and the `QuickTimePlayer`. These classes provide ScriptX object wrappers to the Video For Windows runtime and the QuickTime runtime, respectively.

The `VFWPlayer` works only on a Microsoft Windows machine with Video For Windows installed. It can play AVI files directly, without importing them. Video For Windows is responsible for doing everything from reading the video information off disk, deinterleaving it, decompressing video frames and displaying them on the screen. The ScriptX `VFWPlayer` class simply provides the movie with an area of screen real-estate on which to display the video and supports behavior similar to that of the ScriptX `MoviePlayer` class.

The `VFWPlayer` is provided as loadable classes and is not part of the ScriptX Core Classes. It must be explicitly loaded before they it can be used.

The `VFWPlayer` class inherits from `TwoDPresenter` and `Player`.

Be cautious when building a title that uses the `VFWPlayer` class. A title that depends on this player will not be portable to different platforms, and will only work so long as the files for the movies played by the player resides in the place where the player expects to find them. Note also that instances of loadable media player classes cannot be saved to the object store.

Loading the VFWPlayer Class

`VFWPlayer` is a scripted class that needs to be loaded dynamically before it can be used. The script files required to load the `VFWPlayer` class reside in a directory called `vfwplayr` in the directory called `loadable.`

To load the `VFWPlayer` class on a Microsoft Windows system, open the `loadme.sx` file in the `/LOADABLE/VFWPLAYR` directory using the **Open Title** command in the ScriptX **File** menu.

Using the Loadable Media Player Classes

After loading a loadable Media Player Class, you can use it to play the appropriate media on the appropriate platform without importing the media into ScriptX.

Create an instance of the appropriate class by calling the `new` method on the appropriate class.

The following script is an example of how to create a new instance of `VFWPlayer`:

```
myPlayer := new VFWPlayer \
```

```
dir:theStartDir \
filename:"RodDance" \
masterClock:topPlayer
```

The variable myPlayer points to the newly created video for windows player, which can be used to play the movie "RodDance" that resides in the ScriptX startup directory. The player's master clock is topPlayer.

The new method uses the keywords defined in init.

You can play the movie associated with a loadable media player instance by using the usual Player and MoviePlayer methods. For example:

```
-- append the player to a visible window
global w := new windows boundary:myPlayer.bbox
show w
append w myPlayer

-- Play the movie associated with myPlayer from beginning to end
gotobegin myPlayer
playUntil myPlayer myPlayer.duration
```

init

init *self* [dir:*dirRep*] [fileName:*string*] [masterClock: *topPlayer*] [title:*titleContainer*] ⇨ *self*

self	QuickTimePlayer or VFWPlayer object.
dir:	DirRep object for the directory containing the movie to be played.
fileName:	String object for the name of the QuickTime or AVI file containing the movie to be played.

The Clock superclass uses the following keywords:

masterClock:	Clock object to be used as the master player
scale:	(Ignored by the loadable media player classes)
title:	(Ignored by the loadable media player classes. Loadable media player instances cannot be saved to title containers.)

Initializes the loadable media player object *self* so that it can play the movie in the file specified by fileName which resides in the directory specified by dir. If supplied, masterClock is used as the master clock for the new player.

If you omit an optional keyword its default value is used. The defaults are:

dir:theStartDir
fileName:undefined
masterClock:undefined (which means the root hardware clock)

Instance Variables

Inherited from Clock:

callbacks	rate	ticks
effectiveRate	resolution	time
masterClock	scale	title
offset	slaveClocks	

Inherited from Player:

audioMuted	globalContrast	globalVolumeOffset
dataRate	globalHue	markerList

duration	globalPanOffset	status
globalBrightness	globalSaturation	videoBlanked

Inherited from Presenter:

presentedBy	subPresenters	target

Inherited from TwoDPresenter:

bBox	globalBoundary	target
boundary	globalRegion	transform
changed	globalTransform	width
clock	height	window
compositor	imageChanged	x
direct	position	y
eventInterests	stationary	z

Instance Methods

Inherited from Clock:

addPeriodicCallback	clockAdded	pause
addRateCallback	clockRemoved	resume
addScaleCallback	effectiveRateChanged	timeJumped
addTimeCallback	forEachSlave	waitTime
addTimeJumpCallback	isAppropriateClock	waitUntil

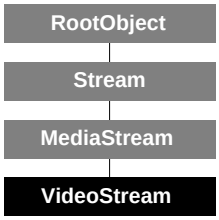
Inherited from Player:

addMarker	goToBegin	playPrepare
eject	goToEnd	playUnprepare
fastForward	goToMarkerFinish	playUntil
getMarker	goToMarkerStart	rewind
getNextMarker	pause	stop
getPreviousMarker	play	

Inherited from TwoDPresenter:

addChangedRegion	draw	refresh
adjustClockMaster	getBoundaryInParent	windowToLocal
createInterestList	localToWindow	

VideoStream



Class type: Core class (concrete)
Resides in: ScriptX and KMP executables
Inherits from: `MediaStream`
Component: Media Players

The `VideoStream` class provides objects that manage the specific details of video data. Video streams can select and use type-specific codecs to decompress incoming video data into frames with the desired attributes. Several characteristics of the data are set for the video stream when it is opened on that data; the stream passes some of these on to the codec used to decompress the stream.

`VideoStream` allows you to retrieve the attributes of the decompression process, set playback to a position in the data stream, and get frames of video data. For the `VideoStream` class the unit for reading is a frame, although seeking through the stream is done in terms of time.

Note – Depending on implementation details, it may not be possible for the stream to support some of the methods described below. For example, it may not be possible to set clip regions if the codec being used will not support it.

Creating and Initializing a New Instance

`VideoStream` instances are created during the process of importing movies into ScriptX. See the description of `MoviePlayer` for a discussion of importing movies.

The `AudioStream` class does support the `new` method, whose argument is `inputStream`, which needs to be a `byteStream`. You would not normally call the `new` method on `VideoStream` to create a new instance. Instead you would import a movie, which creates a video stream as part of the importing process. However, for the sake of completeness, the following script illustrates how to create a new instance of the `VideoStream` class by calling the `new` method. For this example, you would need to have previously created the byte stream `myStream`, which should contain digitized video data.

```
myVideoStream := new VideoStream \
                 inputStream:myStream
```

The variable `myVideoStream` points to the newly created video stream, which has the `ByteStream` instance `myStream` as its input stream.

The `new` method uses the keywords defined in `init`.

init

`init` *self* [`inputStream:`*stream*] ⇨ *(none)*

self	`VideoStream` object
`inputStream:`	`ByteStream` object

Initializes the `VideoStream` object *self*, applying the `inputStream` argument as the source of video data. This argument may be a file or buffer opened as a stream and containing video data. Do not call `init` directly on an instance—it is automatically called by the `new` method.

If you omit an optional keyword, its default value is used. The defaults are:

```
inputStream:undefined
```

Instance Variables

Inherited from `MediaStream`:

`dataRate`	`sampleType`	`inputStream`
`markerList`	`scale`	`variableFrameSize`
`rate`		

The following instance variables are defined in `VideoStream`:

accuracy

self`.accuracy` (read-write) `NameClass`

Specifies the accuracy with which the decompression will be done for the video stream *self*. The values that this instance variable can take are `@maximum`, `@minimum`, `@low`, `@normal`, `@high`, and `@lossLess`.

cursorFrame

self`.cursorFrame` (read-only) `Integer`

Specifies the frame index corresponding to the cursor value in the video stream *self*. The frame index is 0-based.

direction

self`.direction` (read-write) `Integer`

Specifies the value for the direction of the video stream *self*. The value can be positive, negative, or zero (corresponding to forward play, reverse play or stationary). Positive direction means that successive reads will obtain frames corresponding to forward movements in time. Negative direction implies that the returned frames will be in reverse order. Zero direction will result in the same frame being returned for each read.

frameHeight

self`.frameHeight` (read-only) `Integer`

Specifies the height (in pixels) of a video frame in the video stream *self*. Note that this value may change during the course of a stream.

frameRate

self`.frameRate` (read-only) `Fixed`

Specifies the frame rate (in frames per second) of the video stream *self*. This value may change during the course of a stream.

frameWidth

self.`frameWidth` (read-only) `Integer`

Specifies the width (in pixels) of the current video frame in the video stream *self*. Note that this value may change during the course of a stream.

numFrames

self.`numFrames` (read-only) `Integer`

Specifies the number of frames in the video stream *self*.

srcRect

self.`srcRect` (read-write) `Rect`

Specifies the source rectangle that identifies the portion of the compressed frames to be decompressed from the video stream *self* to the bitmap surface.

Instance Methods

Inherited from `Stream`:

`cursor`	`next`	`seekFromStart`
`flush`	`plug`	`setStreamLength`
`isAtFront`	`previous`	`streamLength`
`isPastEnd`	`read`	`write`
`isReadable`	`readReady`	`writeReady`
`isSeekable`	`seekFromCursor`	
`isWritable`	`seekFromEnd`	

Inherited from `MediaStream`:

`addMarker`	`isSeekable`	`isWritable`
`isReadable`		

The following instance methods are defined in `VideoStream`:

frameDone

`frameDone` *self surface* ⇨ `Boolean`

self	`VideoStream` object
surface	`BitmapSurface` object

Relinquishes the surface used by the video stream *self*.

prepareStream

`prepareStream` *self surface param* ⇨ `Boolean`

self	`VideoStream` object
surface	`BitmapSurface` object
param	(unused)

Prepares the stream *self* to play its media. In media-production terms, this may be thought of as "pre-roll." As implemented in `VideoStream`, this method allocates resources required to play the video. The second argument, inherited from the `MediaStream` class, is used to represent a surface that the video will be drawn on. The third argument, *param*, is ignored.

readFrame

`readFrame` *self surface transform refresh* ⇨ `BitmapSurface`

self	`VideoStream` object
surface	`BitmapSurface` object
transform	`TwoDMatrix` object
refresh	`Boolean` object indicating whether the surface will be refreshed or not

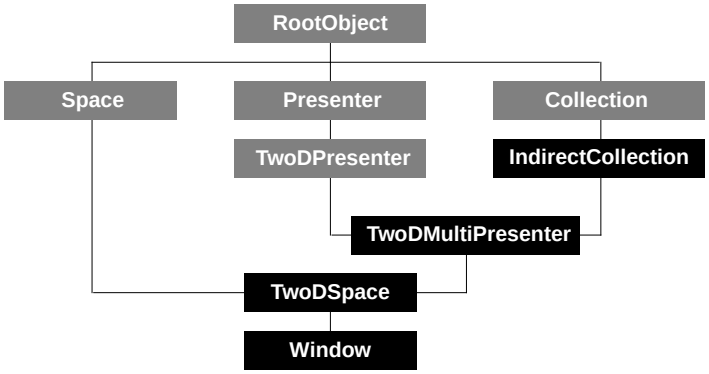
Returns a single frame of video at the current position within the video stream *self*. If appropriate, the transform matrix is applied during decompression. The surface returned is owned by the stream and must be returned to it.

unprepareStream

`unprepareStream` *self* ⇨ (`Boolean`)

Releases resources needed to play the data in the video stream *self*.

Window



Class type: Core class (concrete)
Resides in: ScriptX and KMP executables
Inherits from: `TwoDSpace`
Component: Spaces and Presenters

The `Window` class represents any of several different styles of window found in a standard user interface, including dialog boxes, notice windows, and palette windows (as determined by the `type` instance variable). Full-screen windows are implemented by `FullScreenWindow`, a subclass of `Window`. The appearance of a window's title bar and border are defined by the underlying operating system, as shown in the following figure.

