ScriptX Components Guide

December 1995

Kaleida Labs

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

Printed in the USA.

Kaleida Labs, Inc.

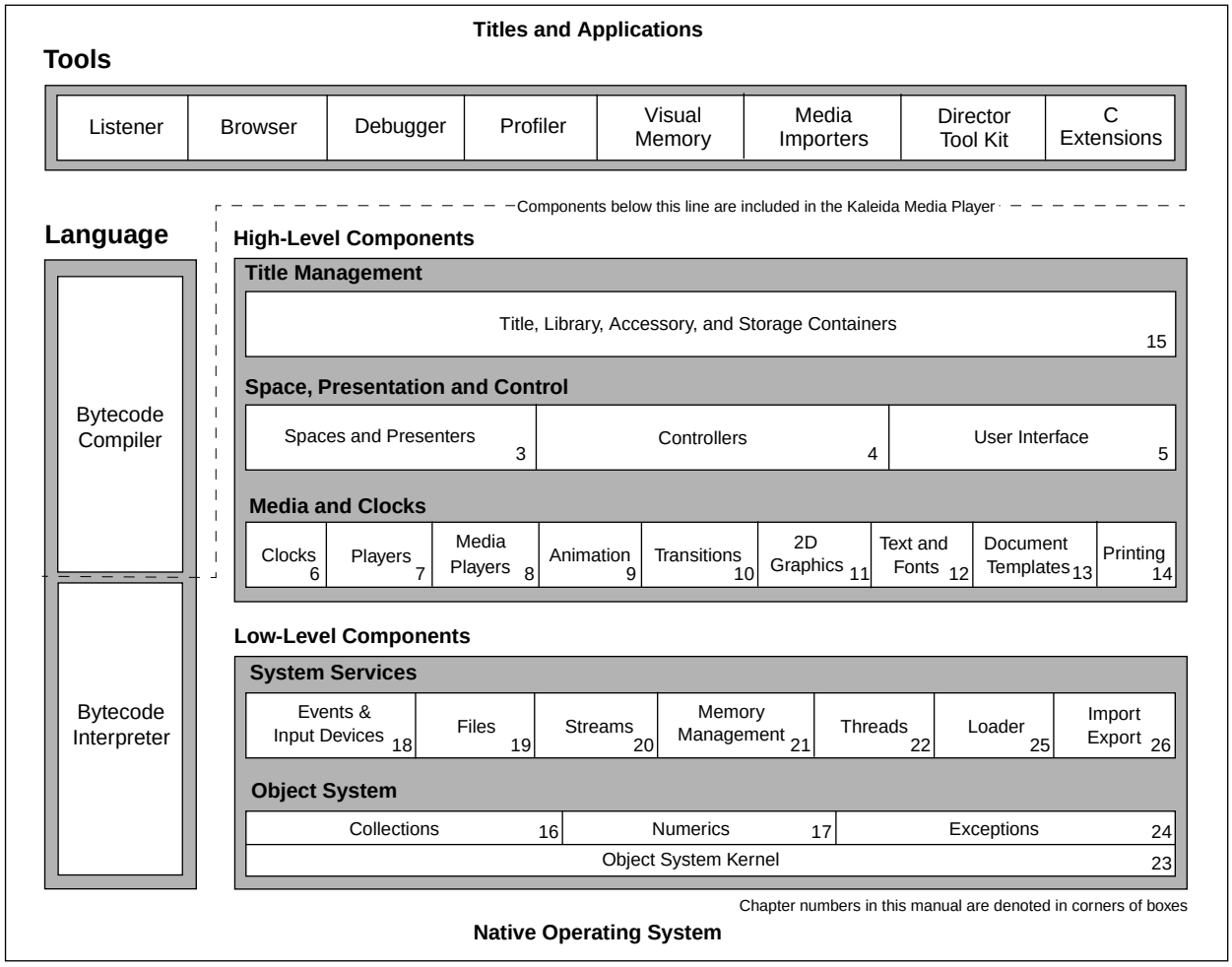
c/o Apple Computer

1 Infinite Loop

Cupertino, CA 95014

Quick Contents

Graphic Overview



This diagram depicts not only the components that make up ScriptX, but also the organization of this book. Each box in this diagram represents a separate component, which is documented as a separate chapter. This diagram arranges the ScriptX components in the following groupings:

- **Tools** – for aiding in the development of ScriptX titles and applications. Some tools are written in C and others in ScriptX.
- **Language** – for compiling ScriptX scripts into bytecode, which is then interpreted by the Kaleida Media Player. This component is documented in the *ScriptX Language Guide*.
- **Title Management** – for creating and managing titles.
- **Space, Presentation and Control** – for defining underlying models, controlling objects in them, and displaying them for viewing.
- **Media and Clocks** – for playing animation, transitions, audio, video, and movies, displaying text and graphics, and printing.
- **System Services** – for supporting services common to many operating systems.
- **Object System** – for fundamental collections, numerics and exceptions, as well as the foundation for the object-oriented and dynamic nature of ScriptX, including multiple inheritance, first class object system, dynamic binding and incremental compilation.

Notice the Object System sits on the native operating system, such as Macintosh, Power Macintosh, Windows or OS/2.

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 to run on the Kaleida Media Player. This series includes the following documents:

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

Audience for this Book

This book is intended for ScriptX programmers, both title developers and tool developers, who need to understand the fundamental programming concepts and features provided by the ScriptX architecture and core classes.

Summary of Contents

The *ScriptX Components Guide* provides a conceptual overview of the ScriptX architecture, along with detailed discussions of the components implementing that architecture. The architectural discussion describes the high-level abstractions in ScriptX, including motivating factors behind the design of ScriptX, the key features of the architecture, and examples of the types of applications ScriptX is designed to implement. The component descriptions provide details to help programmers navigate the architecture, including discussions of concepts embodied by classes in each component, and annotated code examples to demonstrate common usage of the component.

After the first two introductory chapters, this book is divided into the following parts:

Part 1, "High-level Components" describes the motivations behind the design of ScriptX, its primary differentiating factors from other authoring and programming tools, and examples of the unique capabilities ScriptX provides for title and tool development.

- **"Title Management,"** discusses features of ScriptX for managing a title, including startup, shutdown, interacting with menus, saving and opening files, and quitting the ScriptX runtime environment.
- **" Presentation, and Control,"** discusses the central features of ScriptX for presenting multimedia models to the user.
- **"Media Sources,"** describes the various types of media available through ScriptX, including players for sound and video, the 2D graphics imaging model, and text handling and font management.

Part 2, "Low-level Components, "System Services," describes operating system interface features provided by ScriptX for timing, file and data access, title storage, and managing memory.

- **"Object System,"** describes classes implementing the ScriptX object model and primitive object types.
- **"Tools,"** describe the facilities provided by ScriptX for importing and exporting media in various formats.

Note – The Property Manager is no longer supported in ScriptX. Instead, the `systemQuery` function provides access to information about features and capabilities of the underlying hardware platform. Please see Chapter 2, "Global Functions," in the *ScriptX Class Reference* for more information.

Manual Conventions

This manual is set primarily in Palatino and Avant Garde, except code samples, method names, and other code-like elements are in `Courier.`

Note – Notes to the user look like this.

C H A P T E R

ScriptX Features

1

The ScriptX Platform is a rich, dynamic, extensible, cross-platform, multi-threaded, object-oriented environment that enables you to develop multimedia titles, tools, and other interactive applications for desktop computers. This platform contains both a dynamic language and a built-in library of core classes which are intimately connected.

The Kaleida Platform currently runs on Microsoft Windows, Macintosh 68K, Power Macintosh, and OS/2. Later, support will be added for other widely-used hardware and operating system combinations, as well as for dedicated multimedia devices.

The core classes library defines a wide range of classes designed for title management, modeling, presenting, controlling, data management, text, graphics, animation, video, audio, timing, user-interaction, and system services. ScriptX implements all of these features in software, using hardware assist when available.

ScriptX Executables

Kaleida ships both a development environment, which enables the creation of ScriptX multimedia applications, and a runtime executable for their delivery:

- The ScriptX Development Environment consists of the ScriptX Language, the bytecode compiler, for compiling scripts, the bytecode interpreter, and the core classes library. The compiler compiles ASCII scripts and media into cross-platform bytecode methods and data which can be stored to a file that is binary compatible across all platforms. This environment also includes programming tools for developing and debugging programs at the language level.
- The ScriptX Player (KMP) is the consumer runtime executable for running titles and applications developed with ScriptX. It contains the bytecode interpreter and core classes library.

This book focuses on the components that make up the ScriptX class library and define the ScriptX development environment.

This chapter examines the objectives that led to the development of ScriptX and the categories of user interaction that drove many of the architectural requirements. It then provides a brief overview of the major components of the ScriptX development environment.

Chapter 2 looks at some hypothetical examples to show how the components interact from a programmer's perspective. The remaining chapters look at each system component in greater detail. The *ScriptX Class Reference* is the companion book, which describes the public application programming interface (API) for all the classes in the core classes library.

Titles, Tools and Applications

With ScriptX you can develop multimedia titles, tools, and applications. These terms have somewhat well-defined meanings. For example, an interactive book is considered a title, an object inspector is a tool, and a text editor is an application. However, rather than repeat these terms, throughout this book we loosely use the term *title* to include tools and applications.

To look at what a title is, it can help to describe the primary model underlying all ScriptX titles:

> A title is composed of objects interacting in, or associated with, multiple, concurrent spaces in which users can participate.

The three important parts of this model are the objects, spaces and users. While spaces are simply objects themselves, they provide a location or place for objects to interact with each other and with the user. A window is the most obvious example of a space.

The user might just be an arrow that responds to mouse movements, or might be represented in the title by a surrogate animated or video graphic image.

This model is powerful enough to enable any multimedia presentation to be constructed and played. Developers can invent new metaphors based on this model.

Design Objectives of ScriptX

The design and development of ScriptX has been driven by the following objectives:

- To provide broad, creative range for developers, and thereby a wide variety of experiences for users
- To provide a platform for developing and delivering new forms of information and entertainment media in a compelling, easy-to-use, and reliable format
- To provide a general-purpose, metaphor-independent scripting language, built-in media support, and a flexible customization paradigm so that developers can express the design of their products fully and naturally
- To create a software abstraction of a multimedia engine that enables the playback of content on a variety of hardware and software platforms
- To encourage the adoption of ScriptX as a standard across existing and emerging markets

To understand what features would meet these objectives, the architects of ScriptX studied a broad range of information models and user interactions. The following section describes the major categories of user experiences that they considered.

Categories of User Experience

Each of the following five categories represents an independent experience for the user, with its own information model and style of user interaction. These categories together capture a broad range, though not the complete range, of applications for which ScriptX can be used. We fully expect developers and authors to reach beyond these specified paradigms, to develop new and interesting experiences, such as real-time collaboration between users.

ScriptX enables the authoring of a wide range of experiences including, but not limited to, the following. A single title can contain any number of these experiences within it. These experiences are described in the next sections:

- Modular Composition – A collection of multimedia objects composed in different groupings, along with a means for users to reorganize those objects to make their own compositions.
- Virtual Space – A fixed, seamless, multi-dimensional space populated by objects through which the user navigates.
- Conversational Interaction – The user and computer carry on a conversation—both have initiative, information, and intentions.
- Constructive Experience – A collection of objects with behaviors that interact when the objects are juxtaposed or connected.
- Multitrack Sequencing – A linear stream of multiple, simultaneous tracks of video, audio, data, scripts and so forth, with which the user can interact.

Each of the features in the above titles is implemented in one or several components of ScriptX. The following is a brief listing of which components are responsible for features common to most of these titles:

- Dynamic binding through the Object System Kernel, and loading of extensions to ScriptX through the Loader component
- Database management, search, and query through the Text and Collections components
- User interaction through the User Interface, Events, and Controllers components
- Dynamic screen layout and composition through the Spaces and Presenters component, the Document Template component, and the Transitions component
- Audio, video and MIDI synchronization and delivery through the Players, Media Players, and Streams components
- Animation and simulation through the Animation component
- Synchronization of objects through the Event and Clock components
- Mouse and keyboard interaction through the Events component
- Simultaneous, independent execution of complex, real-time interactions through the Threads component
- Storage of objects to the underlying file system through the Title Management, Files, and Streams components

- Use of standard dialog boxes such as Open, Save and Print belonging to the underlying platform through the Title Management component

Modular Compositions

To describe a modular composition, see Figure 1-1, which represents a collection of beads to form necklaces. This modular composition is a collection of objects (bowl of beads), and several compositions (catalog page, timeline, and necklace). It includes tools to enable users to rearrange the compositions.

Each of the compositions is *modular*—which means that when users encounter a new composition, such as a book or movie, with different objects, they can set the objects aside to form new arrangements and compositions.

Modularity is the ability to organize objects and information into well-defined structures that can be put together in clean, understandable ways. A *modular composition* combines these independent structures to give users alternatives for interacting with objects. Within a composition, the structures require no direct knowledge of each other and can be independently modified or replaced to form new or different types of interactions.

Dynamic binding enhances the interactive nature of modular compositions in the playback environment. Dynamic binding lets users add completely new elements, authored elsewhere, into the run-time environment without building, compiling, linking, or quitting the title. This facility lets modular compositions accept new types of objects to organize in real time. Thus, not only can the compositional structure be modified, but its content can be as well.

Many types of applications qualify as modular compositions: product catalogs (interactive shopping); reference material (multimedia database); expanded books, periodicals and magazines; and buying and selling guides. While modular compositions can be self-contained titles, they can also be independent software components designed to perform a specific function across a variety of contexts. This type of modular composition is known as an *accessory*.

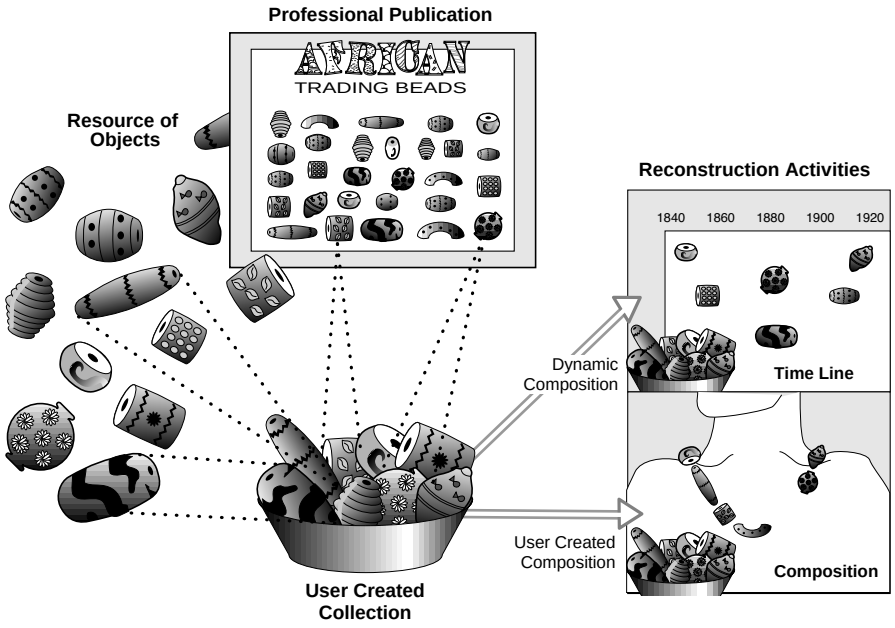


Figure 1-1: Modular compositions

Virtual Spaces

A *metaphor* is a collection of authoring abstractions that relates to some aspect of the real world and gives an author a particular point of view from which to construct a title. While authoring metaphors are useful, the ScriptX architects did not want to limit developers to a handful of authoring abstractions. Their goal was to let developers create new ways of organizing information that might be more appropriate for the activity at hand. One flexible alternative for organizing information is a virtual space.

A *virtual space* is a fixed, multi-dimensional area populated by objects, which users can navigate. A virtual space might have a real-world counterpart, such as an underwater canyon, or it might represent some imaginary area, such as a galaxy in a 3D adventure game. Since the structure of the space is fixed (unlike a modular composition), users can develop intuitions about how to get around. Figure 1-2 illustrates a virtual space. The user interface, images, and audio may provide a perceptually seamless portrayal of the space. This virtual space can be a metaphor for the physical world that can be enhanced or altered.

One way in which a virtual space can differ from modular compositions is that it can be time-based. The simulations and animations it contains are driven by a clock. The unfolding of events might respond to the user, but autonomous objects (fish) might operate with a mind of their own. Thus, events change dynamically over time around the user.

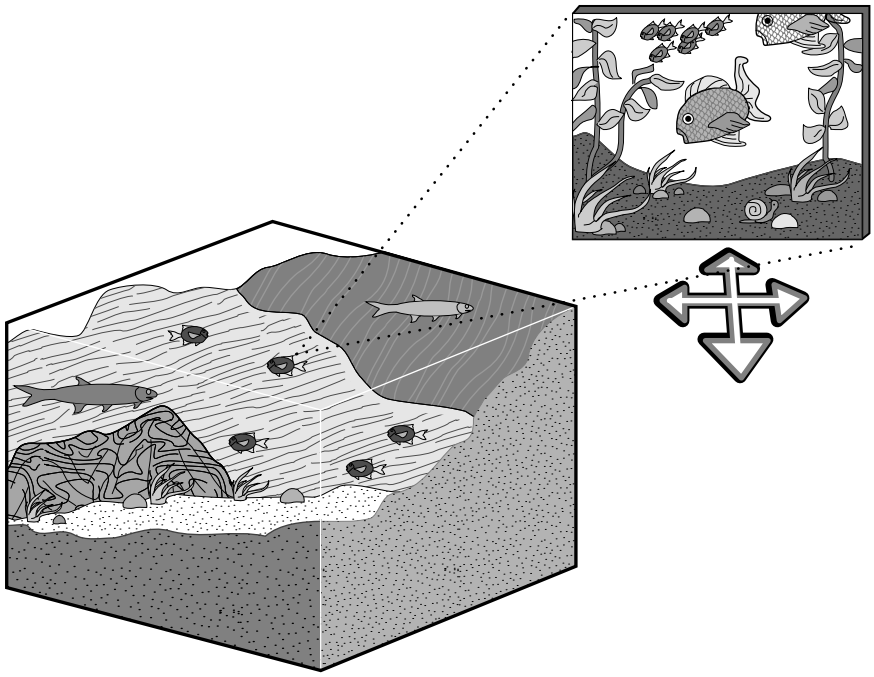


Figure 1-2: A virtual space

Conversational Interactions

Conversational interactions are experiences that engage both users and computers in a conversation—an exchange of information where both have initiative and intentions. As illustrated in Figure 1-3, computers respond to users' requests in real time and might direct users toward a specific goal or target. In this way, users direct the flow of information, and even the complexity of the interaction. Examples include interactive books, professional training, "how-to" guides, tutorials, and help systems.

Characteristics of a good conversational system are high-bandwidth interaction, the ability of the application to present multiple representations of information, the ability of the application to initiate, and the ability of the user to interrupt at any time.

One key feature of successful conversational interactions is their ability to adapt to a variety of different users. For example, a tutorial might have one presentation for novices and another for experts. Or a help system might provide information in several different languages, based on user requests. ScriptX can define a context that provides the most appropriate representations to users on demand.

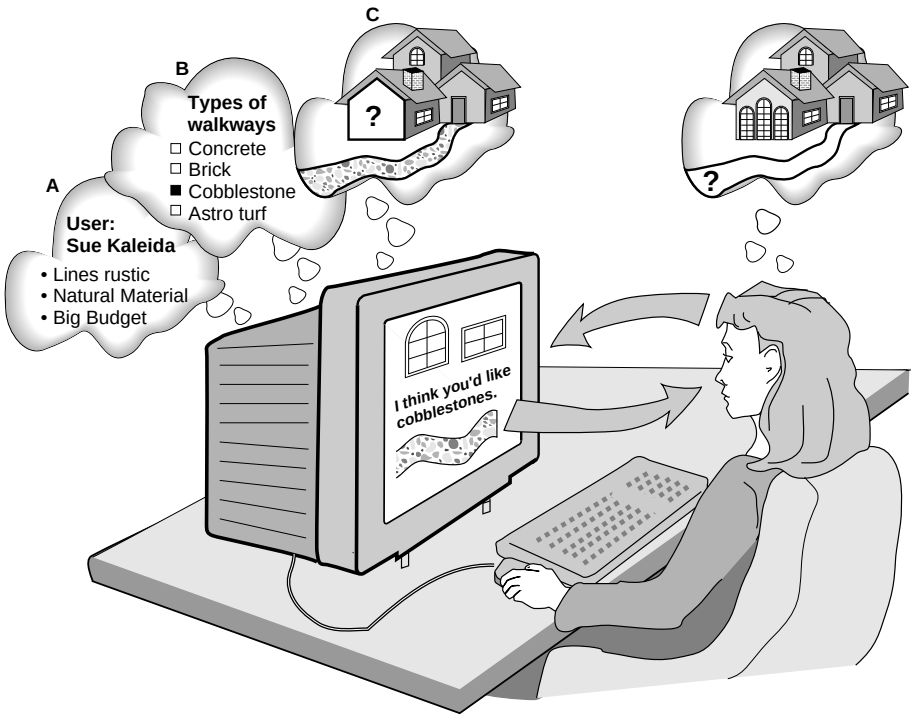


Figure 1-3: A conversational interaction

Constructive Experiences

In a constructive experience, users combine collections of multimedia objects to see how they interact when juxtaposed or connected. In Figure 1-4, the bird pecks at the food (A), which causes the door to open (B), causing the cat to enter the room (C), which causes the dog to awaken (D).

In rigorous systems, like physical simulations, the objects in the collections might have well-defined, repeatable behaviors. In other systems, the objects might have a wide range of unpredictable simple behaviors. That is, they might respond differently depending on the context.

Examples include retail product modeling and simulation, children's play environments (Rube Goldberg contraption), and interactive storytelling.

A feature that makes constructive experiences interesting is the ability for objects to change their actions over time. Clocks allows different objects to be synchronized, making this kind of experience run smoothly.

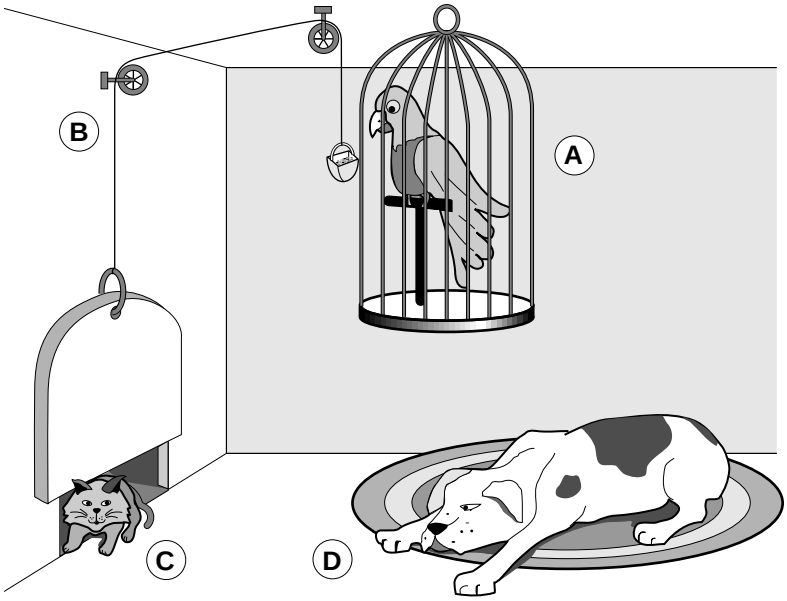


Figure 1-4: A constructive experience

Multitrack Sequencing

Multitrack sequencing is a linear stream of multiple simultaneous tracks, as shown in Figure 1-5, containing text, still images, animation, video, audio, data, commands, scripts, objects, transitions, and similar media elements. The tracks either play in a straightforward sequential fashion, or allow the user to interactively control time jumping and screen layout. For example, the user could switch between cameras at a sports event or at the production of a movie.

Personal movies, professional presentations, and video editing fall into this category.

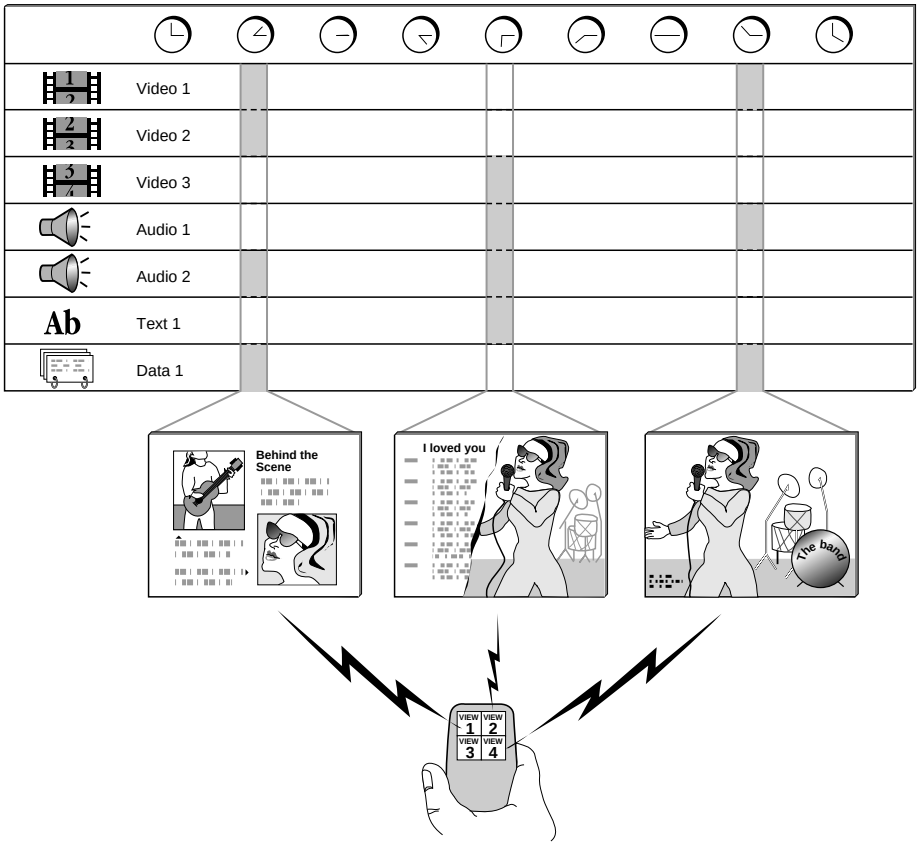


Figure 1-5: Multitrack sequencing

Key ScriptX Features

To achieve wide acceptance from hardware manufacturers, tool developers, title developers, and consumers, ScriptX must be general enough to accommodate all of the experiences described in the preceding sections and to adapt to new experiences that emerge in the future. It must define standard ways to present images, generate sounds, access files, use various media formats, interact with the windowing environment, and so on. It must also provide a core of multimedia-specific services and tools. The following sections describe the components that deliver these features to the ScriptX development environment in more detail.

The ScriptX Object-Oriented Programming Model

The ScriptX development environment is based on a modular, dynamically extensible object-oriented programming model. Objects in ScriptX represent all the framework, from physical devices controlled by hardware to the individual components of a multimedia piece. Once an object is defined, it can be reused by any other client in the framework through well-specified interfaces.

A key feature of object-oriented programming is *inheritance*—the ability to define new classes of objects by refining existing ones. ScriptX supports multiple inheritance, allowing developers to fully modularize their programs and leverage their work across a wide range of uses. Inheritance makes programming with ScriptX objects easier by allowing developers to build upon the existing framework and tune the underlying technology to new situations.

For example, a generic player, defined in the `Player` class, implements common player features, such as standard methods for starting and stopping. Specialized types of players—audio players, video players, and others—are defined as subclasses of `Player`. These subclasses inherit the general player behavior and adapt that behavior to work with specific media types. For example, the generic function `play` will do the right thing when called on a video player and also do the right thing when called on an audio player. The name stays the same, but the implementation is changed appropriately for different subclasses. Thus, a developer can use `play` with any of the ScriptX core class players and know that it will do the appropriate thing. If there were a new media type, the developer could create a new subclass of `Player` and specialize the behavior of `play` to accommodate that new media type.

The entire ScriptX development environment is itself defined in terms of the object model. The components of this environment are implemented as groups of related classes that are easily extensible. All user-written code is assimilated dynamically into the ScriptX environment and receives the same treatment as predefined ScriptX objects.

Metaphor-Based Authoring Facilities

The ScriptX language and class library directly support two authoring paradigms based on well-known metaphors. Metaphor-based authoring facilities help developers construct particular styles of multimedia interactions.

The Document Templates metaphor is a full-fledged component of the ScriptX development environment consisting of built-in classes, spaces, presenters, and related mechanisms for creating document-centered applications. Titles constructed using this metaphor resemble traditional books, magazines, annual reports, catalogs, and other such paper documents. However, by using underlying ScriptX technology, they can support all types of embedded media in their layouts—text, images, animation, audio, and video.

The Director Translator Kit is a set of scripted classes that enable the conversion of Macromedia Director™ titles into ScriptX. The model that drives this metaphor is a linear stream of multiple simultaneous tracks (channels). Tracks can contain video, audio, data, commands, scripts, objects, transitions, and so forth. The tracks present information sequentially and can also respond in real time to user-selected sequencing.

Spaces, Presenters, Controllers, and the Compositor

One of the primary goals of the ScriptX architecture is to provide a rich set of building blocks that allows authors to express their designs in ways that closely correspond with users' experience of them.

To that end, the ScriptX development environment includes a suite of authoring abstractions, some of which are not tied to any specific authoring technology or metaphor, since high-level abstractions limit the tools available to an author. With these building blocks, authors can define new tools and metaphors and can even "mix metaphors" for a richer experience.

The most fundamental authoring abstraction in ScriptX is a *space*. A space defines and organizes the behavior of objects. In ScriptX, spaces provide the environments in which objects live and interact. Two- and three-dimensional geometric and physical simulation spaces, cards, stacks, cast sheets, system simulation environments, catalog databases, and maps are all examples of possible spaces.

Spaces might also define other properties, such as timelines, spatial relationships, and state monitoring capabilities. Authoring tools and metaphors can be characterized in terms of spaces and the associated objects they provide.

In addition to the conceptual framework for organizing objects, available through spaces, ScriptX provides two special types of abstract interfaces to objects:

- *Presenters* provide concrete ways of experiencing objects in a space, such as through screen graphics or sound effects. They translate the abstract definition of an object into a form users can see and hear.
- *Controllers* manipulate objects in spaces. Controller behavior depends upon the control protocols defined by a space or by an object itself.

It is possible for a single object to be simultaneously represented by a presenter and manipulated by a controller. Presenters and controllers can be combined into a single object or can be embodied in many objects.

Compositors are internal media managers that map the audio and graphics of a title onto presentation hardware and, in doing so, coordinate the output of many presenters. Whenever a script associates a display surface with a space, the system creates a compositor to render the presenters contained in those spaces onto the display surface. The compositor manages the spaces associated with the display according to well-specified protocols. The compositor defines the general presentation behavior, and the display surface specializes that behavior to match its particular functionality.

Time-based Media Classes

Key to a viable multimedia standard is the rich representation of time-based media. This representation must take into account the interaction among different media types and different time scales. The object-oriented nature of ScriptX allows it to encapsulate time and media intuitively and naturally as objects.

ScriptX time is represented by clock objects that synchronize to the hardware clocks of the underlying platform. Programmers can use clock instances directly for synchronizing real-time media within a title; more commonly, however, they will choose the clock behavior embodied through ScriptX's built-in media players, which are special types of clocks.

ScriptX players represent the playback of time-based media, such as digital video, sound, videodiscs, and VCRs. They have predefined protocols for play, stop, fast forward, rewind, and other playback operations.

In addition to media players, ScriptX defines a number of media data types, including text, vector and bitmap graphics, animation, audio, and video. With a wide range of native media objects to draw from, developers can create a rich media mix with a minimum of programming.

Text and Collection Searching

Multimedia content often needs fast access to the media resources it uses. ScriptX addresses this issue with protocols for locating objects in a title. The Text component provides search-and-retrieval operations for matching strings and for finding the nth word, sentence, or paragraph in any given text. The Collection component provides methods for searching and retrieving objects within any collection.

Title Management

The Title Management component provides the mechanism for loading a title from a CD, disk drive, server or other storage device, and for saving objects used by a ScriptX title across multiple user sessions. This component can store title data, configuration information, and program state in the underlying storage system so that the data remains available from one invocation of a title to the next.

Object System Kernel

A deep understanding of ScriptX requires an understanding of the Object System Kernel, which defines the fundamental behavior of classes and objects. ScriptX objects are defined in terms of *classes*. Each class is itself an object, an instance of another class called its *metaclass*. Together with the metaclasses, the three root classes `RootObject`, `Behavior`, and `RootClass` comprise the metaclass network.

In the core classes, the distinguished class `RootObject`, root superclass of all ScriptX classes and objects, defines behaviors that are shared by all objects. Any method that is defined by `RootObject` is inherited by all objects in the system. In `RootObject`, ScriptX defines protocols for operations such as initialization, instance variable access, comparison, copying, coercion between classes, and printing to a debugging stream.

The `Behavior` class defines common operations shared by all classes. In particular, it defines the method for creating new objects in ScriptX. Through a metaclass network, similar to Smalltalk, ScriptX implements class variables and methods. The distinguished class `RootClass` is the superclass of all metaclasses and defines the protocol for creating new classes.

Several additional "system" classes are considered to be part of the Object System Kernel. Among them are the function classes, `Delegate`, `ModuleClass`, and `NameClass`. These classes provide essential services that are used by all other ScriptX objects.

C H A P T E R

Approaching ScriptX

2

This chapter presents top-level views of how ScriptX system components interact within two typical user experiences—a modular composition and a virtual space. Readers should consider these examples not as strict templates, but as aids for navigating the ScriptX system when implementing these styles of titles.

More detailed information on the components that organize the ScriptX class library appears in the chapters that follow. The *ScriptX Class Reference* presents detailed information about the classes that implement these components.

Introduction

As you can tell from the thickness of this volume, the ScriptX class library is very comprehensive. It provides a rich assortment of features and functions to help authors and developers create complex titles and tools.

To orient yourself to the ScriptX class library, consider the conceptual diagram in Figure 2-1:

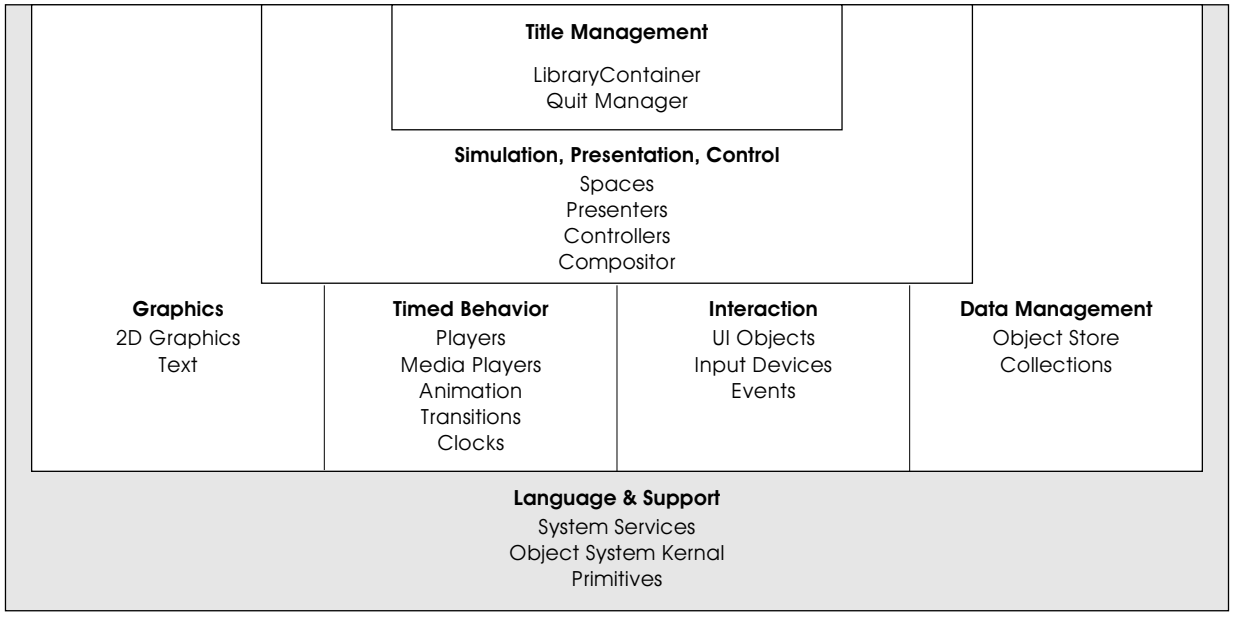


Figure 2-1: Interplay of components in ScriptX development framework

Figure 2-1 is a simplified overview of the development framework that groups key components in the architecture in two ways:

- By the roles they play in the design and implementation of ScriptX applications

- By their interrelatedness in performing those roles

From the bottom up, Figure 2-1 organizes the components of the ScriptX development framework into these natural groupings:

- Language facilities, the foundation of every application in ScriptX, and even of ScriptX itself
- Graphics facilities, the geometric shapes and text of applications
- Animation facilities, for adding time-based actions to applications
- Interaction facilities, for involving users in applications
- Data management facilities, for storing, retrieving, and organizing application data and for managing run-time memory usage
- Simulation and composition facilities, for defining a model simulation and presenting that model in ways that users can see and manipulate
- Display management facilities which define the visual presentation and interaction of applications
- Title management facilities, for creating and managing ScriptX titles

This way of thinking about ScriptX is task-oriented and emphasizes the dynamic nature of ScriptX. The examples that follow discuss the ScriptX development framework in terms of this organization.

Note, however, that this diagram is a simplification and does not provide a detailed view of every ScriptX component. The architecture is actually more complex than this diagram allows. For a more complete picture of the ScriptX system, see “Graphic Overview” on page v.

Modular Compositions

As introduced in Chapter 1, a modular composition is a collection of independent well-defined structures that provide alternatives for organizing and interacting with multimedia objects. Modular compositions encompass a wide range of user experiences, including conversational and constructive interactions. Interactive shopping catalogs, multimedia periodicals and reference books, tutorials and self-help guides are all flavors of modular compositions.

Figure 2-2 looks at a modular publication in its most general form. It groups the ScriptX components used to implement the modular composition into five primary categories:

- The language and support facilities provide the overall foundation of ScriptX—its object-oriented programming model. Developers can define intelligent, active ScriptX objects with context-sensitive behaviors that can be dynamically assimilated into a run-time environment. The object model pervades every component of ScriptX.
- The data management facilities are those ScriptX components that provide ways of organizing, partitioning, and storing ScriptX objects so that they can be easily abstracted and manipulated. The Collections component and the

storage capabilities of the Title Management component work together to provide a full range of persistent data structures that can be searched, iterated over, and operated on as a unit. The Memory Management component defines mechanisms for pulling saved objects from a file into memory and freeing them when done.