`Window` is a subclass of `TwoDSpace`, and thereby has a clock, and can have an associated set of controllers. A `Window` object has an instance of `DisplaySurface` which it displays. When you create a new window, it automatically creates this display surface, to which it draws. The window thereby becomes the top presenter for a new presentation hierarchy. Since every window is a top presenter, you cannot add a window to another window.

The `Window` class does not use the operating system's scroll bars. To add scroll bars to a window, you can use either `ScrollingPresenter` or `ScrollBar`, which are both defined in the core classes. Adding objects to a `Window` object is the same as to adding them to a `TwoDSpace` object.

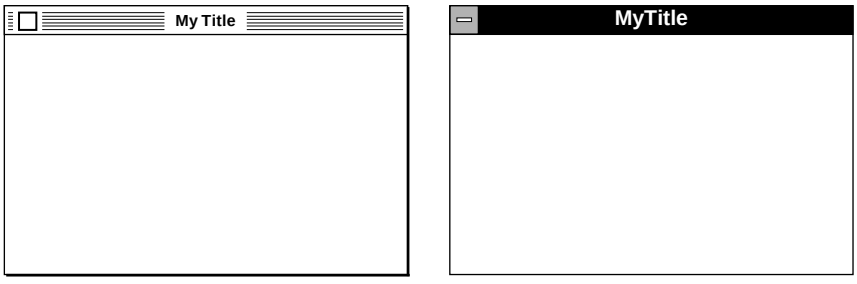


Figure 4-33: A window's appearance is determined by the underlying operating system.

The `title` instance variable stores a reference to the title container that manages the window. This allows the window to be closed automatically when the title container is closed, among other things. Note that this does not necessarily imply that the window will be saved with the title container. A title container is a collection, so a program must explicitly add the window to the title container to save it with the title. When you create a window, this instance variable is set to the value of `theScratchTitle`, unless you specify otherwise.

Types and Subclasses of Window

The window class has four subtypes, as determined by its `type` property: `@dialog`, `@palette`, `@notice`, and `@normal`. These four types correspond with the standard window types that are defined by the operating system for each of the underlying platforms on which the Kaleida Media Player runs. For example, a window whose `type` is set to `@dialog` is a modal window with no title bar and no resize box. A window's type is set at instantiation. For a side-by-side description of these types, see the "Spaces and Presenters" chapter in the *ScriptX Components Guide*.

Window also has one subclass that is built into the ScriptX core classes, `FullScreenWindow`, which fills the entire screen. For more information on this class, see its definition in this volume.

Creating and Initializing a New Instance

The following script is an example of how to create a new instance of the `Window` class:

```
myWindow := new Window      \
            title:myTitle    \
            centered:true    \
            boundary:(new Rect x2:250 y2:250)          \
            resizeBoundary:(new Rect x2:100 y2:100)    \
            name:"Window of Opportunity"
```

The variable `myWindow` contains an initialized instance of `Window`. Since no value is supplied for the `type` keyword argument, its type is `@normal`. The window belongs to the title container `myTitle`, is centered on-screen, has a boundary of 250 x 250 pixels, and is resizeable to a minimum size of 100 x 100 pixels. The name "Window of Opportunity" appears in its title bar. The `new` method uses keywords defined by the `init` method.

Note – If you omit the `boundary` keyword, the window is moved down so that its title bar is showing. If you include the `boundary` keyword, the window is moved up so that its title bar is hidden behind the menu bar. If you omit the `resizeBoundary` keyword, the window is not resizeable.

For performance reasons, when creating an instance of `Window`—actually, any instance of `TwoDMultiPresenter` or its subclasses—you should not specify the `targetCollection` keyword. Allow the class to create its default collection. Such a presenter requires a target collection that can be traversed quickly when drawing subpresenters or handing events. Performance could suffer if you change `targetCollection` to something other than what is generated by default.

init

`init` *self* [`type:`*name*] [`title:`*titleContainer*] [`centered:`*boolean*] [`name:`*string*] [`compositor:`*twoDCompositor*] [`resizeBoundary:`*stencil*] [`boundary:`*stencil*] [`fill:`*brush*] [`scale:`*integer*] [`stroke:`*brush*] [`target:`*object*] [`targetCollection:`*sequence*] ⇨ *(none)*

self	`Window` object
`type:`	`NameClass` object
`title:`	`TitleContainer` object
`centered:`	`Boolean` object
`name:`	`String` object
`compositor:`	`TwoDCompositor` object
`resizeBoundary:`	`Rect` object representing minimum resize boundary

Superclasses of `Window` use the following keywords:

`boundary:`	`Rect` object representing the window perimeter
`fill:`	`Brush` object to fill the background
`scale:`	`Integer` object
`stroke:`	Ignored by `Window`
`target:`	Ignored by `Window`
`targetCollection:`	`Sequence` object
`stationary:`	`Boolean` object

Initializes the `Window` object *self*, applying the values supplied with the keywords to the instance variables of the same name, except `centered` and `resizeBoundary`, which have no corresponding instance variable.

When the value of `centered` is `true`, the window is centered in the middle of the screen. When `false`, the `x` and `y` values inherited from `TwoDPresenter` determine where the window is located.

The value supplied for `resizeBoundary`, which must be a `Rect` object, determines whether the window is resizeable, and what its minimum size is. If the `resizeBoundary` argument is not supplied, the window is not resizeable. Windows of type `@dialog` and `@notice` are not resizeable, so the resize boundary is ignored. The subclass `FullScreenWindow` also ignores this keyword, since it is not resizeable.

You normally omit the `compositor` keyword; if no value is supplied, a new compositor is created automatically. You can specify `compositor:undefined` to prevent a compositor from being created, or supply your own compositor. The `target` keyword is ignored by `TwoDSpace`, but you are free to use it in a subclass of `Window`. Do not call `init` directly on an instance—it is automatically called by the `new` method.

If you omit an optional keyword, its default value is used. The defaults are:

```
type:@normal
title:theScratchTitle
centered:false
name:"ScriptX Display Surface"
compositor:[creates a new 2D compositor]
resizeBoundary:unsupplied
boundary:(new Rect x1:0 y1:0 x2:600 y2:400)
fill:whiteBrush
scale:24
targetCollection:(new Array initialSize:14 growable:true)
stationary:false
```

Class Methods

Inherited from `Collection`:

`pipe`

Instance Variables

Inherited from `Collection`:

`bounded`	`maxSize`	`size`
`iteratorClass`	`minSize`	`uniformity`
`keyEqualComparator`	`mutable`	`uniformityClass`
`keyUniformity`	`mutableCopyClass`	`valueEqualComparator`
`keyUniformityClass`	`proprietored`	

Inherited from `IndirectCollection`:

`targetCollection`

Inherited from `Presenter`:

`presentedBy`	`subPresenters`	`target`

Inherited from `TwoDPresenter`:

`bBox`	`height`	`transform`
`boundary`	`IsImplicitlyDirect`	`width`
`clock`	`isTransparent`	`window`
`compositor`	`isVisible`	`x`
`direct`	`needsTickle`	`y`
`eventInterests`	`position`	`z`
`globalBoundary`	`stationary`	
`globalTransform`	`target`	

Inherited from `TwoDMultiPresenter`:

`clock`	`fill`	`stroke`

Inherited from `Space`:

`clock`	`controllers`	`tickleList`

Inherited from `TwoDSpace`:

`protocols`

The following instance variables are defined in `Window`:

authorData

self`.authorData` (read-write) *(object)*

Specifies an argument, which can be any ScriptX object, that is supplied as the first argument to the window's `resizeAction` when that function is called. When `resize` is called on the window *self*, it automatically invokes the function stored in `resizeAction`.

colormap

self`.colormap` (read-write) `Colormap`

Specifies an instance of `Colormap` that describes the bit-depth and color encoding of pixels in the window *self*. Pixel encodings in the data of a bitmap are assumed to represent keys into this color map whose values are instances of `Color`. By default, this value is set to `defaultColormap`.

If the `colormap` of a bitmap image added to a window is not the same as the `colormap` for the window, performance can suffer greatly (in a platform-dependent way). To improve performance, make the window's colormap the same as the bitmap's:

```
myWindow.colormap := myBitmap.colormap
```

For more information on color maps and bitmaps, see the chapter "2D Graphics" in the *ScriptX Components Guide*.

displaySurface (TwoDPresenter)

self`.displaySurface` (read-only) `DisplaySurface`

Specifies the display surface onto which the window *self* is displayed. This display surface is automatically created when the window is created.

fill (TwoDMultiPresenter)

self.fill (read-write) Brush

Specifies the brush for rendering the background of the window *self*. For windows, its default value is whiteBrush.

hasUserFocus

self.hasUserFocus (read-only) Boolean

Specifies whether or not the window *self* is the frontmost of all open windows (except palette windows). This flag is read-only, and is automatically set to true in three cases: when the window is first created or opened, when the window gets a mouse down event, and when bringToFront is called on both a window and its title container. When this flag is true, the window *self* and its title are given user focus, and the window is placed at the front of the title's windows list.

Only one window for each ScriptX runtime environment can have user focus at a time. That is, if several titles are open, each with several windows, only one window can have user focus. The only window that cannot have user focus is a window of type @palette. This is because a palette window is the only window which can be in front of other windows while a background window has user focus.

Conversely, when any other window gets user focus, the hasUserFocus flag of the window that loses focus is set to false.

To set this flag true from a script, call bringToFront on the window. The bringToFront method brings a window only to the front of its layer. The three window layers are the notice/dialog layer, the palette layer, and the window/full-screen window layer.

name

self.name (read-write) Text

Specifies the text that is presented in title bar of the window *self*.

resizeAction

self.resizeAction (read-write) *(function)*

If defined, resizeAction specifies a function, which can be any generic, global, or anonymous function, that is invoked automatically when resize is called on the window *self*. Use resizeAction to specify any changes to the window (and to the objects it contains) that occur when the window is resized. The compositor is disabled when this function is invoked.

Specify a function that takes three arguments for the window's resizeAction. This function is called automatically when resize, an instance method defined by Window, is invoked. The first argument is referenced by the instance variable authorData. The second argument is the window *self*. The third argument is a rectangle describing the bounding box of the window *self* after the window has been resized. The function is passed a copy of the rectangle referenced in the window's bBox instance variable.

func *authorData window boundingBox*

systemMenuBar

self.systemMenuBar (read-write) SystemMenuBar

Specifies the system menu bar that is active when the window *self* has user focus. Is set to undefined in the current version of ScriptX. In the future, this may point to a SystemMenuBar object which is associated with the window (rather than the title).

title

self.title (read-write) TitleContainer

Specifies the title container that manages the window *self*. This causes the window to be closed when the title container is closed, the title to get user focus when the user gets user focus, and the window's compositor to pause and resume when the title is paused and resumed. When you create a window, this instance variable is set to the value of theScratchTitle, unless you specify otherwise with the title keyword of the new method.

The title instance variable is set to a particular title if and only if the window is listed in that title's windows instance variable.

When you add a window to a title container, the title instance variable is updated to that title container. However, you can set this value directly if you want it to be managed by another title container.

Notice that the title instance variable does not necessarily determine that the window will be saved with that title container—that is determined by the title the window is contained in (by adding the window to the title container with a collection method such as add, append, or prepend).

Instance Methods

Inherited from Collection:

add	forEach	iterate
addMany	forEachBinding	localEqual
addToContents	getAll	map
chooseAll	getAny	merge
chooseOne	getKeyAll	pipe
chooseOneBinding	getKeyOne	prin
deleteAll	getMany	removeAll
deleteBindingAll	getOne	removeOne
deleteBindingOne	hasBinding	setAll
deleteKeyAll	hasKey	setOne
deleteKeyOne	intersects	size
deleteOne	isEmpty	
emptyOut	isMember	

Inherited from IndirectCollection:

isAppropriateObject	objectAdded	objectRemoved

Inherited from TwoDPresenter:

adjustClockMaster	inside	show
createInterestList	localToSurface	surfaceToLocal
draw	notifyChanged	tickle
getBoundaryInParent	recalcRegion	
hide	refresh	

Inherited from TwoDMultiPresenter:

draw	findFirstInStencil	moveToBack
findAllAtPoint	isAppropriateObject	moveToFront

findAllInStencil	moveBackward	objectAdded
findFirstAtPoint	moveForward	objectRemoved

Inherited from Space:

isAppropriateObject	objectAdded	objectRemoved

Since a Window object is an indirect collection, you can also use any methods defined in the class specified by targetCollection. The target collection is typically an instance of Array, which inherits from Sequence, so the following instance methods are redirected to a window:

Accessible from LinearCollection:

chooseOneBackwards	deleteSecond	getMiddle
chooseOrdOne	deleteThird	getNth
deleteFifth	findRange	getNthKey
deleteFirst	forEachBackwards	getOrdOne
deleteFourth	getFifth	getRange
deleteLast	getFirst	getSecond
deleteNth	getFourth	getThird
deleteRange	getLast	pop

Accessible from Sequence:

addFifth	moveBackward	setFourth
addFirst	moveForward	setLast
addFourth	moveToBack	setNth
addNth	moveToFront	setSecond
addSecond	prepend	setThird
addThird	prependNew	sort
append	setFifth	
appendNew	setFirst	

The following instance methods are defined in Window:

bringToFront

bringToFront *self* ⇨ Boolean

If the window is visible, this method moves the window to the front of its layer in the title container, both in the windows list and visually on the screen, and then returns true. The three window layers are the notice/dialog layer, the palette layer, and the window/full-screen window layer.

If the window is not visible, this method does not change the window order, and returns false.

Note that this method brings the window forward only among other windows within its title—it does not also bring the window's title to the front of other titles. If what you really want is to give the window user focus, in front of windows in all other open titles, instead set hasUserFocus to true.

A user clicking on a window calls bringToFront on the window and again on its title.

clearSelection

clearSelection *self* ⇨ *(object)*

Deletes the currently selected objects in the window *self*.

The implementation of the clearSelection method is empty in the Window class. It is your responsibility to subclass Window and override clearSelection to delete the selected objects. By convention, this method should return the object being cleared. The clearSelection method has no effect on the clipboard.

See the `clearSelection` method in the `TitleContainer` class for a description of how this method is called when a user chooses **Clear** from the ScriptX menu bar.

copySelection

`copySelection` *self* ⇨ *(object)*

Makes copies of the currently selected objects in the window *self*, and places those copies on the clipboard; the previous contents of the clipboard are deleted.

The implementation of the `copySelection` method is empty in the `Window` class. It is your responsibility to subclass `Window` and override `copySelection` to copy the selected objects to the clipboard, using `setClipboard`. By convention, this method should return the object being copied.

See the `copySelection` method in the `TitleContainer` class for a description of how this method is called when a user chooses **Copy** from the ScriptX menu bar.

cutSelection

`cutSelection` *self* ⇨ *(object)*

Removes the currently selected objects from the window *self*, and places them on the clipboard; the previous contents of the clipboard are deleted.

The implementation of the `cutSelection` method is empty in the `Window` class. It is your responsibility to subclass `Window` and override `cutSelection` to cut the selected objects to the clipboard, using `setClipboard`. By convention, this method should return the object being cut.

See the `cutSelection` method in the `TitleContainer` class for a description of how this method is called when a user chooses **Cut** from the ScriptX menu bar.

hide (TwoDPresenter)

`hide` *self* ⇨ *self*

Hides the window *self*. In addition to the default behavior for `hide` that is inherited from `TwoDPresenter`, the `hide` method defined by `Window` sets `hasUserFocus` to `false`, gives user focus to the next window on the screen, and disables the window's compositor. Calling `hide` also sets `isVisible` to `false` and removes the window's compositor's interest in `PaletteChangedEvent` objects from the interest list. The window's clock continues to run.

When a user closes a window by clicking in the close box (Macintosh) or choosing **Close** from the system menu (Windows and OS/2), this invokes `hide` on the window. Although we speak of "closing" windows, you call `hide` on a window, not `close` (`close` is for storage containers).

Notice that clicking the close box is not equivalent to choosing the **Close** menu command on the File menu. The **Close** menu command closes the title and all its windows.

pasteToSelection

`pasteToSelection` *self* ⇨ *(object)*

Makes a copy of the objects on the clipboard and places the copy in the window *self*. The contents of the clipboard remain unchanged.

The implementation of the `pasteToSelection` method is empty in the `Window` class. It is your responsibility to subclass `Window` and override `pasteToSelection` to paste from the clipboard into the window. This method should typically check the clipboard's

`typeList` instance variable to see if the types available on the clipboard are appropriate for the insertion point in the window, and if they are, should call `getClipboard` to do the paste. By convention, `pasteToSelection` should return the object being pasted.

See the `pasteToSelection` method in the `TitleContainer` class for a description of how this method is called when a user chooses **Paste** from the ScriptX menu bar.

printWindow

`printWindow` *self* ⇨ *(object)*

This method is called by the default implementation of `printTitle` if the window *self* is the topmost window in the title. By default, `printWindow` does nothing. Override this method in a subclass for a different implementation.

refreshRegion

`refreshRegion` *self addedRegion* ⇨ *(none)*

self	`Window` object
addedRegion	`Region` object

The Kaleida Media Player calls `refreshRegion` automatically when any portion of a ScriptX window needs to be refreshed. The screen might need to be refreshed, for example, after the user has closed or moved a window that belongs to another open title or application.

For example, the following code forces the entire window `myWin` to refresh:

```
refreshRegion myWin (myWin.globalBoundary as Region)
```

As defined by `Window`, `refreshRegion` calls a method of the same name on the compositor attached to the window *self*. The compositor, in turn, adds the region specified by *addedRegion* to the changed region it keeps internally until the next composite cycle, when it re-draws all presenters that intersect the changed region.

If the value of `compositor` is `undefined`, you can override `refreshRegion` to perform any drawing operation you want.

resize

`resize` *self bBox* ⇨ *self*

self	`Window` object
bBox	`Rect` object, the new bounding box of the window

Do not call `resize` directly from the scripter. It is called automatically when the window *self* is resized by the user. Specialize `resize` to change the behavior of an entire subclass of `Window`. Since `resize` automatically invokes the function referenced by `resizeAction`, if one is defined, creating a resize action is the better route to instance-level specialization. ScriptX disables the compositor before calling `resize`, and re-enables it immediately afterwards. Thus, any layout changes that are undertaken by `resize` or by the function referenced by `resizeAction`, do not appear until they are completed.

Note that the user resizes a window using gestures that are defined by the underlying operating system. While the user is resizing a window, the operating system receives and processes mouse events directly. Windows and OS/2 allow the user to resize a window by grabbing and moving its outside boundary. This boundary is outside the Kaleida Media Player display area. The Macintosh user interface specifies a resize box in the lower right corner of the window, in a region where the Kaleida Media Player would otherwise receive mouse events. These mouse events are not processed by the Kaleida Media Player.

sendToBack

`sendToBack` *self* ⇨ `Boolean`

If the window is visible, this method moves the window to the back of its layer in the title container, both in the `windows` list and visually on the screen, and then returns `true`. The three window layers are the notice/dialog layer, the palette layer, and the window/full-screen window layer.

If the window is not visible, this method does not change the window order, and returns `false`.

Note that this method sends the window to the back only among other windows within its title—it does not position the window's title behind other titles.

show (TwoDPresenter)

`show` *self* ⇨ *self*

Shows the window *self*. In addition to the default behavior for `show` that is inherited from `TwoDPresenter`, the `show` method defined by `Window` registers an interest in `PaletteChangedEvent` for the window's compositor, enables the compositor, gives the title user focus, sets `isVisible` to true, calls `notifyChanged` on itself, and finally, calls `bringToFront` on the window's title.

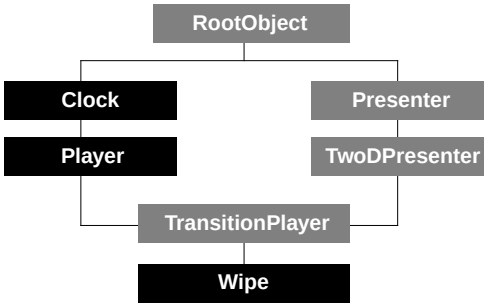
snapshot

`snapshot` *self surface* ⇨ `BitmapSurface`

self	`Window` object
surface	`BitmapSurface` object

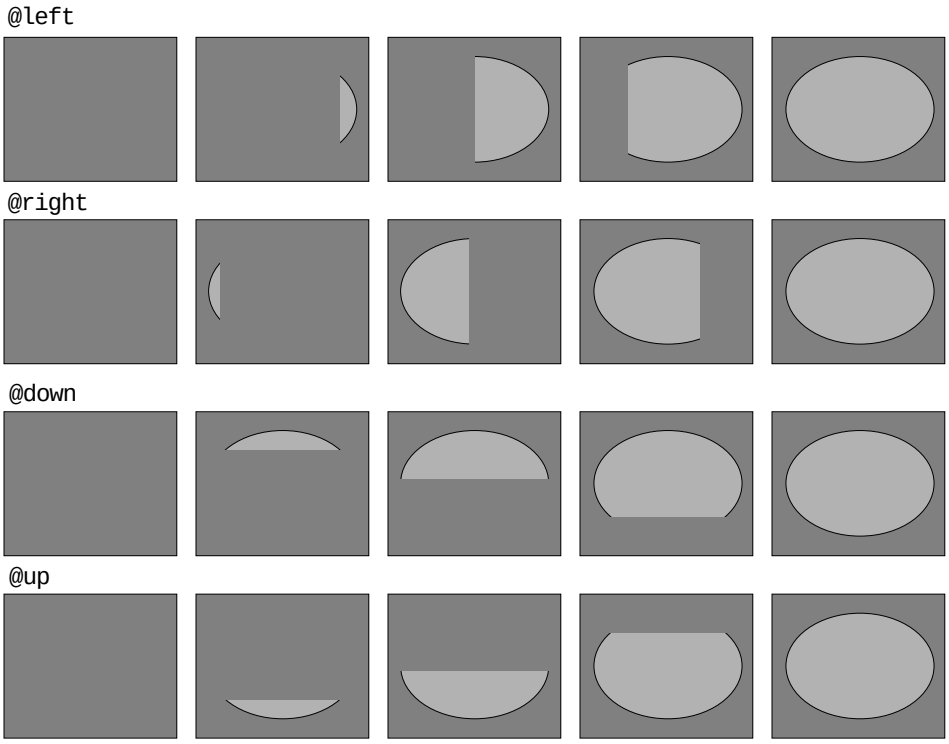
Returns a bitmap snapshot of the current state of the window *self*. If a `BitmapSurface` object is passed in, the snapshot is rendered onto it. If `undefined` is passed in, the method returns a `BitmapSurface` object that might be created on-the-fly. In the case where a `TwoDCompositor` object is associated with the window *self*, the compositor's offscreen `BitmapSurface` object is returned.