- The search-and-query facilities, available through the Collection and Text components, provide the interfaces for locating specific items in the multimedia database.
- The display management facilities represent the metaphor-based classes and scripts for presenting multimedia information in a unified form that users can easily navigate. The ScriptX Document Templates component offers a familiar metaphor for interacting with multimedia. The Spaces, Presenters, and Controller components provide the building blocks for developers to define new metaphors more natural to their title's needs.
- The interaction facilities provide the means for users to directly manipulate objects within a presentation. Facilities from the User Interface component provide control mechanisms that are sensitive to user input events, such as mouse moves and key presses. Developers use the Presenters, Controllers, and 2D Graphics components to add their own look-and-feel to these widgets. The Input Devices and Events components encapsulate the underlying hardware to communicate user-level activity throughout ScriptX.

To pull the pieces of the modular publication together into an integrated whole, developers must provide a navigation path for users to experience the composition. Figure 2-2 illustrates the dynamic structure of a possible modular publication. The double-stemmed arrow indicates a possible navigation path. The "data flow" paths indicate the movement of objects across facilities using built-in and specialized ScriptX operations. The "process flow" indicates the exchange of program control that occurs during the data query and retrieval protocol.

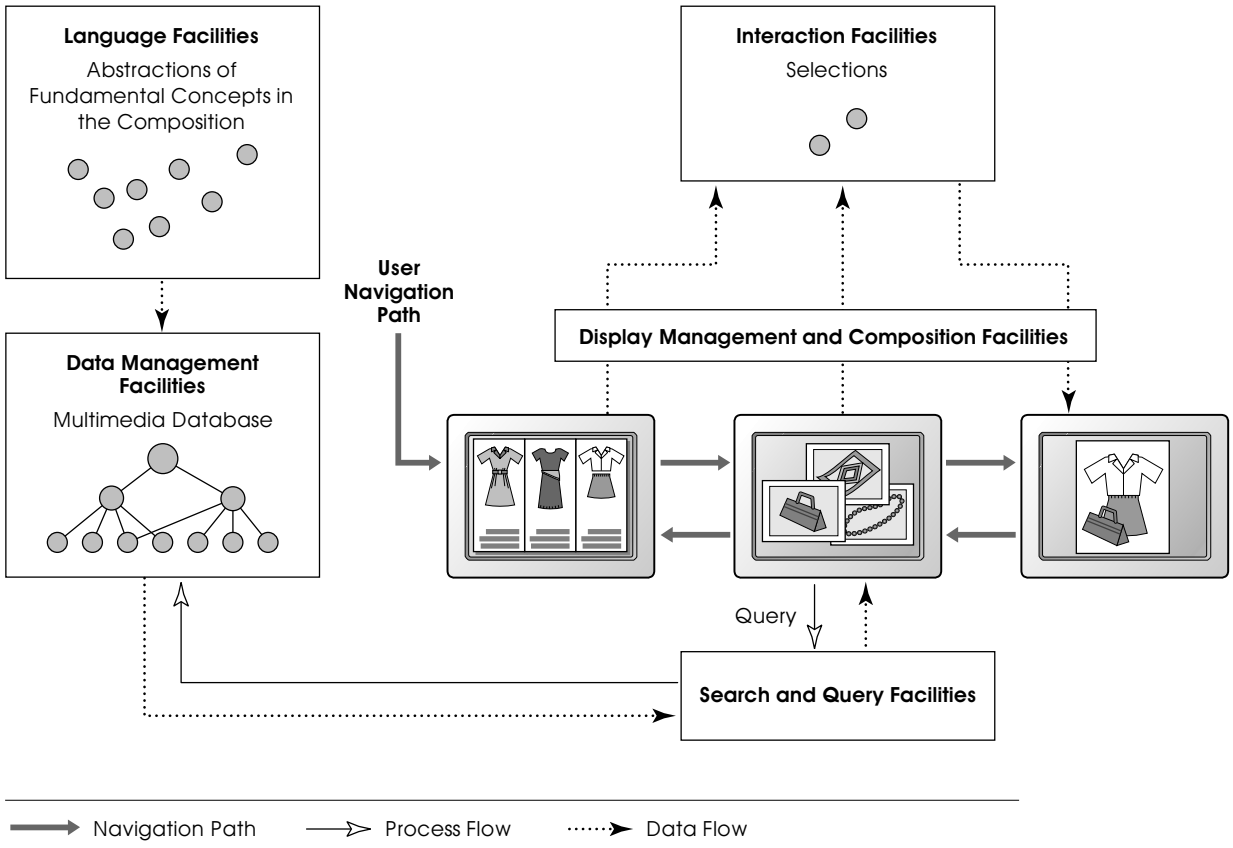


Figure 2-2: A generalization of a modular composition

To understand further how ScriptX components interact in a modular composition, consider the following hypothetical situation. Assume you were implementing an interactive shopping title that would allow users to browse through mail-order catalogs from leading clothes manufacturers, select accessories from other sources, and mix and match the pieces to create complete ensembles for purchase.

The objects of interest in this composition are all types of clothes that a user might expect to find in such a catalog—dresses, suits, slacks, shirts, blouses, shoes, ties, purses, and belts. These objects could be called the essential content objects of the composition. The first task of the developer is to define a rich set of behaviors and portrayals for these objects so that users' interactions with them are interesting.

Language Facilities of Modular Compositions

ScriptX provides an object definition framework that is broad enough for content objects to have many different portrayals and behaviors in a modular composition. As the user's experience changes in the composition, so can the action and appearance of the content objects.

A key feature of ScriptX is its ability to separate an object's structure from its presentation. Developers can model an object directly and present it in different forms. In ScriptX, objects called presenters are responsible for

translating the abstract definitions of content objects into forms users can see and hear. An object can define multiple portrayals —pictures, movies, line drawings—for different presentation contexts. And it can embody different behaviors that change with the presentation. The richer the object definition, the more alternatives for users to experience. For more information about presenters, see the chapter “Spaces and Presenters.”

In contrast, objects in existing authoring environments have a fixed universe of behaviors and portrayals. Authors have no way of creating different abstractions from what is prepackaged with the system. Authors can simulate more complex behavior, but they cannot directly model this behavior as in ScriptX.

Subclassing and inheritance provide other avenues for integrating complex behavior into a composition. Developers can modularize the behavior and appearance of ScriptX objects across several classes, and then specialize these behaviors and portrayals through subclassing or combine them through multiple inheritance.

Examining how users want to use content objects provides some clues about what information they might contain. For example, users might want to select clothes by retailer, such as Saks Fifth Avenue or Macy’s. Some might want to shop by designer—DKNY or Calvin Klein. Some might be more interested in looking at clothes by category, such as casual, work clothes, or evening wear. Some might want to shop for clothes of a particular complimentary color or price range, or match shoes and accessories to an existing outfit. Figure 2-3 show how these influences can be reflected in an object definition. Content objects can define many ways of selecting and organizing the information.

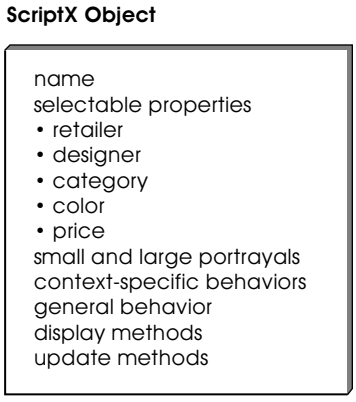


Figure 2-3: The richer the object definition, the more user alternatives

Data Management Facilities of Modular Compositions

All built-in ScriptX classes transparently incorporate persistence through the storage part of the ScriptX Title Management component. Persistence means that objects defined in ScriptX can remain available for a title’s use across multiple sessions of ScriptX. The storage mechanism provides for bringing objects into memory from disk when needed, and purged from memory when

done. Whenever a collection is stored, all the objects within it automatically become persistent, though optimal storage solutions are left to authors and developers. See the chapter “Title Management” for more information.

The Memory Management component provides facilities for measuring, monitoring, and calibrating memory usage. By using memory management operations in conjunction with the storage operations defined by the Title Management component, you can manage the run-time load of the title to optimize performance. See the chapter “Memory Management” for more information.

While persistence provides one ingredient of data access organization in ScriptX, the developer plays a large role in selecting the right data structures for different situations, and in managing the comings and goings of objects in memory. To define efficient means of storing and managing content objects, developers must consider their possible uses, including future uses. The ScriptX Collections component provides a broad range of built-in data structures: arrays, linked lists, hash tables, B-trees, fixed-size sequences, and byte strings. The section “Which Collection Should I Use?” in the chapter “Collections” discusses ways of selecting the most appropriate data structure.

All of the built-in collection classes in ScriptX have direct mechanisms for gaining access to the elements they contain—methods such as `chooseOne` and `chooseAll`. Developers can hand-craft their search operations by supplying matching functions for their particular needs.

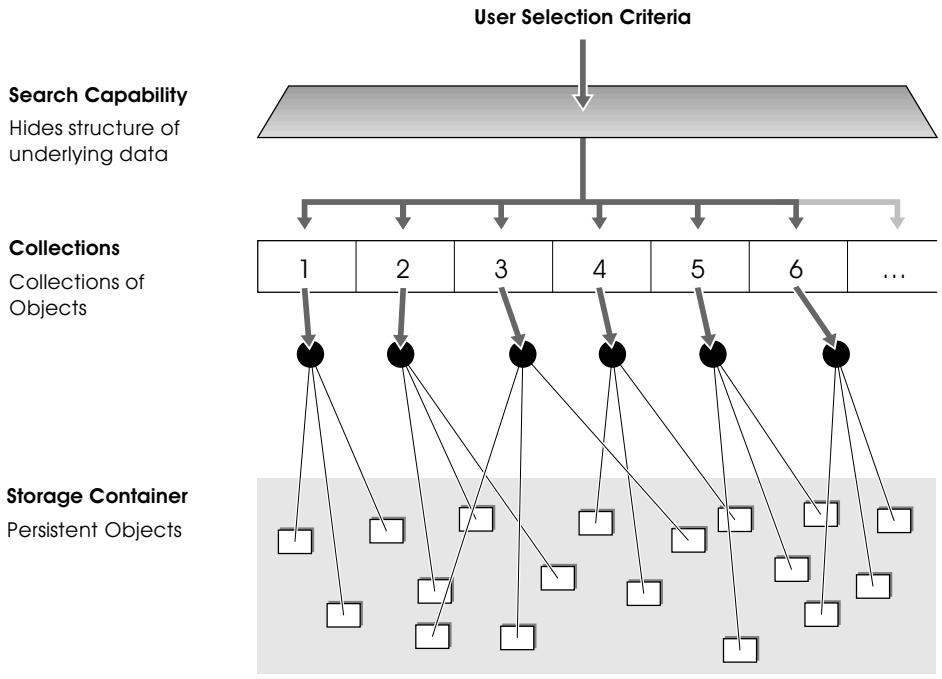


Figure 2-4: The interaction of components involved in object storage

To understand how these ScriptX components organize data access and storage, consider our interactive shopping example.

The ScriptX objects presented by this example define many selectable properties. Users can view clothing by retailer, by designer, by category, or by color—or through a combination of categories. These user choices translate into search criteria for locating the relevant objects in the database, by use of methods such as `chooseOne`.

Search-and-Query Facilities of Modular Compositions

The User Interface component in ScriptX allows a developer to display a selection of text choices to a user, using objects such as pop-up menus, pushbuttons, or scrolling lists. Developers can specialize all User Interface widgets for custom use appropriate to a title.

The text choices that a user makes from the top-level interfaces constitute the selection criteria for locating an object or group of objects in the content database. Each built-in ScriptX collection has its own methods for navigating, appropriating, and updating its data.

Text searching is divided into two categories:

- Parsing (searching for the nth word, sentence, or paragraph). Parsing is done by the global function `findNthContext`. A special feature of this function is that you can specify any character as a delimiter and search for the nth word, sentence, or paragraph bounded by that delimiter.
- Matching (searching for a match to a given string). The global function `searchIndex` searches for a match very efficiently because any text to be searched has been previously indexed, and the function searches the index rather than the text itself. An instance of the class `StringIndex` provides the index by automatically generating a signature index for the text supplied to its `string` instance variable. There is no limit to the size of the text to be indexed.

Figure 2-5 looks at the search-and-query facilities using the interactive shopping example:

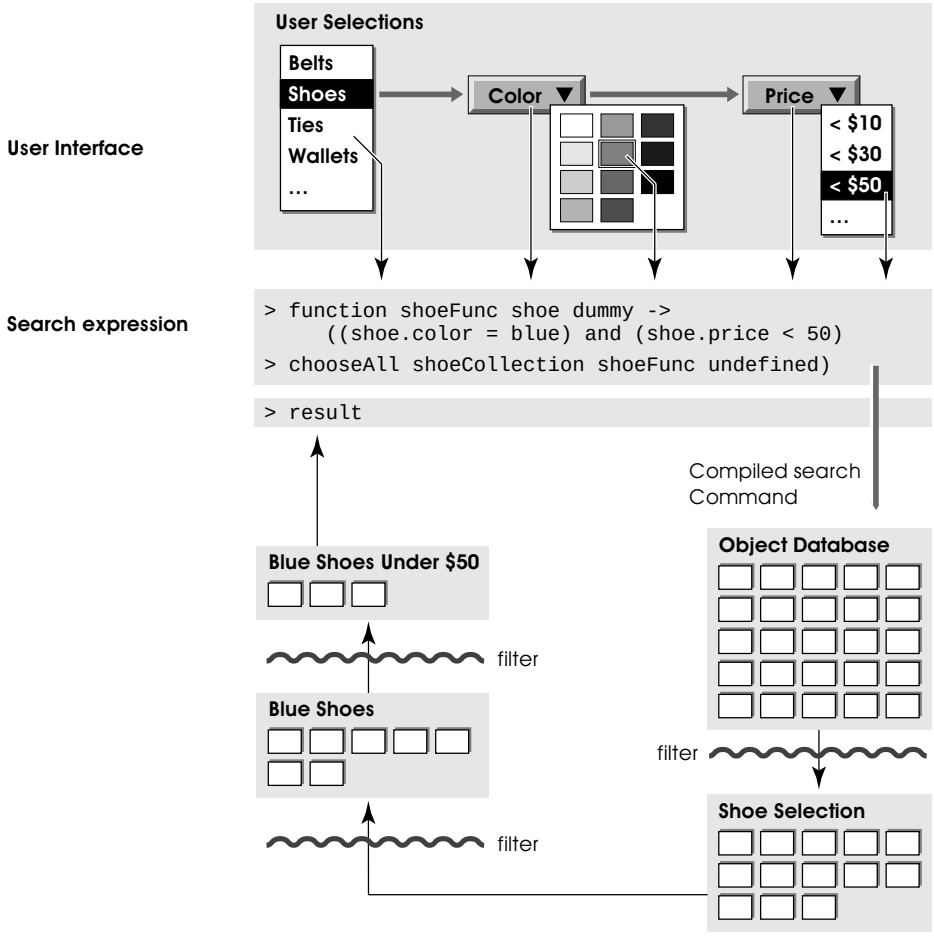


Figure 2-5: A typical ScriptX search-and-query operation

Suppose users can choose to search for shoes or accessories from a menu, which might cause a panel of buttons to appear. These buttons narrow the search by color and price, or a combination of the two if both buttons are pressed. The color button brings up a color palette, while the price button brings up a menu of price ranges.

A user on a budget trying to find shoes to match a particular outfit would first select the `shoes` category from the main menu, then push the `color` and `price` buttons, and finally click the right shade in the color palette and the appropriate price category. The expression for executing those choices might be something like this:

```
function shoeFunc shoe dummy ->
   ((shoe.color = blue) and (shoe.price < 50))

chooseAll shoeCollection shoeFunc undefined
```

This expression defines the context—shoe—and the criteria—blue color and moderate price—for the search.

Once the search for a match begins, the shoe objects are returned in the form of a selection, which is filtered on the basis of both color and price.

For detailed information on searching, see the “Text and Fonts” and “Collections” chapters.

Display and Composition Facilities of Modular Compositions

Having a rich assortment of well-defined objects is only interesting if there is some way for users to experience them. ScriptX provides display management facilities for presenting multimedia data to users in easy-to-navigate formats based on some common multimedia authoring metaphors. It also has composition facilities for creating new formats and metaphors that are more closely attuned to the experience an author wishes to project.

As Figure 2-6 shows the hypothetical interactive shopping example displays information to users in three separate formats:

- As pages from a catalog, which can be organized by retailer, designer, or clothes category (casual, work clothes, evening wear)
- As index cards that display shoes and accessories and can be organized by color and price
- As a virtual dressing room that shows how user selections work together as an outfit

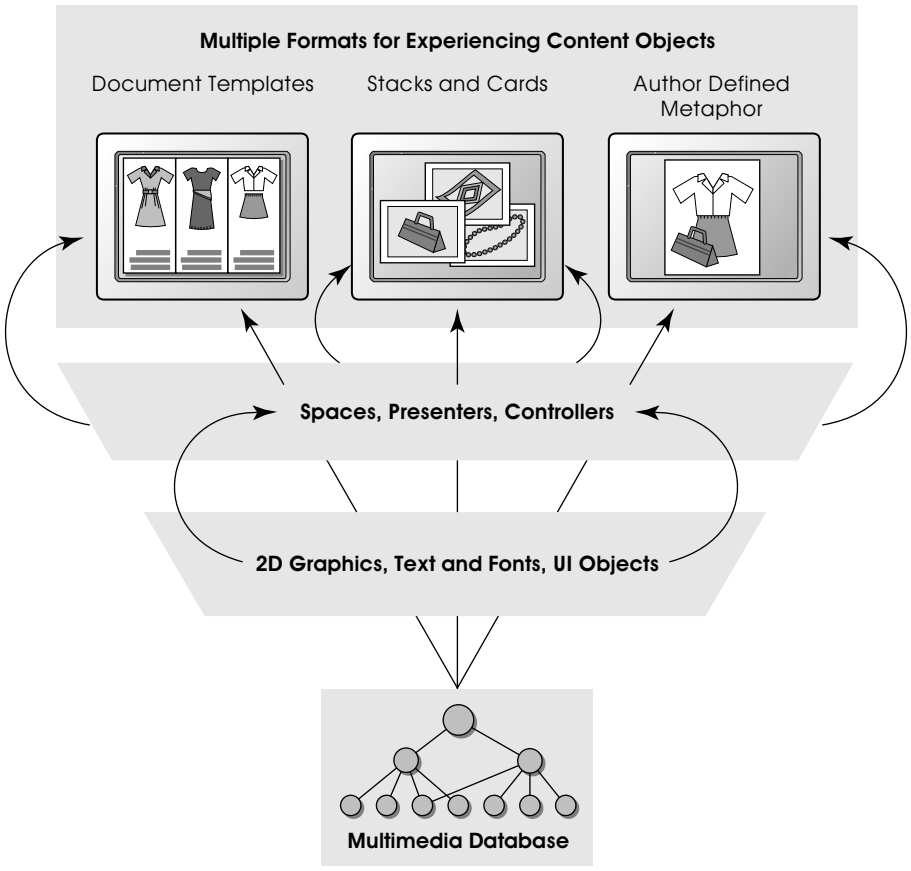


Figure 2-6: Interaction of display management and composition facilities

The Document Templates component is the basis for the first of these formats, the catalog pages. Document templates use a book metaphor to orient users to the interaction. There are four principal structuring mechanisms:

- The data, which has an intrinsic structure, such as text divided into chapters, paragraphs, and sentences
- A page for storing specific data, much like a single page in a paper document
- Page templates, page layers, and page elements, which collectively define how data is laid out and displayed
- Documents, which organize a sequence of pages

A document contains references to the raw data that it presents and provides built-in facilities for querying and searching that data.

Developers can specialize the behavior of document templates as necessary. For example, the interactive shopping title would define multiple page objects, each presenting the data in a unique graphical layout. To make the layout more exciting, the pages might embed animations, using facilities from the Players and Animations components, or incorporate graphical elements from the 2D Graphics component. It could include pop-up menus or other widgets from the

User Interface component, to aid in user selection and navigation. Yet the base functionality of document templates is sufficient to implement a catalog without specialization.

See the chapter “Document Templates” for more information about the Document Templates component.

The final interaction in the catalog example—the virtual dressing room—lets users construct their own experience of the catalog data. They can select a pair of pants from one catalog, a shirt from another, shoes from one index card, and a tie from another, and then bring them together in an uncluttered virtual dressing room to see how the pieces look as an outfit. To support this constructive experience requires a context for organizing the information, ways of manipulating the information, and mechanisms for displaying the information. The ScriptX Spaces, Presenters, and Controllers components define the composition facilities for implementing this experience. The section “Virtual Spaces” on page 33 provides more complete information about creating new authoring metaphors with ScriptX.

One important aspect of implementing the virtual dressing room would be scaling all the different clothes items to the proper relative size. For example, a skirt on a catalog page might appear in a 2x3” photo, while shoes on a card might appear in a 3x2” photo. Size in these contexts is relative to the page or card. When put together in the dressing room, however, these images would be disproportionate. In this case, the different objects should be sized relative to each other. Authors can build in this type of scaling as part of the class definition for the space. The space would pass this information on to the presenters, who would provide mechanisms for tailoring their graphic representations of the objects to correspond to the coordinated view of the outfit. Controllers assigned to the space could allow users to move the objects around and change the viewing perspective.

All metaphor-based interfaces—whether built-in or scripted—are implemented through the composition facilities of the Spaces, Presenters, and Controllers components. The metaphors simply impose a fixed interface structure above these elements, using graphics, text, and user-interface widgets provided by the 2D Graphics, Text and Fonts, and User Interface components.

User Interaction Facilities of Modular Compositions

The user interaction facilities include input devices, controllers, presenters, and events. They let users manipulate content objects directly, which in turn control the underlying model.

ScriptX offers many built-in ways to present selection choices to a user, such as pop-up menus, pushbuttons, or scrolling lists. Developers are expected to add their own look (using the Presenter and 2D Graphics components) and feel (using the Controller component). While default controllers exist for all User Interface objects, developers can create interesting new behaviors through subclassing and specialization. For information about the 2D Graphics and Presenters components, see the chapters “2D Graphics” and “Spaces and Presenters” respectively.

User Interface objects let users control which content objects they experience in a composition. The actual selection process is communicated to the system through the mouse and keyboard, defined by the Input Devices component, in conjunction with the ScriptX event system, defined by the Events component.

The default controllers for all User Interface objects hide the operation of input devices and events, so developers who use these built-in facilities from ScriptX need not concern themselves with them. The chapters “User Interface” and “Controllers” describe high-level features provided by ScriptX for user interaction, while the chapter “Events and Input Devices” covers the implementation details of these features.

Figure 2-7 shows how the interaction facilities apply to the shopping example:

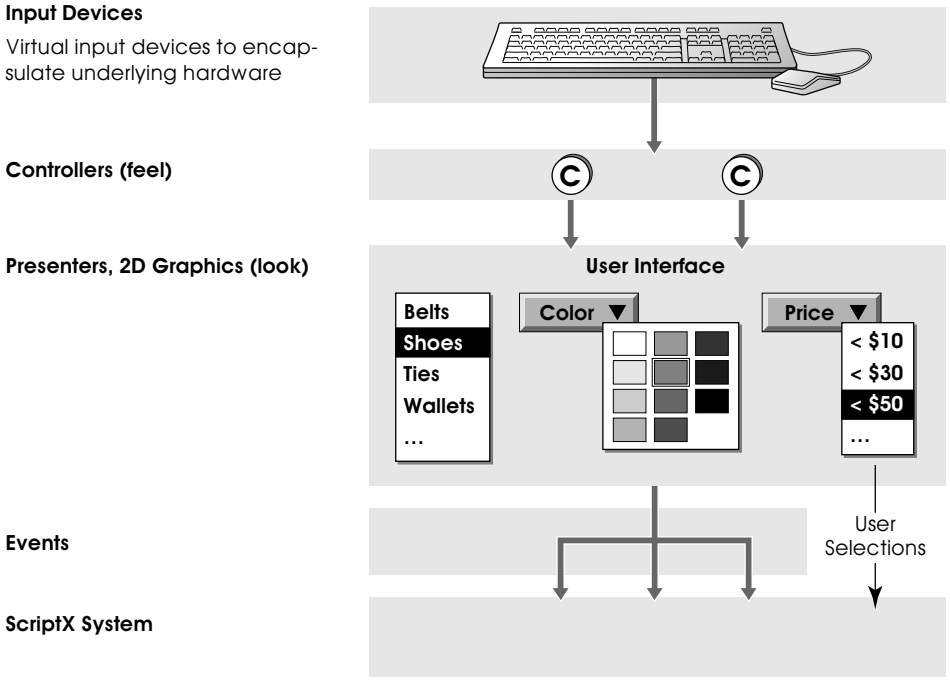


Figure 2-7: User interaction—direct and programmatic manipulation of data

Virtual Spaces

The previous part of this chapter described Modular Compositions. Another kind of title design is Virtual Spaces.

A key goal of ScriptX is to provide the foundation for developers to create new ways of organizing information to closely reflect the activity at hand. Unlike today's metaphor-based authoring frameworks, ScriptX provides simulation-based building blocks that let developers have real models behind their scenes, instead of being essentially a movie with a few twists. These building blocks underlie the concept of virtual spaces.

In its simplest form, a ScriptX space is an empty collection with an associated clock, and a list of controllers. As you add objects to the space, they can be controlled by the controllers, whose rate can be controlled by the clock. The controllers can move the objects, allow them to respond to user interaction, or whatever. The developer can define criteria which determine what kinds of objects are allowed into that space, and what happens to them when they enter and leave.

A virtual space is an imaginary environment created from a ScriptX space, with a well-defined geometry and physics, that users can view and interact with. A virtual space might have a real-world counterpart, such as a department store dressing room, or it might represent some chimerical area, such as a distant planet in a fantastic galaxy. In any case, the fixed geometry of the environment lets users intuit how to navigate the area on their own. Likewise, the physics of the virtual space dictates how its objects will respond to forces within the environment, both those defined by the space and those imposed by users.

Virtual spaces need not be sophisticated models of a world. In fact, their geometry and physics can be very crudely implemented. The department store dressing room, for example, can be a simple two-dimensional space whose objects do not directly interact but simply scale proportionately to each other. Regardless of the complexity of the space, it provides users with reference points for understanding the range of interactions available to them.

Metaphor-based authoring systems, such as the document templates described in the preceding modular composition example, are information driven. While they might be dynamically constructed as the title runs and contain embedded movies or other time-based animations, their overall composition is static. In a virtual space, however, the unfolding of events need not be directly under user control; that is, events can change over time. The simulation can have its own sense of time that is independent of any specific presentation on a particular platform.

Figure 2-8 shows a general form of a virtual space. The term model in the figure means the imaginary world represented by our virtual space. The model is separate from the presentation that users see, as Figure 2-8 indicates.

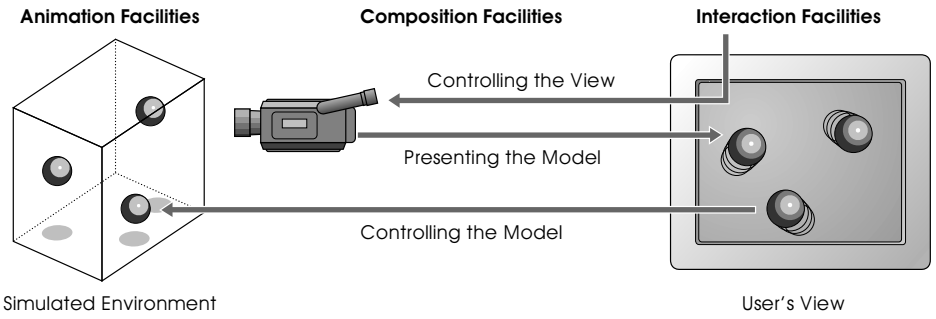


Figure 2-8: A generalization of a virtual space

A model is a group of objects within a virtual environment that interact with each other and with users according to some well-defined geometry and physics. Objects in the model are known as model objects.

The prime example of a virtual space consists of a three-dimensional simulation. Special model objects act as "cameras" that translate the simulation into a form the user can experience. Cameras are implemented through two different objects:

- Camera model objects that participate in the simulation
- Camera presenter objects that portray their views to users in the presentation

The cameras generate view projections to provide a three-dimensional feel to the experience, even though the users' view is a two-dimensional approximation of the model, constrained by the limitations of their output screen.

The model objects themselves obey certain "natural laws" of behavior defined by the virtual space. Controllers within the model cause objects to behave in the expected ways as a result of causal events that occur internally to the simulation, as well as those imposed externally by users.

Figure 2-8 groups the ScriptX facilities used to implement virtual spaces into three main categories.

- Simulation facilities define the structure and behavior of the model and the objects within it. The Spaces, Controllers, and Clock components provide the building blocks for defining the model and determining how objects within it interact over time.
- The model comes to life when presented to the user dynamically through the composition facilities—the interaction of the Spaces, Presenters, Controller, and Compositor components. Spaces not only organize the model but provide a basis for viewing it. Presenters translate the abstract simulation into an externally visible and audible form, and controllers let users manipulate both the simulation, and their view of it. The compositor ensures that all composition takes places smoothly on the output devices.

- The interaction facilities let users experience the model through the views presented to them. Using 2D Graphics component transformations, developers can specialize User Interface component classes to control the simulation through the presented views and, potentially, to modify the views or the action as events unfold.

To understand how a virtual space works, consider the following example. Your title lets earthbound users explore the surface of a distant moon. A robotic all-terrain vehicle, equipped with a movable camera eye, serves as their exploratory capsule. Users can travel across the moon's surface by driving the vehicle across the virtual plains, valleys, and mountains or through the imaginary seas. They can change their camera angle to get different views. Should they spot anything that requires closer inspection, they can zoom in on it. They can even operate a mechanical arm of the vehicle to collect a specimen.

The first task of the developer is to create an interesting simulated world that represents this new moon.

Modeling Facilities of Virtual Spaces

The modeling facilities are those ScriptX components that help you define an underlying model that can be viewed from different perspectives. A model can have a timeline, a coordinate system, and equations to determine how parameters interact. An example would be a simulation of a physical system, such as a heat exchange between a flame and a beaker of water.

In the most simplistic view, the model consists of these pieces:

- The unseen, underlying space that holds the model itself and has a clock and coordinate system
- The objects that populate the model
- Controllers that implement the physical behaviors of the objects in the model
- The cameras that live within the model, that present a view of the model to users

In our example, the model is the three-dimensional surface of the moon and the atmosphere surrounding it. The model objects are the rocks, hills, valleys, and seas of the moon, plus any extraterrestrial creatures that populate the landscape.

The Spaces component provides the facilities for defining the basic model structure. In most cases, you would mix in behavior from the Collections component to define the qualifications for validating which objects belong to a space. You can control the physical interaction of the space through Controller component facilities. Spaces inherit mechanisms from the Clock component to tie these interactions together into a continuous sequence of events. The chapters "Spaces and Presenters" and "Clocks" describe the interplay of these components.

The camera is our moonwalker, with its camera eye. Like the other model objects, it is constrained by the physics of the model—the lack of gravity, for example. Yet unlike the other model objects, it perceives the geometry of the

model in a special way, because it is associated with a special 2D presenter that lives in the presentation space. The camera presenter can translate what it sees back to the users viewing the model from their output screen. In addition, the control of our camera model object might be sensitive to direct user manipulation, whereas users can experience the other model objects only indirectly through the view provided by the camera eye.

Often more than one camera will exist in the model, to provide different views on the simulation. For example, one camera might offer a projective view, while another might show a top-down radar view. Projective cameras can supply wire frame views, shaded views, perspective views, or orthogonal views.

Cameras must understand how to transform the geometry of the simulation into the different presentation forms that users will see. The 2D Graphics component provides methods for computing the transformations required to render the objects in the appropriate size and position. Objects from the Presenter component can be used to compute and form the apparent views and adjust the sizes and positions accordingly. The chapter "2D Graphics" describes the 2D Graphics component, while the Presenter component is part of the "Spaces and Presenters" chapter.

A developer has a lot of latitude in defining a virtual space. The following questions are the most fundamental:

- Is the virtual space a realistic simulation that immerses the user in a different world, or a simplistic simulation intended to provide an intuitive way of navigating the model?
- Is the geometry Euclidean (corresponding to our idea of the world) or non-Euclidean (mazelike geometries, or independent geometries connected through some hyperspace)?
- Are the physical interactions predictable (again, corresponding to our common expectations) or magical?
- How does the user experience the space? Are the cameras passive viewers, or do they directly represent the user in the simulation?
- Is there an avatar, separate from a camera, that represents the user in the model?
- With respect to the camera, how many views are possible? What controls will the user have over the views, and what constraints limit the views in certain situations?

Composition Facilities of Virtual Spaces

The composition facilities let users see what's happening in the simulation, control the view, and, in some cases, influence the sequence of events that occur in the simulation.

Controllers work in concert with the camera presenters to navigate the model and manipulate model objects. From a presentation perspective, controllers map their interactions through the cameras' external views, which serve as the user's points of reference. In some cases, these controllers might also directly

influence the model objects in the simulation. While the cameras' views show what users can manipulate, the kinds of manipulations that are possible are defined by the model itself. Users might be able to change the location, color, or other properties of the objects that appear in the presentation via controllers that propagate those changes back to the model. The camera presenters then display the modified results.

Users can manipulate both the model objects and the cameras through controllers. Changes to the objects affect the behavior of the simulation, while changes to the camera affect the users' orientation to the space.

In our distant moon title, users can control what they see by manipulating the moonwalker itself or by panning, tilting, and zooming the camera. Both activities are transmitted to the space through controllers. In conjunction with the cameras, the controllers understand what objects should be visible in what locations, and they choose which objects to affect at the appropriate time. If the user zooms in for a closer look, the controller informs the camera to recompute the image, showing a magnified view.

Users can also collect specimens for later study. Controllers remove such objects from the simulation and propagate the ramifications through the model. For example, if a user "kicks a stone", a controller might cause the stone to catapult into space.

For a complete discussion of the components that drive these interactions, see the chapters "Spaces and Presenters" and "Controllers."

Our moon exploration title represents time as a continuous stream of events that begins at some point within the title and runs forward in time. When the title is played on hardware, however, our moon space becomes simply bit locations on a screen and time is metered out by built-in hardware clocks. The simulation is tied to the underlying hardware clocks through their root presentation space, which inherits from the Clock component. The Compositor component coordinates the playback of time-based components with the overall presentation. It guarantees that the presentation occurs smoothly across different output devices. The chapter "Clocks" describes the clock-compositor interaction in complete detail.

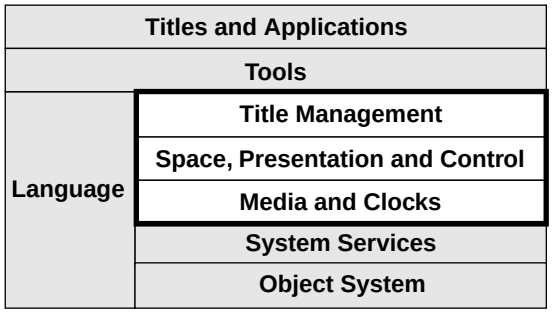
User Interaction Facilities of Virtual Spaces

A special set of user input controllers are sensitive to user input events, such as mouse moves and button presses, and act as mediators between the user and objects they control. In a virtual space, user control of model objects is constrained by the current views, as described in the preceding section. Using 2D coordinate space and transformations defined by the 2D Graphics component, developers can specialize the user input controllers to map interactions through the cameras' views. See the chapter "2D Graphics" for more information.

User interface controllers, and the whole notion of user interaction, is discussed in detail in the section "User Interaction Facilities of Modular Compositions" on page 31.

PART ONE

Higher-Level Components



C H A P T E R

3

Spaces and Presenters

The Spaces and Presenters component provides the object modeling and viewing facility for ScriptX. You can model any kind of system in a title and display it to the user using this component. This model could be a simple slide presentation, a complex interactive simulation, a virtual space, a media database retrieval system, or any other model a title needs. The model can be constructed apart from any presentation, perhaps in its own space, and displayed through one or more separate presentation spaces, or views.

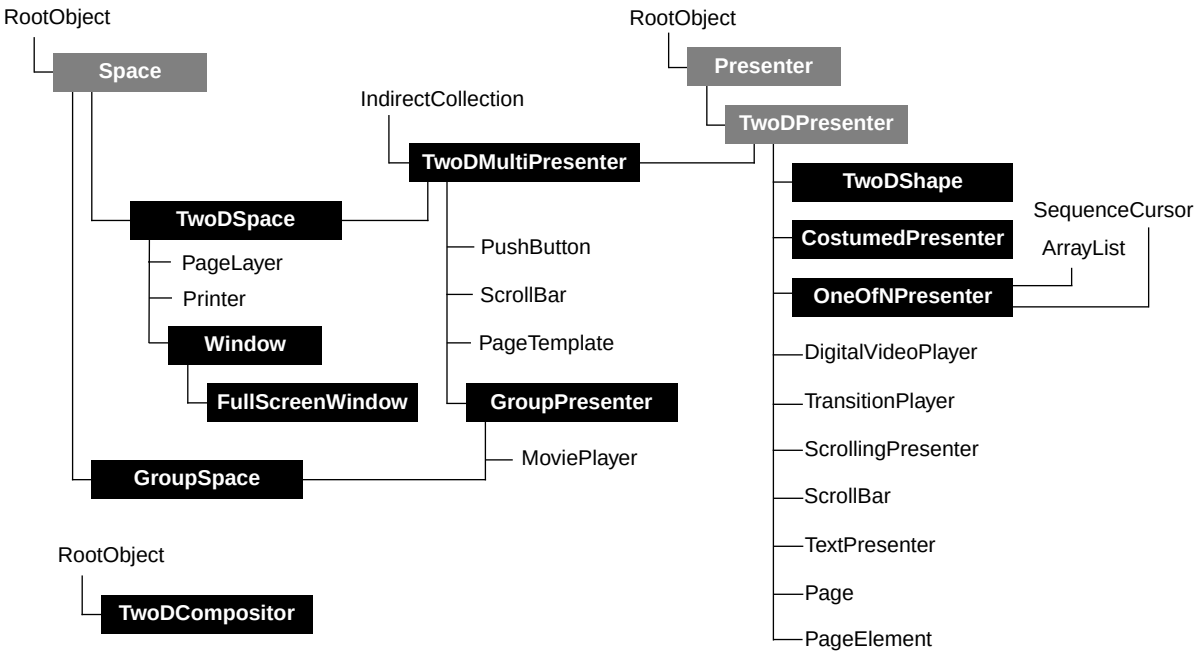
This chapter describes modeling, as implemented in the `Space` class, and presentation, as implemented by `Presenter` and its subclasses. It also describes the `TwoDCompositor` class, which manages the drawing of presenters to a display surface.

Spaces and presenters use features defined in a number of other ScriptX components. Collections provide for multiple objects in a model or presentation, and define the behavior used to add objects to and access objects in a space. Clocks and Players define the timing behavior available to objects in a space. The 2D Graphics component provides the fundamental coordinate system, imaging model and display surface for a 2D space.

Three other components allow the user to manipulate, change and monitor the model. The Controller component can apply a behavior to many objects—the "laws" or rules of that space. The User Interface component provides a standard set of user input controllers and presenters that allow users to directly manipulate objects in a 2D space and thereby interact with objects in either the model or 2D space. This component has built-in event-handling to accept mouse events. Similarly, the Text and Font component provides the presentation of text and event-handling to accept keyboard events.