Wipe



Class type: Core class (concrete)
Resides in: ScriptX and KMP executables
Inherits from: `TransitionPlayer`
Component: Transitions

The `Wipe` class creates the visual effect of having the target presenter drawn incrementally onto the screen. You specify the direction in which the transition occurs by setting its `direction` instance variable, defined by `TransitionPlayer`. Possible values include `@left`, `@right`, `@up`, and `@down`, as shown in the graphic. For example, `@left` means that the target image wipes onto the screen moving from right to left.



Creating and Initializing a New Instance

The following script is an example of how to create a new instance of `Wipe`:

```
myWipe := new Wipe \
          duration:60 \
          direction:@left \
          target:myShape
```

The variable myWipe contains the initialized wipe transition. The transition wipes the image myShape onto the screen from right to left, and has a duration of 60 ticks.

You determine which space the transition will take effect in by adding this instance into that space. Then, when you play the transition player, myShape is "transitioned" into that space.

The new method uses the keywords defined in init.

init

init *self* [duration:*integer*] [direction:*name*]
[movingTarget:*boolean*] [useOffscreen:*boolean*] [target:*twoDPresenter*]
[boundary:*stencil*] [masterClock:*clock*] [scale: *integer*] ⇨ *(none)*

This method is inherited from TransitionPlayer with no change in keywords—refer to that class for details. Do not call init directly on an instance—it is automatically called by the new method.

Instance Variables

Inherited from Presenter:

presentedBy	subPresenters	target

Inherited from TwoDPresenter:

bBox	height	transform
boundary	IsImplicitlyDirect	width
clock	isTransparent	window
compositor	isVisible	x
direct	needsTickle	y
eventInterests	position	z
globalBoundary	stationary	
globalTransform	target	

Inherited from Clock:

callbacks	rate	ticks
effectiveRate	resolution	time
masterClock	scale	title
offset	slaveClocks	

Inherited from Player:

audioMuted	globalContrast	globalVolumeOffset
dataRate	globalHue	markerList
duration	globalPanOffset	status
globalBrightness	globalSaturation	videoBlanked

Inherited from TransitionPlayer:

autoSplice	direction	movingTarget
backgroundBrush	duration	target
cachedTarget	frame	useOffscreen

The following instance variables are defined in Slide:

direction (TransitionPlayer)

self.direction (read-write) NameClass

Specifies the direction in which the wipe transition *self* should be applied. Possible values are @left, @right, @up, @down, @southeast, @northeast, @southwest, or @northwest.

Instance Methods

Inherited from `TwoDPresenter`:

`adjustClockMaster`	`inside`	`show`
`createInterestList`	`localToSurface`	`surfaceToLocal`
`draw`	`notifyChanged`	`tickle`
`getBoundaryInParent`	`recalcRegion`	
`hide`	`refresh`	

Inherited from `Clock`:

`addPeriodicCallback`	`clockAdded`	`pause`
`addRateCallback`	`clockRemoved`	`resume`
`addScaleCallback`	`effectiveRateChanged`	`timeJumped`
`addTimeCallback`	`forEachSlave`	`waitTime`
`addTimeJumpCallback`	`isAppropriateClock`	`waitUntil`

Inherited from `Player`:

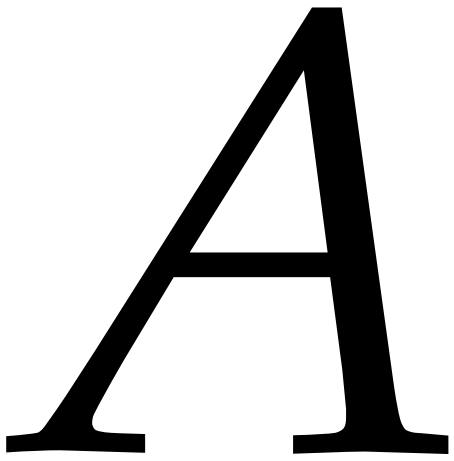
`addMarker`	`goToBegin`	`playPrepare`
`eject`	`goToEnd`	`playUnprepare`
`fastForward`	`goToMarkerFinish`	`playUntil`
`getMarker`	`goToMarkerStart`	`resume`
`getNextMarker`	`pause`	`rewind`
`getPreviousMarker`	`play`	`stop`

Inherited from `TransitionPlayer`:

`playPrepare`

APPENDIX

Loadable Extensions



This section contains an overview of the loadable extensions that can be loaded and run in the Kaleida Media Player. (Other extensions that run only in the ScriptX Development Environment are described in the *ScriptX Tools Guide*.)

ScriptX contains two kinds of loadable extensions:

- C-Loadable extensions – These classes and functions are written in C and compiled separately for each platform. These are often referred to simply as "loadable extensions."
- Scripted extensions – These classes are are written in ScriptX and saved and distributed in library containers (which run on all ScriptX platforms).

C-Loadable Extensions

The following are the loadable extensions available with this version of ScriptX that run with the Kaleida Media Player:

Loadable Transitions

Transitions provide the capability for visual effects when changing what's on the screen. Loadable transitions provide variety beyond the core set of transitions.

Classes: `Blinds`, `Checkerboard`, `DiamondIris`, `Dissolve`, `Fan`, `GarageDoor`, `Push`, `RandomChunks`, `RectIris`, `RectWipe`, `StripSlide`, `StripWipe`

For more information: See the Transitions chapter in the *ScriptX Components Guide*.

Printing

The Printing loadable extension provides the basic building blocks you need to write custom printing methods for your title. You can design custom printing methods to print a `Window` view to a page (as a bitmap), and print a `TextPresenter`, `OneOfNPresenter`, or `Document` object to a series of pages.

Classes: `Printer`, `PrinterSurface`

For more information: See the Printing chapter in the *ScriptX Components Guide*.

Loadable Media Players

We provide three loadable media players classes:

- `VFWPlayer` and `QuickTimePlayer` can be used to play movies directly from hard disk without importing them into ScriptX. These classes provide ScriptX object wrappers to the Video For Windows runtime and the QuickTime runtime, respectively.
- `CDPlayer` class is a loadable extension class which lets you play audio CD in a CD-ROM drive while running ScriptX.

Classes: `QuickTimePlayer`, `VFWPlayer`, `CDPlayer`

For more information: See the Media Player chapter in the *ScriptX Components Guide*.

External Command Interface Extension

The MCICMD extension provides access to Multimedia Command Interface commands for Microsoft Windows systems.

Classes: *(none)* However, there is one global function: `mciCommand`

For more information: See the description in the Loadable Extensions appendix of the *ScriptX Components Guide*.

Scripted Extensions

Widget Library

The Widget Library provides a set of simple user interface controls to save you time creating your user interface. In general, Widget Kit objects use fewer presenters and therefore perform better than core User Interface objects, but they are also less customizable; they are less complex, but also less flexible. For example, buttons in the Widget Library derive their appearance from stencils and not from subpresenters.

Classes: `ColorScheme`, `FontContext`, `Frame`, `Label`, `GenericButton`, `RadioButton`, `CheckBox`, `StencilButton`, `TextButton`, `PopUpButton`, `PopUpMenu`, `RadioGroup`, `ListSelection`, `ScrollBox`, `MultiListBox`,`ScrollListBox`, `ListBox`, `SimpleScrollBar`, `SmallTextEdit`,`ScrollingTextEdit`

For more information: See the User Inteface chapter in the *ScriptX Components Guide*.

APPENDIX

Exceptions

B

Exceptions

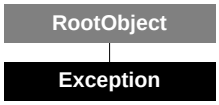
This appendix describes the `Exception` family of classes. It begins with the `Exception` class. Instances and subclasses follow alphabetically:

Exception
ClockException
CollectionException
DevicesException
DirRepException
EventException
General Exception
ImportExportError
LoaderException
MathException
MemoryException
ObjStoreException
PlayerException
ScriptError
StreamException
SystemError
TextException
ThreadException
TwoDGraphicsException

List Of Exceptions

The following list shows all the exceptions alphabetically, along with their class, and the page number where you can find more information about them in this chapter.

Exception



Inherits from: `RootObject`
Class type: Concrete
Component: Exceptions

`Exception` is the top-level exception class. An exception indicates that an error of some kind has occurred. Exceptions can be created directly as instances of `Exception`, or as instances of its subclasses.

ScriptX has many global instances of `Exception`, which are listed according to subject, in the following pages.

Each `Exception` instance has a format string that provides the default error message for the exception. This string can contain substitution characters that are replaced by real values when the exception is reported. All occurrences of `%*` in the format string are replaced by an object when the string is printed, and each occurrence of *%n* is replaced by the *n*th item in a linear collection.

To report an exception to indicate that an error has been detected, call the `report` method on an `Exception` instance. The second argument to the `report` method is an object that contains information about the error that occurred. This information is passed to the format string to replace its substitution characters. This information can also be used by any code that wants to find out the details of the exception.

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 linear collection object containing at least *n* objects.

For example, the format string for the hypothetical user-defined instance `cantBePurple` might be:

```
"I am sorry. %* clashes with my hair color."
```

When this exception is reported, the second argument to `report` should be the unsuitable color, for example:

```
function putColor object color ->
(
    if (color = "Purple" or color = "Violet" or color = "Mauve")
    do report cantBePurple color
    else object.color := color
)
```

If `putColor` is called with arguments of `person1` and `"Violet"`, a `cantBePurple` exception is reported. The exception's format string is:

```
"I am sorry. Violet clashes with my hair color."
```

See the *ScriptX Language Guide* for more information on reporting exceptions.

What Happens if You Use a Non-existent Instance Variable

Most of the system-defined exceptions are named and used intuitively, for example, the `divideByZero` exception is reported when an attempt is made to divide a number by zero. The format string says `"Attempt to divide` *n* `by zero"` where n is the number being divided.

However, if you try to access an instance variable that does not exist, you will get an `undefinedFunction` exception, which may not initially seem very intuitive. Misspelling an instance variable name is a very common mistake when writing scripts, so you might find it helpful to understand the error message that is reported in this situation.

All instance variables have a "getter" function that gets the value of the variable, and a "setter" function that sets the value of the variable. Usually these getter and setter functions are transparent to the user; in most cases they operate behind the scenes and you don't need to use them directly. However, if you try to access a non-existent instance variable, the system reports that it can't find the setter or getter function for that instance variable.

For example, suppose you enter the following:

```
global myplayer := new player
myplayer.tme -- misspelling of time
```

An `undefinedFunction` exception is reported. Its format string is:

```
tmegetter does not have a function definition
```

You can interpret this message as "There was an attempt to get the value of the `tme` instance variable, which does not exist."

Similarly, you can interpret the following exception string:

```
tmesetter does not have a function definition
```

as "There was an attempt to set the value of the `tme` instance variable, which does not exist."

Exceptions in the Kaleida Media Player

When a ScriptX title is running in the Kaleida Media Player (that is, the runtime player without the development environment), if an uncaught exception occurs, a dialog box opens up warning the user of the problem. The user's only option is to to click `OK` to close the dialog box.

Creating and Initializing a New Instance

The following script is an example of how to create a new instance of the `Exception` class:

```
exception := new Exception \
       name:"cantBePurple" \
       format:"I am sorry. %* clashes with my hair color."
```

This example creates a new instance of exception that can be used to indicate that a "cant be purple" exception has occurred.

The `new` method uses the keywords defined in `init`.

init

`init` *self* [`name:`*nameString*] [`format:`*formatString*] ⇨ *self*

self	`Exception` object
`name:`	`String` of the name for the exception
`format:`	`String` of the exception's format, which provides the default error message. This string should contain useful information about the error.

A format string consists of text and substitution characters, `%*` and `%`*n*, which are replaced by real values when the format string is printed. All occurrences of `%*` in the format string are replaced by an object when the string is printed, and each occurrence of `%`*n* is replaced by the *n*th item in a collection.

If you omit an optional keyword, its default value is used. The defaults are:

```
name:"anonymous"
format:"%*"
```

Instance Methods

prin

`prin` *self styleOrInfo stream* ⇨ *(none)*

self	`Exception` whose error message is to be printed. Note that the global variable `throwTag` is always bound to the most recently reported exception object.
styleOrInfo	Either a style argument or the input object needed by the exception's format string
stream	`Stream` to which to print. Pass this as `debug` to print to the screen.

The exception classes have a specialized implementation for the `prin` method. 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 instances, the second argument can be either:

- A style argument (one of `@normal`, `@complete`, `@debug`, `@unadorned`) in which case the exception is printed in the indicated style or
- A non-style argument. In this case, the exception's format string is printed, using the given argument to fill in the substitution characters in the format string.

You can use the global variable `throwTag` to refer to the most recently reported exception, and you can use the global variable `throwArg` to refer to the extra argument passed by `report` when it reported that exception.

For example, the default error message for the `divideByZero` exception instance is:

```
"Attempt to divide %* by zero"
```

Suppose the most recently reported exception was reported by:

```
report divideByZero 6
```

In this case, the following statement:

```
prin throwTag throwArg
```

prints the following message to the output window:

```
"Attempt to divide 6 by zero"
```

You can use the derivations of `prin`, such as `prinln` (which adds a newline after printing the format string), as normal. See the section "Output" in the chapter "*Working with Objects*" in the *ScriptX Language Guide* for more information on using the prin method and its derivatives.

report

`report` *self info* ⇨ `Exception`

self	`Exception` object
info	An object containing information about the exception that occurred

This method reports an exception. It prints the call stack (that is, the sequence of function or method calls that lead up to the function or method that caused the exception), and then prints the format string for the exception.

For example, the following statement would be called when an attempt was made to divide 10 by zero:

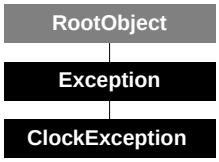
```
report divideByZero 10
```

If the exception is reported inside a `guard/catching` language construct, the system checks if the reported exception can be caught, and if so, executes the indicated consequences.

The use of exceptions is closely tied to the `guard/catching` language construct. Please see the "Exceptions" chapter of the *ScriptX Language Guide* for a full explanation of how to report and catch exceptions.

The *info* object is used by the exception's default error message. It can be an arbitrary object, in which case the object is substituted into all occurrences of `%*` in the format string, or a collection object, in which case items in the list are substituted into occurrences of `%`*n* in the format string, where *n* is a number indicating the position in the list.

ClockException



Class type: Core class (concrete)
Resides in: ScriptX and KMP executables
Inherits from: `Exception`
Component: Exceptions

`ClockException` is the class of clock exceptions.

Global Constants

cantInitClock

Indicates that the system cannot initialize the root clock in the Windows environment.

The format string is:
`"Can't initialize Windows clock: %*."`

cantStartClock

Indicates that the system cannot start the root clock in the Windows environment.

The format string is:
`"Can't start Windows clock: %*."`

cantStopClock

Indicates that the system cannot stop the root clock in the Windows environment.

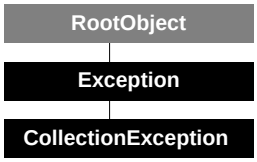
The format string is:
`"Can't stop Windows clock: %*."`

clockPeriodTooLow

Indicates that the given clock period is too low.

The format string is:
`"Root Clock period desired: (%*) ms. too low."`

CollectionException



Class type: Core class (concrete)
Resides in: ScriptX and KMP executables
Inherits from: `Exception`
Component: Exceptions

`CollectionException` is the class of collection exceptions.

Global Constants

badByte

Indicates that a bad byte was encountered in a collection.

The format string is:
`"Bad byte value %2 for collection %1."`

badKey

Reported when a bad key is given for a collection.

The format string is:
`"Bad key (%2) for collection (%1)."`

badValue

Reported when an attempt is made to add an unacceptable value to a collection.

The format string is:
`"Bad value (%2) for collection (%1)."`

boundedError

Indicates that a collection has reached its bounds, that is, it has the minimum or maximum number of items.

The format string is:
`"Collection (%*) has reached its bounds."`

collectionMem

Indicates that a collection has a memory problem.

The format string is:
`"Collection (%*) has a memory problem."`

immutable

Reported when an attempt is made to modify an immutable collection.

The format string is:

"Modification attempted on immutable collection (%*)."

inappropriateObject

Reported when an inappropriate object is passed to another object.

The format string is:
"(%1) is not an appropriate object for (%2)."

invalidTargetCollection

Reported when a non-collection object is specified in the place where a collection is needed.

The format string is:
"Target (%*) is not a collection."

iteratorBoundary

Reported when an attempt is made to use an iterator out of bounds.

The format string is:
"Attempt to use iterator (%*) out of bounds."

iteratorMismatch

Reported when an attempt is made to use an iterator on a collection that does not support the iterator.

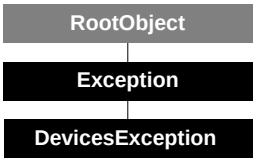
The format string is:
"Iterator (%*) does not match Collection."

notReversible

Reported when an attempt is made to traverse a non-reversible collection in reverse.

The format string is:
"Collection (%*) cannot be traversed in reverse."

DevicesException



Class type: Core class (concrete)
Resides in: ScriptX and KMP executables
Inherits from: `Exception`
Component: Exceptions

`DevicesException` is the class of exceptions for errors related to using the `Devices` class.

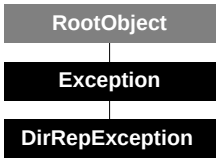
Global Constants

invalidDeviceID

Indicates that an invalid device ID was received by a method where a valid device ID was expected.

The format string is:
`"Invalid device ID, %1, given to method %2 of class %3."`

DirRepException



Class type: Core class (concrete)
Resides in: ScriptX and KMP executables
Inherits from: `Exception`
Component: Exceptions

`DirRepException` is the class of exceptions for errors related to using the `DirRep` class.

Global Constants

dupName

Indicates that the file name specified is already in use.

The format string is:
```
"Duplicate name: %3 within %2 while calling %1."
```

fileAlreadyOpen

Indicates that an attempt was made to open a file that is already open.

The format string is:
```
"File is already open: %3 within %2 while calling %1."
```

fileLocked

Indicates that an attempt was made to open a locked file.

The format string is:
```
"File is locked: %3 within %2 while calling %1."
```

fileOrDirBusy

Indicates that the file specified is busy or or the directory specified is not empty.

The format string is:
```
"File is busy or directory is not empty: %3 within %2 while calling
%1."
```

nameNotFound

Indicates that the given name is not found.

The format string is:
```
"Directory or file not found: %3 within %2 while calling %1."
```

notADir

Indicates that the name specified for a directory does not match any existing directory.

The format string is:
```
"Not a directory: %3 within %2 while calling %1."
```

notAFile

Indicates that the name specified for a file does not match any existing file.

The format string is:
```
"Not a file: %3 within %2 while calling %1."
```

osErr

Indicates that an unknown operating system error has occurred.

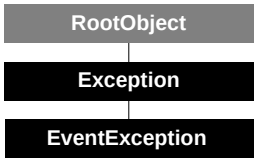
The format string is:
```
"Unknown OS error: #%3: involving %4, within %2 while calling %1."
```

pathTooShort

Indicates that a path name is too short.

The format string is:
```
"Path too short: while calling %1 on %2."
```

EventException



Class type: Core class (concrete)
Resides in: ScriptX and KMP executables
Inherits from: `Exception`
Component: Exceptions

`EventException` is the class of event exceptions.

Global Constants

cantBroadcast

Reported when the event system can't broadcast events it has been asked to broadcast.

The format string is:
`"Can't broadcast %* events."`

cantSignal

Reported when the event system can't signal events it has been asked to signal.

The format string is:
`"Can't signal %* events."`

invalidEventPriority

Reported when an invalid event priority is given.

The format string is:
`"Invalid event priority %*, must be between 1 and 15."`

noEventReceiver

Reported when there is no receiver for an event.

The format string is:
`"There is no eventReceiver defined for %*."`

notAValidRejectQueue

Reported when an attempt is made to return a non-queue, or a locked queue, to a reject queue pool.

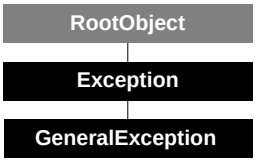
The format string is:
`"%* is not a valid queue to return to the pool because it is not a queue or is locked."`

wrongRejectQueuePool

Reported when a rejected event queue is sent to the wrong reject pool.

The format string is:
`"%* is being returned to the wrong reject queue pool."`

GeneralException



Class type: Core class (concrete)
Resides in: ScriptX and KMP executables
Inherits from: `Exception`
Component: Exceptions

`GeneralException` is the class of general exceptions. Exceptions that are relevant to more than one component belong in this class.

Global Constants

badParameter

Indicates that a bad parameter has been passed to a method. When this exception is reported, the fourth item in the collection passed to the exception's `report` method should be a string explaining why the parameter is bad.

The format string is:

```
"Invalid parameter (%1) given as argument to %2 on %3. \n-- ** %4"
```

where:

%1 is the invalid parameter.

%2 is the method to which the bad parameter was passed.

%3 is the object that the method was called on.

%4 is a string describing why the parameter is bad.

An example of reporting a `badParameter` exception is:

```
report badParameter #(-7, objectA, methodB, "Expected a non-negative
number")
```

The reported exception would have the following format string:

```
"Invalid parameter (-7) given as argument to methodA on objectB.
"Expected a non-negative number."
```

badFunctionParameter

Reported when an attempt is made to call a function with an invalid argument.

The format string is:

```
"Invalid parameter (%1) given as argument to %2.\n-- ** %3
```

where:

%1 is the invalid parameter.

%2 is the function to which the bad parameter was passed.

%3 is a string describing why the parameter is bad.

badIV

Reported when an attempt is made to install an invalid value in an instance variable.

The format string is:
`"%1 is not a valid value for instance variable %2 of object %3."`

cantCoerce

Reported when an attempt is made to coerce an object into a class that it cannot be coerced to.

The format string is:
`"Cannot coerce %1 into %2."`

debugInterrupt

Reported when the debug thread (usually the Listener thread) is interrupted.

The format string is:
`"Thread interrupt (%*)."`

generalError

Indicates that a general error has occurred. Report this exception when you want to indicate that an error has occurred, but it is not classifiable as any pre-defined exception. When calling `report` on this exception, pass a string that describes the problem in details.

The format string is:
`*%*"`

ivAccessDenied

Reported when an attempt is made to write a value into a read-only instance variable.

The format string is:
`"IV access denied: calling %1 on %2 is not allowed."`

%1 is the method that tried to do write to the read-only instance variable, and %2 is the object that the method was called on.

When you try to access an instance variable that does not exist, you do not get an `ivAccessDenied` exception. Instead, an `undefinedFunction` exception is reported, since the system cannot find a setter or getter function for the non-existent instance variable.

keywordRequired

Indicates that an attempt was made to call a function or method that takes keyword arguments where a required keyword argument was missing.

The format string is:
`"Keyword %* is required."`

newOnAbstractClass

Reported when the `new` method is called on an abstract class, that is, a class that is not directly instantiable.

The format string is:

```
"Class %* is not directly instantiable".
```

notImplementedBySubclass

Indicates that a method was called on a class, where the method should have been specialized for the class but wasn't.

```
"Method %1 must be implemented by a subclass of %2, but is not
implemented by %3".
```

shouldntHappen

Reported when something happens that shouldn't be allowed to happen. You should never encounter this error when using the ScriptX core classes. If you do, call Kaleida Technical Support immediately.