Classes and Inheritance

The class inheritance hierarchy for the Spaces and Presenters component is shown in the following figure.



The following classes form the Spaces and Presenters component. In this list, indentation indicates inheritance.

`Space` – a container class for holding objects, with a clock for timing and controllers to manipulate the objects.

`Presenter` – the root abstract class for objects that display images and live in presentation spaces.

`TwoDPresenter` – the root abstract class for 2D graphic objects that can display themselves in a 2D space.

`TwoDShape` – the class of simple 2D geometric and graphic presenters.

`CostumedPresenter` – a 2D presenter that uses another presenter as its costume, or appearance.

`OneOfNPresenter` – an ordered list of 2D presenters, only one of which is shown at a time.

`TwoDMultiPresenter` – a 2D presentation container of a fixed size that can display and clip multiple presenters at once. It has no clock or controllers.

`GroupPresenter` – a 2D presentation container that groups multiple presenters together, without clipping.

`GroupSpace` – a 2D presentation space that groups other 2D presenters together so that they can be treated as a single object, without clipping.

`TwoDSpace` – a 2D presentation space of fixed size that can display and clip multiple presenters. It also has a clock and keeps a list of controllers operating on it.

`Window` – General purpose window that can handle palette, dialog, and notice windows as special cases.

`FullScreenWindow` – Full-screen non-modal window

`TwoDCompositor` – Manages the rendering of 2D presenters to the window's display surface

Conceptual Overview

An authoring metaphor is a model of user interaction in a multimedia presentation that is based upon familiar objects and behaviors from the everyday world. A bowl of necklace beads is familiar to users. They know from their real-world experience that they can pick beads from a bowl and arrange them. A multimedia title could use necklace beads as a metaphor for arranging objects in a collection, placing objects in sequential order.

Authoring metaphors are the basis on which a user interacts with a title. ScriptX supports multiple, concurrent authoring metaphors in a title. Some metaphors encompass a single gesture or activity, such as turning on a switch or moving a slider control. Others are global in scope, such as turning the pages of a book or flipping cards in a stack. A metaphor is useful to the extent that it feels familiar to a user. Chapter , "ScriptX Features," describes several authoring metaphors that are the basis for multimedia title design. ScriptX developers are free to use well-known metaphors, or to create their own.

Any authoring metaphor can be best understood by separating its constituent objects into three basic functions, or roles: model, presenter and controller. In the design phase of a title, it's useful to analyze what role objects play, to more clearly define them.

Model-Presenter-Controller System

ScriptX titles are based on interaction between three kinds of objects: models, presenters, and controllers. The `Space` family of classes provides the foundation for models. The `Presenter` family of classes provides the foundation for presentations, or views. The `Controller` family of classes provides the ability to manipulate objects in space and time.

Any particular object is a model object, a presenter object, a controller object, or a combination of the three.

Figure 3-1 shows an example of a model-presenter-controller system. The model space on the left contains model objects that are presented to the user by objects in the presentation space (on the right). The user manipulates user interface objects which can affect both the model and its presentation. The compositors present the objects to the user via the screen and speaker. The 2D graphic compositor is represented by the `TwoDCompositor` class; the audio compositor is internal only, and has no representation as a ScriptX class.

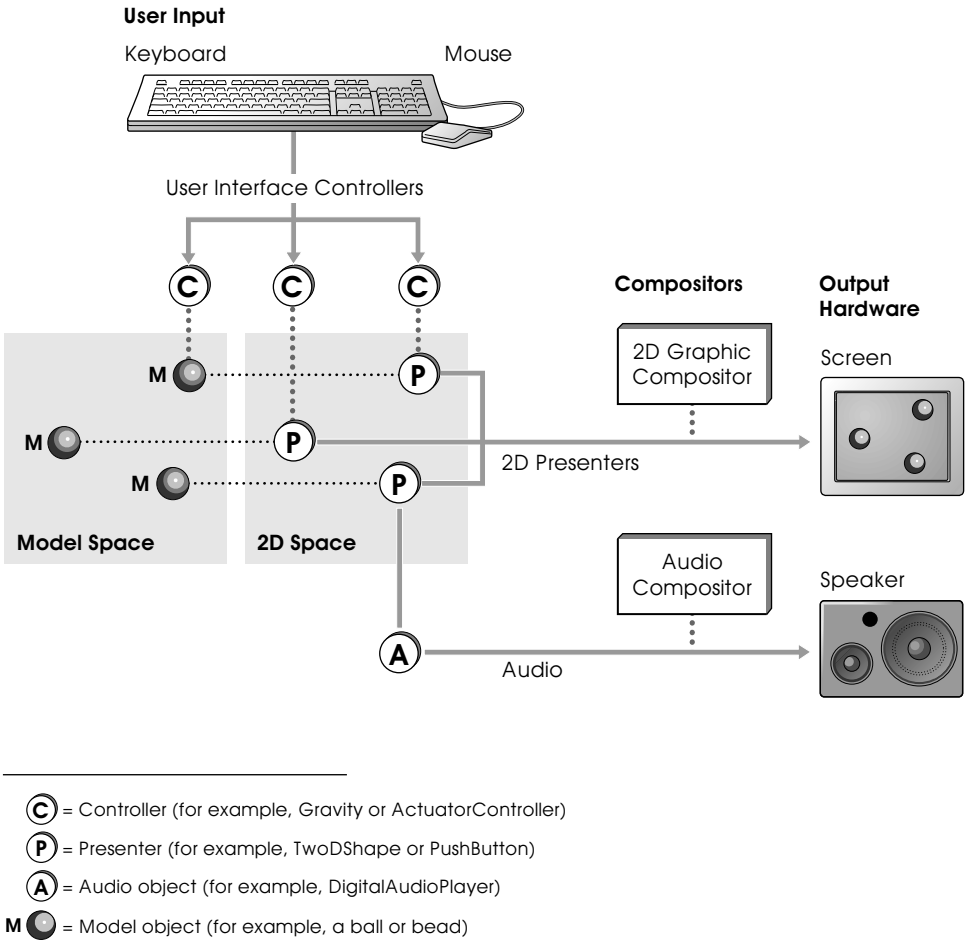


Figure 3-1: Example of a model-presenter-controller system

This overview describes models, presenters and controllers, assuming a separation exists between the model and its presentation. The separation of model object from presenter object is important because it allows for multiple presentations of an object, and it allows you to work with the presenters as separate model objects in their own window. For example, separate presenter objects allow you to move the view around on the screen without moving the model itself.

In simpler titles with no distinct, underlying model, the presentation is the model, with the objects in the model serving as both model objects and presenter objects.

Models

A model is a group of objects within a conceptual framework that interact with each other and with the user in some well-defined way. When a model is strictly separated from its presentation, you have the flexibility of creating multiple views on the same model.

By subclassing the `Space` class, you can create a class that provides a useful, though not necessary, container for a model. The `Space` class is deliberately abstract, open and flexible. A space can contain and organize the so-called model objects. It controls which objects are allowed into the model, performs actions on objects as they enter, and provides a clock and controllers to control the objects while they are there. Spaces are designed to allow their objects to be both time-driven and to respond to user events.

Models are more general than spaces. A model does not have to exist in a space. Time-based models generally reside in spaces, since a space conveniently provides a clock. Simulations that can map to physical coordinates can also be modeled in a space. However, a model could be a completely mathematical abstraction, even with its own clocks, and have no need to be contained in a space.

Presenters

Presenters are a second kind of object on which any title depends. Presenters provide the graphic views the user has of the model. A view of a model contains a set of presenters shown from a certain perspective. A user can view objects that exist in an underlying model space only when they are represented by presenters.

For example, in a model that illustrates the principles of thermodynamics, the image of a thermometer is a presenter object that can graphically represent the internal, calculated heat of a flame. In another view, a graph is a presenter that shows the rising temperature over time. Separating a model from its presentation allows creating such multiple views on the same underlying model.

The abstract class `Presenter` is the root class for all presenters. `TwoDPresenter`, its direct subclass, provides the structure for drawing to a two-dimensional surface, such as a display monitor or a printed page. The `DigitalAudioPlayer` class, which is outside of the `Presenter` family of classes, provides for the audio portion of a title.

Objects are made visible to a user in a presentation container; the container can be an instance of `TwoDMultiPresenter` or any of its subclasses, such as `TwoDSpace`, `Window`, `GroupSpace`, or `PageLayer`. User interface objects live in this container as well, available for the user to act on.

Whether there is an underlying model or not, the presentation container provides a framework for displaying the visible content of a title as images from various visual media. Each presenter has its own 2D graphical coordinate system with its origin relative to the presenter that contains it. Presenters also define an event handling structure that enables a user to interact with the title.

Controllers

Controllers can define standard behavior for a group of objects in three ways: they define the behavior of objects over time, they monitor objects, and they respond to user events. Controllers provide a uniform way of enforcing the same behavior on a group of objects. The `Controller` class is the abstract

representation of all controllers, and has a subclass `TwoDController` for operating on objects in 2D spaces. The three ways in which controllers manage other objects are elaborated here:

- Controllers can define time-driven behavior governing objects within a space. Time-based controllers, also called "ticklish" controllers, implement a method for `tickle`, which is called once with each tick of the space's clock. The `tickle` method implements time-based behavior as some periodic action that is performed at each tick of the clock. For example, `Interpolator`, a subclass of `TwoDController`, can cause objects to move within the space by moving them incremental distances, calculated at rapid intervals. See "The Ticklish Protocol" on page 106 in Chapter 4, "Controllers" for more information on time-based controllers.
- Controllers can also monitor a model. For example, a controller could monitor the distance between objects and performing some operation based on proximity for collision detection or magnetic repulsion. Controllers that monitor objects also implement a `tickle` method. `Bounce` is such a controller—it watches for an intersection between a projectile and the walls of a container, and when they intersect, it changes the projectile's direction.
- Controllers can respond to user input. Certain kinds of events, such as mouse clicks, mouse movements, and keyboard presses, originate with the user. Controllers make the objects being controlled, such as buttons and menus, respond to these events. For example, an `ActuatorController` object waits for mouse events. When the user presses the mouse over an actuator (button) it is controlling, it calls `press` on that actuator. See the "User Interface" chapter for more information on how user interface controllers operate on model objects in a space.

Controllers require a space in which to operate—they cannot operate on objects outside a space. Thus, a model that embodies some set of behavior or natural laws that all model objects must obey is best implemented in a space. Each time there is a change of state that drives activity in the space—the clock ticks, the user presses a mouse button—the controller acts on every object it controls within the space. See Chapter 4, "Controllers" for more details on how controllers operate on model objects within a space.

The next two sections—"How Spaces Work for Modeling" and "How Presenters Work"—describe modeling and presentation as separate, independent aspects of design.

How Spaces Work for Modeling

A *space* is an environment with a clock where objects live, interact, and can be controlled and presented to the user. Spaces are very common in titles—examples include simulation spaces, cards and stacks, timeline spaces, catalog databases, and maps.

Spaces are a fundamental component of ScriptX titles, and are used in both modeling (with `Space`) and presentation (with `TwoDSpace`). This section describes how spaces are used for modeling.

The Space Class

The `Space` class represents a container in which objects can interact with each other and indirectly with the user. The concept of space is part of what gives ScriptX its flexibility and extensibility as an authoring environment.

The `Space` class is an abstract class, so any utility is derived by creating a subclass of it. The `Space` class by itself is not a collection—it must be mixed in with a collection so it can hold multiple objects. Subclasses of `Space` rely on being mixed in with `IndirectCollection` (or one of its subclasses). The `IndirectCollection` class is quite flexible in that it can represent any collection class, through delegation. When you create an instance of the space, you can select the type of collection most appropriate for the model you are constructing, as the target collection. Throughout this chapter, the term "space" (all lowercase) means an instance of a subclass of the `Space` class, and so is assumed to be a collection.

You create a space by first subclassing the `Space` class, mixing in `IndirectCollection`, and defining its methods and instance variables. To create a new class called `ModelSpace`:

```
class ModelSpace (Space,IndirectCollection)
-- define methods and instance variables
end
```

To create an instance of this space, you would call `new` on it, optionally specifying `targetCollection` and `scale`, such as:

```
mySpace := new ModelSpace targetCollection:(new HashTable) scale:20
```

This statement creates both a space and a clock, as shown in Figure 3-2. The `targetCollection` determines what kind of collection `mySpace` is, and the `scale` determines the resolution of the space's clock. The clock's rate is initially 1. The `targetCollection` keyword is required for direct subclasses of `Space`.

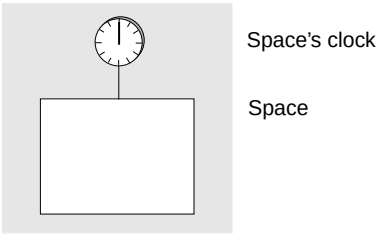


Figure 3-2: Every space that is created has a clock attached to it.

Do not specify `targetCollection` if the subclass inherits from `TwoDMultiPresenter` (or any of its subclasses, such as `TwoDSpace`, `GroupSpace`, `Window`, or `PageLayer`), unless you have some good reason to need a different collection data structure. By default, these multi-presenters use an array as their target collection. Performance could suffer if you set the `targetCollection` to something other than the default.

However, when creating an instance of any other subclass of `Space` that is strictly a model and not a presentation space (one that does *not* inherit from `TwoDMultiPresenter`), choose whatever `targetCollection` is most appropriate for the model you are creating. This collection does not need to be traversed for drawing or handling events.

The State of a Space

A space has a state represented by four characteristics: members, protocols, controllers, and a clock, as shown in Figure 3-3:

- members – A space is first and foremost a collection of objects, or members. Members are objects added to a space.
- protocols – Protocols are used to determine eligibility for membership in a space. These are kept in the space's `protocols` instance variable.
- controllers – Controllers define the standard behavior, or natural laws, governing some or all members of the space. These are kept in the space's `controllers` instance variable.
- clock – A space's clock drives the model. At each tick, the clock gives a slice of time to every controller that implements a method for `tickle`, allowing the controller to perform some periodic action on its target objects. The space's clock is kept in its `clock` instance variable.

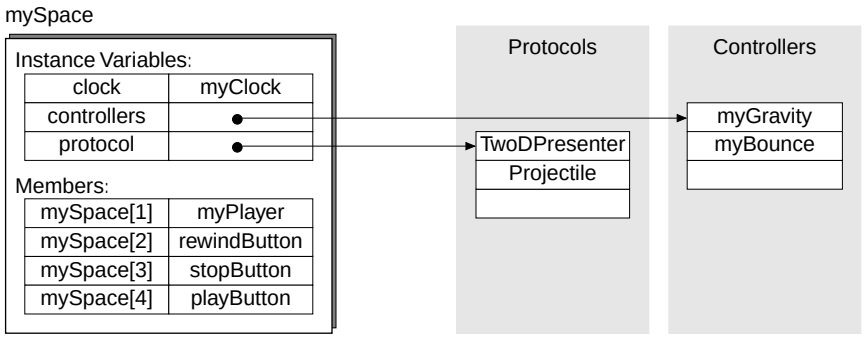


Figure 3-3: An instance of a subclass of Space

Protocols

A general description of protocols is included in the "Information Common to All Classes" chapter of the *ScriptX Class Reference*.

As implemented in ScriptX, every class represents a protocol. The `Space` class defines a `protocols` instance variable, which is a list of classes that you specify. This list forms the necessary protocols for objects to be added to the space, and you can add or remove classes from this list to raise or lower the admissions requirements to the space. This list allows the space to restrict its membership and allow in only objects created from certain classes. The space uses `isAKindOf` to test if the candidate has *all* protocol classes among its superclasses before it is admitted to the space. If the candidate does not match all protocols, it is rejected. Testing against this list is the current means of protocol-checking in ScriptX (future versions may develop other means).

For example, if the space were controlling a physical simulation, you might allow in only objects that were projectiles. You would do this by adding the `Projectile` class to the protocols list. If you further restrict the members to be 2D presenters, you would also add `TwoDPresenter` to the protocols list. In this case an object would have to be both a projectile and a 2D presenter in order to be added to the space. You can change these rules for membership at any time by adding or removing classes from the `protocols` list.

The `protocols` instance variable holds an instance of `Array`; therefore, to add or remove classes from the `protocols` instance variable, use the methods from `Array`. For example, the following code adds the `Projectile` class to the `protocols` instance variable:

```
append mySpace.protocols Projectile
```

The order of classes in the `protocols` list doesn't matter, since an object must have all classes as its superclasses to be added to the space.

Members of a Space

Add objects to a space just like you would to any collection—using one of the methods defined by the space's `targetCollection` for adding an item to the collection. These include `add`, `append`, `prepend`, `setOne`, and so forth.

```
mySpace := new TwoDSpace
myGroup := new GroupPresenter
prepend mySpace myGroup
```

`IndirectCollection` specializes these methods to call two additional methods on the space each time an object is added—`isAppropriateObject` and `objectAdded`—as shown in Figure 3-4. Using `prepend` as an example, this flowchart shows the four steps:

1. You call `prepend` on the space to add an object.
2. `IndirectCollection` specializes `prepend` and other collection methods that can add an object to the space to call `isAppropriateObject`, as implemented by the space, to check that the candidate object conforms to the protocols for the space. If `isAppropriateObject` returns `true`, then the procedure continues.
3. The `prepend` method as implemented by the target collection itself is called. This actually adds the object to the space.
4. The `objectAdded` method is called.

The `IndirectCollection` method `objectAdded` iterates over all controllers listed in the space's `controllers` instance variable, adding the object to only those controllers for which the value of `wholespace` is `true` and whose protocols match. You can specialize `objectAdded` to perform any action you want to occur every time an object is added to the space. For example, you might center a presenter that is being added in a space, or make sure that the presenter that is being added is visible, by calling `show` on it.

The `IndirectCollection` method `objectRemoved`, also shown in Figure 3-4, is called automatically whenever an object is removed from the space. This method deletes the object from all controllers. You can specialize `objectRemoved` to perform any action you want to occur every time an object is removed from the space.

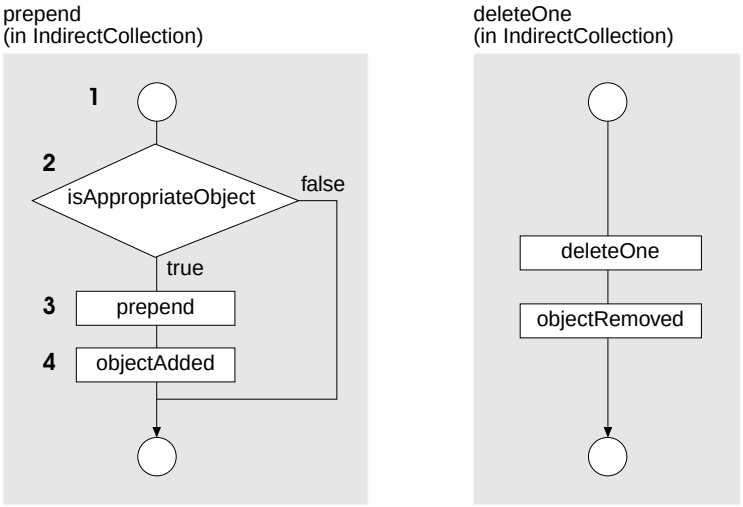


Figure 3-4: Methods prepend and deleteOne are specialized in IndirectCollection.

For more general information on the `IndirectCollection` class, see the discussion on page 469 of Chapter , “Collections.”

Models and Model Objects

A *model* is a group of objects within a conceptual framework. A *model object* is any object that is a member of that model. Model objects interact with each other and the user in some well-defined manner.

A model can live in a space, but it doesn’t have to. A space is simply a convenient and useful container for a model, with a clock and, potentially, controllers for manipulating the model objects over time. It lets you exclude certain objects, and possibly modify other objects as they are added, as described earlier. Each space can be customized to provide an appropriate environment for its objects, including a coordinate system. This environment can specify the protocols that an object must have to live in the space.

Most spaces embody some kind of coordinate system by which relationships between objects can be measured. This can be a 2D or 3D coordinate system, a timeline, or any other rule by which objects can be measured.

Clocks and Timing

Whenever a space is created, a clock is automatically created for it. This clock determines the following:

- At what time clock callbacks execute. A callback executes a function at a pre-determined time. See the "Clocks" chapter for details about setting up callbacks.
- How often the `Ticklish` controllers act on objects in that space (using the `tickle` method). The space schedules the controller callback at every tick of its clock. When the callback runs, it sequentially invokes the `tickle` method on every controller, to perform its periodic action.

In specialized subclasses of `Space`, the clock can have callbacks that invoke specialized behavior. For example, where the space is an instance of `TwoDSpace` at the top of a presentation hierarchy, the clock has a compositor callback that determines how frequently the presentation is refreshed.

If a title contains multiple spaces, it will contain multiple clocks—one for each space. In general, if a space is contained within another space, their clocks should be synchronized. For presentation spaces, ScriptX does this for you automatically. For example, if you add presentation space A to another presentation space B, ScriptX automatically slaves the clock for space B off the clock for space A, as shown in Figure 3-5. More specifically, when you add a 2D space to a window, the clock of the 2D space is slaved off the clock of the window. For a description of the mechanism of how clocks are automatically slaved, see "Synchronizing Clocks" on page 95.

However, the clocks for non-presentation spaces are not automatically slaved. This means that if you have a model space to which you add a subspace, you must set up the master/slave clock relationship yourself.

There are three main benefits to slaving subspace clocks to the top space's clock:

- When you slow down, speed up, pause and resume the top space, the other clocks will follow.
- You can move groups of objects into other spaces and they will still run.
- To minimize "temporal aliasing" in a model (described in the section "Synchronizing Clocks" on page 95).

As shown in Figure 3-5, there may be more than one way to connect clocks, depending on the needs of the title. The left-hand figure shows a top model space slaved off the top presentation space. When you stop the presentation, the model also stops. The right-hand figure shows both top spaces slaved off a separate title clock, which means you can independently stop either the model or presentation.

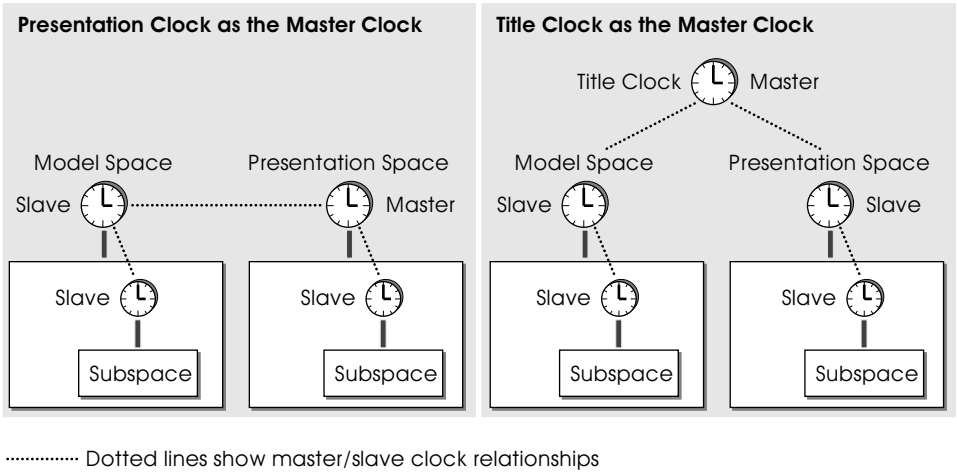


Figure 3-5: Two ways to synchronize the model space and presentation space.

When two clocks have a master-slave relationship, changing the rate of the parent clock increases or decreases the speed of both clocks. However, the two clocks can still have different scales and rates—the rate of the master clock merely acts as a multiplier for the rate of the slave.

Controllers in a Space

Each space maintains a list of controllers that manipulate objects in the space. To attach a controller to a space, you assign the space to the controller's `space` instance variable. That space in turn automatically adds the controller to its read-only `controllers` instance variable. The `space` instance variable determines which space the controller is controlling. It can contain only one space at a time and thereby ensures that the controller manipulates only one space at a time.

The following code assigns the space `mySpace` to the controller `myController`, which means that `mySpace` will then automatically add `myController` to its array of controllers:

```
myController.space := mySpace
```

A controller has a `wholeSpace` instance variable to indicate which objects in a space are to be controlled by it. If `wholeSpace` is set to `true`, the controller will control all appropriate model objects in the entire space specified by the controller's `space` instance variable. If `wholeSpace` is set to `false`, the controller will control only objects which are explicitly added to the controller. The default value for `wholeSpace` is `false`.

In addition, when an object is added to a space, if the value of the controller's `wholeSpace` instance variable is `true`, the space notifies each controller, and adds the object to the controller only if it is appropriate for that controller. See the chapter on controllers for information on what makes an object appropriate for a controller.

Time-based controllers respond to ticks of the clock by specializing `tickle`, an instance method that can be defined by subclasses of `Controller`. Once every tick of the space's clock, its periodic callback calls `tickle` on the space, causing it to iterate through every one of its controllers that implements a method for `tickle`. This allows the controllers to perform some periodic action on target objects. (Controller actions are always periodic, and they may also be incremental.)

A controller that needs to be informed or "tickled" each time the clock ticks should implement a specailization of `tickle`. A `tickle` method should run to completion within a fraction of a tick, since all controllers run once with every tick. If too much is going on in the model space, so that controllers that implement `tickle` cannot finish running before the clock's next tick, then the callback skips the next cycle, causing all of the controllers that are attached to the space to be skipped. Thus, either all controllers attached to the space get tickled, or none get tickled. (In the same way, the compositor skips if presentation would start too late.)

When controllers get skipped, it's up to the particular `tickle` implementation as to whether they catch up or not—it could be smart and look at the time of the space's clock, or it could be dumb and make the same increment even when it skips.

Controllers are described in greater detail in the chapter "Controllers."

How Presenters Work

Presentation is obviously an important part of any title—it is the graphic view the user has of the model, the stage where a multimedia title and the user meet and interact. The only way a user can view objects that exist in an underlying model space is if those objects are represented with presenter objects.

Presenter

While this section uses an example with `TwoDPresenter` objects, its main focus is to describe features that the `Presenter` class provides on its own, irrespective of `TwoDPresenter`.

Perhaps surprisingly, the `Presenter` class does not know anything about graphics, stencils, brushes, drawing, compositors, display surfaces or coordinate systems, and has no interest in mouse events. Those are all functions of `TwoDPresenter`. `Presenter` is responsible only for establishing the presentation hierarchy—it is left up to subclasses to determine what kinds of objects can be presented and how they are to be presented.

In the current release of ScriptX, `Presenter` has only one direct subclass, `TwoDPresenter`, from which all other presenter subclasses inherit. The `TwoDPresenter` class inherits from and builds on the features of `Presenter`. In future versions, you might imagine a `ThreeDPresenter` class, also inheriting from `Presenter`, with its own 3D objects and ways of presenting them.

Presentation Hierarchy

A complete 2D presentation hierarchy is determined by a top presenter and all its subpresenters. The `Presenter` class provides a mechanism for establishing a presentation hierarchy among presented objects in a title. This hierarchy can be used to determine the order in which the presenters are ordered, accessed, queried, operated on, and presented. A title can have any number of presentation hierarchies, as illustrated in the "Title Management" chapter. Each hierarchy has a window as its top presenter.

An example of a 2D presentation hierarchy is shown in Figure 3-6. This example has four presenters: instances of `Window`, `MoviePlayer`, `DigitalVideoPlayer`, and `TwoDShape`. An instance of `Window` is the top presenter and contains the other presenters.

A presentation hierarchy is not to be confused with a class inheritance tree. A presentation hierarchy is an illustration of how presented objects contain other presented objects.

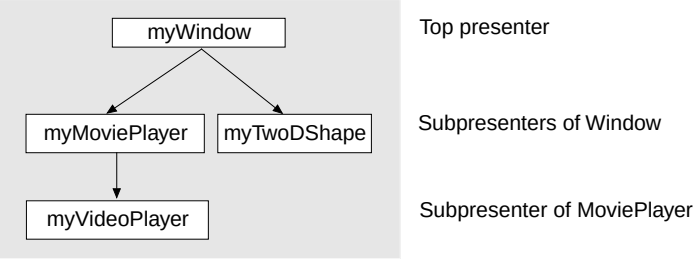


Figure 3-6: A simple presentation hierarchy

A *presenter* is an instance of a concrete subclass of `Presenter`. Presenters come in two varieties: "containers," which can hold and display other presenters, and "simple" presenters, which cannot. In general, a container corresponds to a node of the presentation hierarchy that branches to multiple subpresenters, while a simple presenter corresponds to a leaf at the end of a branch. (Specific classes for simple and container presenters are shown in Figure 3-16.)

In the previous example, `myTwoDShape` and `myVideoPlayer` are simple presenters, while `myWindow` and `myMoviePlayer` are container presenters—`myWindow` contains `myMoviePlayer` and `myTwoDShape`, while `myMoviePlayer` contains `myVideoPlayer` (as well as an instance of the non-graphic class `DigitalAudioPlayer`, not part of the hierarchy).

Simple and container presenters are described further in the section "Simple Presenters vs. Container Presenters" on page 76 later in this chapter.

Subpresenters

Each presentation hierarchy is made up of a top presenter and any number of other presenters. The top presenter is always a window. The `Presenter` class has two instance variables that establish this hierarchy: `subpresenters` and `presentedBy`.

Working down the hierarchy, some presenters have so-called *subpresenters*—a list of presenters that it presents—held in the `subpresenters` instance variable. The presenter at the top of a hierarchy (a window) can have any number of subpresenters, some of them may have subpresenters, and so forth. Every presenter in a presentation hierarchy is a subpresenter, except for the top presenter.

Working up the hierarchy, all presenters except the top presenter are contained within, or presented by, another presenter. This is the presenter that appears above it in the hierarchy. The `presentedBy` instance variable holds this presenter. For the top presenter, this value is `undefined`.

While some presenters can have multiple subpresenters, all presenters are presented by at most one presenter. In other words, while `subpresenters` can be a list, `presentedBy` is a single value.

Since the `subpresenters` instance variable can hold a list, a presentation hierarchy can be thought of as lists within lists, nested to any depth. The previous example is illustrated as lists of lists in Figure 3-6—the top presenter has two subpresenters, one of which is `MoviePlayer`, which itself has one subpresenter.

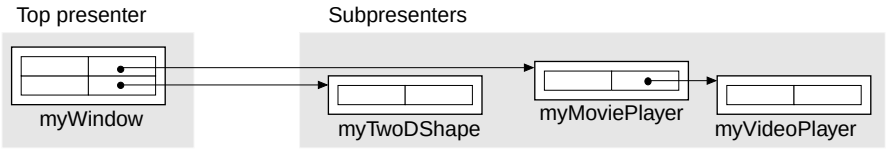


Figure 3-7: Subpresenters are lists within lists.

Some subclasses of `Presenter` allow subpresenters, while others do not because they present only themselves and no other presenters. A presenter which presents only itself has its `subpresenters` instance variable set to `undefined`.

An entire presentation hierarchy can be traversed, encountering every one of its members, by starting at the top and working sequentially downward through each `subpresenter` list. Similarly, starting at any presenter, this hierarchy can be traversed up to the top presenter using `presentedBy`. This is one of the fundamental uses of a presentation hierarchy—to provide an orderly way of visiting every presenter.

The `Presenter` class also has a `target` instance variable, which can hold a source object to be presented—either a presenter or a non-presenter object. A target is not connected directly to the presentation hierarchy, and is implemented differently by different subclasses.

The `Presenter` class is abstract and does not specify how the presentation hierarchy is to be used; various subclasses of `Presenter` can implement different uses. For example, `TwoDPresenter`, the root abstract class for the presentation of all 2D graphics, uses this hierarchy to draw its subpresenters to a window. In addition, it uses the order of subpresenters to determine the front-to-back order in which overlapping subpresenters appear. You could define your own subclass of `Presenter` and give your own meaning to the presentation hierarchy.

Notice that the Presenter class is quite abstract—it specifies only the presentation hierarchy. The Presenter class has no notion of a coordinate system, no interest in mouse events, and no connection to a display surface onto which instances can be rendered—these are left to TwoDPresenter.

TwoDPresenter

The TwoDPresenter class is the only direct subclass of Presenter in the core classes. The TwoDPresenter class provides for the display of two-dimensional objects in ScriptX on 2D graphic devices, such as display monitors. 2D presenter objects include any of the visual media: text, graphic shapes, bitmap images, transitions, animation, and video.

A *2D presenter* is an instance of a concrete subclass of TwoDPresenter, and represents an object that is presented to the user through graphic images.

How the Presentation Hierarchy is Used

The TwoDPresenter class uses the presentation hierarchy, described previously, in two specialized ways not encompassed in the Presenter class:

- For drawing the presenters in an orderly manner.

  The hierarchy organizes the drawing of multiple 2D presenters to a single display surface attached to a window in a title. This hierarchy provides an ordered way of calling draw on all presenters, ensuring each is presented exactly once.

  The compositor calls draw on the top presenter, which in turn calls draw on each successive subpresenter, giving overlapping presenters a front-to-back visual ordering.

- For receiving mouse events. This is described in the Events chapter. When the user performs a mouse action, the hierarchy enables a search for the front-most presenter at that location to receive the mouse event.

It is important that a presenter appear only once in the presentation hierarchy—otherwise, one of the presenters cannot receive events, and time is wasted drawing the presenter more than once. TwoDMultiPresenter and its subclasses have safeguards to prevent them from containing a presenter more than once—the objectAdded method ensures that every time a presenter is added to a container, the presenter is removed from its previous presentedBy container.

A title can have any number of presentation hierarchies, but needs a window (display surface) for each one, as shown in Figure 3-8.

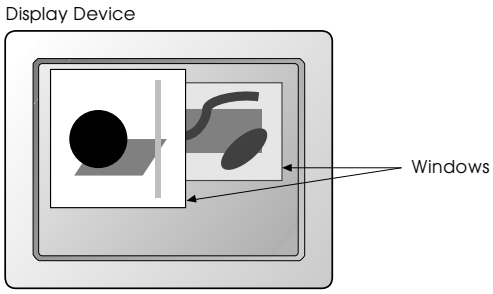


Figure 3-8: Every window has its own presentation hierarchy.

Drawing To a Window's Display Surface

Instances of `TwoDPresenter` by themselves are not sufficient for viewing; they must draw themselves to a display surface in a window, where they are made visible to the user. `TwoDPresenter` defines the generic imaging function `draw` that the compositor uses to construct image frames by rendering the presenter to a display surface.

The generic function `draw` is central to compositor imaging of all ScriptX objects. `TwoDPresenter` itself does not implement a method for `draw`. Each subclass must implement a `draw` method for any drawing to occur. Every concrete subclass of `TwoDPresenter` should implement its `draw` method in the manner most efficient for that class. `TwoDPresenter` does not manage the drawing of any of its subpresenters, either—the `TwoDMultiPresenter` class specializes `TwoDPresenter` to manage subpresenters.

For each presenter in a window, the compositor automatically calls `draw` on that presenter to tell the presenter to render itself to the window's display surface. For more control of on-screen drawing, you can disable the compositor and explicitly call `draw` so that the presenter is rendered directly to a display surface. Or, for off-screen drawing, you can call draw on a bitmap surface and then transfer the off-screen bitmap to a display surface for viewing.

The next section of this chapter describes windows, which act as both a container space and as the top presenter for the presentation hierarchy. Other instances of `TwoDPresenter` are described later in this chapter. For more discussion of display surfaces, see Chapter , "2D Graphics."

Window

Windows are an essential part of any title that has visible objects—all 2D presenters must be contained in a window to be composited and displayed. A window is a collection that holds 2D presenters and has a rectangular display surface on-screen for clipping and displaying them. Every window has a compositor to orchestrate the drawing of all presenters and a clock for timing. A window can also have a set of controllers for manipulating objects within it. The appearance of the window's title bar and border are defined by the underlying operating system, as shown in Figure 3-10.

A window object is any instance of the Window family of classes, which includes `Window` and its specialized subclass, `FullScreenWindow`. A side-by-side comparison of these appears later in this section.

To make an instance of window visible, you create an instance and then call `show` on it:

```
myWin := new Window
show myWin
```

The only objects that can be at the top of a presentation hierarchy are instances of `Window` and its subclasses. When you create a new window, it automatically creates its own display surface to draw to. A window also automatically provides a clock to run the 2D compositor. A window is always the top presenter; in fact, windows cannot be further down in the presentation hierarchy—they can only be at the top. In other words, you cannot add a window to another window, as you can with all other presenters. Being at the top, a window has no parent—that is, its `presentedBy` is `undefined`.

A window forms a rectangular clipping region that allows subpresenters within its boundary to be displayed, and crops away any parts of subpresenters outside its boundary.

Any presenters added to the window become its subpresenters. All 2D presenters are designed to live in windows; this includes user interface objects, text, video players, 2D shapes and document templates. For further details about other properties of windows common to 2D space, such as subpresenters, z-value, clipping, clock, and controllers, refer to “TwoDSpace” on page 87.

Showing and Hiding a Window

As demonstrated in the previous section, a window’s `show` method enables the window’s compositor, sets `isVisible` to `true`, brings the window to the front, and gives the title and window user focus.

Conversely, the `hide` method removes the window from the screen, sets the window’s `hasUserFocus` to `false`, gives user focus to the next window on-screen, disables the window’s compositor, and sets `isVisible` to `false`.

```
hide myWin
```

A user can also hide a window by clicking in the close box (Macintosh only) or choosing “`Close Title`” from the system menu (Macintosh or Windows)—either way calls `hide` on the window. Although people commonly speak of “closing a window,” these operations call `hide`, not `close` on it (`close` is a method for library, title, and accessory containers).

Be sure not to confuse closing a window with closing a title. The **Close** menu command closes the title and all its windows. Notice that clicking the close box is not equivalent to choosing the **Close** menu command on the File menu.

The `isVisible` instance variable is persistent, which means windows remember the setting when the title is closed. Therefore, when you open a title, and then call `load` on a window, it will immediately be displayed if its `isVisible` is set to `true`—you do not have to also call `show` on it.

When a window is hidden, it can be garbage collected by dropping all references to it and making it purgeable—see the section “Freeing a Window from Memory” in the “Title Management” chapter for details.

Managing Windows in a Title

When a window is added to a title container, it can be managed by that title, or it can be reassigned to be managed by a different title. Managing windows includes making sure the windows share user focus with the title, pausing the compositors for its windows when the title is paused, and closing windows when the title is closed.

For more on managing a title, see Chapter , “Title Management.”The title management chapter also describes freeing and saving windows in a title.

Window Subclasses

Windows come in different styles, determined by setting the window’s type when a window is instantiated. Table 3-1 lists the attributes for each type of window. A *modal* window is one that does not allow any action outside the window until the user hides it—in other words, the user must responds to the modal window before continuing. Clicking outside a modal window causes a beep.

The title bar is the horizontal strip along the top of the window, containing a name. A user moves a window by dragging its title bar.

Note that none of the ScriptX window types uses the underlying operating system’s scroll bars. To make a window scrollable, you can either add an instance of `ScrollingPresenter` to that window, or define your own scrolling presenter by creating a space and setting up scrollbars to scroll through that space.

Collection methods are used to add objects to a `Window` object.