The format string is:

```
"Internal System Error: %*."
```

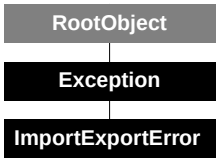
unimplemented

Reported when an attempt is made to use unimplemented functionality.

The format string is:

```
"Not yet implemented %*."
```

ImportExportError



Class type: Core class (concrete)
Resides in: ScriptX and KMP executables
Inherits from: `Exception`
Component: Exceptions

`ImportExportError` is the class of exceptions related to importing and exporting.

Global Constants

importerExporterLoadFailed

Indicates that a failure occurred while loading an importer.

The format string is:

```
"Failed loading the code for the importer/exporter
-- MediaCategory:%1, inputMediaType:%2 and mediaOutputType:%3"
```

importerExporterNotFound

Indicates that an importer or exporter could not be found when one was required. This exception often indicates that incorrect arguments were passed to the `importMedia` method for `theImportExportEngine` global instance. See the chapter about importers in the *ScriptX Developers Guide* for a discussion of the different arguments required for importing various kinds of media.

The format string is:

```
"An importer/exporter could not be found for
MediaCategory:%1, inputMediaType:%2 and mediaOutputType:%3."
```

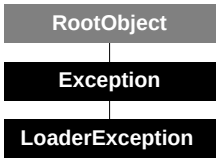
importFailed

Indicates that an attempt to import media failed.

The format string is:

```
"Import from %3 failed (%5).
-- Media category:%1, inputMedia:%2, and mediaCategoryOutputType:%4."
```

LoaderException



Class type: Core class (concrete)
Resides in: ScriptX and KMP executables
Inherits from: `Exception`
Component: Exceptions

`LoaderException` is the class of loader exceptions.

Global Constants

cantLoadClass

Reported when the loader tries to load a class that is not loadable, or has other problems loading.

The format string is:
`"Cannot dynamically load class %*."`

loaderInitError

Reported when an error occurs during initialization of a loadable unit. Specifically, `LoaderInitError` is reported by the `assertClass` function if the asserted symbol was not relocated.

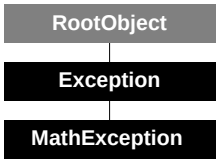
The format string is:
`"Loadable Unit requires %* class to be initialized."`

loaderUnresolvedError

This exception is reported if dynamically loaded code attempts to call a method or function that the Loader system could not resolve during loading.

The format string is:
`"Loader: Unresolved function called."`

MathException



Class type: Core class (concrete)
Resides in: ScriptX and KMP executables
Inherits from: `Exception`
Component: Exceptions

`MathException` is the class of math-related exceptions.

Global Constants

cantSetScale

Indicates that an attempt was made to set the value of the `scale` instance variable on an instance of the `Date` class.

The format string is:

```
"An attempt was made to set the scale instance variable of %*. You
can't set the scale instance variable of an instance of the class
Date."
```

divideByZero

Reported when an attempt is made to divide a number by 0.

The format string is:

```
"Attempt to divide %* by zero."
```

illegalOperation

Indicates that an illegal operation occurred.

The format string is:

```
"Illegal operation: %1 %2 %3"
```

invalidNumber

Reported when an attempt is made to use a value as a number when the value is not a valid number.

The format string is:

```
"Not a valid number: %*."
```

lossOfRange

Reported when an attempt is made to coerce an object, usually a `Number` of some kind, to another kind of object, where that type of coercion is not allowed because it would result in too much loss of range. For example, this exception would be reported by an attempt to coerce 65000 to a `Fixed` object.

The format string is:

```
"%1 can not be represented by a %2."
```

undefinedResult

Indicates an `undefined` result.

The format string is:
```
"Undefined result. %1, %2, %3."
```

unordered

Reported when an attempt is made to compare two unordered objects.

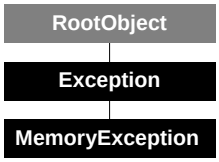
The format string is:
```
"Attempt to compare unordered items (%1, %2)."
```

valueOutOfRange

Indicates that an attempt was made to install a value in an instance variable where the value is out of the acceptable range for the instance variable.

The format string is:
```
"Value %1 is out of range for the iv %2. The value must be between %3
and %4 inclusive."
```

MemoryException



Class type: Core class (concrete)
Resides in: ScriptX and KMP executables
Inherits from: `Exception`
Component: Exceptions

`MemoryException` is the class of exceptions for memory problems.

Global Constants

outOfDynamicMemory

The dynamic memory ran out.

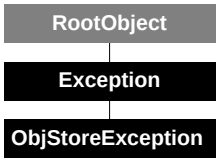
The format string is:
`"Out Of Dynamic Memory."`

outOfMemory

The system is out of memory. ScriptX usually crashes when this exception is reported.

The format string is:
`"Out of memory -- Fatal Error."`

ObjStoreException



Class type: Core class (concrete)
Resides in: ScriptX and KMP executables
Inherits from: `Exception`
Component: Exceptions

`ObjStoreException` is the class of exceptions that represent object store errors.

Global Constants

badContainerVersion

Reported when an attempt is made to use a file whose container is incompatible with this version of ScriptX.

The format string is:

```
""File %1 (with container version %2) is incompatible with this version of ScriptX")"
```

cannotDeflate

Indicates that an attempt was made to deflate an object to store it, when the object cannot be stored.

The format string is:

```
"Object (%*) cannot be made persistent."
```

cannotStore

Indicates that an attempt was made to store an object as a top-level object when it cannot be stored as such.

The format string is:

```
"Object (%*) cannot be stored as a top-level object."
```

cantStoreSubstrateClass

Indicates that an attempt was made to store a substrate class, for example one that was written in C and dynamically loaded. Substrate classes cannot be stored.

The format string is:

```
"Cannot store substrate or loaded OIC class: %*."
```

illegalPurge

Reported when an attempt is made to purge an object that cannot be purged..

The format string is:

```
"It is illegal to purge object (%*)."
```

noFileForContainer

Reported when an attempt is made to create or open a container, specifying the mode as something other than `@create` or `@update`, when the container does not already exist.

The format string is:
```
"Container file %* doesn't exist. (Use @create or @update to create.)"
```

noFileForPath

Indicates that no file can be found that matches a given path in a container search list.

The format string is:
```
"No container file matching path %* found in any directory in
theContainerSearchList"
```

noGlobalObjectWithName

Indicates that a global object was expected but not found.

The format string is:
```
"Global Object named %* expected but not found."
```

noProxyForObject

Indicates that a proxy was expected but not found for an object.

The format string is:
```
"In %1 (%2), no PersistentProxy found for object %3."
```

prematurePurge

Indicates that an attempt was made to purge an object from the object store before it had been stored.

The format string is:
```
"Can't purge object (%*) before it is stored."
```

storageSystemError

Indicates that an error occurred in the Object Store system. The format string includes a `String` that describes the error that occurred.

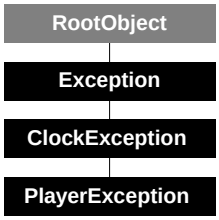
The format string is:
```
"Storage System Error: %*"
```

wrongContainer

Indicates that an attempt was made to put an object in a container when it is already in a another container.

The format string is:
```
"Can't put object %1 into container %2 (It is already in container
%3)."
```

PlayerException



Class type: Core class (concrete)
Resides in: ScriptX and KMP executables
Inherits from: `ClockException`
Component: Exceptions

The `PlayerException` class is the class of all exceptions due to problems with media players, including exceptions due to problems in a player's stream or problems encountered by devices or drivers when playing media.

Global Constants

badAIFF

Reported when an invalid AIFF stream is encountered when importing a file that is supposed to be an AIFF file containing digitized sound.

The format string is:
`"Stream (%1) is not a valid AIFF stream. (%2)"`

badInputStream

Reported when the value of the `inputStream` instance variable of a `MediaStream` is a stream that contains incorrectly formatted data. This exception is usually reported when a player tries to play but can't play because its media stream has an input stream that contains incorrectly formatted data.

The format string is:
`"%1 contains incorrectly formatted data for %2 on %3. (%4)"`

where:

`%1` is the stream that is unacceptable.

`%2` is the method reporting the exception.

`%3` is the media stream containing the badly formatted input stream.

`%4` is a string describing why the input stream was unacceptable.

cantBackward

Reported when an attempt is made to play a player backwards when the player does not have the capability to play backwards.

The format string is:
`"Player %* cannot play backwards."`

cantCloseWaveDev

Reported when a problem occurs while trying to close a WAVE device.

The format string is:
"Cannot close WAVE device: %*."

cantInitWaveDev

Reported when a problem occurs while trying to initialize a WAVE output device.

The format string is:
"Cannot initialize WAVE device: %*."

cantOpenWaveDev

Reported when a problem occurs while trying to open a WAVE output device.

The format string is:
"Cannot open WAVE device: %*."

cantQueueWaveBuffer

Reported when a problem occurs when trying to send a buffer to a WAVE device.

The format string is:
"Can't queue buffer to WAVE device: %*."

chunkBounds

Reported when a an attempt is made to read past the end of a chunk stream while playing an interleaved movie.

The format string is:
"Chunk bounds error in %1 on %2. (%3)"

codecCantDealWithParameters

Reported when a trying to use a codec with parameters that it cannot deal with.

This exception could be thrown as a result of using video or compressed bitmaps if ScriptX is running on a monochrome screen. In this case, ScriptX might throw an exception saying that the codec can't deal with the bit depth of the incoming screen. Since the required configuration for ScriptX includes 8-bit or higher displays, this should be rare.

The format string is:
"Codec %1 cannot deal with the %2 parameter: %3"

codecFormatNotSupported

Reported when an attempt is made to use codec format on a platform that does not support codec format.

The format string is:
"Codec format %* is not supported by ScriptX or the native codec manager. Video cannot be played."

invalidInputStream

Reported when the `inputStream` instance variable on a `MediaStream` contains an object that is not a stream (or other class appropriate to the particular media stream in question). This exception is usually reported when a player tries to play but can’t play because its media stream has an invalid input stream.

The format string is:
`"%1 is not a valid input stream for %2 on %3. (%4)"`

where:

%1 is the stream that is unacceptable.

%2 is the method reporting the exception.

%3 is the media stream whose `inputStream` instance variable contains the invalid stream.

%4 is a string describing why the input stream was unacceptable.

invalidMediaStream

Reported when the `mediaStream` instance variable on a `MediaStreamPlayer` contains an object that is not a `mediaStream` (or other class appropriate to the particular `MediaStreamPlayer` in question). This exception is usually reported when a player tries to play but can't play because its media stream has an invalid value.

The format string is:
`"%1 is not a valid media stream for %2 on %3. (%4)"`

where:

%1 is stream that is unacceptable.

%2 is the method reporting the exception.

%3 is the mediaStreamPlayer whose mediaStream instance variable contains the invalid value.

%4 is a string describing why the media stream was unacceptable.

midiDeviceError

Indicates that an error has occurred while trying to write to a MIDI device.

The format string is:
`"A MIDI device error occurred. The MIDI device is either busy or unavailable."`

missingMediaInformation

Reported when a stream that a media player is attempting to play is missing some media information that is needed to enable the media to play.

The format string is:
`"%1 does not contain all of the media information (%2) needed to play this stream. It may be necessary to import the data again.")"`

noMIDIDrivers

Indicates that no MIDI drivers are available.

The format string is:
`"No MIDI drivers are available. Your system might not have MIDI support."`

noSoundResources

Indicates that no sound resources are available for playing sound.

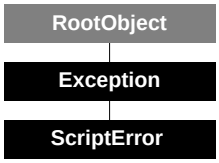
The format string is:
"All sound resources are in use or do not exist."

unsupportedMidiFileType

Indicates that the given file type is not supported.

The format string is:
"Standard MIDI file type %* not supported. Only type 0 supported."

ScriptError



Class type: Core class (concrete)
Resides in: ScriptX and KMP executables
Inherits from: `Exception`
Component: Exceptions

`ScriptError` is the class of exceptions that represent ScriptX script compilation errors.

Global Constants

assignToConstant

Reported when an attempt is made to assign a value to a read only variable.

The format string is:
`"Illegal assignment to constant %*."`

changeOfOwnership

Reported when an illegal attempt is made to change the ownership of an object.

The format string is:
`"%1 is owned by %2 and cannot be redefined."`

classNotModifiable

Reported when an attempt is made to redefine a class that already has instances or subclasses or is a ScriptX core class.

The format string is:
`"Class not modifiable because it is a core class or has instances or subclasses: %*."`

loadNameClash

Reported when a name clash occurs during loading.

The format string is:
`"Can't load %1 -- name already bound: %2."`

moduleChangeOfContainer

Reported when a an attempt is made to define a module that is already defined by another title container.

The format string is:
`"Module %1 already exists and cannot be defined by container %2."`

moduleRedefinition

Reported when an attempt is made to redefine a module.

The format string is:
"%* cannot be redefined, because redefinition hasn't been implemented yet."

noSuchModule

Reported when a an attempt is made to use a module that does not exist.

The format string is:
"No module named %u*."

notAClass

Reported when an attempt is made to redefine a class that is not really a class.

The format string is:
"Attempted class redefinition on non-class: %*."

notEnoughArguments

Reported when an attempt is made to call a method or function with too few arguments.

The format string is:
"Not enough arguments to %1. %2 supplied, at least %3 required."

syntaxError

Reported when a syntax error is encountered in a script.

The format string is:
"Syntax error: (%1) at %2."

tooManyArguments

Reported when an attempt is made to call a method or function with too many arguments.

The format string is:
"Too many arguments to %1. %2 supplied, no more than %3 allowed."

tooManyVariablesInModule

Reported when an attempt is made to reference too many variables in a module.

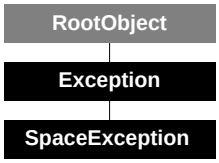
The format string is:
"Too many variables referenced in %u*."

uninitializedVariable

Reported when an attempt is made to use a variable that has not been initialized.

The format string is:
"%* does not have a variable value."

SpaceException



Class type: Core class (concrete)
Resides in: ScriptX and KMP executables
Inherits from: `Exception`
Component: Exceptions

`SpaceException` is the class of exceptions that represent errors to do with using spaces.

Global Constants

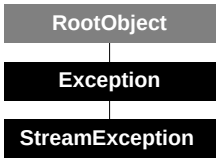
notSameSpace

Reported when an attempt is made to use a controller on an object that is not in the same space as the controller.

The format string is:

```
"(%1) is not in the same space that (%2) is controlling."
```

StreamException



Class type: Core class (concrete)
Resides in: ScriptX and KMP executables
Inherits from: `Exception`
Component: Exceptions

The `StreamException` class is the class of exceptions reported when problems occur with most kinds of streams. (Exceptions caused by problems related to streams used by media players are instances of the class `PlayerException`.)

Global Constants

cantAlterSize

Reported when an attempt is made to change the size of a stream whose size cannot be changed.

The format string is:
`"Stream size cannot be altered: %*."`

cantPlug

Reported when an attempt is made to plug a stream that cannot be plugged.

The format string is:
`"Stream not plugable: %*."`

cantRead

Reported when an attempt is made to read a non-readable stream.

The format string is:
`"Stream not readable: %*."`

cantSeek

Reported when an attempt is made to seek a non-seekable stream.

The format string is:
`"Stream not seekable: %*."`

cantWrite

Reported when an attempt is made to write to a non-writable stream.

The format string is:
`"Stream not writable: %*."`

noSpace

Reported when an attempt is made to perform a task on a stream, where the stream is too small, or there is not enough space available in the stream, to support the task.

The format string is:
`"Cannot oblige request; stream too small or space not available: %*."`

readError

Indicates that an error occurred while reading a stream.

The format string is:
`"Read error: %*."`

seekBounds

Reported when an attempt is made to seek to a place in a stream that is out of the seekable bounds of the stream.

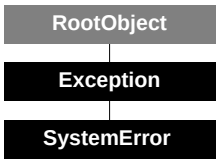
The format string is:
`"Seek out of bounds: %*."`

writeError

Indicates that an error occurred while writing to a stream.

The format string is:
`"Write error: %*."`

SystemError



Class type: Core class (concrete)
Resides in: ScriptX and KMP executables
Inherits from: `Exception`
Component: Exceptions

`SystemError` is the class of system exceptions. When a system error is reported, it indicates that a problem occurred at the substrate (OIC) level. You should not write code to report these exceptions in your ScriptX script; they will be reported automatically by the substrate as needed.

Global Constants

deletingUsedModule

Reported if you call `deleteModule` on a module that is used by another module.

The format string is:
`"%1 cannot be deleted, because these modules use it: %2"`

invalidObjectUse

Reported when a method is called on a non-object.

The format string is:
`"Method %1 called on %2"`

newNotClass

Reported when an attempt is made to call the `new` method on a non-class.

The format string is:
`"New on non-class (%*)."`

noClassMethod

Reported when an attempt is made to call a class method on a class that does not implement that method.

The format string is:
`"No %1 class method for %2."`

noMethod

Reported when an attempt is made to call a method on an object that does not implement that method.

The format string is:
`"No %1 instance method for %2."`

superNotClass

Reported when an attempt is made to initialize a class when the superclass specified for it is not a class.

The format string is:
```
"*Attempting to use %1 as a superclass for %2."
```

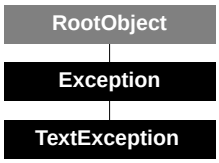
undefinedFunction

Reported when an attempt is made to call a function that is not defined.

The format string is:
```
"%* does not have a function definition."
```

Note that when you try to access an instance variable that does not exist, an `undefinedFunction` exception is reported, since the system cannot find a setter or getter function for the non-existent instance variable.

TextException



Class type: Core class (concrete)
Resides in: ScriptX and KMP executables
Inherits from: `Exception`
Component: Exceptions

`TextException` is the class of exceptions that can be reported when using text and fonts.

Global Constants

badFirstUTF

Reported when the first byte in a UTF sequence is bad.

The format string is:
`"Bad first byte (%*) in UTF sequence."`

badFontName

Reported when the given font name has not been specified as a string.

The format string is:
`"Font name (%*) is not a String."`

badTextAttr

Reported when the given font name has not been passed as a string.

The format string is:
`"Bad value for Text attribute %1: %2"`

`%1` is the attribute, for example, `@size`, and `%2` is the invalid value for the attribute.

badOtherUTF

Reported when the byte that should come first in a UTF sequence is not the first byte.

The format string is:
`"UTF first byte (%*) occurs after first position in sequence."`

notAFont

Reported when a given font is not a font.

The format string is:
`"%* is not a font."`

notAString

Reported when a non-string value was received when a string was expected.

The format string is:
"%* is not a string."

unknownFont

Reported when ScriptX cannot find a font that was supplied as a parameter where a font was expected.

The format string is:
"Font (%*) not found."

unimplementedFontOp

Reported when an attempt is made to perform an unimplemented operation (such as rotation) on a font.

The format string is:
"Unimplemented font operation: %*."

UTFBadRune

Reported when an invalid character is found in a UTF sequence.

The format string is:
"Invalid character value (%*) for UTF sequence."

UTFNoFirst

Reported when an no first byte can be found in a UTF sequence.

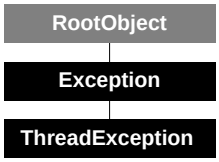
The format string is:
"Could not find first byte in UTF sequence."

zeroInString

Reported when a nul byte is found in the middle of a string. (

The format string is:
"Nul (zero) byte occurred inside a String."

ThreadException



Class type: Core class (concrete)
Resides in: ScriptX and KMP executables
Inherits from: `Exception`
Component: Exceptions

`ThreadException` is the class of exceptions related to using threads.

Global Constants

alreadyOwner

Reported when an operation is attempted to make a thread own a lock it already owns.

The format string is:
`"Lock (%*) is already owned by this thread."`

brokenPipe

Reported when an operation is attempted on a broken pipe.

The format string is:
`"Operation attempted on broken pipe (%*)."`

noMoreInterruptItems

Reported when the thread system runs out of interrupt items.

The format string is:
`"Thread system has run out of interrupt items."`

notLocked

Reported when an attempt is made to unlock a lock, which is not actually locked.

The format string is:
`"Lock (%*) is not locked."`

stackOverflow

Reported when a thread overflows its stack.

The format string is:
`"Thread (%*) overflew its stack."`

threadProhibited

Reported when a thread attempts to perform an action that it is prohibited from performing.

The format string is:

```
"Thread (%*) prohibited from action."
```

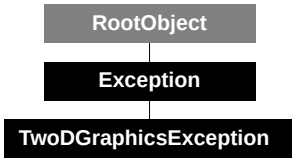
wrongOwner

Reported when a relinquishing thread attempts to unlock a lock that it does not own.

The format string is:

```
"Lock (%*) is not owned by the relinquishing thread."
```

TwoDGraphicsException



Class type: Core class (concrete)
Resides in: ScriptX and KMP executables
Inherits from: `Exception`
Component: Exceptions

`TwoDGraphics` is the class of exceptions related to using 2D Graphics.

Global Constants

cantDropData

Reported when the `dropData` method is called on a `Bitmap` object that is either not compressed or was not loaded from a title or library container.

The format string is:

```
"Can't dropData for %*. It needs to be a persistent object or a
compressed bitmap."
```

A P P E N D I X

Glossary

C

Glossary

The following is a list of terms used in this manual. The `Point` class is used throughout as an example to illustrate these terms.

abstract class– A type of class designed for subclassing rather than instantiating. An abstract class can range from containing no implementation (a true abstract class) to containing full implementation (a true mixin class). Contrast with *concrete.*

The term "abstract method" means a method that has no implementation.

accessory – A set of classes or instances that can be dynamically added to a running title. It is intended to incrementally add data or behavior to titles. Like a library, it may be usable in only one title or in many titles. Examples include a tape measure that can be used to measure the size of objects, and an inspector that can analyze the state of objects in a title.

attribute – A property of an object that has a special implementation, through its own accessor methods. Examples are the attributes of `Text` and `TextPresenter` objects.

class – An object that defines a set of variables and methods for a set of similar objects, called instances. Instances can be created from a class. For example, `Point` is a class that defines instance variables (`x`, `y`) and instance methods (`copy`, `transform`). ScriptX allows you to define and specialize your own classes.

class method – A method that operates on a class. A class method defines a behavior of its class. For example, `new` is a class method that operates on the `Point` class to create an instance:

```
pt := new Point   -- Creates a new instance of Point
```

class variable – A variable of a particular class; this variable holds some state information for that class. ScriptX has very few class variables in its core classes.

concrete class – A type of class that can be instantiated. Some concrete classes are instantiated by the system and cannot be instantiated by the author— `Boolean`, for example, can have only two instances, `true` and `false`. In general, a concrete class has a `new` class method for creating instances. For some concrete classes, this method is not visible at the scripter level. For example, new instances of the `Number` subclasses are automatically generated by the compiler as it encounters numbers in a script. Contrast with *abstract.*

core class – Any class that resides in the Kaleida Media Player executable file; therefore it does not include loadable or scripted classes. Technically, ScriptX has another set of core classes that belong to the ScriptX executable (development environment), which is a superset of the KMP core classes.

frame – One complete image in the sequence of images that makes up a time-based graphic presentation. A typical presentation might run at 10 to 30 frames per second.

frame buffer – See "off-screen buffer."

generic function – A function that calls one of several methods according to the class passed into the function (see Figure 4-34). For example, `init` is a generic function. When `init` is called on an instance of `Point`, the generic function redirects the call to the Point class's implementation of `init`. In this way, a generic function can call a uniqe implementation for each kind of instance. The result is that each class has its own way of initializing its instances—see *polymorphism*.