Table 3-1: Attributes for the Window family of classes

Type	Class	Modal?	Title Bar	Close Box or Menu?
General	`Window`	No	Yes	Yes
Palette	`Window`	No	Yes	Yes
Full Screen	`FullScreenWindow`	No	No	No
Dialog	`Window`	Yes	Yes	No
Notice	`Window`	Yes	No	No

The open windows that are managed by a particular title container are listed in the title container's `windows` instance variable. This list is sorted by user focus—the frontmost window for the title appears first in the list. Conversely, each window in the list will have its `title` instance variable set to the title container which manages it.

Different types of windows are displayed in different layers, as shown in Figure 3-9. A palette window always appears in front of a normal window or full-screen window. A dialog window or notice window always appears in front of other windows. Within a given layer, windows can appear in either order, so a notice window can appear either in front of or behind a dialog window.

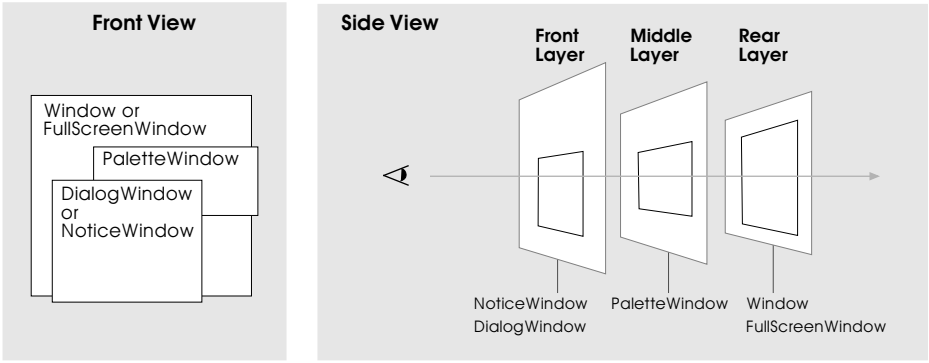


Figure 3-9: Windows are displayed in three different layers.

In Microsoft Windows 3.1, most 2D graphics resources (such as `Bitmap`, `Brush`, `Region`, and `Window` objects) are allocated out of a 64K GDI (Graphics Device Interface) memory heap. This 64K GDI heap is shared by all Windows 3.1 applications running at any given time. When this heap starts filling up, many 2D operations in ScriptX will start failing.

The number of ScriptX windows that can be opened at once depends on what kind of 2D presenter objects you have created in each window and what portion of the system's GDI resources are being used by each. It is a good rule of thumb for a title in Microsoft Windows 3.1 to open at most 5 windows; more can be opened if they contain simple graphics, fewer if they contain complex graphics.

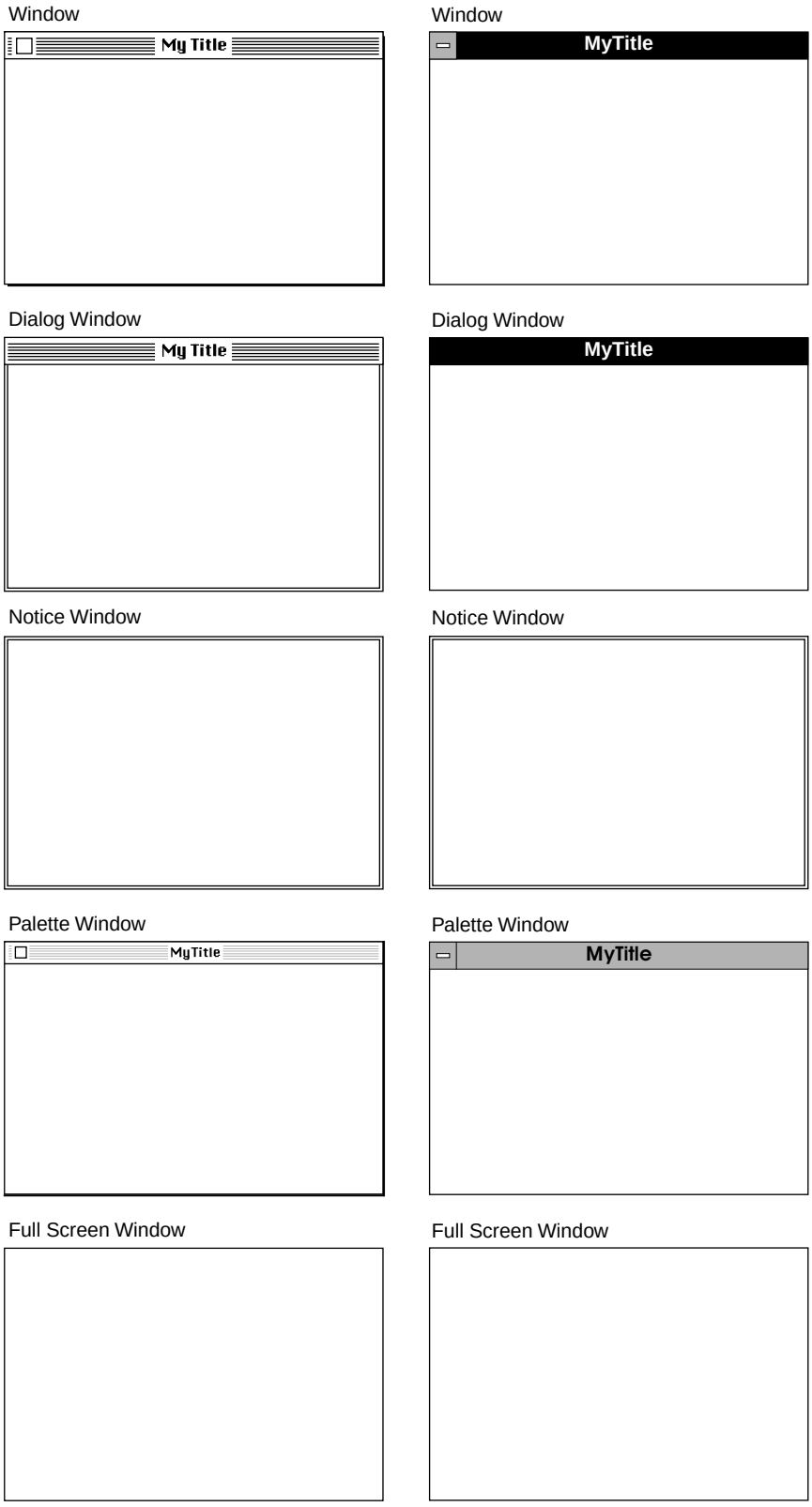


Figure 3-10: The appearance of a window is determined by the underlying platform.

Display Surface, Compositor, and Clock

When you create an instance from the `Window` family of classes, a new display surface, 2D compositor, and clock are automatically created and attached to the window.

A new display surface is automatically created and connected to the window. This display surface provides an area of the display screen visible to the user for the 2D presenters to be drawn to; any presenter drawn to the display surface is thereby visible to the user. A reference to the display surface is stored in the `displaySurface` instance variable:

```
myWindow.displaySurface
```

A new 2D compositor is also created automatically; it orchestrates the drawing of the presentation hierarchy to the display surface. The window becomes the top presenter for that 2D compositor. The 2D compositor is held in the `compositor` instance variable:

```
myWindow.compositor
```

The display surface and 2D compositor are described later in this chapter in “How Compositors Work” on page 89.

A new clock is automatically created for all spaces, so a window always has a clock by virtue of being a space. A window’s clock determines the timing for the compositor and also for any controllers that might be added to the window. It is held in the `clock` instance variable:

```
myWindow.clock
```

Adding Objects to a Window

In the following example, `myWindow` is created as a window with a display surface of 100 x 100 pixels. It contains a movie player and pushbuttons for making the movie play or stop, as in Figure 3-11. The movie player and pushbuttons are subpresenters of the 2D space. (This example assumes you have previously defined pushbutton presenters `myRewind`, `myStop`, and `myPlay`.)

```
myWindow := new Window boundary:(new Rect x2:100 y2:100)

myVideoPlayer  := new DigitalVideoPlayer \
                      boundary:(new Rect x2:100 y2:100)
rewindButton   := new Pushbutton releasedPresenter:myRewind
stopButton     := new Pushbutton releasedPresenter:myStop
playButton     := new Pushbutton releasedPresenter:myPlay

append myWindow myVideoPlayer
append myWindow rewindButton
append myWindow stopButton
append myWindow playButton

show myWindow
```

Figure 3-11 shows three different views of the same set of objects. The User's View is the view the user sees of the instance; the Internal State is a list of the instance variables and numbers of the window, name (left column) and their values (right column). The Presentation Hierarchy shows how the presented objects can contain other presented objects.

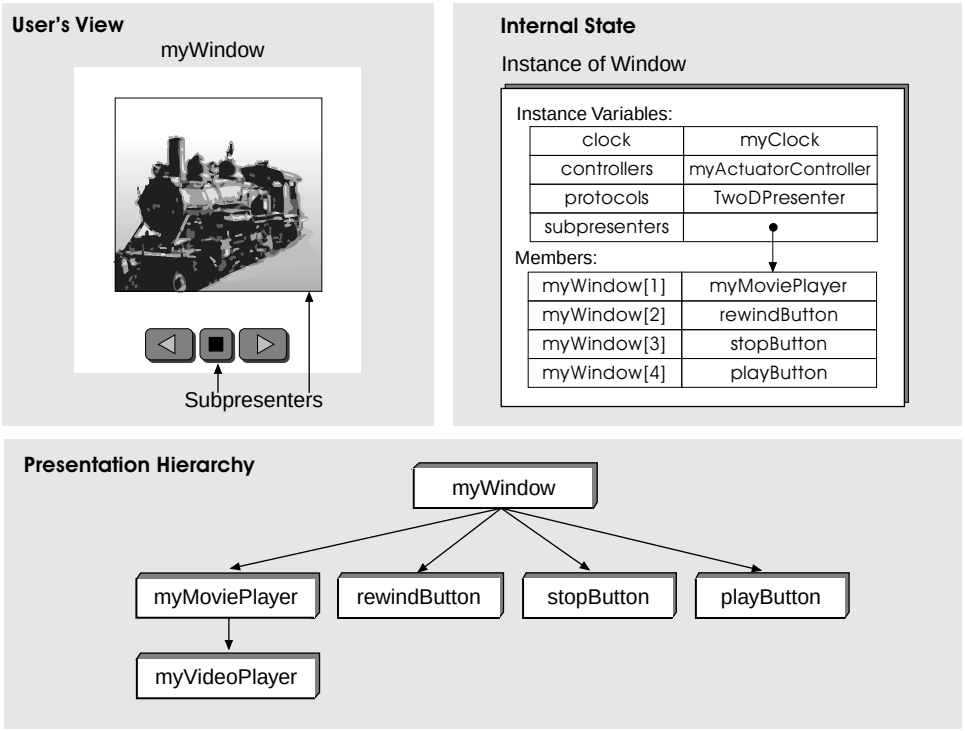


Figure 3-11: Three views of a window.

To do "off-screen" imaging, draw to an instance of `BitmapSurface` instead of `DisplaySurface`. You could then transfer this image to a display surface. This technique could be useful when you want to build an image and control the time at which this image is displayed.

Coordinate Systems

Three different basic coordinate systems exist in a ScriptX display, as shown in Figure 3-12. For example, when the user clicks the mouse on a presenter, that point can be returned in the local coordinates of the presenter, the coordinates of the display surface, or the screen coordinates of the ScriptX Player. For all three of these, x increases to the right, and y increases downward. These three coordinate systems have origins located as follows:

- Screen coordinates – origin is located at the upper-left corner of the ScriptX Player. For the Macintosh, this is always the corner of the screen; however, for Microsoft Windows and OS/2, the ScriptX Player lives in its own window that can be made smaller than the screen.

- Surface coordinates (abbreviation for display surface coordinates) – origin is located at the upper-left corner of the usable display surface. For all windows except instances of `FullScreenWindow`, this origin is just below the title bar at the upper-left corner of the window. For instances of `FullScreenWindow`, the origin is the upper-left corner of the display surface, and the instance variables `screenCoords` and `surfaceCoords` are identical.

  The surface coordinates are used for measuring `globalBoundary`, `globalRegion`, and `globalTransform`.

- Local coordinates – origin is located at the upper-left corner of each 2D presenter. Also called "presenter coordinates."

The `TwoDPresenter` class has two methods for converting a point between surface and local coordinates: `surfaceToLocal` and `localToSurface`.

Note that you can hide the menu bar that displays the words **File** and **Edit** — see the `SystemMenuBar` class. Doing so on the Macintosh exposes the previously hidden area behind the menu bar to the user — the rest of the screen stays as-is.

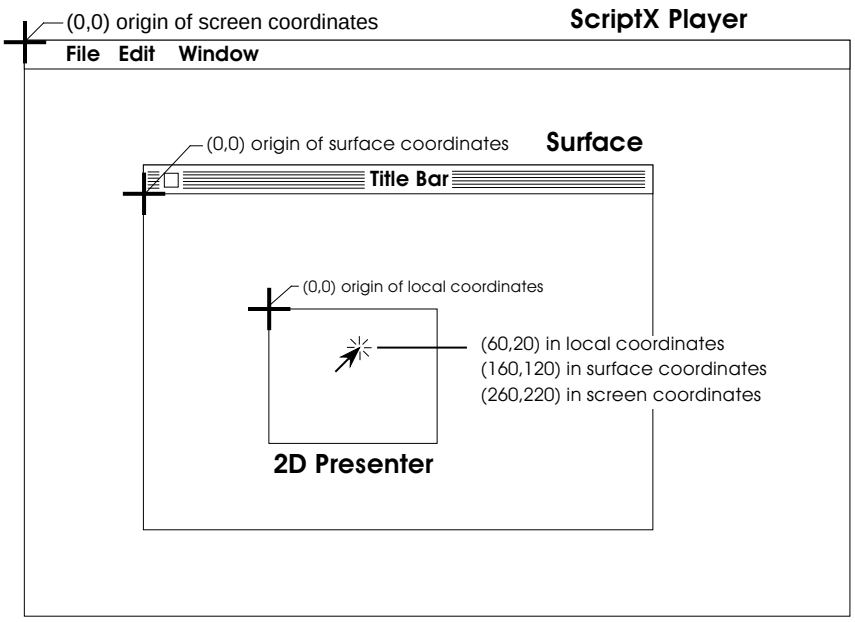


Figure 3-12: The ScriptX Player with the menu bar showing.

Machine-Specific Screens

In Microsoft Windows and OS/2, the origin of the surface coordinates is located at the top edge of the system menu bar, but shifts upward when the system menu bar is hidden. On the Macintosh, the origin remains fixed at the top of the system menu bar, regardless of whether it is showing or not.

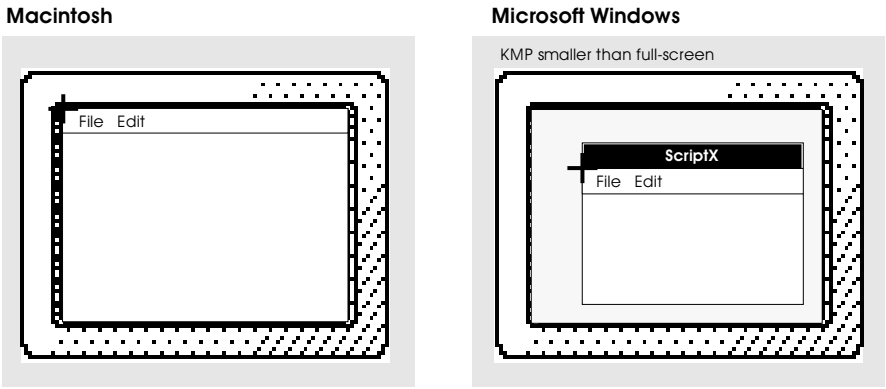


Figure 3-13: The Microsoft Windows ScriptX Player has a title bar and is re-sizeable.

Also, in Microsoft Windows, the ScriptX Player itself is located in a movable window with a title bar, and so can be resized and moved around on the screen, as shown in Figure 3-13. On this platform, the only way to hide the title bar is to use an instance of `FullScreenWindow`. The Macintosh version of the ScriptX Player has no title bar.

Properties of 2D Presenters

All 2D presenters have in common the properties described in the following sections. These properties are implemented as instance variables and include `target`, `transform`, `boundary`, `globalBoundary`, `bBox`, `position`, `x`, `y`, `z`, `size`, `window`, `compositor`, `clock`, `needsTickle`, `isTransparent`, `stationary`, and `direct`.

A 2D Presenter's Target

In general, the `target` is the model object that the 2D presenter is providing a view to. For example, an instance of `String`, `StringConstant`, or `Text` is the target for a `TextPresenter` object, as in Figure 3-14. Instances of these classes cannot draw themselves—they require a presenter such as `TextPresenter` to display them.

For `TwoDShape`, the target is the stencil it is drawing—the oval, rectangle, line, path, bitmap or region.

In other cases, the `target` itself is a presenter. For example, the target of `CostumedPresenter` is a `TwoDPresenter` object.

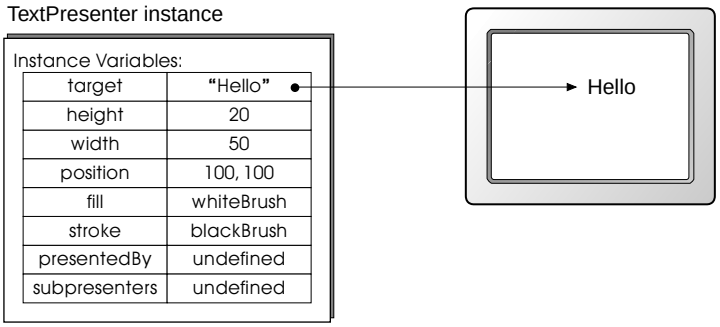


Figure 3-14: The target of this `TextPresenter` is an instance of `Text`.

Not all presenters require a target. The `target` instance variable is not directly used by `TwoDMultiPresenter`, `TwoDSpace`, or `GroupSpace`—it is available for use by subclasses. When a target is not needed, `target` should be set to `undefined`.

Transform, Position and Size

Each `TwoDPresenter` object has a `transform` instance variable that holds a `TwoDMatrix` instance that specifies the presenter's x-y location and scale. The 2D presenter also has the following instance variables that directly access values derived from the `transform` matrix: `position`, `x`, and `y`. The `bBox`, `height`, and `width` instance variables are separate and are not derived from the `transform` matrix.

While the transform matrix is capable of rotation for stencils, this operation is not currently available for 2D presenters. To get this effect, you can create an instance of `TwoDShape` and instead rotate its target stencil.

2D presenters use the same basic coordinate system described in the 2D Graphics component. Every presenter has a local coordinate system with its own origin (0,0) based on the underlying stencil. If the stencil's values for `x1` and `y1` are `0`, then the local origin (upper left corner of the presenter) coincides with the origin, as shown for `mySpace` in Figure 3-15; otherwise, the stencil's upper left corner will be offset from the origin by the stencil's values for `x1` and `y1`. The x-axis is positive going to the right and the y-axis is positive going downward. The unit of measure for 2D presenter coordinate systems is pixels. As illustrated in Fig. 3-15, the stencil for `myMulti` has an `x1` value of 3 and a `y1` value of 3, putting the local origin of `myMulti` at the point 3,3 in the coordinate system of `mySpace`.

Any presenters in a container are positioned relative to the origin of that container. For example, the left-hand side of Figure 3-15 shows a 2D space, `mySpace`, that contains a 2D multipresenter, `myMulti`, that holds a rectangle. (Both the 2D space and 2D multipresenter have `x1` and `y1` set to `0`.) Notice that the rectangle is positioned relative to `myMulti`, not `mySpace`. Any subpresenter of `mySpace` is positioned relative to the origin of `mySpace`. Moving `mySpace` moves its contents as well.

The right-hand side of the figure shows a 2D space, `mySpace`, that contains two rectangles, both of which are positioned relative to `mySpace`.

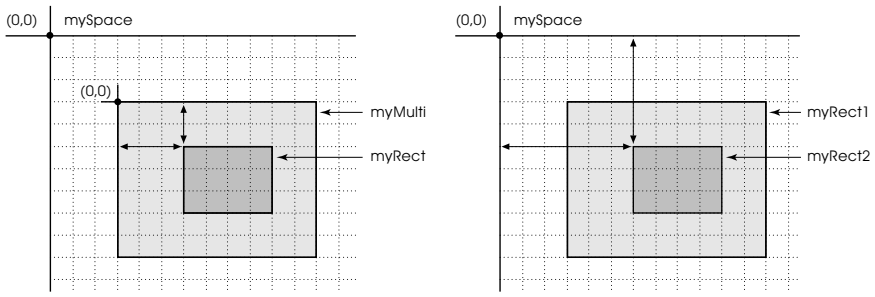


Figure 3-15: The local origin of a container is its upper left corner.

Every 2D presenter has an x-y position relative to the origin of the presenter it is immediately contained in, except windows, which have an x-y position relative to the origin of the display coordinates. The x-y position is specified by two instance variables: `x` and `y`.

The `position` instance variable, a `Point` object representing both `x` and `y`, is a convenient way of accessing both values at the same time. If you were to move an object by setting first its x value and then its y value, you might see the object shift twice: first in the x-direction, and then again in the y-direction. However, by setting the `position` instance variable, the object shifts once diagonally to its proper destination.

The `z` instance variable specifies the front-to-back position of a 2D presenter, relative to other overlapping presenters within their parent container. If the 2D presenter is not contained in an instance of `TwoDMultiPresenter` or one of its subclasses, `z` is ignored.

The `bBox` instance variable is an abbreviation for "bounding box," the smallest rectangle that can fully enclose the presenter. The `bBox` rectangle is not visible, but is used in certain operations. The `height` and `width` instance variables are the dimensions of the 2D presenter's bounding box.

The `boundary` is a stencil that describes the perimeter of a presenter. While `bBox` is always rectangular, `boundary` can be non-rectangular, because it can be any stencil. Each subclass of `TwoDPresenter` has its own restrictions on which shapes are possible. For `Window` and its subclasses, this boundary must be a rectangle. However, for `TwoDMultiPresenter`, `TwoDSpace`, and `TwoDShape`, the boundary can be a region or any of the stencils available in ScriptX, such as a line, oval, rectangle, rounded rectangle, path, or bitmap.

The instance variable `globalBoundary` contains a rectangle like `bBox`, except that its points are specified in surface (window) coordinates. (The presenter's global boundary is unrelated to the `boundary` instance variable, which is a stencil.) This rectangle specifies the clip rectangle that is used when drawing to a surface—only the part of the presenter inside the clip rectangle is drawn. Likewise, the value of `globalTransform` specifies the transformation matrix for the 2D presenter. This matrix determines where the 2D presenter is located in surface (window) coordinates.

Window, Compositor, and Clock

The `window` instance variable specifies the window that the 2D presenter is contained in, either directly or indirectly. If the presenter is not in a window, this value is `undefined`.

The `compositor` instance variable specifies the 2D graphic compositor used to draw the 2D presenter to its window. In fact, the 2D compositor is responsible for drawing all members of a presentation hierarchy to their window. The compositor and display surface are automatically created when you create the window. Thereafter, when a presenter is added to a presentation hierarchy, its value of `compositor` is automatically filled in.

The `needsTickle` instance variable should be set to `true` to indicate that the presenter needs to have its `tickle` method called with each composite cycle; otherwise, it should be set to `false`. The 2D graphics compositor maintains a private list of presenters that need to be tickled, and setting a presenter's `needsTickle` instance variable to `true` automatically adds the presenter to this list. By maintaining this list, the compositor avoids having to poll each presenter in a space.

If you want to create your own `tickle` method, refer to the entry on TwoDPresenter in *ScriptX Class Reference*.

The `clock` instance variable is read-only and specifies the clock used to control this presenter—it is `undefined` for instances of most core class 2D presenters. Exceptions include those that inherit from `Space` and `Scrollbar`. Thus, `clock` has a value for `TwoDSpace`, `GroupSpace`, `PageLayer`, `Window`, and their subclasses.

The compositor is described later in this chapter under "How Compositors Work" on page 89. The top presenter's clock is described under "Compositor Clock" on page 92".

Notifying the Compositor of Changes

The `TwoDPresenter` method `notifyChanged` is the mechanism for notifying the compositor about changes to presenters. This method does two things:

- It causes a `TwoDPresenter` instance to compute its changed region and pass this region on to the compositor.
- It provides an optimization hint to the compositor. The first argument to `notifyChanged` is an instance of `TwoDPresenter`; invoking `notifyChanged` indicates that there has been some change to the presenter specified in this first argument. The second argument is a `Boolean` object indicating whether or not the change involves the presenter's position or boundary. Giving `true` as the second argument indicates that the presenter's boundary has changed, the presenter's position has changed, or both have changed. Supplying `false` as the second argument tells the compositor that only the image has changed—neither the position nor the boundary of the presenter has changed. When only the image has changed, the compositor can skip several steps and render the presenter more efficiently. Consequently, calls to `notifyChanged`, which may be quite

costly, are more efficient when the second argument is `false`. The following code tells the compositor that `myTwoDPresenter` has changed only its image:

```
notifyChanged myTwoDPresenter false
```

In most cases, `notifyChanged` gets called 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.

There are cases, however, where you must explicitly call `notifyChanged`, For example, if you specialize a presenter so that it makes changes, and those changes do not automatically call `notifyChanged`, you will have to include an explicit call to `notifyChanged`. If `notifyChanged` is not called, either explicitly or implicitly, the changes you implemented will have no effect on presentation.

You must also call `notifyChanged` explicitly when you change an indirect attribute. An indirect attribute is an attribute of an attribute, rather than an attribute of an object directly. For example, `y` is a direct attribute in "`myRect.y`" and an indirect attribute in "`myRect.target.y`". The following code illustrates this situation:

```
-- Create a window and put a rectangular 2D shape in it
myWin := new Window boundary:(new Rect x2:200 y2:200) fill:whiteBrush
myWin.y := 40
show myWin

myRect := new TwoDShape target:(new Rect x2:50 y2:50) fill:blackBrush
append myWin myRect
myRect.y := 50
-- Now make the rectangle half as wide by changing its stencil
myRect.target.x2 := 25    -- Notice this has no effect
show myWin
notifyChanged myRect true    -- The shape is now redrawn
show myWin
```

In this example, `myRect.target` is the stencil. Setting the stencil's width to 25 has no effect on the display until `notifyChanged` is invoked because the stencil's `x` value is an indirect attribute of `myRect`. The following code changes the width of `myWin` using a direct attribute, which means that the setter method for `width` will call `notifyChanged` automatically:

```
myWin.width := 25 -- changes the width and calls notifyChanged
```

Note – The `TwoDPresenter` method `notifyChanged` replaces the flags `changed` and `imageChanged`. ScriptX 1.5 still recognizes `changed` to allow for backward compatability, but it just redirects to `notifyChanged`.

Re-Drawing the Changed Presenters

At each frame, the display must show all presenters in their most current state—that is, any changes to presenters since the previous frame must be drawn to the window. Rather than re-drawing the entire presentation hierarchy every frame, the system draws only the fewest presenters necessary to meet this end.

There are basically two kinds of changes that can cause a presenter to be redrawn:

- The presenter itself changes shape, position or image
- Another presenter that was in front of it moves away, uncovering more of it

Only presenters meeting one or both of these conditions should be redrawn. To meet this requirement, the system keeps track of which presenters have changed (those which have had `notifyChanged` called on them), and then draws only those presenters that intersect the changed presenters.

`TwoDPresenter` has two instance variables, `isTransparent` and `stationary`, which give hints that the compositor uses when it is trying to be smart about what to re-draw:

- `isTransparent` indicates whether a 2D presenter is transparent, and therefore, whether presenters or regions of presenters beneath it are visible. (A presenter is transparent if it does not render all of the pixels encompassed by its boundary.) Set `isTransparent` to `true` if the 2D presenter is transparent; otherwise, set `isTransparent` to `false`. If an object is not visible because it is occluded, then it is not necessary to draw changes to it. The following code informs the compositor that `myTwoDPresenter` occludes any presenters directly beneath it:

```
myTwoDPresenter.isTransparent := false
```

When `notifyChanged` is called on a transparent presenter, presenters directly beneath it need to be re-drawn also.

The default value for `isTransparent` is `false`.

- `stationary` is set to `true` when an instance of `TwoDPresenter` satisfies one or both of the following conditions:
  - it doesn't move much
  - it occupies a large space

In all other cases, `stationary` should be set to `false`.

A window is an example of a presenter that is stationary because it is large; objects like scrollbars, pushbuttons, or `TextEdit` text are examples of presenters that are stationary because they are regions that don't change location much.

The default value for `stationary` is `false`.

When `stationary` is set to `false`, the compositor will not spend time figuring the fewest number of presenters that need to be re-drawn.

Optimizing When a Presenter Is Re-drawn

The compositor uses hints provided by `notifyChanged`, `isTransparent`, and `stationary` to help optimize drawing.

The following conditions help the compositor decide which presenters should be re-drawn:

- `notifyChanged myTwoDPresenter true` —Since the second argument is true, it indicates that the presenter's position or boundary has changed, either of which necessitates calculating the area of the presenter that needs to be redrawn.
- `myTwoDPresenter.stationary := false`—This indicates that the presenter is either small or moves a lot, either of which causes the presenter to be re-drawn completely. If an object is small and moves often, it is generally cheaper just to re-draw it rather than to calculate the area needing to be re-drawn and then re-drawing only that specific area. If an object is large, however, it may be more efficient to calculate the area which has changed and to re-draw only regions which have changed.
- `myTwoDPresenter.isTransparent := true` (and presenters beneath it have changed)—Since the presenter is transparent, any changes to objects directly beneath it are visible and therefore need to be re-drawn.

The compositor looks for situations where it does not need to re-draw. In simplest terms, these are situations in which changes are not visible.

For example, assume there are two presenters, A and B. A is not transparent and completely covers B, which is beneath it. If B changes, those changes will not be visible because A occludes B. If only A's image changes, A needs to be re-drawn, but the changes to B don't need to be re-drawn because they are still not visible. If A moves, however, so that changes to B become visible, then both A and B need to be re-drawn.

The compositor also tries to draw as little as possible. As mentioned above, it ends up being more efficient to simply re-draw all small objects, even if the change affects only a small area. Consequently, if a presenter's `stationary` instance variable is set to `false`, it will always be completely re-drawn when it is visible (when any presenters directly above it are transparent). When a large object changes, however, it may be worth calculating which regions changed and re-drawing only those regions.

Improving Drawing Speed with Direct Presenters

Some 2D presenters have certain characteristics that lend themselves to drawing optimization by the compositor, to speed up performance. To achieve optimization, it may be important to set the `direct` instance variable on certain 2D presenters.

The `direct` instance variable should be set to `true` for video and other rectangular presenters that have a fixed location and boundary but with quickly moving images within that boundary. This flag optimizes the drawing speed by drawing directly to a display surface, bypassing the compositor's off-screen frame buffer.

This flag is set to `false` by default for all new presenters. If you want an instance of `DigitalVideoPlayer` to be `direct`, for example, you must explicitly set its `direct` flag to `true`.

```
myDigitalVideoPlayer.direct := true --causes compositor to draw
--directly to the display surface
```

Direct presenters always appear in front of non-direct presenters (and so override z-ordering). Direct presenters should not overlap other direct presenters—if they do, then the user will see flickering as they both struggle to update the overlapped area. Do not use `direct` for video that flies across the screen or has other presenters overlapping it. Direct presenters are also restricted to being rectangles.

The major benefit with direct presenters is reducing the amount of drawing that takes place for each frame. When a non-direct presenter is asked to draw, it draws the *entire* presenter within the clip region. In contrast, when a direct presenter is asked to draw, it can draw just the parts of it that have changed since the previous frame, with assurance that no other part of the image has been disturbed by other objects. Since the compositor prevents other presenters from drawing within direct presenter areas of the display surface, a direct presenter is the only one allowed to draw to its region of the display surface.

For example, as a direct presenter, a digital video player can do "incremental" updates at each frame, just redrawing the bits of the surface that have changed. Similarly, a transition effect can run more efficiently as a direct presenter, blitting only the part of the image that changes.

The `refresh` method (defined by `TwoDPresenter`) is called on direct presenters 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 entirely redrawing itself (that is, not doing just an incremental update, but a complete update). The default behavior of the `refresh` method is just to call `draw`.

Diagnostics for Improving Drawing Speed

ScriptX provides several diagnostic functions that can help you optimize performance. These are available in the development environment, but not with the ScriptX Player. The `showChangedRegion` instance variable, defined by `TwoDCompositor`, can be set `true` to mark a red outline around any subsequently changed regions in the compositor's window:

```
myWindow.compositor.showChangedRegion := true
```

The red outlines accumulate until the window is refreshed (which you can do by calling `refresh` on the window or hiding and showing it). You can set `showChangedRegion` to `true` to help optimize your title to do the least amount of updating possible. For example, instead of animating by switching between large bitmaps, an author could have a large bitmap and several smaller ones that layer on top of it successively.

The `warnings` function is a diagnostic that displays warnings that can affect execution of a script or slow performance. The following script demonstrates how to turn on `warnings`.

```
warnings true
```

A warning is issued when a bitmap transfer occurs with mismatched color maps. For example, the warning appears when a bitmap image with one colormap has been added to a window with a different colormap. While this works, such a mismatch can cause performance to 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
```

Receiving Mouse Events

A 2D presenter is associated with a region on the display surface, and it can receive mouse events that occur within that region. Clicking on, moving over, or dragging a presenter could cause the presenter to change colors, change images, begin an animation, bring in a new scene, or initiate some other action.

The `eventInterests` instance variable in `TwoDPresenter` is a read-only list of event interests associated with a given 2D presenter. A full description of this instance variable and how a 2D presenter can receive mouse events is given in the Events chapter.

Classes like `ActuatorController` and `DragController`, which inherit from `TwoDController`, provide an easy interface to events for presenters.

Subpresenters of TwoDPresenter

As described earlier, every presenter has a `subpresenters` instance variable. For presenters that have only themselves and no other presenters to display, the `subpresenters` value is `undefined`; for other presenters, it can contain a list of presenters to be displayed. The presentation hierarchy is made up of a window and a network of subpresenters, where a subpresenter can have its own subpresenters, nested to any level deep. Each list of subpresenters forms a branch of the presentation hierarchy. Putting a presenter in a subpresenters list places it in the presentation hierarchy.

If you want to make a presenter be displayed, you can add it to a presenter container in the presentation hierarchy—the presenter will automatically show up in the container's subpresenters list. For example, if you prepend an

instance of `TwoDShape` to an instance of `Window`, the shape is displayed as part of the window's presentation hierarchy because a window's `subpresenters` instance variable points to the window and its contents.

At this point it's useful to elaborate on the impact that the `subpresenters` instance variable can have on the character of a presenter—it distinguishes between simple presenters and container presenters.

Simple Presenters vs. Container Presenters

Figure 3-16 lists the class trees for "container" presenters and simple presenters. Container presenters are 2D presenters, such as windows, that can have multiple subpresenters. This means they can hold and display multiple 2D presenters together. Adding a presenter to a container presenter automatically adds the presenter to the container's subpresenters list. This happens because the `subpresenters` instance variable points to the members of the presenter container, as shown on the left side of Figure 3-18.

Figure 3-18 compares the two kinds of 2D presenters—"containers," which can hold and display other presenters, and "simple" presenters, which can hold only their targets and can display only one target at a time:

- A "container" presenter is an instance of the `TwoDMultiPresenter` family of classes, which itself is a collection of presenters. The container presenter itself and all its members are presented (subject to clipping). A container forms a node in the presentation hierarchy, with its subpresenters below it. Examples are instances of `Window`, `TwoDSpace`, and `GroupPresenter`.
- A "simple" presenter inherits from `TwoDPresenter` but is *not* a member of the `TwoDMultiPresenter` family. A simple presenter presents only one object at a time. Examples are instances of `TwoDShape`, `TextPresenter`, and `OneOfNPresenter`.

An important attribute of container presenters is that any transform on the container affects all of its member presenters—the members all transform as a group. In other words, the x-y coordinate position of all members is relative to the container. Thus, moving the container moves all of its contained objects. In addition, its contained objects can be clipped by the container (`TwoDMultiPresenter`, `TwoDSpace`, `Window`) or not clipped (`GroupPresenter`, `GroupSpace`).

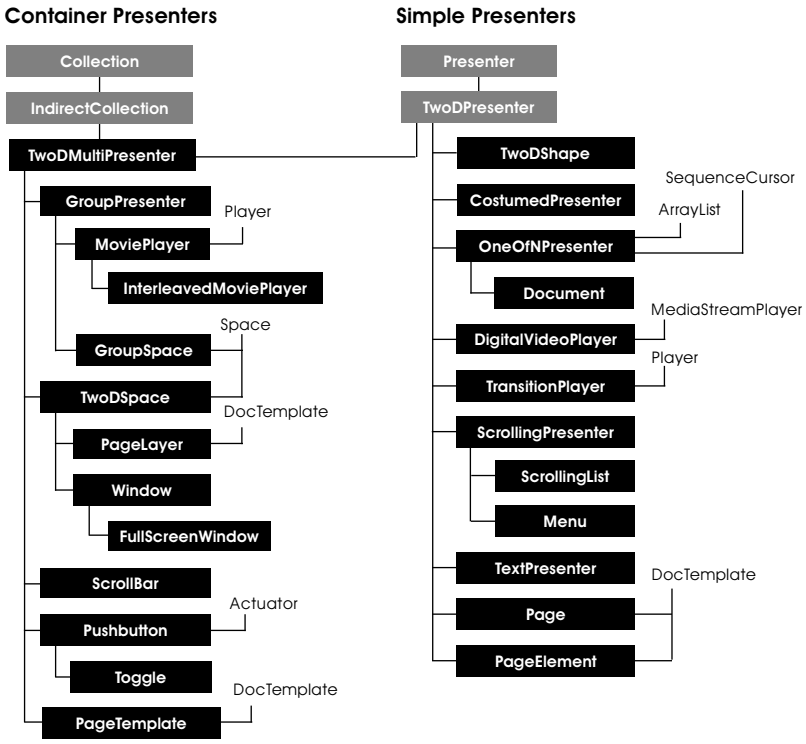


Figure 3-16: Container presenters inherit from TwoDMultiPresenter, simple presenters do not.

In general, a container presenter has its own visible features—a window has a border, fill, and stroke. The members of the window, which are the subpresenters, determine which additional objects are to be presented, and in what front-to-back visual order. The order of the objects in the members list is the order in which they are drawn to the screen, starting with the last object in the list (in back) and working to the first (in front).

Figure 3-18 compares the internal states of container presenters and simple presenters. In containers, the `subpresenters` instance variable points to the members of the container; simple presenters do not have multiple subpresenters. To simplify the diagram, not all instance variables are shown.

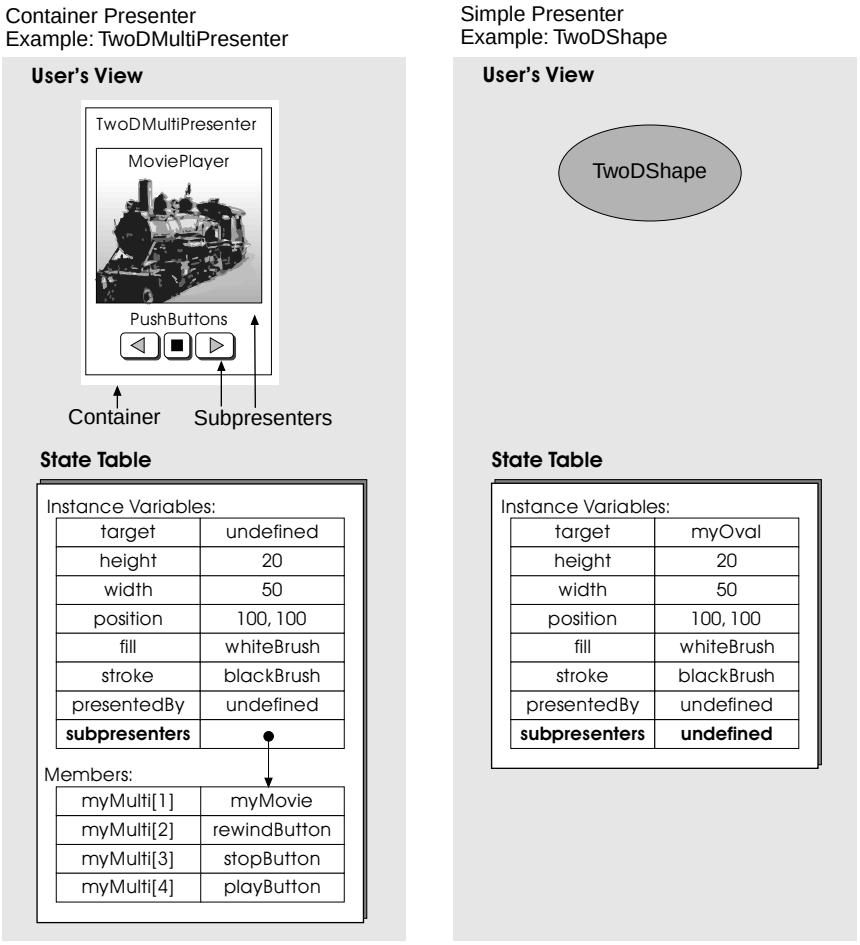


Figure 3-17: Comparison of a container and a simple presenter.

For containers, the mechanism for subpresenters is straightforward: the `subpresenters` instance variable points to the container itself, meaning the subpresenters are the members in the container. Therefore, any object added to the container is automatically in its list of subpresenters, where it is part of the presentation hierarchy, and appears to the user; these multiple objects are presented together.

For example, a new instance of `TwoDSpace` is by default an array (because its `targetCollection` is an instance of `Array`). The subpresenters of the 2D space are the members of that array. Any object added to the 2D space is automatically a subpresenter, as the following script shows. In particular, the shape `myBox` is a subpresenter of `mySpace`.