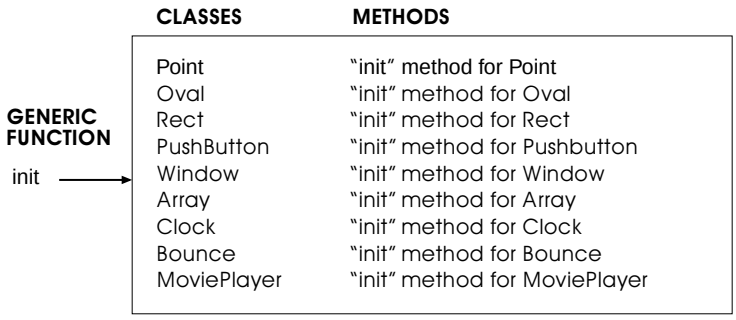


Figure 4-34: When the generic function `init` is called on an instance, it calls the `init` method corresponding to that class.

garbage collector – An independent process that operates incrementally and invisibly while ScriptX is running, reclaiming memory that is assigned to objects that are no longer in use.

implement – To provide functionality by way of a script or other code.

inheritance – A relationship between classes where a class shares the variables and methods defined in its superclasses. The further down the inheritance hierarchy you go, the more specialized classes become.

Methods and variables inherit differently in ScriptX—while a class inherits entire methods (both syntax and implementation) from its superclasses, it inherits only the name and not the value of instance variables from its superclasses.

instance – An individual object that is created from a class. An instance can have its own instance variables and instance methods. The following example creates an instance of the `Point` class:

```
new Point         -- Creates a new instance of Point
```

Also see "object."

instance method – A method that operates on an instance. In the following example, `xSetter` is an instance method that operates on `pt`, an instance of the `Point` class.

```
pt := new Point   -- Creates a new instance of Point
xSetter pt 100    -- 'xSetter' is an instance method
```

instance variable – A variable of a particular instance; this variable holds some state information for that instance. The value of an instance variable is kept with the instance (in contrast to a class variable, where the value is kept with the class). One instance of `Point` might have an x-location of 100, another instance might have an x-location of 50. Thus, the values of instance variables distinguish instances of the same class. In the following example, `x` is an instance variable.

```
pt := new Point   -- Creates a new instance of Point
pt.x := 100       -- Sets x of pt to 100
```

instantiate – To create an instance from a class. In ScriptX, you generally do this with the `new` method. Some instances, such as numbers and strings, can be created automatically by ScriptX without explicitly using the `new` method.

Kaleida Media Player – The runtime environment, including loadables, where ScriptX titles and applications can be played.

kind of– Any class or superclass of an instance. For example, an instance of `Point` is a kind of `Stencil`, because `Point` is a subclass of `Stencil`.

KMP– See "Kaleida Media Player."

library – ScriptX code, saved in a library container, that is intended to be reused by one or more titles. A library can contain any kinds of objects or classes, including text, graphics, animation, audio, video or ScriptX code.

metaclass – The class of a class. For example, the `metaPoint` class is the class of the `Point` class.

method – A function that is defined and implemented in a class or an instance of a class. A *class* method requires a class as its first argument; an *instance* method requires an instance as its first argument. When used alone, the term method could apply to either—its meaning depends on the context. The methods of a class or instance are often called its *behavior*.

You can easily distinguish between class methods and instance methods by looking at whether its first argument is a class or an instance. In the following example, `new` is a class method, since it operates on `Point`, a class. However, `xSetter` is an instance method, since it operates on `pt`, an instance.

```
pt := new Point   -- 'new' is a class method
xSetter pt 100    -- 'xSetter' is an instance method
```

method instance variable – A "method" instance variable is an instance variable that is accessed with underlying "getter" and "setter" methods instead of accessed directly. (In fact, there might be no memory allocated for a particular instance variable.) When you read the variable, an underlying "getter" method is called, and when you write to it, a "setter" method is called. For example, the methods to set and get the `x` instance variable are `xSetter` and `xGetter`.

All accessible instance variables in ScriptX are method instance variables. A method instance variable may or may not have a slot (see "slot instance variable").

mixin class – A type of class that can usefully be mixed into other classes. All classes in ScriptX are technically mixin classes, although they don't all add functionality. The classes `Dragger` and `SequenceCursor` are true mixin classes in that they contain a full implementation—by mixing them in you automatically get the added functionality you want without further implementation.

module – Aname space, similar to the packaging system in the Lisp language. Names defined in one module are visible to another module only if they are exported from the first module and imported into the second. For example, unless you export it, a global variable is accessible only within the module where it is defined.

multiple inheritance – The ability for a class to be defined directly from two or more classes. This resulting class inherits variables and methods from these direct superclasses and is said to "multiply inherit" from them.

name literal – A series of characters that begins with an "at" sign (`@`), which you can use as a value to make your code more readable, where you might otherwise use strings. A name literal is an instance of `NameClass`, and is efficient than a string. For example, instead of using the Boolean `true` to indicate a procedure has succeeded, you could create a name for it: @succeeded.

object – Any class or instance of a class. Each object is represented by some memory for a set of variables and methods. In ScriptX, not only are all instances objects, but all classes are also objects, because a class is an instance of its metaclass. Throughout this manual, the term "object" generally refers to instances, not classes. For example, the term "`Point` object" means an instance of the `Point` class, rather than the `Point` class itself. Also see "instance."

off-screen buffer – The off-screen area of memory where the changed parts of a frame are constructed. Once all changed presenters have drawn to the frame buffer, the image is transferred to the display surface for viewing by the user.

override – Given a method defined in a class, to override that method means to redefine that method in a subclass such that it takes precedence. Instances of that subclass will then have the new behavior defined by the overriding method.

persistent object – An object actually stored in a `StorageContainer`.

polymorphism – The ability of separate objects to execute their own implementations of methods in response to a common generic function (see *generic function*). Thus, `new Point` and `new Rect` execute different `new` methods.

reference to – When object A has a reference to object B, that means either object A has an instance variable that holds B, or A is a collection that contains B. For example, a 2D shape has a reference to the bitmap it displays, by way of its `target` instance variable that holds the bitmap.

ScriptX development environment – The ScriptX executable, Listener, tools and loadables in which the ScriptX source scripts can be compiled and run. ScriptX titles, applications, and tools can be played in this environment.

ScriptX runtime environment – The portion of the ScriptX executable that runs the ScriptX compiled language and core classes. Both the ScriptX development environment and the Kaleida Media Player contain the ScriptX runtime environment.

sealed class – A type of class that cannot be subclassed. Very few classes in ScriptX are sealed. The `Number` class is an example of a sealed class. Note that sealed classes can have predefined subclasses—for example, `Number` has the subclass `Fixed`.

slot instance variable – Also called a *slot*. A "slot" instance variable is an instance variable that has a slot of memory that holds the state. A slot instance variable is accessed with underlying "getter" and "setter" methods instead of accessed directly, the same as method instance variables. (In fact, there might be no memory allocated for a particular instance variable.) When you read the variable, an underlying "getter" method is called, and when you write to it, a "setter" method is called. For example, the methods to set and get the `x` instance variable are `xSetter` and `xGetter`.

Not all instance variables in ScriptX are slot instance variables. (See "method instance variable").

specialize – To define methods or variables for a particular instance or class. This can involve creating new methods and variables, redefining existing ones, or inheriting existing ones through multiple inheritance.

state – The condition defined by instance variables, class variables, and global variables. Given an instance of the `Point` class, its state is defined by its x and y values.

subclass – *(noun)* A class that inherits variables and methods from one or more classes. `Point` is a subclass of `RootObject`. A subclass is generally more specialized than the classes it inherits from. *(verb)* To create a new class that inherits from an existing class. The keyword to do this in ScriptX is `class`.

superclass – A class that has one or more classes inheriting variables and methods from it. `RootObject` is a superclass of `Point`. A superclass is generally less specialized than classes that inherit from it.

title – A complete, stand-alone, interactive multimedia ScriptX application or program. Examples include modular compositions, virtual spaces, conversational interactions, constructive experiences, and multitrack sequencing.

tool – A complete, stand-alone application or program used to develop titles. Examples include browsers, debuggers, and compilers. Some tools, such as the borwser and debugger, are written in ScriptX.

Index

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, Tyler, Rufus, Kiri, Frisky and Iggy

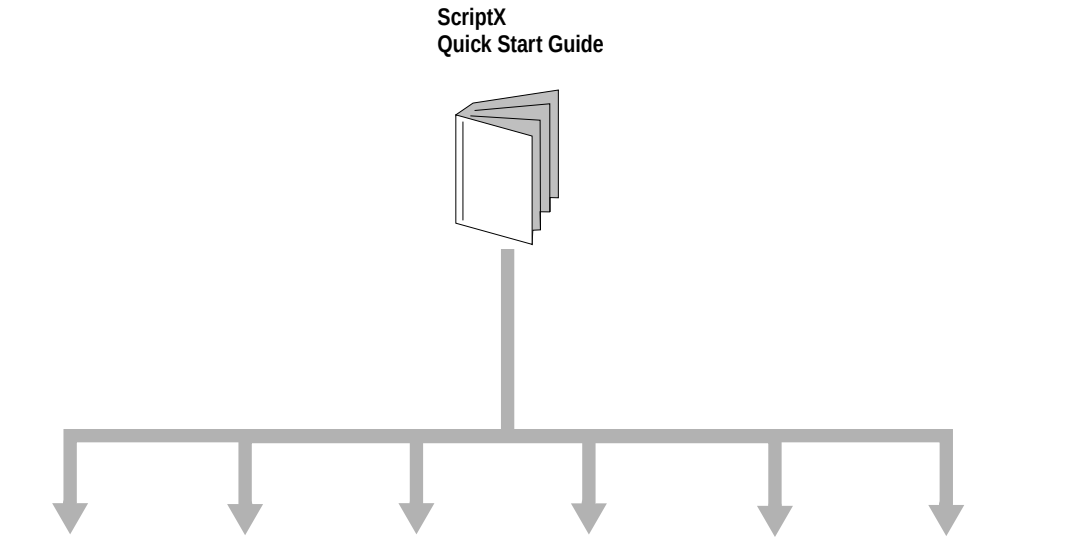
THIS DOCUMENT

Writing • Douglas Kramer, Howard Metzenberg, Jocelyn Becker, Alta Estad, Maydene Fisher

Book production • Sandra Ware, Jacki Dudley, Diana Harwood, Beth Delson

This book was created electronically using Adobe FrameMaker on Macintosh Quadra computers.

Documentation Roadmap



ScriptX Language Guide

Expressions, conditionals, class definitions, etc.

ScriptX Components Guide

User's guide for ScriptX

ScriptX Class Reference

Classes from A-Z, Functions, Variables, Constants

ScriptX Quick Reference

Brief overview of language, components and classes

ScriptX Tools Guide

Listener, browser, debugger, profiler, importers, etc.

Online ScriptX University

Lessons in ScriptX basics

Acrobat versions are available for most manuals