```
mySpace := new TwoDSpace
myBox := new TwoDShape boundary:(new Rect x2:50 y2:50) \
    fill:blackBrush
prepend mySpace myBox
```

Each concrete subclass of `Presenter` manages how objects are moved into and out of the subpresenters list. For example, for `TwoDMultiPresenter` (and its subclasses), all of its members are also subpresenters and are displayed together; however, for `OneOfNPresenter`, only one of its members is moved 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. Presenters that inherit from `TwoDMultiPresenter` can have multiple subpresenters, while presenters that don't inherit from `TwoDMultiPresenter` typically have either zero or one subpresenter. Presenters that inherit from `TwoDMultiPresenter` are called presentation *containers* because they can contain more than one subpresenter at a time.

- The simplest non-container presenter, `TwoDShape`, presents its target stencil, which is not a presenter, and so `subpresenters` is set to `undefined`.
- More complex non-container presenters, such as `OneOfNPresenter` and `CostumedPresenter` are themselves not visible, and require one subpresenter to display. `OneOfNPresenter` maintains a list of presenters to present, but can show only one at a time—only the one that is held in the `subpresenters` instance variable. `CostumedPresenter` can also change the object it is presenting—again, the object being presented is the one currently held in `subpresenters`.
- Container presenters, such as `TwoDMultiPresenter`, `TwoDSpace`, `Window`, `GroupPresenter`, and `MoviePlayer`, can have multiple subpresenters, which are displayed together.

If you create a subclass of `TwoDPresenter` or `Presenter` rather than `TwoDMultiPresenter`, the resulting class will by default have no subpresenters—you must add the extra functionality required to handle subpresenters.

Simple Presenters

The Spaces and Presenters component contains three classes that are simple presenters: `TwoDShape`, `CostumedPresenter`, and `OneOfNPresenter`.

TwoDShape

The `TwoDShape` class provides the means for presenting and displaying 2D graphic objects, such as bitmaps, rectangles, ovals, curves, and other shapes. `TwoDShape` objects operate as presenters for 2D graphic objects, using the 2D compositor.

You create graphics using a stencil and brush technique, as described in Chapter 11, "2D Graphics." When creating an instance of `TwoDShape`, you specify the appropriate stencil. Among the stencil classes that the `TwoDShape` class can present are the `Line`, `Rect`, `RoundRect`, `Path`, `Oval`, `Region`, and `Bitmap`.

When a TwoDShape object is asked by the graphics compositor to draw itself, it renders its stencil onto a surface, using its fill and stroke (outline) brushes. The stencil determines the area's shape. One brush can determine the color and pattern for the fill, while another brush can determine the color, pattern and line width for the stroke. (These terms are described in Chapter 11, "2D Graphics.")

Sample Script for TwoDShape

The following is a sample script for creating a rectangle and displaying it in a window. The new method on Rect creates the rectangular stencil, which by itself cannot be added to a window. The new method on TwoDShape specifies myRect as its target, and sets its fill and stroke (outline). The next two lines position the shape. After the window is created, the prepend method adds the shape to the window.

```
-- Create a rectangle stencil
global myRect := new Rect x2:50 y2:50

-- Create a rectangle presenter from the stencil and position it
global myBox := new TwoDShape target:myRect fill:blackBrush\
stroke:blackBrush
    myBox.x := 20
    myBox.y := 20

-- Create a window
global myWindow := new Window boundary:(new Rect x2:300 y2:300)

-- Add the 2D shape to the window
prepend myWindow myBox
show myWindow
```

CostumedPresenter

CostumedPresenter is a presenter that by itself is not visible—it needs another presenter to be its "costume." The object being presented, or "costume," is held in both the target and subpresenters instance variables. Figure 3-18 shows the state of an instance of CostumedPresenter.

Use a CostumedPresenter object when you want a presenter that can change appearance, say from text to bitmap, while retaining position information and connections to other objects, such as controllers. For example, you can use CostumedPresenter to model a picture hanging on a wall—the costumed presenter represents the blank place on the wall, and its costume is the picture.

CostumedPresenter instance

Instance Variables:

target	myCostume
height	20
width	50
position	100, 100
fill	whiteBrush
stroke	blackBrush
presentedBy	undefined
subpresenters	**myCostume**

Figure 3-18: The subpresenter for CostumedPresenter is its costume.

`CostumedPresenter` implements a kind of delegation.Another ScriptX option for delegation is the `Delegate` class, a utility class that is part of the Object System Kernel. For general information on delgation in ScriptX, see “Delegation” on page 625 of Chapter 23, “Object System Kernel.”

OneOfNPresenter

`OneOfNPresenter` is a presenter that contains a list of presenters, only one of which can be displayed at a time. Figure 3-19 shows the state of an instance of `OneOfNPresenter`. While `OneOfNPresenter` is a collection, it can display only one object at a time, so its `subpresenters` instance variable holds a single item and does not point to a collection.

OneOfNPresenter instance

Instance Variables:

target	undefined
height	20
width	50
position	100, 100
fill	whiteBrush
stroke	blackBrush
presentedBy	undefined
subpresenters	●

Members:

myOneOfN1]	MyTrout
myOneOfN[2]	myCrab
myOneOfN[3]	**myLobster**
myOneOfN[4]	mySnail

Figure 3-19: The subpresenters for OneOfNPresenter points to a single presenter, not a list.

`OneOfNPresenter`, like `CostumedPresenter`, also implements a kind of delegation. `OneOfNPresenter` embodies a one-to-many relationship, in which one member from a collection of presenters is the current presenter. For general information on delgation in ScriptX, see “Delegation” on page 625 of Chapter 23, “Object System Kernel.”

TwoDMultiPresenter

`TwoDMultiPresenter` is a subclass of `TwoDPresenter` that mixes in `IndirectCollection` and thereby provides a way of presenting a collection of `TwoDPresenter` objects. TwoDPresenter provides the following features, beyond the basic functionality of the `TwoDPresenter` class:

- A `TwoDMultiPresenter` object contains multiple subpresenters—all of which can be presented at the same time and have positions relative to the 2D multipresenter.
- `TwoDMultiPresenter` sorts its subpresenters by their `z` value, and displays them front-to-back in this order.
- A `TwoDMultiPresenter` object defines properties for `boundary`, `fill`, and `stroke`, and uses them to render itself, just as a `TwoDShape` object does. However, the boundary forms a clipping region. Any part of a subpresenter that would be drawn outside this boundary is clipped.
- A `TwoDMultiPresenter` ensures that an object appears only once in the presentation hierarchy. Every time a presenter is added to a container, it is removed from its previous container, if it has one, as determined by the value of `presentedBy`.

It is important that a presenter appear only once in the presentation hierarchy—otherwise, time is wasted drawing the presenter more than once, and only the first instance that is traversed during event delivery can receive events. Ensuring that an object cannot be duplicated in the presentation hierarchy is one of the main features built into `TwoDMultiPresenter`. Whenever you add a member object to an instance of `TwoDMultiPresenter`, it is removed from its previous container.

Containment

`TwoDMultiPresenter` forms the basis for container presenters. It inherits its container properties from `Collection`, by way of `IndirectCollection`. `TwoDMultiPresenter` classes "contain" their subpresenter objects visually. `TwoDMultiPresenter`, `TwoDSpace`, `GroupPresenter`, and `GroupSpace` are container presenters, while `TwoDShape`, `CostumedPresenter`, and `OneOfNPresenter` are not.

Subpresenters within any container share the same x-y coordinate system and origin as their container. When you move a 2D space, all objects in that space move with it. Both of these effects are due to the concatenation of matrices during the presentation cycle.

All presenters that are added to a 2D multipresenter at a location within its boundary are displayed. Presenters that are added, but appear outside the multipresenter's boundary, are clipped by that boundary. Members of the collection are presented because the `subpresenters` instance variable points to them (by way of pointing to the object itself), as shown in Figure 3-20.

In this example, an instance of `TwoDMultiPresenter` contains a collection of images, arranged with cars in front and a movie in the background.

`TwoDMultiPresenter` is useful when you need to manage multiple `TwoDPresenter` objects, but don't want the overhead of managing controllers and a clock that `TwoDSpace` requires. It's also a simple way of clipping a presenter or group of subpresenters.

Figure 3-20 shows three different views of a `TwoDMultiPresenter` object:

- User's view – Shows how the user would view and interact with this instance of `TwoDMultiPresenter`.
- Internal state – Shows the instance variables and member objects.
- Presentation hierarchy – Shows the relationship of presenters and subpresenters

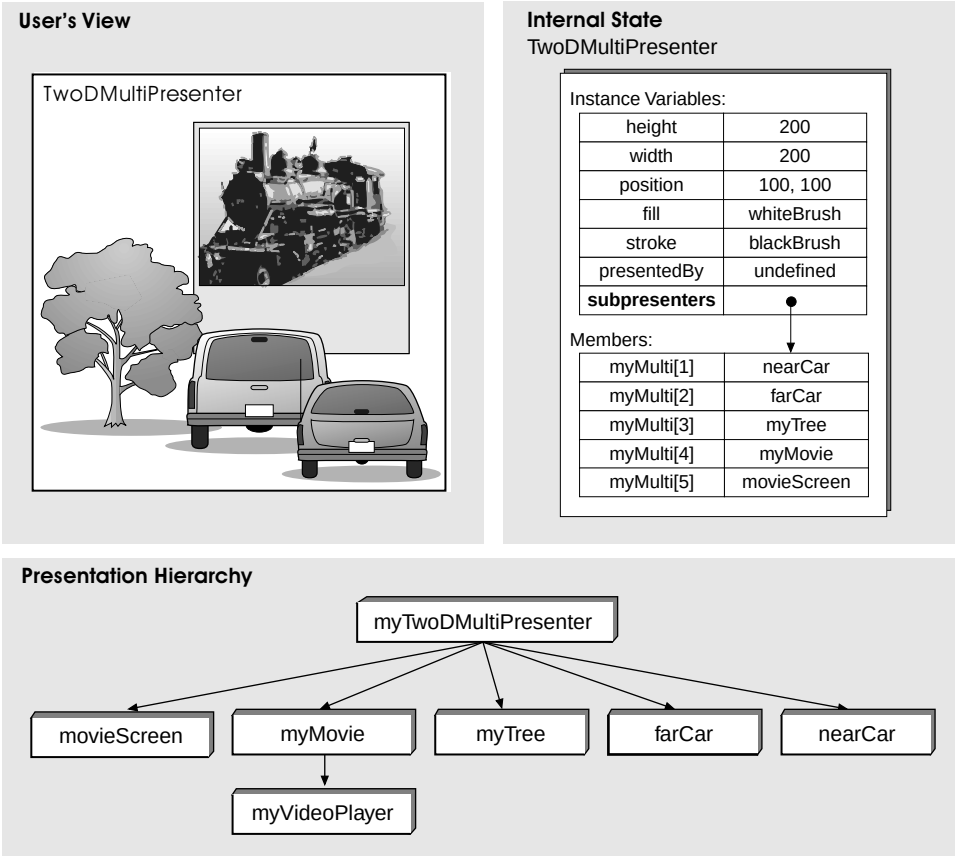


Figure 3-20: Three views of an instance of TwoDMultiPresenter

Creating a TwoDMultiPresenter

To create an instance of `TwoDMultiPresenter`, call `new` and supply a value for the `boundary` keyword. For example, to create the `TwoDMultiPresenter` instance shown in Figure 3-20, assuming the subpresenters `myMovie`, `myTree`, `myCar1` and `myCar2` already exist, you first create the multipresenter and then add its subpresenters to the container individually.

```
myMulti := new TwoDMultiPresenter \
       boundary:(new Rect x2:200 y2:200)
prepend myMulti movieScreen
```

```
prepend myMulti myMovie
prepend myMulti myTree
prepend myMulti farCar
prepend myMulti nearCar
```

For performance reasons, when creating an instance of `TwoDMultiPresenter` (or any of its subclasses, such as `TwoDSpace`, `GroupSpace`, `Window`, or `PageLayer`), you should generally omit the `targetCollection` keyword and allow the class to set its target collection by default. Multipresenters require collections that can be traversed easily for drawing presenters and handling events. Generally, the default target collection is an instance of `Array`, with the keyword `initialSize` set to some optimal level, given the expected size of the collection. Performance could suffer if you change the target collection to something other than the default. Bounded arrays, such as `Pair`, `Triple`, and `Quad`, can be specified as a target collection where appropriate.

Drawing

The `draw` method defined by `TwoDMultiPresenter` renders its image onto the display surface of the window, with clipping defined by a clip stencil. The `draw` method performs three steps when drawing an instance of `TwoDMultiPresenter`:

- Fills the instance, using the Brush object defined by fill, which forms a background
- Iterates `draw` on each subpresenter
- Strokes the instance's boundary

The boundary is drawn last, because its thickness can overlap presenters contained inside the instance of `TwoDMultiPresenter`.

Finding Presenters Within a Container

`TwoDMultiPresenter` has four methods for finding presenters that it contains:

- `findAllAtPoint` – Finds all objects that intersect the specified point, given in local coordinates of the 2D multipresenter.
- `findFirstAtPoint` – Finds the first (front-most) object that intersects the specified point, given in local coordinates of the 2D multipresenter.
- `findAllInStencil` – Finds all objects contained in the 2D multipresenter that intersect the specified stencil.
- `findFirstInStencil` – Finds the first (front-most) object contained in the 2D multipresenter that intersects the specified stencil.

These are useful for determining which objects are located at a specific place in the `TwoDMultiPresenter` object. For example, you could use `findAllInStencil` to determine which presenters are contained inside a rectangle that the user draws—this allows the user to drag-select objects. When

the user clicks, you could use `findFirstAtPoint` to do hit detection of the front-most presenter. The `findAllAtPoint` method will return an array of all presenters that contain the specified point.

Clipping

`TwoDMultiPresenter` and many of its principal subclasses within the core class—`TwoDSpace`, `PageTemplate`, and `Window`—allow the programmer to set the multipresenter's boundary directly. These multipresenters use the parent presenter's boundary to clip subpresenters. In this way, `TwoDMultiPresenter` extends the meaning of `boundary`, an instance variable defined by `TwoDPresenter`.

Several subclasses of `TwoDMultiPresenter`—`GroupPresenter`, `GroupSpace`, `PushButton`, and `Toggle`—override this behavior. These presenters grow to encompass the union of the objects they contain. In these presenters, the boundary is determined automatically based on the boundary of subpresenters. The `boundary` keyword to the `init` method is ignored, and attempting to set the value of `boundary` reports an exception.

Z-Ordering

When 2D presenters overlap on the screen, their front-to-back display order is determined by their order in the presentation hierarchy, which is their position in subpresenters lists. Within a given list, a presenter at the front of the list (that is, with 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 presenters toward the top of the list the draw later, and hence, on top. Subpresenters of any `TwoDMultiPresenter` class are ordered by z-value, the value of its `z` instance variable, as follows.

Initially, and unless you specify otherwise, all presenters have by default a z-value of 0. Thus, if you never explicitly specify a z-value, all presenters will have a z-value of 0, and have their front-to-back order determined, as always, by their order in the list. To add a new presenter in front of other presenters, you `prepend` it to the list of subpresenters; to add it behind other presenters, you `append` it.

To change the position of a presenter in the front-to-back order, you can either:

- Move it directly
- Specify a `z` value

To move it directly, call `moveForward`, `moveBackward`, `moveToFront` or `moveToBack` on it. You can also use collection methods, such as `prepend`, `append`, or `setNth`, to specify its position in the subpresenters list when you are adding it to a `TwoDMultiPresenter`.

To specify a `z` value, set the presenter's `z` instance variable. Setting this value actually moves the presenter in the subpresenter list, so that once again the order of presenters in the list determines their draw order. A higher, more positive value of `z` corresponds to a position in front of other presenters.

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 displays it in front of them.

The following table shows this relationship between index numbers in a subpresenters array, z-values, and visual ordering:

Table 3-2: Relation between index, z-value and visual ordering

Index in subpresenters array: (from 1 to n)	1	2...	n
z-value:	high	medium	low
Visual ordering:	front	middle	back

The value for `z` can be any integer from –2,147,483,647 to +2,147,483,648. The initial z-value for a new presenter is 0.

Setting z-values is the way to set the front-to-back order of presenters so that this order is an attribute of the presenters themselves. If you want a presenter to stay in the background, you can give it a negative z-value; then if it gets moved to another space, it can maintain its z-position.

If you specify a z-value and then use a contradictory method to add the object to a 2D multipresenter, the z-value takes precedence. For example, if all the presenters in a 2D multipresenter have a z-value of 0, and you append a presenter with a z-value of 1, it is inserted in front of all the presenters. You would expect the `append` method to insert it at the back, but its z-value is higher than the other presenters. However, the `append` method does make sure that the presenter is inserted at the back of any presenters that have the same z-value.

Example of PushButton as a TwoDMultiPresenter

As another example, a toggle pushbutton can be created from the `Toggle` class, which is a subclass of `Pushbutton` (both are defined in the User Interface component). `PushButton` in turn inherits from `TwoDMultiPresenter`.

An instance of `Toggle` has two states, on and off. In addition, it has a third visible state that appears when the mouse button is down on it. Thus, it can display at most three subpresenters simultaneously—one from each of the following groups:

- `pressedPresenter` or `releasedPresenter`
- `toggledOnPresenter` or `toggledOffPresenter`
- `disabledPresenter` or `undefined`

For example, a pushbutton can display its `releasedPresenter`, `toggledOnPresenter`, and `disabledPresenter`. Figure 3-21 shows the internal state of a typical pushbutton and its view to users.

For more information on `Pushbutton` and `Toggle` classes, refer to the section on actuators in the “User Interface” chapter.

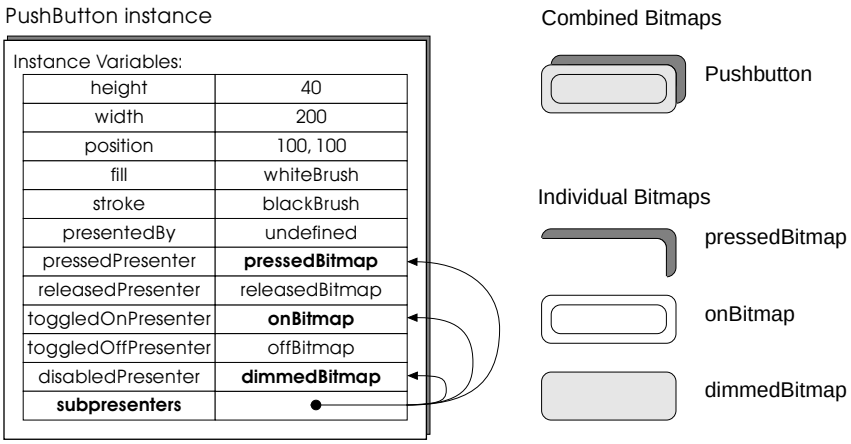


Figure 3-21: The internal state and user view of a pushbutton that is pressed, on and dimmed.

TwoDSpace

`TwoDSpace` is the simplest two-dimensional space for use inside a window. This class combines the presentation and collection attributes of `TwoDMultiPresenter` with the clock, controllers and protocols attributes of `Space`. Use a 2D space wherever you want an area within a window to have its own clock and controllers.

A `TwoDSpace` object creates an environment where presenter objects can live, including presenters that display bitmaps, shapes, video images, animations, text, and other media. A space is an interactive environment, with any object being potentially clickable or draggable, including windows, toggles, pushbuttons, scrolling lists and so forth. As the user manipulates objects in a space, those objects can control objects in that space or any other space—or even objects that are not in any space. Since it is a subclass of `TwoDSpace`, the `Window` class shares all of these attributes.

`TwoDSpace` inherits its container and presenter features from `TwoDMultiPresenter`. Any object successfully added to a `TwoDSpace` automatically becomes a subpresenter of the space and is added to the presentation hierarchy. Each `TwoDSpace` object has a boundary that is fixed and clips its subpresenters. Every ScriptX presenter defines two instance variables that determine its position in the presentation hierarchy. You can iterate down through this hierarchy by means of the `subpresenters` instance variable of each `presenter`, or up by means of the `presentedBy` instance variable. The ordering of subpresenters at any given level in the hierarchy can be specified by setting their `z` values.

As with any instance of `TwoDMultiPresenter`, `TwoDSpace` implements a `draw` method that first fills the space, using the brush specified by `fill`, to form a background. Next, it renders the outline of the space, using the brush specified by `stroke`. Finally, it iterates over all of its subpresenters in the presentation hierarchy, calling `draw` on each subpresenter.

TwoDSpace inherits its timing and simulation features from the Space class. Space defines three instance variables—clock, controllers, and protocols—that a space uses to simulate behavior in an environment over time. Its clock is used for timing. Its controllers operate on objects that are added to the space, modifying their behavior over time, or in response to some other stimulus.

TwoDSpace also inherits from Collection, by way of IndirectCollection. A TwoDSpace object is a collection which contains other objects. IndirectCollection defines a notification protocol for controlling what objects can be added to a space. A space uses isAppropriateObject to determine which objects are allowed into the space, based upon the value of protocols. It uses its objectAdded and objectRemoved methods so that it can respond each time an object is added to or removed from the space. For more information on IndirectCollection, see page 469 of Chapter , “Collections.”

Presenters in a 2D space are rendered in the reverse order that they appear in this collection, with the last item in the collection rendered first. When objects overlap, the first presenters in the collection are displayed in front of the last presenters in the collection. Thus overlapping objects within a 2D space present themselves in a manner similar to most drawing programs, with some objects occluding others.

All TwoDPresenter objects are designed to live in a 2D space; this includes instances of subclasses of TwoDPresenter, such as user interface objects and 2D shapes. A TwoDSpace object cannot be a top presenter, unless it is a window as well. Thus, it can be contained within another instance of TwoDSpace. By contrast, a Window object, which is a kind of TwoDSpace, cannot be a subpresenter—a window must be the top presenter in a presentation hierarchy.

The visible shape of a TwoDSpace object within its parent space is determined by the value of its boundary instance variable. This property, defined by TwoDPresenter, contains a Stencil object. All presenters within a space are clipped to the space’s boundary. The position of a 2D space is given by its transform matrix, which includes its x-y position in its parent container.

Protocols

For any instance of TwoDSpace, the protocols instance variable is defined to include the TwoDPresenter class. This means that objectAdded checks to make sure that any object added to the space inherits from TwoDPresenter. If not, then objectAdded prevents the object from being added to the space.

How Compositors Work

Important – This section, which describes the interaction of the `TwoDCompositor`, `DisplaySurface`, and `Window` classes, has not been updated to reflect changes in ScriptX 1.5. For up-to-date information on the compositor in ScriptX 1.5, see the `TwoDPresenter` and `TwoDCompositor` classes in the *ScriptX Class Reference*.

Every ScriptX window has its own compositor which initiates the drawing of its presenters. Compositors orchestrate the visual presentation of a title.

Note – The 2D graphic compositor is a low-level part of the presentation system which manages the drawing of presenters to a window's display surface. It can be important to understand the 2D compositor for optimizing a title's performance.

Once compositing starts, each presenter in the hierarchy draws its changed region to the window's display surface, including those that have moved, changed image, or changed boundary. The compositor is designed to ensure in general that regions of the screen that have not been altered do not get redrawn. The compositor accumulates changes to the window internally.

The window's presentation hierarchy ensures that presenters are drawn in the correct order, from background to foreground, building up the composition as each presenter is drawn. Compositing is just another word for the mechanism by which presenters are drawn to the screen in an orderly, time-driven manner. The class that embodies this functionality is `TwoDCompositor`.

Introduction to the 2D Compositor

A compositor assembles discrete 2D presenters into a composite image once every tick of its clock. A 2D compositor is necessary to continually refresh the graphic images. If a compositor were not present, then any moving image would erase other images it passes in front of. The background image would not be restored because it was drawn only once. The compositor handles the updating of all the images for you, and attempts to do it in an efficient manner, updating only what has changed since the last drawing cycle.

A compositor works closely with other components in ScriptX, as shown in Figure 3-22. This diagram has four major components: a presentation hierarchy, a 2D graphic compositor, a frame buffer, and a window's display surface. The compositor initiates the `draw` method on the presenters to render them to the off-screen frame buffer and then transfer that image to the window's display surface.

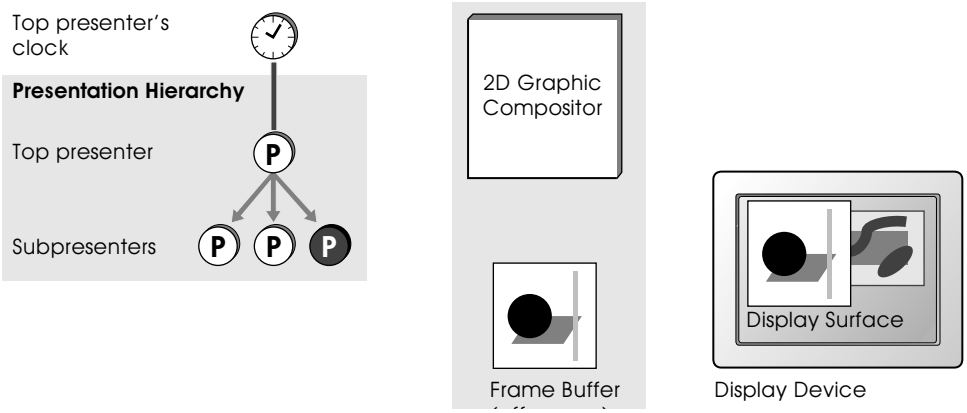


Figure 3-22: Graphic compositor controls the display of the presentation.

As described earlier, the *presentation hierarchy* is an ordered structure of 2D presenters. Its purpose is to provide an order to the presenters for capabilities such as front-to-back ordering in the user's view, controlling the order in which events traverse presenters, and providing an ordered way of reaching every presenter exactly once during the presentation phase. Each presenter in the presentation hierarchy is capable of presenting itself to the frame buffer or to the display surface.

The *frame buffer* is the off-screen area of memory that maps directly to the display surface. The frame buffer gives the compositor a place to construct the changed parts of a frame; it is the area where each of the changed presenters draws itself. A time-based presentation consists of a number of visual frames per second; a frame is one complete image in this sequence of images. A typical presentation might run at 10 to 30 frames per second. The frame buffer is not directly accessible at the scripter level.

The *display surface* is the visible part of a window where the presenters are ultimately drawn. The `draw` method (defined in `TwoDPresenter`) operates on a display surface, not a window. A window is the collection of presenters to be displayed. When you add presenters to a window, they are displayed on its display surface. A title can have multiple windows, and, hence, display surfaces, and they can overlap each other. A display surface is represented by an instance of the `DisplaySurface` class, and a window by an instance of the `Window` class (or its subclass).

The *2D graphic compositor* orchestrates the timing and drawing of the presentation hierarchy onto the display surface during the presentation phase. There is a separate compositor per display surface. A compositor is represented by an instance of the `TwoDCompositor` class.

Creating a Compositor

You do not create a compositor directly from a script. When you create an instance of `Window`, a compositor is automatically created, and attached to that window, to manage the compositing of presenters in that window. This compositor, an instance of `TwoDCompositor`, is accessible through the

`Window` class's `compositor` instance variable. Each window has its own compositor and clock—the clock drives timing of both the compositor and the window's controllers.

A window and its compositor contain references to each other, and to the window's display surface, through instance variables defined by `Window` and `TwoDCompositor`. Although the links between a window, its compositor, its display surface, and the presenters it contains are sometimes redundant, they allow for greater speed and flexibility in the mechanisms of modeling and presentation.

`TwoDCompositor` defines the instance variables `topPresenter` and `displaySurface`. These instance variables store references to the window, as top presenter for a presentation hierarchy, and to its display surface. The `displaySurface` instance variable is writable, but the preferred mechanism for controlling whether or not the compositor draws to a frame buffer or directly to the display surface is through the instance variable `useOffScreen`, a flag that is defined by `TwoDCompositor`.

`Window` defines the instance variables `compositor` and a `displaySurface`. These instance variables store references to the window's associated compositor and display surface. In addition, each presenter in the presentation hierarchy, for which the window is the top presenter, defines a `compositor` instance variable. In other words, every presenter has its own compositor.

If a window needs to be refreshed, the ScriptX Player calls `refreshRegion` on the window automatically, supplying a `Stencil` object that encompasses the area of the screen to be refreshed. As defined by `Window`, `refreshRegion` calls `refreshRegion` on the window's compositor. For example, if you change the height of an object, that object's setter method will automatically call `notifyChanged`, which causes the object to compute its changed region and pass this region on to the compositor.

During each composite cycle, the compositor maintains an internal record of changes to the presentation space. It adds the window's changed region to this internal structure, which it uses to update the screen during the presentation phase.

If your presentation space includes an isolated or border region, such as the border of a `FullScreenWindow` object or some other large window, that you want to draw to once and then never update, you can reduce memory requirements by creating a window without a compositor. If a window has no compositor, less memory is required because no off-screen frame buffer is created. To create a window without a compositor, specify that its `compositor` keyword be `undefined` at instantiation:

```
new Window boundary:myRect compositor:undefined
```

If the window's compositor is `undefined`, you can override `refreshRegion` on `Window` to perform some drawing or update task (for example, to fill the refresh region with a solid color or pattern). It would be important that no presenter be allowed to move into this region, for if it did, its image would be erased.

Compositor Clock

The compositor uses the top presenter's clock to drive its timing mechanisms. (The top presenter must be an instance of `Window`, and its clock is stored in the window's `clock` instance variable. A compositor does not have a clock of its own.) The compositor is scheduled to update the screen once with every tick of the window's clock, but if it has too much work to do for a given cycle, and consequently does not keep up with the clock, then it takes longer to update the screen, and the frame rate slows. Thus, increasing the speed of the window's clock can speed up the number of frames per second only to the point where the compositor no longer has any slack time.

Temporarily Disabling the Compositor

`TwoDCompositor` defines the instance variable `enabled`, which gives you control over whether the compositor is running or not. When a compositor is created, it is enabled by default. Set `enabled` to `false` to stop the compositor. You might want to disable the compositor if you were constructing an image and wanted to freeze the display until the image was completed.

```
myWindow := new Window name:"ScriptX Forever" boundary:myRect
myWindow.compositor.enabled := false
```

For all presenters in a window, the compositor automatically calls `draw` to render the presenter to the window's display surface. For more control of on-screen drawing, you can disable the compositor and call the `draw` method explicitly on each presenter, supplying the window's display surface as the second argument to the `draw` method. Or, you can perform off-screen drawing by calling draw on a bitmap surface, then transfer the off-screen bitmap to a display surface for viewing.

The compositor has a feature associated with enable/disable that you can take advantage of. After you disable the compositor, you can remove or add objects to the presentation hierarchy, and then re-enable the compositor. Because the compositor was disabled when the objects were changed, the compositor will not know that any changes have occurred, and the screen will not update until an object is next moved or changed. This feature is useful for transitions in collections of presenters. For an example of this technique, refer to the "Transitions" chapter.

The Modeling/Presentation Cycle

As shown in Figure 3-23, a title alternates between modeling and presentation. First the model runs and then the results are applied to presenters that are composited to the output devices. Either phase can take more time than the other, depending on the complexity of modeling and presentation. In short, the two parts of the cycle are as follows:

- The model runs, performing calculations, doing comparisons, tickling the controllers, and running simulations—at some point before presentation begins, it delivers its results by storing values into variables that are accessible to presenters. This modeling generally occurs without any direct presentation.
- The presentation occurs, which causes the modified presenters to draw themselves to the screen using the updated values supplied by the model. This is called the presentation, or composition, part of the cycle. The compositor is the mechanism by which presentation occurs.

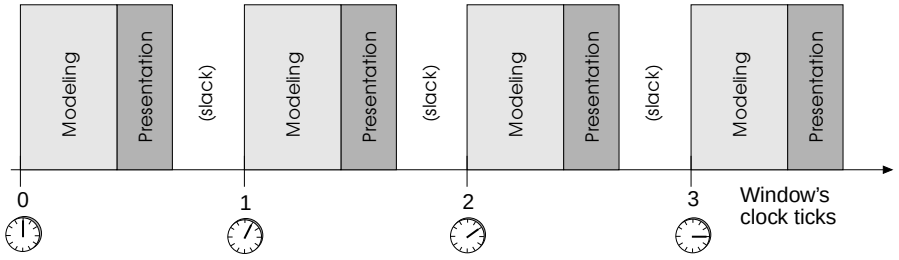


Figure 3-23: The modeling/presentation cycle

Each tick of the window's clock drives a complete modeling/presentation cycle. The modeling and presentation run in the same thread, and alternate, as shown. The modeling runs, and when it's done the presentation runs, followed by slack time at the end of each cycle, which is when other threads run. The slack interval is filled with other operations, like garbage collection. Note that Figure 3-23 is greatly simplified, as it combines all modeling into one block, ignores interleaving of other threads that can occur during modeling, and assumes there is only one compositor running.

Both modeling and presentation are triggered by clock callbacks. Controllers that have a `tickle` method also need to be triggered once each tick of the clock. Each space initially sets up a controller callback to call `tickle` on its controllers. Likewise, each compositor initially sets up a composite callback to draw all of the changed presenters. As a result, at each tick of the window's clock, the controller callback tickles all of the window's "ticklish"controllers (ones that have defined a `tickle` method), and then the compositor callback draws all changed presenters. If a controller callback and a compositor callback are scheduled to happen at the same time, the controller callback is given priority.

In a similar manner, if a model or presentation hierarchy contains a sub-space, that space initially sets up a controller callback on its clock. When the controller callback triggers, it tickles *all* controllers in the space that implement a `tickle` method before compositing begins.

A single thread manages both controller callbacks and the compositor callback for all spaces, including spaces in other models and presentation hierarchies. This thread is labeled the "user priority callbacks" thread. The reason these callbacks are managed in the same thread is to ensure strict serialization, meaning that every scheduled controller is tickled at least once before the compositing begins. The modeling in this thread cannot be interrupted by the compositor—all scheduled `tickle` methods run to completion. This means,

however, that while the simulation is being run, no compositing can take place and, perhaps more significantly, while compositing is taking place, no simulation can be expected.

This serialization eliminates the temporal aberrations caused when a model is in an incomplete state during compositing. Note that modeling that occurs outside of that thread can still have this problem of being interrupted by a compositor. Such is the case for event processing and independent threads.

Skip Compositing If Late

Once every tick of a window's clock, its periodic callback calls `composite` on the 2D compositor, which then calls `draw` on presenters that need to be re-drawn. (A presenter needs to be re-drawn if `notifyChanged` has been called on it or it has been newly exposed by movements in other presenters.) You can specify that a callback not be called if it would call `composite` later than its specified time. In this case, if the callback calls `composite` later than its scheduled time, the callback skips that cycle. Thus, either all presenters get drawn or, if late, none get drawn.

The mechanism for doing this is the `PeriodicCallback` instance variable `skipIfLate`. Setting it to `true` causes the compositor to do nothing during a cycle in which the callback is behind schedule. The compositor does nothing because the callback will wait for the next appropriate time to call `composite` instead of trying to catch up by calling `composite` repeatedly. Note that it is possible for a callback never to catch up.

Access to `skipIfLate` is illustrated in the following code:

```
myWindow.clock.callbacks[1].skipIfLate := true
```

For the compositor callback, the default value for `skipIfLate` is `true`, whereas, the default for periodic callbacks in general is `false`.

Running Separate Threads

In general, you should not manipulate a model from a separate thread. The compositor would not be aware of other threads, and would begin its phase at its normal time, at the next tick, when the modeling in its own thread is complete. The compositor does not wait for operations from another thread to be completed. The result could be compositing a state that should never have existed in the model, where objects appear in unexpected places, like the robot/ball example in "Temporal Aliasing" on page 96, only to greater degree.

The Modeling Phase

The modeling phase of the compositor's cycle is a slice of time during which model objects interact and the controllers manipulate model objects. A title can contain any number of spaces, each of which can have its own controllers; the `Space` class maintains a list of controllers in its `controllers` instance variable. Each space schedules a periodic callback on its clock to "tickle" its

contollers at one tick intervals. This callback iterates through the space's list of controllers, calling `tickle` on each controller, beginning with the first controller in the list. (The order in which the controllers are listed matters only when one object in the space depends on another object in the space.) It is the use of controllers which are activated by the space's clock that makes precise synchronization possible in ScriptX. For scenes where timing and synchronization are important, as in a time-based simulation, all model updates should be managed by controllers.

During the modeling phase, the compositor accumulates a record of changes to presenters, which it uses to redraw the screen during the presentation phase. Each time any `TwoDPresenter` object's boundary, image, or position changes, that presenter calls its `notifyChanged` method to inform the compositor that it should add the changed region to its refresh region. All methods defined in the core classes that cause a change to the appearance of a presenter call `notifyChanged` automatically.

TwoDMultiPresenter Drawing Order

Instances of TwoDMultiPresenter draw in the same order as other presenters:

- Fill the `TwoDMultiPresenter` object's background
- Stroke the `TwoDMultiPresenter` object's outline
- Draw its subpresenters that intersect the changed region (skipping presenters whose `direct` instance variable is set `true`)

Note that since the subpresenters are drawn last, they can draw over the border of the 2D multipresenter.

As with all other presenters that have the instance variable `fill`, the 2D multipresenter saves time if its fill is `undefined`. If no part of its fill is visible because its contents cover it, then leave the fill set to `undefined`.

Synchronizing Clocks

Instances of both `TwoDSpace` and `Window` have clocks that determine at what rates they are composited. When you put a 2D space into a window, if the two clocks are not synchronized, visual abberations can occur, such as temporal aliasing. To prevent aberrations, it's important that the clock for the 2D space be slaved off the window's clock. When you add the 2D space to the window, this master-slave connection of clocks is automatically made for you.

In other words, to ensure that clock rates are relative to the rate of the top clock, it is important for "sub-clocks" to be slaved off some top clock. This synchronizes the clocks at all rates. The following statement makes the window's clock be the master of the space's clock:

```
mySpace.clock.masterClock := myWindow.clock
```

The mechanism for auto-connecting clocks is found in the `presentedBy` instance variable. Any presenter that is added to or removed from another presenter has its `presentedBy` instance variable changed; the `presentedBySetter` method automatically calls `adjustClockMaster` to connect the new clock as a slave to the next clock above it.

Note that if you have done any customization to the timing hierarchy, that customization will not be overridden. That is, if you have changed the clock of the 2D space to be slaved off another clock besides its default, that connection will not be disturbed.

Not all presenters have clocks; however, all members of the `Space` family of classes have clocks, which includes windows, 2D spaces, and group spaces, as indicated by the presence of a `clock` instance variable.

When a time-driven model does not live in a window, the model should have its clock connected to the clock hierarchy of the window displaying it. This connection would not be made automatically, since the model is not in a presentation hierarchy, and hence has no `presentedBy` instance variable. You need to make the master-slave connection yourself, as is done in the following code:

```
modelClock.masterClock := myWindow.clock
```

However, there may be titles where you want to stop the window's clock but continue running the model space; in this case, the model space would have its own top clock, rather than a clock which is slaved off the window's clock.

The next section describes how temporal aliasing occurs and the kinds of effects it causes. For related information, see the earlier section "Clocks and Timing" on page 52.

Temporal Aliasing

If a title contains one or more objects driven by clocks, an effect called *temporal aliasing* can occur. Temporal aliasing is an aberration of the model due to timing. Temporal aliasing commonly shows up as a jerkiness in the movement of animations and other time-based effects. In special cases it can cause more obvious, but still momentary, aberrations.

For example, suppose you have a top space containing a bouncing ball and a sub-space that is designed to hold just a robot. The robot has legs that enable it to move around, and an arm that can swing at the ball. It also has a collision-detection controller that enables it to determine when the ball is near. Temporal aliasing could show up as a disconnect between the robot and the ball, where the robot swings at the ball and the ball appears momentarily to move *past* the robot's arm before it bounces off the arm.

Temporal aliasing occurs when the ratio of the scales of two clocks in a title is not an integer. From the perspective of the compositor's clock (the window's clock), one of the following must be true to avoid temporal aliasing:

- If the compositing clock's scale is greater than the other clock's scale,

compositing clock's scale / clock scale = integer

- If the compositing clock's scale is smaller than the other clock's scale,

  clock scale / compositing clock's scale = integer

Thus, a compositor's clock scale of 1, 2, 4, or 8 is compatible with a clock scale of 2 and 4, but a scale of 3, 5, or 7 for the compositor's clock causes temporal aliasing.

To ensure that clock rates remain at these ratios as you change the rate of the top clock, it is important for "sub-clocks" to be slaved off the window's clock. This synchronizes the clocks at all rates.

To demonstrate temporal aliasing, Figure 3-24 shows the timing diagram for the robot/ball example, described earlier, with the ball set to a scale of 3, the robot set to a scale of 2, and the robot space's clock slaved off the ball space's clock. The shaded regions are the modeling intervals where the robot and ball controllers run. Temporal aliasing is caused by two effects:

1. The ball goes through *two* modeling cycles between the first and second presentation phases.
2. The ball's modeling phase shifts in time relative to its compositing phase. At its third presentation phase, the ball is being displayed in a relatively old state (.33 ticks old). If the ball were moving fast, it could appear to pass the robot's arm.

Note that the robot controller's callback and the compositor's own callback are both scheduled to begin at 1 tick. Because controllers have priority over compositors, the robot's callback occurs first, followed by the compositing phase.

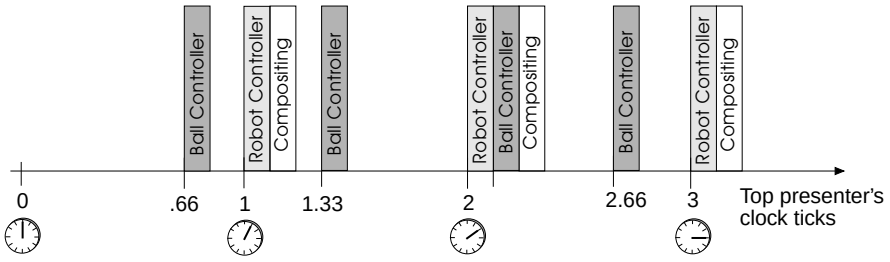


Figure 3-24: Temporal aliasing due to non-integer ratios of clock scales

Note that Figure 3-24 shows the timing for a single display surface with its presentation hierarchy. If there were another display surface, then its compositor callback would be scheduled at each tick of its window's clock. Independent compositing blocks would appear in this figure at those times.

Spaces and Presenters Examples

This section gives examples that use classes in the Spaces and Presenters component.

A Simple Notice Window with a Pushbutton

A `Window` object of type `@notice` represents a modal window—that is, it does not allow access to any other windows until it is closed or hidden. Every notice window requires some mechanism that allows it to close. This example creates a new class called `ClosableNoticeWindow` that includes a pushbutton that allows the user to close the window, as shown in Figure 3-25. When a user clicks on the close button, the window is hidden and then automatically garbage collected (unless some reference to it still exists in the program).

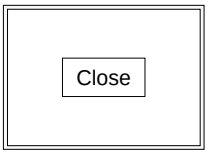


Figure 3-25: An instance of ClosableNoticeWindow with a close pushbutton.

```
class ClosableNoticeWindow (Window) end

method init self {class ClosableNoticeWindow} #rest args ->
    apply nextMethod self type:@notice args -- calls init on
     --superclasses

method afterInit self {class ClosableNoticeWindow} #rest args -> (
    apply nextMethod self args -- calls afterInit on superclasses

    -- create an actuator controller for any pushbuttons in the window
    new ActuatorController space:self wholespace:true

    -- create the pushbutton
    local closeButton := new Pushbutton
    closeButton.x := 75
    closeButton.y := 90
    closeButton.stroke := blackBrush

    -- define the released presenter
    local buttonText := new TextPresenter \
        boundary:(new Rect x2:50 y2:20) target:"Close"
    setDefaultAttr buttonText @alignment @center
    closeButton.releasedPresenter := buttonText

    -- define the pressed presenter
    closeButton.pressedPresenter := new TwoDShape \
        target:(new Rect x2:50 y2:20) fill:blackBrush

    -- define the window to hide when mouse button is released
    closeButton.activateAction := (notUsed button ->
        hide button.presentedBy)

    -- append the button to the window and show the window
    append self closeButton
    show self
)

new ClosableNoticeWindow centered:true boundary:(new Rect x2:200
y2:200)
```

C H A P T E R

Controllers

4

The Controllers component defines classes that can monitor and manipulate model objects in a space. Controllers are themselves non-visible, and can operate on either visible or non-visible objects.

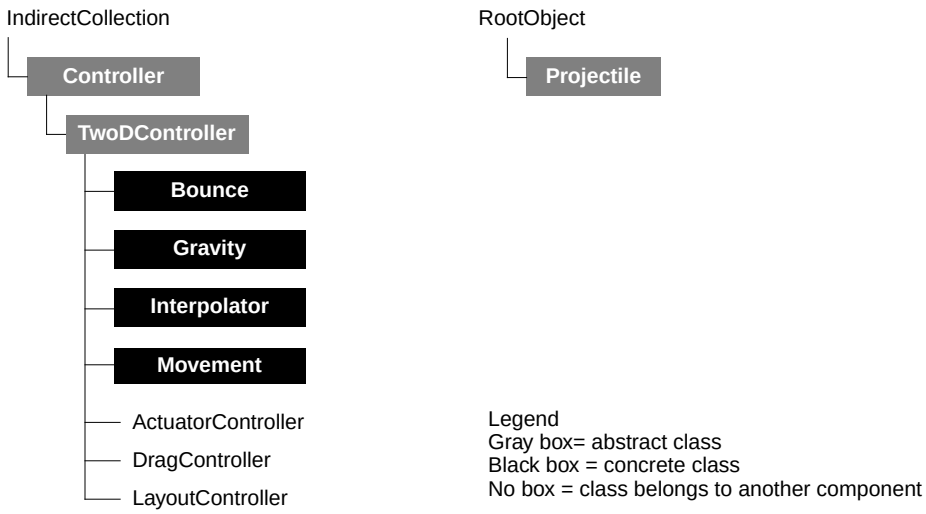
The concrete controllers described in this chapter (some of which are loadable) are `Bounce`, `Gravity`, `Interpolator` and `Movement`. They directly control the state of model objects in a space without user interaction. For example, you could attach `Bounce` and `Movement` controllers to a 2D space, then add into the space a 2D presenter that inherits from `Projectile`; this object would be a target of those controllers, and would bounce off the edges of the space.

Controllers can also be driven by user interaction. The User Interface chapter describes a set of controllers that respond to input events, such as mouse movements and mouse clicks. These controllers receive and process mouse events, changing the state of the presenters they control based on user input.

Controllers use features defined in a number of other ScriptX components. The Spaces and Presenters component provides the fundamental `Space` family of classes; a space is the only environment in which a controller can operate. The Collections component provides a framework that allows a controller to hold multiple objects, and defines the protocol for adding, setting, removing, and accessing model objects.

Classes and Inheritance

The class inheritance hierarchy of the Controllers component is shown in the following figure.



The following classes form the Controllers component. Other controllers are defined as part of the User Interface component. In this list, indentation indicates inheritance:

`Controller` – the root abstract class for controllers, `Controller` manipulates some or all of the objects in a space.

- `TwoDController` – similar to `Controller` but also ensures that objects in the controller's collection are in the same order as they appear in the 2D space, guaranteeing that objects are controlled in their front-to-back order.
  - `Bounce` – causes one or more presenters to bounce off the edges of the container.
  - `Gravity` – causes one or more presenters to accelerate in a specified direction.
  - `Interpolator` – moves one or more presenters to a specified point in a specified amount of time, along a straight or curved path.
  - `Movement` – moves one or more projectiles a distance calculated from the object's velocity vector.

`Projectile` – an abstract class that contains instance variables and methods that need to be controlled by `Bounce`, `Gravity`, or `Movement`.

Conceptual Overview

Most any title can be best understood by separating its constituent objects into three basic functions, or roles: model, presenter and controller. The `Space` family of classes provides the foundation for models. The `Presenter` family of classes provides the foundation for presentations, or views. The `Controller` family of classes provides the ability to control model objects in time and space. These three roles are described in greater detail in the "Conceptual Overview" section of the "Spaces and Presenters" chapter.

Controllers can both monitor and manipulate objects in a space. For example, a `Gravity` controller might look to see how fast an object is moving before determining how far it should move that object.

Controllers often define "natural laws" governing members of the space. In physical simulations, controllers can define the force on objects due to gravity. Controllers can watch the distance between objects and perform some operation based on proximity, such as collision detection or magnetic repulsion.

Developers can create their own controllers. The controllers described in this chapter are sample controllers included basically for demonstration purposes, and some of them ship with the ScriptX Language and Class Library as loadable classes rather than in the core classes:

- `Bounce` – causes presenters to bounce off the edges of their container.
- `Gravity` – causes presenters to accelerate in a specified direction.
- `Interpolator` – moves presenters to a specified point in a specified amount of time, along a straight or curved path.

- `Movement` – moves projectiles a distance calculated from the object's velocity vector.

Controller Class

The abstract `Controller` class provides the set of instance variables and instance methods common to all controllers. The instance methods are empty in `Controller`; concrete subclasses provide individual implementations of these methods.

One key protocol that controllers can implement is the Ticklish protocol. A controller that implements an instance method for `tickle` is said to be "ticklish." This method is used to specify a repeated action that a controller performs on its target objects each time the space's clock ticks.

A `tickle` method can perform both monitoring and control activities. For example, in the `Bounce` class, a ScriptX loadable class, `tickle` checks to see if the object has hit a side of the container. If so, it changes the object's velocity so that it moves in a new direction.

The other controller methods, `objectAdded`, `objectRemoved`, and `isAppropriateObject`, along with the `protocols` instance variable, operate similarly to their counterparts in the `Space` class. These methods allow you to determine which objects are added to a controller, and to perform any specified action when an object is added to a controller or removed from it.

Whenever an object is added to a controller, the object is checked to ensure that its protocols match those expected by the controller. This means that each object added must be an instance of or inherit from the classes listed in the controller's `protocols` instance variable. If the object does not have the correct protocol, then the object is not added.

TwoDController Class

The `TwoDController` class inherits from `Controller` and has the same features of that class but with one difference: it ensures that objects being controlled are ordered in the controller's collection in the same order as they appear in the 2D space. Since this order determines the z-order of objects, it guarantees that objects are controlled in their front-to-back order.

When to Use Controllers

Since there are many ways to control objects in a space, the question arises—when should you use a controller? You should use a controller when there is a single outside force affecting more than one object. For example, you would use an `ActuatorController` when you have several pushbuttons in a space, since you want them all to operate identically. It doesn't make much sense to use a controller unless there is a one-to-many relationship of controller to objects.

Imagine a space where a flock of birds is flying. If you used a controller to control the wing motion of a bird, then all birds would have identical wing motion. However, if you wanted each bird to have unique wing motion, then a single controller would not provide the behavior you would want for all birds. Nor would it make sense to create a separate controller for each bird. In this case, it would be more appropriate for the class of birds to define the general motion and have each bird individualize that motion.

It would, however, be appropriate to create a controller that represents the wind, which blows all birds along relative to the ground. The wind is a force outside the birds that has an impact on each bird it encounters.

How Controllers Work

Controllers operate on objects in spaces. Spaces and controllers work together closely—to understand controllers, it is important to understand how they work with spaces.

Attaching a Controller to a Space

Controllers require a space on which to operate—they cannot operate on objects outside a space. Controllers rely on the space's clock for their timing.

Each space keeps a list of controllers that manipulate objects in the space. To attach a controller to a space, you assign the space to the controller's `space` instance variable. The space in turn automatically adds the controller to its read-only `controllers` instance variable. The `space` instance variable determines which space the controller is controlling, and ensures that the controller manipulates only one space at a time.

Here's an example of creating a space and attaching the space to the controller using the `space:` keyword:

```
myWindow := new Window boundary:(new Rect x2:640 y2:440)
myGravity := new Gravity space:myWindow
```

Thereafter, when an object is added to the space, the space notifies each controller, and if the controller's `wholeSpace` is `true`, the object is also added to the controller.

If a controller implements the Ticklish protocol, the controller's space calls `tickle` on that controller each time the space's clock ticks, allowing the controller to perform a repeated action. With every tick of the space's clock, the space iterates through each of its controllers that implement the Ticklish protocol, allowing them to perform some periodic action. A `tickle` method must run to completion without blocking within in a fraction of a tick, since all controllers run once every tick. For information on blocking, see the section "Blocking" on page 591 of Chapter 22, "Threads."

Note that controllers themselves are not added to the space in the way model objects are—controllers remain outside the space but are attached to it. Controllers must eventually have an effect on presenter objects that they control for their effects to be visible.

What is a Controller?

Each controller is a collection that holds the objects it controls. These objects are its so-called target objects. A controller is a collection by virtue of inheriting from `IndirectCollection`. Although controllers provide the Collection protocol, the `Controller` class does not define which `Collection` class is used as a target collection. Each concrete subclass of `Controller` can define its own target collection to store its target objects in the most appropriate way. Within the core classes, all indirect collections currently use an array as a their target collection, and are optimized to work with arrays.

The controller in Figure 4-1 has the two members `myBall` and `myOval`. These are the objects that are controlled by the controller. The controller's `space` instance variable specifies the particular space it is attached to—in this case, the space `mySpace`.

The controller can be disabled by either setting its `enabled` instance variable to `false` or by removing the controller from the space. In addition, objects can be added to or removed from a controller, which allows behaviors to be added to or subtracted from an object over time.

Consider a space that contains four objects, as shown in Figure 4-1. Two of the objects are projectiles. Notice that the controller `myBounce` has `wholeSpace` set to `true`, so all appropriate objects in `mySpace` are added to `myBounce`. This means only `myBall` and `myOval` are successfully added to `myBounce`, since they are projectiles, and hence satisfy the protocol. Because `myBall` and `myOval` are members of `myBounce`, they are being controlled in `mySpace`.

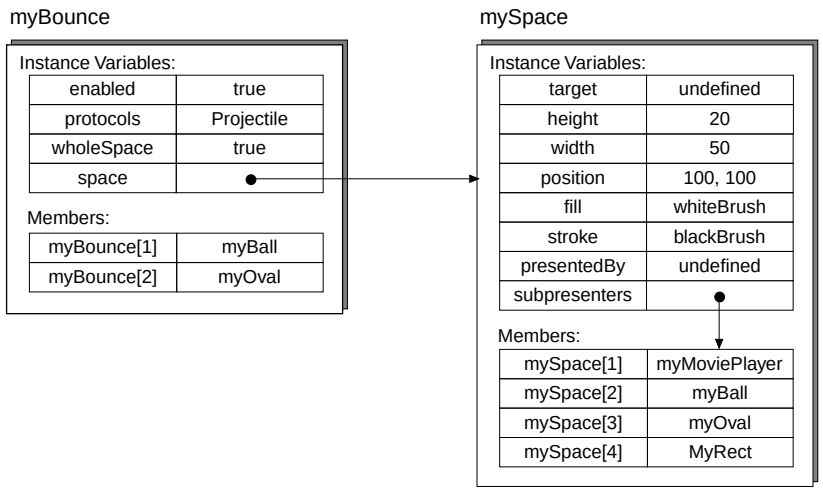


Figure 4-1: The members of myBounce are controlled in mySpace.

Note – The `Bounce` class has no effect unless mixed in with the `Movement` class. Then an object that inherits from the `Projectile` class can bounce off the sides of its container.

Defining Your Own Controller

If none of the controllers in the core classes performs the task you need, you can define a subclass of `Controller` for that task. In any subclass of `Controller`, you typically specialize the instance methods `isAppropriateObject`, `objectAdded`, and `objectRemoved`. If the controller implements the Ticklish protocol, it should also specialize `tickle`. Descriptions of these methods follow.

The Ticklish Protocol

Controllers that implement the Ticklish protocol define a method for `tickle`. A controller class can specialize `tickle` to define an action that a controller performs at each tick of the space's clock. The Ticklish protocol is implemented differently in every concrete subclass of `Controller` that defines a method for `tickle`. A controller that implements this protocol is said to be "ticklish."

A `tickle` method can both monitor and control target objects. For example, in the `Bounce` class (a ScriptX loadable class), the `tickle` method monitors whether the target object has hit a side of the container. If so, it sets a new value for `velocity`, taking into account the value of `elasticity` (both instance variables of `Projectile`). The `tickle` method for the `Movement` class monitors the value of a projectile's `velocity` instance variable, calculates the distance the projectile should have moved with that velocity, and moves the projectile that distance.

Imagine a space that has a ball and two controllers, a `Bounce` controller and a `Movement` controller. The ball is a 2D shape with the `Projectile` class mixed in, which allows it to be controlled by `Bounce` and `Movement`. The `Bounce` class defines a `tickle` method that calculates the velocity that the ball should have, based on whether it has hit the edge of the space—it updates its `velocity` instance variable, but does not move the object. `Movement` defines a method for `tickle` that moves the ball to a point determined by multiplying its `velocity` times the length of a tick.

Figure 4-2 shows a greatly simplified flowchart of the operations that occur: the `tickle` method for `Bounce` is called on all of the bounce target objects, then the `tickle` method for `Movement` is called on all of the movement target objects, and then the presentation of the objects occurs. Under control of the 2D compositor, this process draws the changes to the screen, making them visible to the user. This process is also called compositing. This flowchart shows only the controllers in relation to the compositing—it ignores everything else that is going on in the system.

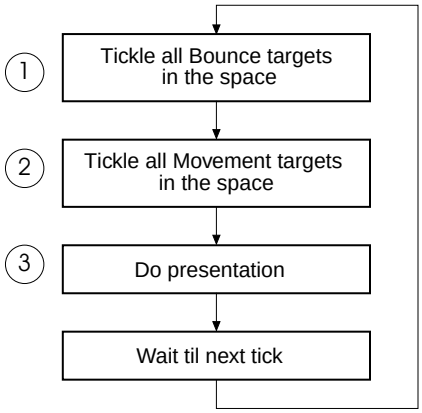


Figure 4-2: The tickle method is called once on each ticklish controller with every tick.

Figure 4-3 demonstrates the same operations as the previous figure, but along a time line. It shows the frequency at which the two `tickle` methods are invoked—once with every tick of the space's clock, just prior to compositing. Presentation and compositing are described in Chapter 3, "Spaces and Presenters."

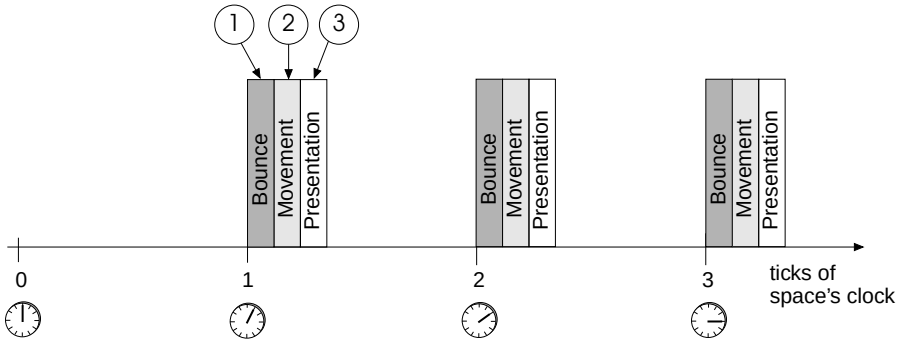


Figure 4-3: Each controller that is ticklish runs a repeated action once each with tick.

The space's clock determines how often objects in that space are tickled. Repeated actions are invoked through callbacks on the space's clock. The space schedules a callback to run at every tick of its clock. When this callback runs, it sequentially calls `tickle` on every controller that implements a `tickle` method, beginning with the first controller in the list. In this particular case, the order in which the controllers are listed is important because `Movement` depends on the velocity set by `Bounce`, which means that `Bounce` must operate before `Movement`.

When a title contains multiple spaces, each space schedules its controllers to run at 1 tick intervals according to its own clock.

To make the ball stop bouncing, you could stop the space's clock, you could disable the `Bounce` and `Movement` controllers, or you could remove the ball from the `Bounce` and `Movement` controllers. The first two choices would stop all objects in the space. When there are two interdependent controllers

affecting an object, they both need to be turned off or on at the same time; otherwise, when they are turned on, the object could jump off the screen. You can define a function that turns them both on or off.

You can implement `tickle` to manipulate the target objects in several different ways. It can watch for changes in the target object's state, it can wait for events that are related to the model objects, or it can operate periodically.

For more information on how controllers interact with the compositor, see "The Modeling/Presentation Cycle" on page 92 of Chapter 3, "Spaces and Presenters."

Specifying an Object To Be Controlled

A controller can be attached to only one space at a time, and can control only objects in that space. The controller is a collection that holds the objects to be controlled.

Once a controller is attached to a space, it can either control all or some of the objects in the space, depending on the value of the controller's `wholeSpace` instance variable:

- When `wholeSpace` is set to `true`, the controller automatically looks at all the objects in the space, and adds objects with the appropriate protocol to the controller.
- When `wholeSpace` is set to `false`, the controller is emptied of its target objects. Objects in the space are not automatically added to the controller. You must explicitly add objects to the controller (using `Collection` methods) if you want them to be controlled.

In either case, any time an object is added to a controller, it is tested with `isAppropriateObject` before actually being added.

Adding an object to a controller is quite similar to adding it to a space; since they both inherit from `IndirectCollection`, they both use the same methods.

You add objects to the controller using the collection methods defined for its `targetCollection` class—methods such as `prepend`, `append`, `addFirst`, and so on. These collection methods are all specialized in `IndirectCollection` to call two additional methods on the controller: `isAppropriateObject` and `objectAdded`, as shown in Figure 4-4. Using `prepend` as an example, this flowchart shows four steps:

1. Call `prepend` on the controller to add an object.
2. The `prepend` method is specialized in `IndirectCollection` to call the `isAppropriateObject` method, as implemented in the controller, to check if the candidate object conforms to the `protocols`. If `isAppropriateObject` returns `true`, then the procedure continues.
3. The `prepend` method is called, as implemented in the `targetCollection` class. This actually adds the object to the controller, causing the object to come into the controller's realm of control.

4. The `objectAdded` method is called.

The `objectAdded` method is automatically called any time an object is added to a controller. You can specialize `objectAdded` in any controller class you define to perform any action you want to occur every time an object is added.

A controller can use `isAppropriateObject` to reject objects and use `objectAdded` to modify objects that it wants to include. This allows the controller to impose constraints on objects that it can control. For instance, the gravity controller might require objects to have a `mass` instance variable—it could either reject objects that have no mass, or send them to a more appropriate space.

The controller's `objectRemoved` method, also shown in Figure 4-4, is automatically called whenever an object is removed from the controller. You can also specialize it to perform any action you want to occur every time an object is removed from the controller.

The object that has been added to or removed from the controller is automatically passed as the second argument to the `objectAdded` and `objectRemoved` methods.

The `isAppropriateObject` method works with the `protocols` instance variable as described in the next section.

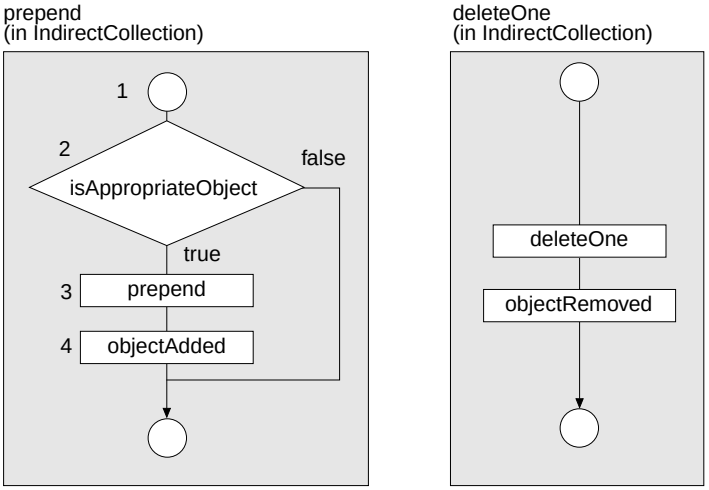


Figure 4-4: Methods prepend and deleteOne are specialized in IndirectCollection.

Protocols

A controller restricts the objects that may be added to it by using protocols the same way a space does. The `Controller` class defines a `protocols` instance variable, which is a list of classes you specify. If the object being added has all the protocol classes among its superclasses, it is added; otherwise, it is rejected.

For example, if the controller were controlling a physical simulation you might allow in only objects that were projectiles—you would do this by adding the `Projectile` class to the protocols list. If you wanted to further restrict the

members to be projectiles that are also 2D presenters, you would add `TwoDPresenter` to the protocols list. Refer to the section “Protocols” in the “Spaces and Presenters” chapter for more details.

User Interface Controllers

The User Interface component also defines several controllers. Unlike the controllers defined by this component, which manipulate a target object automatically with each tick of the clock, user interface controllers respond to user input. These controllers receive user-generated events and process them, modifying the state of their target objects. For example, a button is associated with an actuator controller. This controller receives and processes mouse events, so that the button knows when it is pressed, released, or disabled. For more information, see the “User Interface” chapter.

Contention Among Controllers

Imagine you define a `Ball` class that mixes `Dragger` and `Projectile`, and create an instance of it. You put the ball in a space that is controlled by `Bounce`, `Gravity`, `Movement` and `DragController`. You expect the ball to bounce around in the space, and expect to be able to grab it with the mouse, drag it around, and let go.

You will find that you cannot drag the ball with the mouse, because the `Movement` controller is not designed to release control to the `DragController`.

To make these classes work together, in the `grabAction` method defined in `Dragger`, remove the ball from the `Movement` and `Gravity` controllers. Then as you mouse-down on the ball and drag it around, the `DragController` will be in control. In the `dropAction` method, put the ball back under the control of the `Movement` and `Gravity` controllers.

Controller Example

The following is a complete, working script that demonstrates a few simple controllers in use.

The Bouncing Ball

The following script demonstrates how controllers work. It demonstrates a bouncing ball in a space, using three controllers: `Gravity`, `Bounce`, `Movement`, and the required mixin class `Projectile`, which is required by controllers. (`Projectile` is not itself a controller.)

Once the space is set up with the controllers and the ball is added, the `Gravity` controller makes the ball accelerate downward. The `Bounce` class calculates its new velocity at each collision with the wall of the space. The `Movement` class actually moves the ball to its new position.

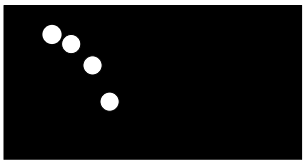


Figure 4-5: The ball falls due to gravity, then bounces off the space's edges.

The Projectile

The `Ball` class mixes `TwoDShape` and `Projectile` to produce a presenter that can have a velocity. The `Projectile` class has instance variables for `elasticity` and `velocity` that `Ball` uses. When `elasticity` is set to 0, the projectile will lose all energy in a collision; when it is set to 1, the projectile is perfectly elastic and will lose no energy. A value greater than 1 causes the projectile to gain energy on collision. The `restart` method allows you to restart the ball, once it has run out of energy and stopped bouncing.

```
-- Make a projectile by defining a class that
-- inherits from both TwoDShape and Projectile
class Ball (TwoDShape, Projectile)
end

-- Method for initializing an instance of Ball
method init self {class Ball} #rest args ->
(
    apply nextMethod self args
    self.x := 50
    self.y := 50
    self.elasticity := 1
    self.velocity := new Point x:-8.0 y:4.0
)

-- Method to restart the ball bouncing
method restart self {class Ball} ->
(
    self.x := 50
    self.y := 50
    self.velocity := new Point x:-8.0 y:4.0
)
```

Set Up the Space and Controllers

This part of the script creates a window, which is a subclass of `TwoDSpace`, and then it creates the controllers `Gravity`, `Bounce`, and `Movement`. For each controller, it sets `wholeSpace` to `true` so that any object added to the space will automatically be added to the controllers.

```
-- Set up the window
global myWindow := new Window boundary:(new Rect x2:400 y2:250)
fill:blackBrush
    myWindow.x := 40
    myWindow.y := 40
```

```
    show myWindow

-- Create the controllers
global myGravity := new Gravity space:myWindow
myGravity.wholeSpace := true

global myBounce := new Bounce space:myWindow
myBounce.wholeSpace := true

global myMovement := new Movement space:myWindow
myMovement.wholeSpace := true
```

Create and Add the Ball

It's important to add the ball to the space after all the controllers are set up, so that all controllers can start controlling the ball at the same instant. This script creates the ball, then adds it to the space

```
-- Create an instance of Ball and add it to the space
global myBall := new Ball target:(new Oval x2:20 y2:20) fill:whiteBrush
prepend myWindow myBall
```

At this point you should see the ball bouncing off the walls of the window. Since the `elasticity` instance variable is set to 1, the ball keeps bouncing without losing energy. Using the code below, you can reset `elasticity` to a value less than 1 to see it slow down, and then after it has slowed down, start it bouncing again:

```
myBall.elasticity := 0.8 -- set so that the ball will lose energy
-- wait several sedonds and then restart it
restart myBall
```

C H A P T E R

User Interface

5

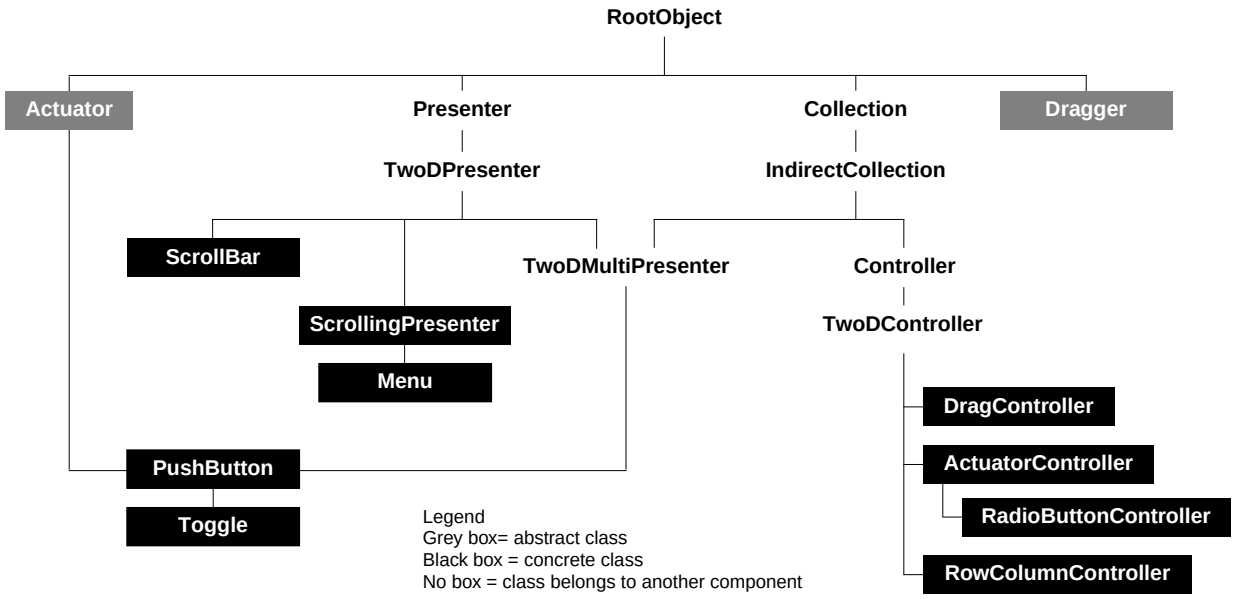
The User Interface component includes a set of classes, mostly presenters and controllers, that can be used to create user interface elements. These classes form all the standard graphical interface controls, including menus, push buttons, and scroll bars, that embody the desktop metaphor.

This component builds directly on two other components covered in earlier chapters—Spaces and Presenters, and Controllers. User interface classes also interact with clocks, and they receive mouse input through the event system. Much of the underlying complexity of the ScriptX presentation model and event system are hidden in the user interface classes, yet these classes are flexible and open, easy to specialize.

The User Interface classes can be put to use in a ScriptX title with very little programming. Although they define many variables and methods that are visible to the scripter, most are not meant to be called directly from a script. This approach allows an author to selectively override any aspect of their behavior. An author can use classes that the User Interface component defines as the basis for creating new kinds of user controls.

Classes and Inheritance

The class inheritance for the User Interface component is shown in the following figure.



The following classes form the User Interface component. In this list, indentation indicates inheritance. Most of the classes in this component inherit from either `TwoDPresenter` or `Controller`. Exceptions are the `Actuator` and `Dragger` classes.

`Actuator` – an abstract class with the basic protocol for pressing, releasing and testing the state of push buttons.

`PushButton` – an actuator that presents a button that the user can press to trigger an action.

`Toggle` – an actuator that implements toggles and radio buttons.

`ScrollBar` – a presenter that displays a bar that a user can manipulate (scroll) to set a numeric value, and to trigger an action each time that value changes.

`ScrollingPresenter` – a presenter which allows a user to view a larger object, scrolling to view all its parts.

`Menu` – a pull-down or pop-up presenter that presents a list of push buttons arranged in a row or column layout, allowing the user to make a selection.

`DragController` – a controller that can be attached to model objects that inherit from `Dragger` and `TwoDPresenter`, to map mouse events to changes in the location of the object.

`ActuatorController` – a controller for actuators that maps mouse events to release and press calls on actuators.

`RadioButtonController` – an actuator controller that controls actuators to ensure that only one of a group is selected at a time.

`RowColumnController` – a layout controller that arranges objects attached to a space in rows and columns.

`Dragger` – an abstract, mixin class with a set of fully implemented methods to make objects dragable.

Note – See the Text and Fonts component for the `TextEdit` class, an additional user interface class. Just as the classes in the User Interface component are presenters that automatically receive and process mouse events, the `TextEdit` class is a presenter that receives and processes keyboard events.

Conceptual Overview

Think of the objects in the User Interface component as user interface building blocks that are constructed from simple parts — pieces that are themselves objects. These “object parts” include presenters, spaces, and controllers. User interface objects can share parts for a common look or feel. Each of these parts can be modified or dynamically changed. In some cases, these parts are supplied automatically; in other cases, they must be supplied by the script that creates a new object.

In general, ScriptX does not give user interface objects any intrinsic appearance. In most cases, their appearance is defined by the presenters they manage. For example, when a button is pressed, the `PushButton` class swaps its `releasedPresenter` for its `pressedPresenter`, and when the button is disabled, it displays its `disabledPresenter`. These three presenters, specified as instance variables for any `PushButton` object, give that button its appearance.

Each object part separates out one aspect of a user interface object's behavior. Change the presenters, and you change the appearance. Change the controller, and you change the behavior or layout of model objects. User Interface objects have a modular, object-based design that can be used to make systematic changes in entire groups or classes of objects. For example, a multimedia title could create a new kind of menu by redesigning the layout controller that controls target presenters in the `Menu` class.

User interface objects are both simple and powerful. A `ScrollBar` object manages a set of stencils, a clock, a controller, and a set of interests in mouse events. Collectively, these objects give an author access to hundreds of instance variables and methods, providing "hooks" so that the scrolling metaphor can be extended in an infinite number of ways. ScriptX has the power to create virtually any appearance or behavior in a user interface. Yet an author can create and manage scroll bars with very little programming.

How User Interface Objects Work

For every class in the User Interface component (except `ScrollBar`), there is an essential relationship between presenters and controllers. `Controller` classes, such as `ActuatorController`, manage presenters, such as `PushButton` and `Menu`. `ScrollBar`, a special case, has characteristics of both a presenter and a controller.

Every class in this component is an instance of `TwoDPresenter` or `Controller`, with two exceptions. The two exceptions, `Dragger` and `Actuator`, are designed to be "mixed in" with a `TwoDPresenter` class. In effect, they are building blocks to add significant features to presenters, to create new kinds of presenters. Indeed, what they really do is make it possible to create a presenter that is controlled by a `DragController` or `ActuatorController` object.

Presenters and User Interface Objects

ScriptX follows the paradigm, established with SmallTalk, of separating a model object from its presentation and control. A presenter is a view of some model object or data that is separate from that data. For more information, see the section entitled "Model-Presenter-Controller System" on page 45 in the Spaces and Presenters chapter.

The User Interface component contains several classes of presenters that display other presenters. For example, a `PushButton` object displays one of three presenters, stored in the instance variables `pressedPresenter`,

releasedPresenter, and disabledPresenter. The Toggle class specializes PushButton by adding two more presenters: the toggledOnPresenter and the toggledOffPresenter.

A button's presenters are layered. More than one presenter can be displayed at a time. Think of these presenters as being superimposed, one on top of another. The compositor draws the background first, then the middle ground (for a toggle), and finally the foreground.

Figure 5-1 depicts this layering effect on a Toggle object. It shows how a toggle that is currently toggled on, but disabled, is composited to a window. First, the compositor draws the released presenter, which forms a backdrop. Next, the toggled-on presenter is superimposed on top. Finally, the disabled presenter, which forms the foreground, is drawn over the others.

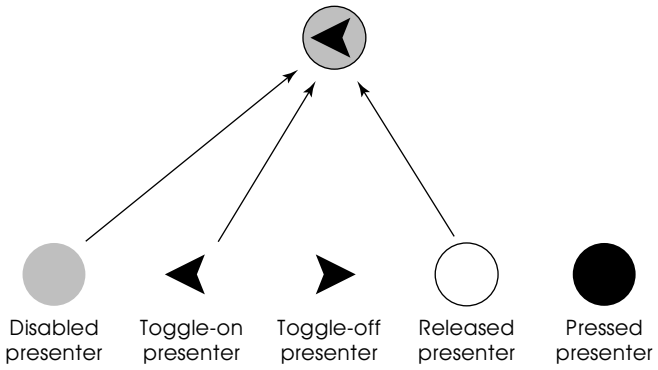
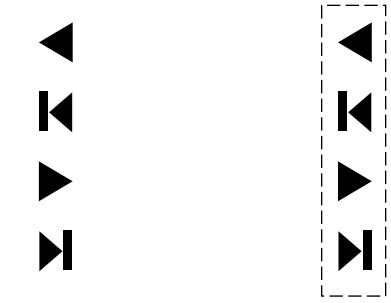


Figure 5-1: When a Toggle object is composited, several presenters are superimposed to form a single view.

Objects in the User Interface component know how to size themselves correctly. Some of these objects calculate their size by calculating the union of the boundaries of their subpresenters. For example, the boundary of a Toggle object is the smallest rectangle that can enclose all of its five presenters.

Similarly, a Menu object, together with the layout controller that is associated with it (a RowColumnController object), automatically lays out the actuators that are its subpresenters and sizes the menu's space correctly to enclose them.



A `ScrollBar` object does the opposite—it lays out and draws all of its stencils according to its own dimensions. When a scroll bar is attached to a scrolling presenter, the scrolling presenter sizes the scroll bar correctly to span the sides of its clipping stencil. The scroll bar, in turn, sizes and lays out its own stencils.

Implications of the Presentation Hierarchy

A single presenter cannot be presented by more than one presenter. Since presenters are in a strict presentation hierarchy, a given presenter can only be in one position in the tree. Two presenters cannot share the same subpresenters.

If several user interface objects share a common look, they do so by defining separate presenters that individually target the same model objects. Another approach is to create a subclass that overrides `draw` in the parent class.

Suppose that you want to create two instances of `Toggle` with the same appearance in a space. In Figure 5-2, each toggle presents an ✘ graphic with its toggled-off presenter, a ✔ graphic with its toggled-on presenter, and a black rectangle as its released presenter. The pressed and disabled presenters are undefined.

In the diagram, one instance of `Toggle` is toggled on, and the other is toggled off. Each `Toggle` object defines three separate and unique presenters, all of which are instances of `TwoDShape`. Different instances of `TwoDShape` can target the same `BitMap` object, allowing the two `Toggle` objects to share the same look. Note that the toggle-on and toggle-off presenters are both layered on top of the released presenter. In Figure 5-2, the two bitmaps represent data, while the presenters that target them represent views of that data.

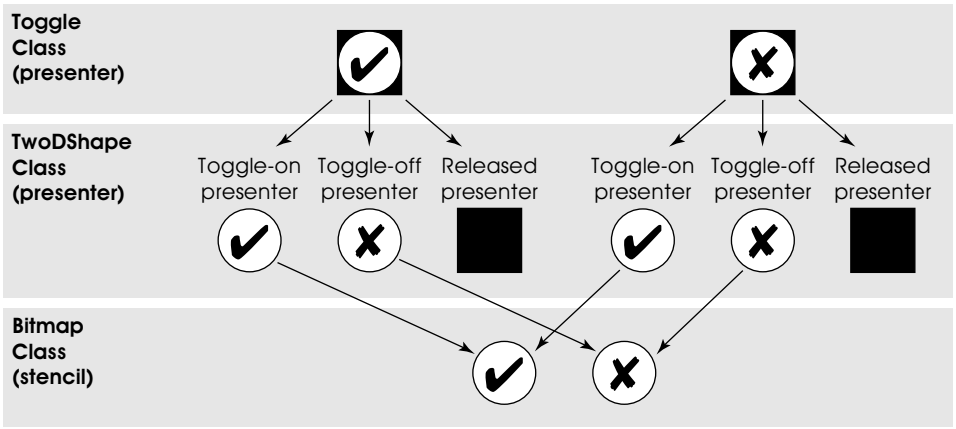


Figure 5-2: User interface objects that present a common appearance should define separate presenters that target the same model objects.

A given presenter can only appear once in the presentation hierarchy. While two separate presenters cannot share or reuse a presenter, a given presenter can reuse the same presenter in certain circumstances. For a toggle, *on* and *off* represent two mutually exclusive states. For this reason, an instance of `Toggle` can define the same presenter as its `toggledOnPresenter` and `toggledOffPresenter`. On and off are mutually exclusive states, so the toggled-on and toggled-off presenters are never presented at the same time. However, the same toggle cannot reuse its `toggledOnPresenter` as its `pressedPresenter`. For more information on sharing or reusing presenters, see the "Spaces and Presenters" chapter.

Layering of Presenters

Push buttons and toggle buttons inherit their ability to display more than one presenter at a time from `TwoDMultiPresenter`, a class which incorporates the Collection protocol through the class `IndirectCollection`. These classes use z order as a layering device.

When an object is added to or removed from the subpresenters list, its z order determines its placement within the list. The `PushButton` and `Toggle` classes impose a predetermined z order on their presenters, which is determined at compile time, but a developer can override this ordering at runtime. For example, suppose that a developer creates a new class of actuators that inherit from `Toggle` and share a common bitmap as a background theme. If these actuators add a new presenter, in its own layer, that presenter must set a lower value of `z` than other presenters in the subpresenter list if it is to be displayed behind the others. For more information on ordering subpresenters, see the discussion of z-ordering that begins on page 85 of the "Spaces and Presenters" chapter.

Controllers and User Interface Objects

Each 2D presenter class in this component is associated with one or more controllers. In the User Interface component, controllers manage presenters. For example, an actuator controller calls the `press`, `release`, `activate`, and `multiActivate` methods on any actuators it controls. The one exception, the `ScrollBar` class, embodies characteristics of both a 2D presenter and a controller.

In this component, the 2D presenter classes exist in a number of a discrete states. For example, a simple push button can be pressed, released, or disabled. For each state, the push button itself determines which presenter is composited to the display surface, and which actions are called. But the controller tells the button when to change its state. It is the controller that manages the underlying event interests, receiving input from the user.

Note that some presenters in this component have more than one controller associated with them, each of which is responsible for a different aspect of behavior or presentation. The `Menu` class requires a layout controller to arrange the objects it controls and an actuator controller to manage the actuator behavior of those objects.

A controller is always attached to a space—it can only control objects that are attached to its space. In the User Interface component, this space is an instance of `TwoDSpace` or `GroupSpace`. To be managed by a controller, a model object must first be added to its space. And if that controller does not manage all objects in its space (`wholeSpace` is `false`), then the model object must be explicitly added to the list of objects controlled by that controller. For more information, see “Specifying an Object To Be Controlled” on page 108 in the “Controllers” chapter of this volume.

Controllers and Hit Testing

Hit testing is the process by which mouse events are matched with a given presenter in the presentation hierarchy. `ActuatorController` and `DragController` perform precision hit testing. These classes can detect that a mouse click occurred not only within the bounding rectangle of a presenter they are controlling, but also within the actual image area of its stencil.

The `Stencil` class defines the `inside` method, used to determine whether a given point is inside its image area. If `inside` returns `true` for a point within the bounding box of a stencil, then precision hit testing applies at that point. For the two presenters depicted in Figure 5-3, all points within the circular image area are inside the stencil, including points within the star for the presenter on the right. For more information about `Stencil` and its subclasses, see the discussion that begins on page 240 in the “2D Graphics” chapter.

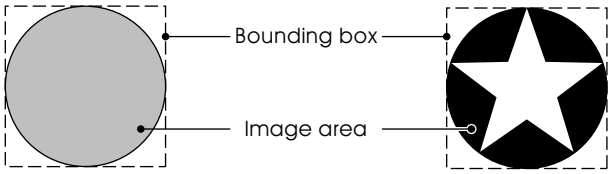


Figure 5-3: Precision hit testing means that hit testing applies only to points that are inside the image area of a stencil.

Since the `PushButton` and `Toggle` classes have multiple presenters, they do hit testing on a rectangular boundary, the boundary of the smallest rectangle that encloses all of the object's presenters. However, it is possible to create a subclass of `Actuator`—a scripted class that inherits from both `Actuator` and `TwoDShape`—that uses the precision hit testing features of the `ActuatorController` class. For an example, see page 135.

Hit Testing Within Subspaces

When the value of `wholeSpace` is set to `true` for a user interface controller, the controller maintains only a single set of event interests, which it associates with the space as a whole. The controller maintains event interests and performs hit testing for the space as a whole. Although this usually saves memory, it can be a source of ambiguity when that space contains another space, and the contained space has its own attached controllers.

The problem occurs with controller classes, such as `DragController` and `ActuatorController`, that receive and process mouse events. Ambiguity arises because the controller attached to the contained space stores interests in mouse events in that space's `eventInterests` instance variables. (Note that `TwoDSpace` inherits from `TwoDPresenter`, which defines an `eventInterests` instance variable for storing interests in mouse events.)

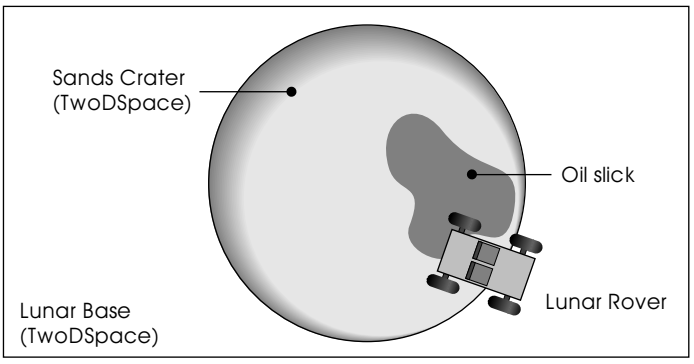


Figure 5-4: Hit testing and drag controllers

Figure 5-4 illustrates how this ambiguity can arise. In Figure 5-4, Lunar Base and Sands Crater are `TwoDSpace` objects, and Sands Crater is contained in Lunar Base. The lunar rover and the oil slick are both `Dragger` objects, but they are not controlled by the same drag controller. The lunar rover is

controlled by a drag controller that is attached to Lunar Base, while the oil slick has a drag controller that is attached to Sands Crater. The lunar rover is in front of Sands Crater in the presentation hierarchy.

Now suppose that we drag the lunar rover so that it is over the oil slick. Since the lunar rover is the frontmost presenter, we expect that when we grab the lunar rover, it will be the presenter that responds. The event system searches for interests in mouse events in depthwise order, beginning with the frontmost presenter. If all interests were stored on the top presenter, Lunar Base, no ambiguity would arise. However, since Sands Crater is a separate `TwoDSpace` object, its attached controllers associate their event interests with Sands Crater.

If the user clicks on the lunar rover over an area that is also covered by the oil slick, another `Dragger` object, the oil slick ends up being the object that is grabbed. Since event interests are not stored on the lunar rover, the event system first searches for interests on the oil slick, which has none, and then on Sands Crater, where it matches the event with an interest. Since the event system examined the interest on the oil slick first, it triggers the drag of the oil slick, and not of the lunar rover.

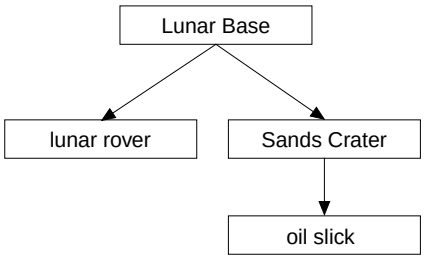


Figure 5-5: The presentation hierarchy for presenters in Figure 5-4

For non-ambiguous hit testing, set the value of `wholeSpace` to `false` and add each model object to its controller separately. When `wholeSpace` is `false`, the event system itself performs the hit testing, and interests are stored with the presenter that is to be controlled. Although this uses more memory, it insures that events are accurately matched and delivered to matching interests. Setting `wholeSpace` to `true` is useful for palettes of controls that have no subspaces.

Actuators

`Actuator` is an abstract class that can be mixed in with other classes to create a presenter that behaves like a button. In ScriptX titles, actuators commonly appear in concrete form as `PushButton` and `Toggle` objects. `Actuator` is not a `TwoDPresenter` class. An author can create a concrete subclass of `Actuator`, but it must inherit from both `Actuator` and `TwoDPresenter` if it is to be controlled by an `ActuatorController` object.

An actuator implements the actuator protocol. Every actuator responds to the `activate`, `multiActivate`, `press`, and `release` methods. These correspond to changes in its state. These methods are called automatically by an actuator controller whenever there is a corresponding change in state. Any 2D Presenter can respond to user input, by posting event interests and

receiving mouse events, but actuators extend the button metaphor much further. An actuator can respond to user input to show its state—pressed, released, or disabled.

PushButton

A push button is a user interface element that can have three different and distinct appearances: pressed, released, and disabled. These three appearances correspond with three states. Buttons modify their appearance by changing their presenters, so a push button defines one presenter for each state. Each presenter should provide visual feedback to the user, indicating the state of the button. For example, one convention in user interface design is to invert a graphic image when a button is pressed and lighten or "gray" that image when the button is disabled.

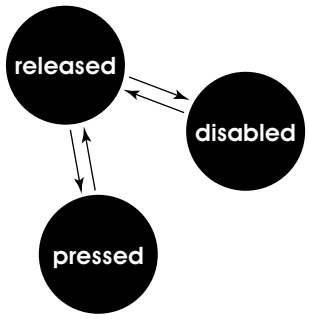


Figure 5-6: A button passes between three states.

The `PushButton` class is flexible enough to create any appearance or behavior that an author desires in a button. In a children's title, the released presenter could be an alert and standing elephant, while the pressed presenter could be an elephant that is kneeling or bowing. The disabled presenter could display a sleeping elephant. Some buttons invoke an action only when they are pressed; others, when they are released.

Most of the methods defined by the `PushButton` class are not meant to be called from the scripter. They are visible to the scripter so they can be overridden. In this way, `PushButton` class can be the basis for almost any kind of behavior that builds upon a button metaphor.

Within the ScriptX core classes, `PushButton` has a specialized concrete subclass, the `Toggle` class. `Toggle` modifies `PushButton` by using two additional states, toggled on and toggled off. These states are associated with two presenters, the `toggledOnPresenter` and `toggledOffPresenter`, that indicate the state of a toggle. A toggle is either on or off at all times. The toggled-on and -off presenters are layered on top of the pressed and released presenters. `Toggle` specializes the `activate` method to call either `toggleOn` or `toggleOff`, methods on `Toggle`. Common applications of `Toggle` are radio buttons and check boxes.

Controllers and Buttons

Controllers maintain a list of objects that they manage—objects that are associated with their space. In the User Interface component, actuator controllers manage groups of buttons, both as stand-alone elements of a user interface and as parts of another larger object, such as a menu or scrolling list. Controllers manage the state of the buttons they control, and thus control their presentation.

Radio buttons are a convention of graphical user interface design. Press a radio button, and it goes into its toggled-on state, automatically turning off other buttons that belong to the same group. Check boxes are a simpler version of radio buttons, controlled independently. Press a check box, and it toggles on, but it has no effect on other buttons near it. The `RadioButtonController` class manages a group of actuators, allowing them to behave as a unit like a group of conventional radio buttons. An actuator controller can be used with the `Toggle` class to create the behavior of a check box. See “ScriptX Widget Kit” on page 138 for more about radio buttons and check boxes.

Detecting Multiple Clicks

ScriptX does not explicitly define a double-click or multiple-click mouse event. A multiple click is a gesture, a series of individual clicks within a given interval of time. A double click is just a special case of a multiple click—a multiple click with only two clicks.

Actuator controllers interpret a series of mouse-up events over a given interval as a multiple click. When an actuator controller detects a multiple click, it calls `multiActivate` on the actuator it is controlling. A script can specify a function that will run each time `multiActivate` is called using the `multiActivateAction` instance variable.

In graphical user interface design, double clicking is treated as an extension of single clicking. A common design convention is that a single click selects an item, while a double click acts on it. In applications that handle editable text, a single click moves the text cursor, a double click selects an entire word of text, and a triple click (if defined) may select an entire line or paragraph.

Table 5-1 indicates the sequence of events when a user double clicks on a mouse button. The actuator controller that is associated with the actuator receives a series of mouse-down and mouse-up events, recorded in column one. It responds with the method call in column three. Assuming that the second mouse-up event is received within a given time, it calls `multiActivate` after the second click. (Otherwise, it repeats the call to `activate`, interpreting the user’s gesture as a pair of single clicks.)

Table 5-1: A double click gesture is a series of mouse events

Event	Event Interest	Method Called	Scripted Response
`MouseDownEvent`	`pressInterest` (`ActuatorController`)	`press` (`Actuator`)	`pressAction` (`Actuator`)
`MouseUpEvent`	`activateInterest` (`ActuatorController`)	`activate` (`Actuator`)	`activateAction` (`Actuator`)
`MouseDownEvent`	`pressInterest` (`ActuatorController`)	`press` (`Actuator`)	`pressAction` (`Actuator`)
`MouseUpEvent`	`activateInterest` (`ActuatorController`)	`multiActivate` (`Actuator`)	`multiActivateAction` (`Actuator`)

By default, an actuator controller is only interested in mouse events that occur on the first mouse button. A developer can modify the event interests associated with an actuator controller to detect and respond to events on other buttons.

Scrolling Presenters

A scrolling presenter presents a target presenter, also a 2D presenter, whose boundary is taller or wider than its own boundary. This target may be an instance of any `TwoDPresenter` class, such as `TwoDShape`, `PageElement`, or `TextEdit`. In effect, the `ScrollingPresenter` class gives the user a mechanism, common to all presenters, for viewing a portion of some larger presenter. Note that a scrolling presenter cannot have a direct presenter as its target—a direct presenter is one that draws directly to the screen rather than through a `BitmapSurface` that acts as a frame buffer.

A scrolling presenter derives its appearance, in part, from the objects it presents, but `ScrollingPresenter`, like its target presenter, is itself an instance of `TwoDPresenter`. The `TwoDPresenter` class defines methods that a scrolling presenter uses to draw itself. The following demonstrates the distinction between the properties and behavior of a scrolling presenter and those of its target.

```
-- this sets the scrolling presenter's stroke
myScroll.stroke := blackBrush
-- this sets its target's stroke
myScroll.targetPresenter.stroke := blackBrush
```

When you create a scrolling presenter, you can supply a horizontal or vertical scroll bar to allow user control of scrolling. You can bring portions of the target presenter into view by calling `scrollTo`. There is no default scroll bar; you must specify a `ScrollBar` object. If present, scroll bars are inset within the boundary of a scrolling presenter, and they are sized automatically. The region within the scroll bars is described by a stencil that is stored in the `clippingPresenter` instance variable defined by `ScrollingPresenter` and is used to clip the target presenter.

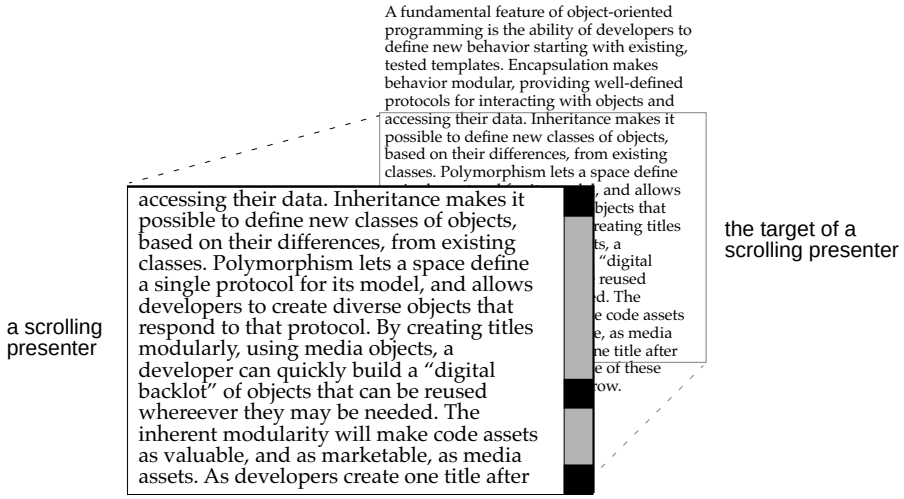


Figure 5-7: A scrolling presenter presents a selected portion of a target object, another 2D presenter, that is potentially much larger than itself.

`TwoDSpace` is a concrete class that inherits from both `TwoDMultiPresenter` and `Space`, combining the behavior of presenters and spaces. A scrolling presenter often has a `TwoDSpace` object as its target, a space to which other presenters can be appended. It is also possible to attach a controller to this space. This strategy is used by subclasses of `ScrollingPresenter` that present and control multiple items, such as `Menu`.

Among the document classes, the class `TwoDSpace` can be instantiated as a `PageLayer`. A scrolling presenter can target a page layer, creating a scrollable view of a page on a virtual document. To create a page layer that is scrollable, create a subclass of `ScrollingPresenter` that also inherits from `DocTemplate`, and make the `PageLayer` object be the target of the new scripted class.

`Window`, also a subclass of `TwoDSpace`, cannot be the target of a scrolling presenter, since a `Window` object must always be at the top of its presentation hierarchy. A script can create a window and append a scrolling presenter of the same width and height to it.

Subclasses of `ScrollingPresenter`, including the `Menu` class, present a list of choices by presenting actuators in the scrolling presenter's space. This approach to selecting items from a menu or list is different from what is found in either the Macintosh and Windows environments. In ScriptX, the menu item is an actuator, and an actuator is itself an object. It may invoke an action, defined by a script, when it is pressed or released. Think of the `Menu` class as a mechanism for presenting a list of actuators. These actuators have the same individual characteristics and behavior as any actuator in ScriptX.

Menus

`Menu` is a subclass of `ScrollingPresenter` that creates its own target presenter, a `TwoDSpace`. An author adds actuators to this space, usually `PushButton` or `Toggle` objects. Since the menu acts as a proxy for its own

target, an author can add objects directly to the menu. In the following example, `MenuOption` is a scripted subclass of `PushButton` that presents a `TextPresenter` object. `Crane`, `Egret`, and `Heron` are `MenuOption` objects.

```
myBirdMenu := new Menu placement:@menuDown
-- Crane and Egret are MenuOption objects, instances of PushButton
addMany myBirdMenu #(Crane, Egret)
```

Every menu has two controllers that operate on objects in its space. One of these is an an `ActuatorController` object that controls the state of the objects in the menu; it controls the actuators in the menu and allows the menu to pop down or out as appropriate. The other is a `RowColumnController` object that controls the layout or position of objects in the menu. By default, this `RowColumnController` object organizes its targets into a single column.

Every menu must have an invoker in order to be displayed. The invoker is usually an actuator, such as an instance of `PushButton`. A given menu can have more than one invoker. The invoker instance variable, defined by `Menu`, stores only a reference to the most recent invoker. A menu does not maintain any collection that stores a list of possible invokers. To make an actuator be an invoker of a menu, set the `menu` instance variable, defined by the `Actuator` class.

In the following example, `pickOne` is a `MenuOption` object, an instance of `PushButton`. The script makes `pickOne` the invoker of the menu that was defined above, `myBirdMenu`, and then adds an additional actuator to that menu.

```
pickOne.menu := myBirdMenu -- pickOne is a MenuOption button object
addOne pickOne.menu Heron -- Heron is another MenuOption button
```

An `ActuatorController` object activates a menu by calling the instance method `popup` on the `Menu` object it is invoking. When it calls `popup`, it automatically supplies the invoker, setting the `invoker` instance variable on the menu that was called. If a menu is invoked from another menu, the menu actuator controller supplies the correct supermenu as an argument to `popup`. Otherwise, the value of `superMenu` is `undefined`.

A menu can use its `invoker` or `superMenu` instance variables to traverse backward, to find the original actuator or the top-level menu that caused it to be invoked. The instance variables `invoker` and `superMenu` each refer to the most recent invoker or supermenu. ScriptX places no restrictions on how a menu is used, so it is quite possible for a given menu to be reused in different places, to have different invokers and different supermenus under different circumstances.

A menu can use its `subMenu` instance variable to traverse forward, to find the bottom of the hierarchy of currently popped-up menus. The instance variable `subMenu` refers to the currently popped-up submenu of the current menu; it is not a collection. If the current menu has no submenus, or if none of its submenus is currently popped up, then the value of `subMenu` is `undefined`.

ScriptX places no restrictions on how a menu is used, so it is quite possible for a given menu to be reused in different places, to have different invokers and different supermenus and submenus under different circumstances.

In the following diagram, `City` is the supermenu of `Neighborhood` and `State` is the supermenu of `City`. As the user selects first a state, then a city, and finally a neighborhood, the `ActuatorController` class calls `popup` on the submenu each time a new menu is invoked. A menu controller manages all of the actuators in its associated space so that they present themselves to the user correctly. For example, `California` in the `State` menu displays its `pressedPresenter` until the user either makes a selection in the final submenu, or clicks somewhere outside of the presenters that the menu actuator controller is controlling.

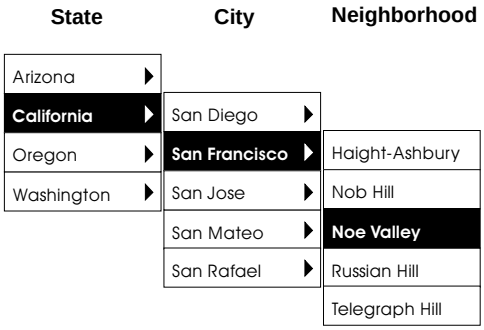


Figure 5-8: A group of super/sub menus

In the figure above, after the user selects `Noe Valley` in the final menu, the following expressions are equivalent:

```
Neighborhood.superMenu.superMenu
City.superMenu
State
```

The following expressions are also equivalent:

```
Neighborhood.superMenu.invoker
City.invoker
California
```

ScriptX supports both the “click” and “drag” styles of selecting from a menu, a design feature it shares with Windows, NextStep, Motif, and many other graphical-user-interface standards. A developer can modify the style of menu selection by creating a specialized version of `ActuatorController` and setting the `actuatorController` instance variable on a menu.

A menu pops up in one of three places: below its invoker, to the right of its invoker, or at the mouse pointer. A script sets the value of the `placement` instance variable, defined by the `Menu` class. To place the menu anywhere else, override the `place` instance method in a subclass. Note that `placement` is a property of a `Menu` object itself, and not of the actuator that acts as its invoker.

A menu's layout controller is concerned with the location of individual actuators within its space. The `RowColumnController` class is a specialization of the `TwoDController` class for use by user interface objects. `RowColumnController` objects control the placement of objects within a space. `RowColumnController` objects may coordinate the layout of many objects, or they may constrain the placement of individual objects. If a `RowColumnController` controls its entire space, it will attempt to modify the space to grow or shrink to fit all contained items.

Scroll Bars

A `ScrollBar` object is not associated with a controller. Instead, it acts as its own controller. Although not explicitly attached to a controller, it shares a similar design with other objects in the User Interface component. For a code example that creates an instance of `ScrollBar`, see page 133.

Presentation and control are separated in the `ScrollBar` class, just as with other classes in this component. A scroll bar derives its appearance from its fill and stroke and from the stencils it presents. It indicates that it is disabled by not drawing its thumb stencil and by drawing its disable brush on top of its track, increment, and decrement areas. Its increment and decrement stencils also have counterparts that can be used when these areas are pressed.

A `ScrollBar` object manages a set of event interests. For example, a scroll bar is interested in mouse-down events that occur over its thumb, increment, and decrement stencils and its track area when these areas are in their released state, and it is interested in mouse-move and mouse-up events when these areas are in their pressed state. A scroll bar adds and removes these interests dynamically as the user interacts with the scroll bar, so that the underlying event system is completely hidden from the user.

You can customize a `ScrollBar` object by modifying its fill and stroke and its stencils. Your script can draw on a wide range of behavior in a scroll bar with very little programming. For example, by omitting the increment and decrement stencils, and by setting the value of `pageAmount` to 0, a `ScrollBar` object can serve as a slider control such as a volume or contrast control.

Dragging the thumb stencil or clicking in the increment, decrement, or track areas changes the value of a scroll bar. This value can be updated continuously as the thumb is moved, or it can update only when the mouse button is released over the thumb, increment, decrement, or track areas. Each time this value changes, the `ScrollBar` object automatically calls a function specified by the author, stored in its `valueAction` instance variable. It calls this function with three arguments: `authorData`, `scrollBar`, and `value`. The first of these is supplied by you, stored in the scroll bar's `authorData` instance variable. The second argument is the scroll bar itself. The final argument is the new value of the scroll bar.

Like other user interface classes, `ScrollBar` defines many instance methods that are not meant to be called directly. They are visible to you so they can be overridden in a scripted subclass.

Draggers

`Dragger` is a mixin class, like several classes defined by other components in ScriptX. Other mixin classes in the core classes include `SequenceCursor`, `DocTemplate`, and `Actuator`, which is also described in this chapter.

A mixin class is never instantiated directly. It is always combined with a base class. By creating a scripted subclass that inherits from both, a developer can add features to the base class. In most cases, there are restrictions on what base classes a mixin class can be joined with. To be controlled by an instance of `DragController`, a `Dragger` object must be a combination of `Dragger` and a `TwoDPresenter` class. A dragger is often a simple presenter such as a `TwoDShape`, but the `Dragger` class can also be combined with more complex presenters, including any of the presenters in the User Interface component.

To make a `DragController` object control a given `Dragger` object, add the dragger to the controller's space. If the value of `wholeSpace` is `true`, the drag controller controls all `Dragger` objects in its space. If this value is `false`, then `Dragger` objects must be explicitly added to the drag controller in order to be controlled by it. A given drag controller may control multiple instances of `Dragger`. This controller manages the location of draggers within a space by modifying their `x` and `y` instance variables as the user moves the mouse.

Note that a dragger can only be controlled if it is in a space with a drag controller attached. It does not have to be constrained to that space. In its `grab` method and its `grabAction` script, a dragger can determine the area it can be dragged over by changing its space. For example, a dragger could remove itself from its current space and add itself to the space that contains the current space, giving itself a greater range of movement.

The `Dragger` class divides a dragging operation into four discrete phases, for which it defines four corresponding methods: `grab`, `beforeDrag`, `afterDrag`, and `drop`. The first and last of these methods, `grab` and `drop` are called only once during each drag operation, at the beginning and at the end. The other two, `beforeDrag` and `afterDrag` are called continuously as an object is dragged. An author can specify a function that will be called each time a `Dragger` object enters one of these phases.

How Controllers Manage Presenters

User interface objects receive and process events automatically, ultimately calling a function that is defined by the user. Most developers never have to go into the internal mechanisms by which controllers manage presenters. The following section is for advanced users who intend to specialize one of the core classes in the User Interface component.

In the User Interface component, controller classes maintain event interests and receive events, while presenter classes respond to their respective controllers. The `DragController` and `Dragger` classes are typical. Suppose that the user clicks on a `Dragger` object. This click, a mouse-down event, sets off a chain of method calls. This initial click on a `Dragger` object is called grabbing, and it will place the object under the `DragController` object's control, allowing the model object to be dragged around until the mouse button is released.

1. When a script creates a `Dragger` object and places it in the control of a drag controller in that object's space, the `DragController` class registers an interest in mouse-down events, which it stores in its `grabInterest` instance variable. As the user clicks the mouse and moves it to various locations on the screen, this controller adds and removes its interest in various classes of mouse events, as appropriate. It associates this `grabInterest` with an event receiver function, its `processGrab` method.
2. The user clicks on the `Dragger` object. Since the drag controller registered an interest in mouse-down events, the `processGrab` method automatically receives this event.
3. The `processGrab` method on `DragController` triggers the `grab` method on `Dragger`. By specializing `grab`, an author could modify the behavior of an entire subclass of `Dragger` objects.
4. The `grab` method on `Dragger` calls a scripted function, specified by the author, stored in the `grabAction` instance variable, using the value stored in `authorData` as its argument. This function also receives an additional argument, a point representing the offset of the mouse pointer that selected it. This function sets the behavior of a particular instance of `Dragger` when the object is grabbed. The function stored in `grabAction` runs in the same thread as `grab`, and returns control to `grab` after it runs.

When `grab` finishes executing, the grab operation is complete. The `DragController` regains control. It adds an interest in mouse-up events to the dragger's `eventInterests` collection, so that it can be informed when the user releases the mouse button. It is ready to receive events that are matched to its `dropInterest`.

As the mouse moves, the `DragController` object polls the mouse device, querying it for its location each time the space's clock ticks. With each tick of the clock, the drag controller checks both the mouse's coordinates and the event dispatch queue to determine whether the mouse has moved. If the mouse has moved, it calls `beforeDrag` and `afterDrag` on the dragger it is controlling.

The following table shows how instance methods and variables defined by `DragController` coincide with instance methods and variables defined by `Dragger`. A `DragController` object automatically processes mouse events and polls the current mouse device for movement of the mouse. Each time `tickle` is called, it calls `beforeDrag` and then `afterDrag`, mapping any events or mouse movements to the corresponding method defined by `Dragger`, and ultimately, to a scripted function that is defined by the developer.

Table 5-2: How a drag controller and a dragger work together.

DragController (instance variable)	DragController (instance method)	Dragger (instance method)	Dragger (instance variable)
`grabInterest`	`processGrab`	`grab`	`grabAction`
	`tickle`	`beforeDrag`	`beforeDragAction`

Table 5-2: How a drag controller and a dragger work together.

DragController (instance variable)	DragController (instance method)	Dragger (instance method)	Dragger (instance variable)
	`tickle`	`afterDrag`	`afterDragAction`
`dropInterest`	`processDrop`	`drop`	`dropAction`

In this table, column one contains an event interest defined by `DragController`. Column two contains either an event receiver or the generic `tickle`. Both `processGrab` and `processDrop` are event receivers, functions that receive an event that matches the interest specified in column one. The `tickle` method is not an event receiver. For more information on `tickle`, see page 106 of Chapter 4, "Controllers." Column three contains an associated method, defined by `Dragger`, which the method in column two calls. Column four contains the name of a function that is called by the corresponding method in column three. All of these methods and variables can be defined or specialized at the scripter level to modify the behavior of an instance of `Dragger`, or an entire subclass of `Dragger` objects.

Each class in the User Interface component is associated with a corresponding set of interests, event receivers, and scripted user actions. A table for the `ActuatorController` class, equivalent to the one above, can be found in the *ScriptX Class Reference*.

User Interface Examples

User Interface objects can serve as building blocks for creating every possible variation on user interface controls—new styles of buttons, menus, scroll bars, and other user interface devices. Objects in the User Interface component can be used, with virtually no coding, by initializing the object and assigning values to the presenters or stencils that determine its appearance. At the same time, these objects can be the basis for creating a library of objects, with different styles and behaviors that can be specified or selected by an author.

Creating an Instance of ScrollBar

This example creates an instance of the `ScrollBar` class without using any specialization. All five of the scroll bar stencils are defined (`thumbStencil`, `incrementStencil`, `decrementStencil`, `pressIncrementStencil`, and `pressDecrementStencil`), as well as the fill and stroke for the enabled scroll bar appearance and the brush for the disabled scroll bar appearance.

The increment and decrement areas are squares when they are not being pressed and are circles when they are being pressed. The thumb of the scroll bar displays the value of the position of the thumb. The text stencil that displays this value has a bounding box that puts the stencil above the scroll bar by default; this example shows the transformation applied to the text stencil to make it visible in the track of the scroll bar.

The stencils are black by default. To change this appearance, you can use bitmaps for these stencils or you can specialize the `draw` method for `ScrollBar`.

This example is available in `DOCEXMP/acguide/userintf/scrllbar.sx`. After you run this file through the Listener, enter

```
sb.enabled := false
```

into the Listener to display the disabled appearance of the scroll bar. With the scroll bar enabled again, observe the behavior of the scroll bar when you click or hold down the mouse button on the increment and decrement areas and on the track area of the scroll bar, and when you use the mouse to drag the thumb.

```
global sq := new Rect x2:20 y2:20
global ov := new Oval x2:20 y2:20
global myThumb := new TextStencil \
    font:(new platformFont name:"Arial" macintoshName:"Helvetica") \
    string:"0"
global myMatrix := new TwoDMatrix ty:15
transform myThumb.bbox myMatrix @mutate
global grayBrush := new Brush color:(new RGBColor red:128 \
    green:128 blue:128)

object sb (ScrollBar) orientation:@horizontal
    incrementStencil:sq, decrementStencil:sq, thumbStencil:myThumb
    settings
    width:200
    stroke:blackBrush
    fill:grayBrush
    disableBrush:grayBrush
    pressIncrementStencil:ov
    pressDecrementStencil:ov
    stepAmount:2
    pageAmount:10
    valueAction:(a b c -> b.thumbStencil.string := (c as String))
end

object w (Window) name:"Scroll Bar Example"
    settings
    width:200, height:40
end

append w sb
show w
```

This example uses the `settings` section of the object definition expression to set values for several instance variables on instantiation. The `settings` notation is especially useful with user interface objects, since they define a large number of instance variables that often determine much about the look and feel of the controls they comprise.

The `settings` notation is useful for assigning a complete object as the value of an instance variable, but it cannot be used to set an element such as a member of a collection. However, a script could create a standard collection of

attributes and assign the entire collection on instantiation to all appropriate objects. This approach is useful for creating a group of objects that share a common look and feel.

Many titles that use user interface objects use bitmaps, presented by `TwoDShape` objects, in connection with user interface controls. Note that bitmaps cannot be resized. A scroll bar can use bitmaps for its thumb, increment, and decrement stencils.

Creating a New Actuator

The second code example creates a new class of `Actuator`. ScriptX defines two concrete classes of actuator among the core classes: `PushButton` and `Toggle`. Although an actuator controller is capable of precise hit testing, `PushButton` and `Toggle` both test for hits in their entire bounding rectangle.

`SimpleButton` is a class that can present any instance of `Stencil` as a button, including bitmaps, ovals, and paths. It is capable of precise hit testing on these objects, even though they are not rectangular in shape. `SimpleButton` is only one of many possible designs for such a button.

```
class SimpleButton (TwoDShape, Actuator)
    instance variables
        authorData
        activateAction
        pressAction
        releaseAction
        currentState -- either @up or @down
    instance methods
        method init self #rest args -> (
            apply nextMethod self args
            self.currentState := @up
        )
end
```

Every actuator implements the `activate`, `press`, and `release` methods. These methods are not actually called from the scripter. The `SimpleButton` class follows the Actuator protocol so that it can be attached to an actuator controller. An actuator controller receives mouse events and calls the associated method on the actuators it manages.

```
method activate self {class SimpleButton} -> (
    nextmethod self
    if self.enabled == false then (
        return self
    )
    else (
        self.currentState := @up
        self.fill := whiteBrush
        handleActivate self
        return self
    )
)
method press self {class SimpleButton} -> (
    nextmethod self
    if self.enabled == false then (
        return self
    )
```

```
    else (
        self.currentState := @down
        self.fill := blackBrush
        handlePress self
        return self
    )
)
method release self {class SimpleButton} -> (
    nextmethod self
    if self.enabled == false then (
        return self
    )
    else (
        self.currentState := @up
        self.fill := whiteBrush
        handleRelease self
        return self
    )
)
```

For brevity, `SimpleButton` does not specialize `multiActivate`, the method in the Actuator protocol that handles multiple mouse clicks. Since it does not define a method for `multiActivate`, it inherits a default implementation from `Actuator`, which does nothing.

`SimpleButton` factors its response to `activate`, `press`, and `release` into two parts: a "handler" method and a scriptable "action" or response. For example, `activate` calls `handleActivate`, which responds by invoking the function stored in `activateAction`. This design reflects both `PushButton` and `Toggle`, the two concrete actuator classes in the core classes. Although there is no reason that `SimpleButton` must work like `PushButton`, there are advantages to consistency in design. For example, a method that is written to override the `handleActivate` method defined by `PushButton` could easily be adapted to specialize `SimpleButton`. This design also allows for easy specialization at both the class and instance level.

```
method handleActivate self {class SimpleButton} -> (
    if self.activateAction <> undefined do (
        self.authorData := "activate"
        (self.activateAction) (self.authorData) self
    )
    return self
)
method handlePress self {class SimpleButton} -> (
    if self.pressAction <> undefined do (
        self.authorData := "press"
        (self.pressAction) (self.authorData) self
    )
    return self
)
method handleRelease self {class SimpleButton} -> (
    if self.releaseAction <> undefined do (
        self.authorData := "release"
        (self.releaseAction) (self.authorData) self
    )
    return self
)
```

To test SimpleButton, we need a window, an actuator controller to control objects in the window, and an instance of SimpleButton. To demonstrate that this script is responding to activate, press, and release, define a simple function called beep. Set activateAction, pressAction, and releaseAction to call beep. Although there is no reason that it must be set up this way, the function signature of an "action" for the SimpleButton class is the same as that for a PushButton or Toggle object. Thus, a function or method that could be the activateAction of a PushButton object could also be the activateAction of a SimpleButton object.

```
object myWindow (Window)
    boundary:(new Rect x2:200 y2:200)
    settings x:300, y:50
end
show myWindow

-- create a controller to control actuators in the window
object myActuatorController (ActuatorController)
    enabled:true, space:myWindow, wholeSpace:true
end
-- create an instance of SimpleButton
object myButton (SimpleButton)
    boundary:(New Oval x2:100 y2:100), stroke:blackBrush
    settings x:50,y:50
end
append myWindow myButton -- append it to the window

-- set up a function for
-- activateAction, pressAction, and releaseAction
function beep x y -> format debug "beep %*\n" x @normal
myButton.activateAction := beep
myButton.pressAction := beep
myButton.releaseAction := beep
```

A more advanced version of SimpleButton could specialize additional methods. For example, the enabledSetter method, which sets the value of the enabled instance variable defined by Actuator, could be specialized to change the appearance of a button when it is disabled. The setter methods for each of the instance variables could be specialized to perform type checking. A multiActivate method could be implemented along the same lines as activate, press, and release.

Creating a Hierarchical Menu

The directory codesamp/hiarcmen/ contains an example of a hierarchical menu. The file textmenu.sx demonstrates how to create a hierarchical text menu. The file bmpmenu.sx demonstrates how to create a hierarchical menu that uses bitmaps as its items. The file reqfiles/button.sx is a mixin class that gives any TwoDShape the functionality of a pushbutton. The media directory contains bitmaps used in bmpmenu.sx. Run the textmenu.sxt or bmpmenu.sxt files or open the loadme files in the Listener.

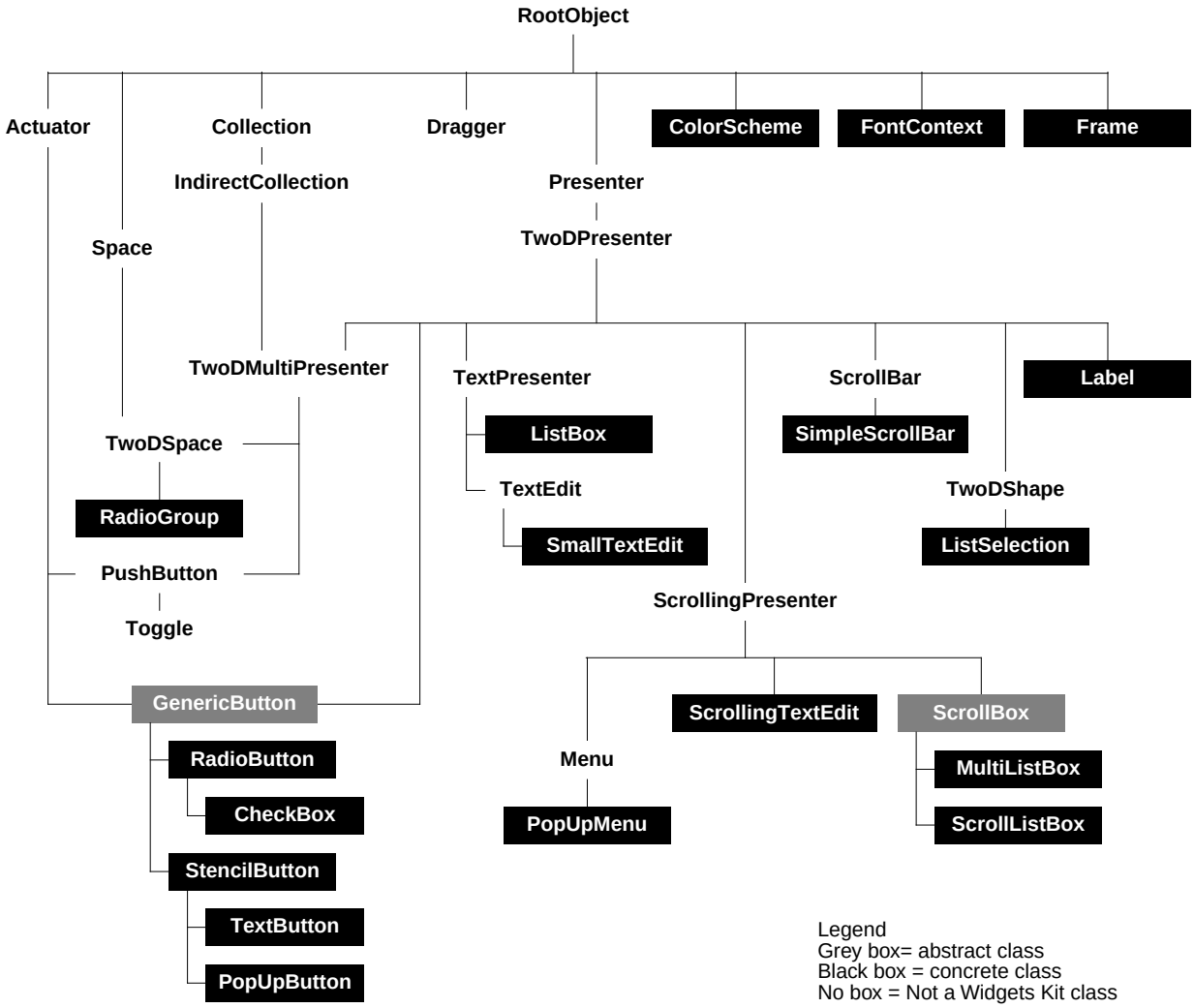
A simpler example of a hierarchical menu is in docs/docexmp/compguid/userintf/menu_eg.sx. Open this file in the Listener.

ScriptX Widget Kit

The ScriptX Widget Kit provides a set of simple widgets to save you time creating your user interface. In general, Widget Kit objects use fewer presenters and therefore perform better than other User Interface objects, but they are also less customizable; they are less complex, but also less flexible. For example, buttons in the Widget Kit derive their appearance from stencils and not from subpresenters.

Classes and Inheritance

The class inheritance for the ScriptX Widget Kit is shown in the following figure.



The following classes form the Widget Kit in the User Interface component. In this list, indentation indicates inheritance. Most of the classes in this component inherit from `TwoDPresenter`.

`ColorScheme` – defines various grayed out and stippled appearances. You should never need to instantiate `ColorScheme` yourself.

`FontContext` – defines two default font contexts. Make a new instance of this class if you want your widgets to use a different font or type style.

`Frame` – defines the three-dimensional look of the button edges. You should never need to instantiate `ColorScheme` or `Frame` yourself.

`Label` – implements simple labels. These objects automatically size themselves to fit the label text.

`GenericButton` – an abstract class that defines common button characteristics.

`RadioButton` – displays a filled circle when selected. Radio buttons can be either selected or not selected (they are on/off buttons, or toggle buttons); they do not automatically execute functions when they are selected.

`CheckBox` – displays a check mark when selected. Check boxes can be either selected or not selected (they are on/off buttons, or toggle buttons); they do not automatically execute functions when they are selected.

`StencilButton` – displays a bitmap or other stencil on a button. A stencil button can execute a function when it is selected.

`TextButton` – displays text on a button. A text button can execute a function when it is selected.

`PopUpButton` – can display a `PopUpMenu` object when it is selected.

`PopUpMenu` – a `Menu` object that is the target of a `PopUpButton` object.

`RadioGroup` – displays a set of `RadioButton` objects or `CheckBox` objects. Only one `RadioButton` object or `CheckBox` object in a group can be selected at a time.

`ListSelection` – is the default value of the `selection` instance variable of the `ScrollBox` class. You should never need to instantiate `ListSelection` yourself.

`ScrollBox` – is an abstract class that provides basic functionality for scrolled list box classes.

`MultiListBox` – displays a box with a multiple-column scrolled list. Its `list` is a collection, and its target is a `TwoDMultiPresenter` that consists of one `ListBox` for each column (for each element in the list collection). The collection of lists scroll together.

`ScrollListBox` – displays a box with a single scrolled list.

`ListBox` – is a very simple class that lists non-selectable items in a non-scrollable box. It is the target of the more interesting and useful `MultiListBox` and `ScrollListBox` objects.

`SimpleScrollBar` – is used in `ScrollingTextEdit`, `MultiListBox`, and `ScrollListBox`. You could instantiate it by itself to use as a slider for contrast or volume control, for example.

`SmallTextEdit` – displays a small editable text field such as on a form.

`ScrollingTextEdit` – displays a scrolled box of editable text.

Widget Kit Example

The example shown in this section demonstrates how to instantiate the widget kit classes. For detailed information on the instance variables and instance methods for these classes, see the *ScriptX Class Reference*.

This example and the title it builds are available online at `utils/widgets/`. In that directory you will find all the Widget Kit source code, plus the following files associated with the example described in this section:

`media` – This subdirectory contains text files and other media that are used by this example. Many of the Widget Kit classes require files in this directory, so this directory needs to be in the same area with any other script you write that uses Widget Kit classes.

`wdgtest.sx` – This is the ScriptX source file that is shown in this section. Open this file in the Listener to produce a title file. By default, the title file will be built in the same directory with the ScriptX application (`theStartDir`).

`wdgtest.sxt` – This is the title that is built when you compile `wdgtest.sx` in the Listener. Open this file in the Kaleida Media Player to skip right to the final result.

This example (and any script you create using the ScriptX Widget Kit) also requires the widget library file, `widgets.sxl`. You will find this file in the same directory with the ScriptX application and the Kaleida Media Player (`theStartDir`). Include it in your script by opening the library as shown at the top of this example.

The `WidgetInterface` module referred to at the top of this example is required by any script that uses the ScriptX Widget Kit. This module is built into the widget library file, `widgets`.sxl.

When you complile this example, you get a message that a particular `HFSStream` object cannot be made persistent. You can ignore this message. The object referred to is the text that is imported into the `ScrollingTextEdit` object near the end of this example; this text stream is not supposed to be saved as a persistent object, but rather is media that is read into the title dynamically. You can change this text (and any of the other media that are imported into this example) after you have created a title file. The new media will be used the next time you run the title; you do not have to rebuild the title to use modified versions of these imported media.

```
-- file wdgtest.sx
open LibraryContainer path:"widgets.sxl"
Module test
    uses ScriptX, WidgetInterface
end
in module test
global w := new Window
w.fill := new Brush pattern:(importDIB "media/bkgnd.bmp")
show w
global ac := new ActuatorController space:w wholeSpace:true
```

```
-- StencilButton
global bitmapButton := new StencilButton \
    stencil:(importDIB "media/kicon.bmp")
bitmapButton.position := new Point x:35 y:14
global sbCaption := new Label text:"StencilButton"
sbCaption.position := new Point x:10 y:60
append w sbCaption
append w bitmapButton

-- TextButton
global textB := new TextButton text:"Kaleida!"
textB.position := new Point x:120 y:30
global tbCaption := new Label text:"TextButton"
tbCaption.position := new Point x:120 y:60
append w tbCaption
append w textB

-- ScrollListBox
global wdgclass_list := #("ColorScheme", "FontContext", "Frame", \
    "Label", "GenericButton", "RadioButton", "CheckBox", \
    "StencilButton", "TextButton", "PopUpButton", "RadioGroup", \
    "ListBox", "SmallTextEdit", "ListSelection", \
    "SimpleScrollBar", "ScrollingTextEdit", "ScrollBox", \
    "MultiListBox", "ScrollListBox", "PopUpMenu")

global slb := new ScrollListBox list:wdgclass_list \
    boundary:(new Rect x2:135 y2:210) \
    hasScrollBar:true
slb.x := 10
slb.y := 150
global slbCaption := new Label text:"ScrollListBox"
slbCaption.position := new Point x:10 y:120
append w slbCaption
append w slb

-- CheckBox
global cb := new CheckBox text:"Check Me" frame:(new Frame)
cb.x := 170; cb.y := 130
global cbCaption := new Label text:"CheckBox"
cbCaption.position := new Point x:170 y:100
append w cbCaption
append w cb

-- Radio Button
global rb := new RadioButton text:"On/Off" frame:(new Frame)
rb.x := 170; rb.y := 210
global rbCaption := new Label text:"RadioButton"
rbCaption.position := new Point x:170 y:180
append w rbCaption
append w rb

-- Radio Group
global rg := new RadioGroup \
    itemList:#(@one:"One", @two:"Two", @three:"Three")
rg.position := new Point x:170 y:290
global rgCaption := new Label text:"RadioGroup"
rgCaption.position := new Point x:170 y:260
append w rgCaption
append w rg

-- SmallTextEdit
global smallTE := new SmallTextEdit text:("120" as Text) \
    boundary:(new Rect x2:35 y2:20)
smallTE.x := 230
smallTE.y := 30
global steCaption := new Label text:"SmallTextEdit"
steCaption.position := new Point x:210 y:60
append w steCaption
append w smallTE

-- PopUpMenu
```

```
global puMenu := new PopUpMenu \
    list:#(@one:"one",@two:"two",@fewWords:"A Few Words") \
    width:104 actuatorController:undefined layoutController:undefined
-- PopUpButton
global popUpB := new PopUpButton menu:puMenu value:@two
popUpB.x := 330
popUpB.y := 50
global pumCaption := new Label text:"PopUpMenu"
pumCaption.position := new Point x:330 y:20
append w pumCaption
append w popUpB

-- MultiListBox
global mlb := new MultiListBox boundary:(new Rect x2:120 y2:120) \
    list:#(#(1,@one), #(2,@two), #(3,@three), #(4,@four), \
        #(5,@five),#(6,@six),#(7,@seven))
mlb.x := 460
mlb.y := 50
global mlbCaption := new Label text:"MultiListBox"
mlbCaption.position := new Point x:460 y:20
append w mlbCaption
append w mlb

-- ScrollingTextEdit
global TEstream := getStream theScriptDir "media/scrollte.txt" \
    @readable
global TEtext := importMedia theImportExportEngine TEstream @Text \
    @ASCIItext @TextEdit
global scrollTE := new ScrollingTextEdit \
    boundary:(new Rect x2:280 y2:160) \
    textWidth:260 textHeight:600 text:TEtext autoRecalc:false
scrollTE.x := 300
scrollTE.y := 200
global scteCaption := new Label text:"ScrollingTextEdit"
scteCaption.position := new Point x:300 y:170
append w scteCaption
append w scrollTE
global txtPres := scrollTE.targetPresenter

global tc := new TitleContainer dir:theScriptDir path:"wdgtest.sxt" \
    name:"Widget Kit Test"
w.title := tc
append tc (getModule @test)
tc.startupAction := (tc -> for i in tc do load i; show w)
close tc
```

CHAPTER

Clocks

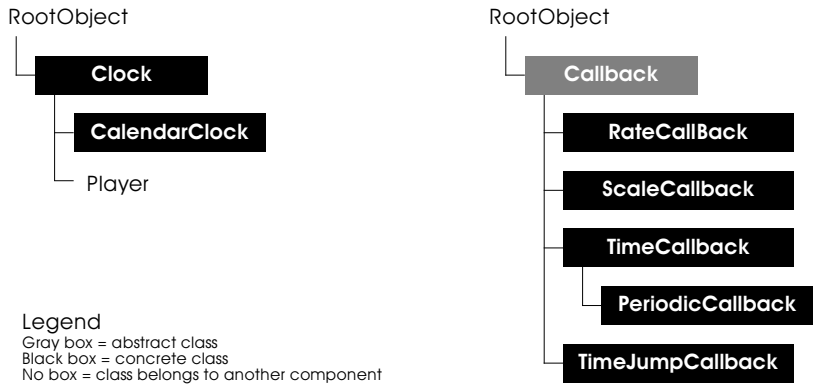
6

The Clocks component provides facilities for representing time, keeping track of time, and scheduling and synchronizing timed sequences of actions. The Clocks component consists of a number of classes, the principal among these being `Clock` and `Callback`.

In ScriptX, the modeling and presentation features of a title are organized and managed by spaces. Each space has a clock to control time-based activities of the objects it contains. While this chapter discusses how to organize the timing hierarchy of a model, the more general topic of spaces and modeling are covered in the chapter "Spaces and Presenters," earlier in this guide. Synchronization of time-based media is provided through the `Clock` subclass `Player` and its subclasses—see the chapters "Players" and "Media Players" for details. To enable an entire title to be started, paused, restarted, and stopped in synchronization, every clock belongs to a specific title container. See the chapter "Title Management" for details on title containers and clocks.

Classes and Inheritance

The class inheritance for the Clocks component is shown in the following figure.



The following classes form the Clocks component. In this list, indentation indicates inheritance.

`Clock` – Defines an object that keeps time and provides a mechanism for controlling sequences of actions. The ScriptX run-time environment provides a global instance, `theEventTimeStampClock`, that can be used to timestamp events.

> `CalendarClock` – Class defining an object that keeps the current date and time of day. There is one global instance of `CalendarClock`, `theCalendarClock`.

`Callback` – A class defining a mechanism to invoke specific functions at certain times or events in a clock's life.

`RateCallback` – A class whose instances invoke a function when a clock's rate changes.

`ScaleCallback` – A class whose instances invoke a function when a clock's scale changes.

`TimeCallback` – A class whose instances invoke a function when a clock reaches a certain time.

`PeriodicCallback` – A class whose instances repeatedly invoke a function at a specific interval.

`TimeJumpCallback` – A class whose instances invoke a function when a clock's time jumps (for example, when it is reset).

Conceptual Overview

Whether modeling complex systems or presenting media, multimedia titles need to accommodate and synchronize time-based behavior. For this reason, the timing facility provided by the Clocks component is a fundamental feature of the ScriptX programming framework. The key features of this timing facility are defined by the classes `Clock` and `Callback`. *Clocks* provide the basic timing mechanism for a ScriptX title. *Callbacks* provide ways for clocks to invoke actions which can control the behavior of other objects over time.

Spaces, described in the "Spaces and Presenters" chapter of this guide, are the basic organizing structure of a ScriptX title. Spaces provide an environment in which other objects can interact for modeling or presentation. To control the timing of these activities, the ScriptX `Space` class defines a `clock` instance variable. Objects inhabiting a space can refer to this clock to control their behavior over time.

In creating a simulation or presentation, you can create any number of clocks and link them together in a *timing hierarchy*, a structure of clocks used to coordinate related actions. Clocks within a hierarchy can be synchronized, yet can locally control the time-based behavior of objects that rely on them. Thus, within a space, you can create clocks to control local behavior of objects, then connect those clocks to the space's clock to synchronize behavior between objects in the space.

Clocks control object behavior through callbacks. You create callbacks by requesting them from a clock. When you request a callback, you specify the *callback script*, a function or method to be called at the appropriate time, and the first argument to the callback script—usually the object to perform the action. Within a callback, instance variables specify other details, such as conditions in which the action is performed and the order in which concurrent callbacks are triggered.

How Clocks Work

Clocks keep time through their instance variables `rate`, `scale`, `time`, and `ticks`. This discussion starts with a look at the meaning of `scale` and `rate`.

Scale and Rate

To provide flexible control over timed behavior of other objects, each `Clock` object can have its own sense of time. This sense is determined by the instance variables `rate` and `scale`. To understand the meaning and interaction of these variables, it helps to visualize a standard clock or stopwatch with a circular face and a single sweep hand.

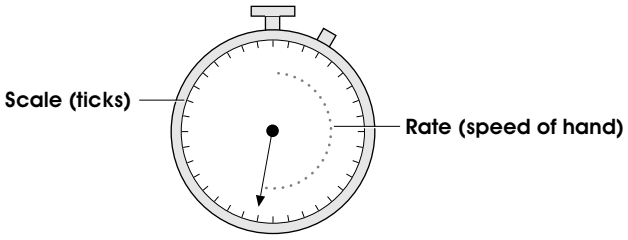


Figure 6-1: Clock scale and rate

As shown in Figure 6-1, the `scale` can be thought of as the number of tick marks on the face of the clock. The `rate` can be thought of as the speed of the hand, or how quickly the hand ticks off the marks on the face.

Here's a simple example: A clock with a scale of 60 can be thought of as having 60 tick marks on its face. If its rate is set to 0, the clock is stopped. If its rate is set to 1, the hand sweeps the face once per second, and the clock ticks 60 times per second. If its rate is set to 5, the hand sweeps the face five times per second, and the clock ticks 300 times per second.

When a clock is created, its rate is zero, which means it is stopped. To start it, set its rate to a non-zero value.

Setting the rate of a clock to a positive value makes the clock run forward. Setting it to a negative value causes the clock to run backwards. The following code starts a clock ticking at a rate of 1:

```
-- create a new clock
clock1 := new clock
-- start the clock ticking
clock1.rate := 1
```

A media-centric way to think about scale is as frames or samples per second. This meaning is used in the `MediaStreamPlayer` class and its subclasses. Similarly, the scale of the clock associated with a `TwoDSpace` instance is set to 24, equivalent to 24 frames per second. This clock is used to drive the compositor, which controls visual presentation of a title.

Reading a Clock's Time

Each clock keeps its time in two instance variables. The `ticks` instance variable represents the number of elapsed ticks since the clock was first started—or from the last time it was reset to zero. Clocks also keep a `time` instance variable, which contains a `Time` object representing local hours, minutes, seconds, and ticks since the clock was first started. You can reset a clock at any time by setting either its `ticks` or `time` to zero.

The relationship between `ticks`, `scale`, and `time` (in local seconds) can be expressed with the formula:

local seconds = ticks / scale

Elapsed time is determined by the rate: one sweep of the clock corresponds to one second in local time. Note however that when a clock is in a timing hierarchy, a clock's *effective rate* may be different than its actual rate, which affects the rate at which the clock sweeps off local seconds. The relationship between effective rate, local time, and actual time are discussed in the next section, "Timing Hierarchies and Synchronization."

In addition to rate, the time kept by a clock in a timing hierarchy may be affected by another factor—`offset`. The `offset` instance variable is used to determine the relationship between the time kept by clocks in a hierarchy. The next section also discusses offset and its effect on a clock's local sense of time.

Timing Hierarchies and Synchronization

To synchronize complex sequences of interrelated actions, clocks can be organized in hierarchical structures, referred to here as *timing hierarchies*.

Master and Slave Clocks

A clock directly above another in a hierarchy is referred to as its *master*. The clock below is referred to as the slave. In the simplest hierarchy, with one master and one slave clock, the rate of the slave is directly controlled by the master. If the master's rate is set to 1, the slave runs at its own rate. If the master's rate is set to 0, the slave stops. If the master's rate is set to -1, the slave runs backwards. In more complex cases, the slave runs at an effective rate, determined by its relationships to the rates of all clocks above it in a hierarchy. Effective rate is described in the next section.

The master clock at the top of a timing hierarchy is referred to as the *top clock*. The top clock is itself synchronized to an underlying hardware clock known as the *root clock*. Since it's hardware dependent, the root clock isn't visible or controllable from ScriptX.

In Figure 6-2, the top clock is the master of clocks a and b, while clock a is the master of clocks c and d. A clock that is directly controlled by another is referred to as its *slave*. In the figure, clocks c and d are slaves of a, which is in turn a slave of the top clock. Each master can have several slaves, while each slave has only a single master.

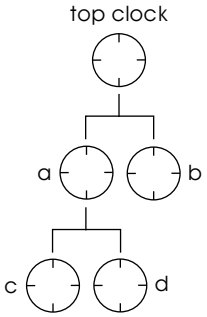


Figure 6-2: A timing hierarchy

Clocks keep track of their position in a timing hierarchy through their `masterClock` instance variable. This instance variable can be set when a clock is first created using the `masterClock` keyword:

```
myClock := new Clock masterClock:mySpace.clock
```

In this example, the new clock is connected as the slave of the clock belonging to a space.

The clock belonging to a space is created automatically when you create the space. A space's clock is always created as a top clock, independent of others. This makes sense for a space that serves as the container for a whole presentation or model. However, if you append one space to another, you need to explicitly set the `masterClock` of the appended space's clock. For more on spaces, see the chapter "Spaces and Presenters". By default, the `scale` of a space's clock is set to 24.

Effective Rate

As mentioned in the previous section, a clock's position in a timing hierarchy affects the global meaning of its local rate and time. When one clock is slaved to another, the rate of the master clock acts as a multiplier for the slave, determining the slave clock's effective rate and actual time by the following formulas:

*effective rate = master's effective rate * slave's rate*

actual seconds = ticks / (effective rate * scale)

A top clock's effective rate is its actual rate, the value found in its `rate` instance variable. A clock further down the timing hierarchy has an effective rate that is the multiple of all the rates of clocks above it and its own rate.

Consider the case of a timing hierarchy that uses three separate clocks to keep track of hours, minutes, and seconds. You could imagine organizing this hierarchy in a couple of different ways. For example, you could create a top-down hierarchy as shown in Figure 6-3, with the top clock being the hour clock, the minute clock as its direct slave, and the second clock as the minute clock's slave.

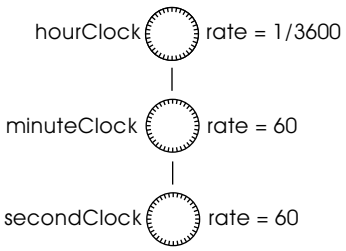


Figure 6-3: A top-down timing hierarchy

The following code would be used to set up this hierarchy:

```
hourClock := new Clock
hourClock.rate := 1.0 / 3600.0
minuteClock := new Clock masterClock:hourClock
minuteClock.rate := 60
secondClock := new Clock masterClock:minuteClock
secondClock.rate := 60
```

Remember that `rate` is equivalent to the velocity at which the metaphorical clock sweeps its face. Thus, the `hourClock` has its rate set to 1/3600, meaning it will sweep 1/3600 of its face per second—making a complete sweep once an hour. The effective rate of the minute clock is 1/3600 * 60 or 1/60 sweeps per second. The effective rate of the second clock is 1/60 * 60, or 1 sweep per second.

The same timing effect could be achieved using the bottom-up hierarchy shown in Figure 6-4, with the `secondClock` as the top clock and the `minuteClock` and `hourClock` as slaves.

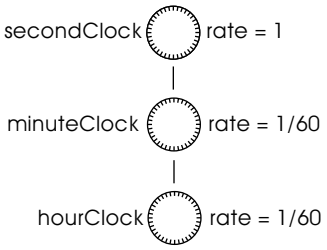


Figure 6-4: A bottom-up timing hierarchy

The following code would be used to set up this hierarchy:

```
secondClock := new Clock
secondClock.rate := 1
minuteClock := new Clock masterClock:secondClock
minuteClock.rate := 1.0 / 60.0
hourClock := new Clock masterClock:minuteClock
hourClock.rate := 1.0 / 60.0
```

Offset

Another relationship between a slave clock and its master is expressed by its instance variable `offset`. This value represents the difference between the ticks of the slave and its master, expressed in ticks of the master. Since master and slave clocks can run at different rates, `offset` specifies this difference at a specific time: the slave's time 0. When the slave's offset is any value other than 0, then the slave's `ticks` value will reach `0` when the value of the master's `ticks` reaches the `offset` value. For example, if you want a slave clock's `ticks` value to be 0 when the value of `ticks` for the master clock reaches 1000, you set the slave's `offset` value to 1000. Setting a slave's `offset` instance variable will affect the values in its `ticks` and `time` instance variables. If the offset is positive, these other values will be negative; for example, setting a clock's `offset` to 10 sets its `ticks` to -10. If the offset is negative, the other values will be positive; that is, setting the `offset` to -10 sets its `ticks` to 10.

When a slave in a hierarchy has its rate set to 0, the slave won't run even if its master is running. If the slave's rate is subsequently set to some nonzero value, the clock and its slaves will synchronize to the master using `offset` to determine the relationship of their start time to their master.

In general, the relationship between the time of a master clock and its slave can be expressed with the following formula:

(slave.time/slave.scale) = (master.time-slave.offset) /master.scale

Setting the offset is only meaningful for a slave clock. If you attempt to set the offset of a top clock, ScriptX reports an exception.

Synchronizing Clocks in a Hierarchy

A clock with a rate of 0 is stopped. In a timing hierarchy, when a master's rate is set to 0, the effective rate of all of its slaves also becomes 0—and they stop too. This effect allows a single master clock to synchronize the timing of a whole hierarchy. Using a master clock in this way, the timing hierarchy shown in the previous examples could be implemented yet another way: with three slaves and a single control clock to synchronize their starting and stopping. Figure 6-5 illustrates this third way of implementing the timing hierarchy.

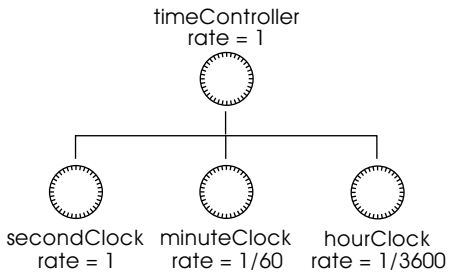


Figure 6-5: A master clock with three slaves

The following code would be used to set up this hierarchy:

```
timeController := new Clock
secondClock := new Clock masterClock:timeController
secondClock.rate := 1
minuteClock := new Clock masterClock:timeController
minuteClock.rate := 1.0 / 60.0
hourClock := new Clock masterClock:timeController
hourClock.rate := 1.0 / 3600.0
timeController.rate := 1
```

When it's created, the `rate` instance variable of `timeController` is set to `0`—the default value for a clock created without a master. Thus, when `secondClock`, `minuteClock`, and `hourClock` are connected to `timeController`, their effective rates become 0. Once all the slaves are connected, `timeController` is started by setting its `rate` instance variable to `1`. This changes the effective rates of the other clocks to the actual value in their `rate` instance variables.

Usually, a script sets other instance variables of a clock—its slaves, master, scale, actions, and so on—before setting its rate. In a timing hierarchy, you set the rates of the slave clocks to prepare them to keep time, then set the master clock's rate to a value other than 0 to start all clocks in the hierarchy.

Timing Hierarchies and Clock Behavior

Certain `Clock` methods and instance variables related to timing are implemented to take the timing hierarchy into account. When you call these methods or set these instance variables for a particular clock in the hierarchy, that clock recursively performs the same action on its slaves (which in turn propagate the action further down the hierarchy). With objects of the `Clock` class, this recursion applies to the instance variables `rate`, `time`, `offset`, and `ticks`. For example, when you explicitly set `time` for a master clock in a hierarchy, the action recurses down the hierarchy, setting the `time` instance variables of all slaves to reflect the new time. Each slave uses the value of its `rate` and `offset` instance variables to calculate the new value for its `time` instance variable.

Generally, a timing hierarchy can be made up of instances of any subclass of `Clock`, including `Player`, `MediaStreamPlayer`, `TransitionPlayer`, and and so on. Any methods defined by the `Clock` class can be invoked with any of its subclasses without untoward side effects. However, within a single timing hierarchy all slaves must be of the same class as, or a descendant class of, their masters. `Clock` and its subclasses implement the instance methods `clockAdded`, `clockRemoved`, and `isAppropriateClock` to check for the appropriate relationship when clocks are moved around in a hierarchy. These methods ensure that method calls and instance variable settings that recurse down the timing hierarchy will be handled properly by all objects.

When a clock's master clock is changed, it attempts to preserve its effective rate, adjusting its `rate` instance variable if necessary. This assures that actual ticks per second produced by the clock will remain constant. Thus, if a clock's previous effective rate was 4, and the clock is connected to a new master whose

effective rate is 1, the clock's `rate` instance variable is set to `4`. If the new master's effective rate is 0, the `rate` for the new slave is set to `1`, regardless of previous effective rate.

Note – When you load a timing hierarchy from a storage container, the top clock's rate is always set to 0, even if the clock was running when stored.

Modeling with Timing Hierarchies

Timing hierarchies are useful for modeling various complex behaviors, and for coordinating the interaction between those behaviors.

One example, implemented through the `Player` class and its subclasses, is the synchronization of virtual players that control the playback of time-based media from various sources. Media-specific players provide timed presentation of audio, video, and animation, using their scale to represent the appropriate media sampling rate. A master player can control a set of these players as a single unit, synchronizing their starting, playback, and stopping. For more on the Media Players component, see the chapter "Media Players."

Timing hierarchies can also be used to implement complex mechanical models. For example, the model of an internal combustion engine might use one clock to drive the crankshaft and pistons, a second to drive camshafts and distributor, others to drive the fuel pump, oil pump, and other accessories, and a master clock to control all the others. Another example of a timing model might be a refinery or other process control system, where distinct parts of the process require different flow rates, sampling rates, rates of heat exchange, and so on.

Timing hierarchies can also be useful for altering the sense of time within a model. For example, a solar system model might use separate clocks for the orbits and rotations of planets and moons, while a master clock contracts the overall sense of time to make motion of the whole visible to the user. Or, an atomic model might use a timing hierarchy to expand the overall sense of time, making the motion of individual electrons around the nucleus available for presentation and interaction in real time.

Clocks Created Automatically by ScriptX

As mentioned previously, every `Space` object has a `clock` instance variable, and a clock is created for each space when it is created. This clock can be used as the top clock for all clocks and timing hierarchies in the space, including player hierarchies managing time-based presentations, hierarchies managing time-based models, and so on.

A global instance of `Clock`, `theEventTimeStampClock`, is created at system start-up. Events use this clock as the source of timestamps that specify the chronological order in which events are generated. A global instance of `CalendarClock`, `theCalendarClock`, is created at system start-up. This

clock's `time` instance variable contains a `Date` object that always reflects the current date and time. You cannot create other instances of `CalendarClock`; instead, use this global instance for current calendar information.

The following code returns the current time and date:

```
theCalendarClock.date
```

The returned value will look something like:

```
Mon Oct 30 18:08:48 1995 as Date
```

Clocks and TitleContainers

To store a clock in a title container, you add it or a master clock to a `TitleContainer`. You need only add the top clock to the title; a slave clock is automatically associated with the title of its master. See the chapter "Title Management" for more on title containers.

The `Clock` instance variable `title` specifies the `TitleContainer` instance to which a clock belongs. (Note that setting this instance variable does not cause the clock to be stored in the title container.)

Note – Creating clock hierarchies that span title containers isn't recommended, since it can lead to inconsistent timing. For example, pausing a title will result in pausing only part of such a hierarchy.

When you call the `TitleContainer` methods `pause` and `resume`, the same methods are automatically called on the active clocks belonging to that title. The `pause` and `resume` methods "freeze" a title's clocks, stopping them without affecting the setting of their `rate` instance variables. This enables you to control the overall operation of a title, without altering the timing relationships you've established in its timing hierarchies.

Using Callbacks To Schedule Actions

To control behavior of objects over time, you attach callbacks to clocks. Callbacks are objects that call scripted functions at specific events or times in the life of a clock.

Callbacks can be scheduled to run a script at a specific time, to run a script at specific intervals, and to run specific scripts whenever the rate, scale, or time of a clock changes.

Types of Callbacks

A number of specialized callbacks are defined by subclasses of `Callback`. Each of these can be attached to a clock using an appropriate method, such as `addPeriodicCallback` and `addTimeCallback`.

Table 6-1: Callback subclasses

Callback subclass	Method to create the callback	When the Script runs
`PeriodicCallback`	`addPeriodicCallback`	At specified intervals
`TimeCallback`	`addTimeCallback`	At specified time
`RateCallback`	`addRateCallback`	When the clock's `rate` is set
`ScaleCallback`	`addScaleCallback`	When the clock's `scale` is set
`TimeJumpCallback`	`addTimeJumpCallback`	When the clock's `time` is set

The callback classes `PeriodicCallback` and `TimeCallback` are associated with time changes in the clock as it runs. The other three classes — `RateCallback`, `ScaleCallback` and `TimeJumpCallback` — are associated with directly setting a `Clock` object's instance variables.

The callbacks that are most commonly used in titles are `PeriodicCallback` and `TimeCallback`. The other callbacks are used in very specific situations that are not so common.

Attaching Callbacks to a Clock

To add a callback to a clock, you call one of the following methods on the clock depending on the kind of callback desired.

- `addPeriodicCallback` — adds a `PeriodicCallback` that fires repeatedly, at periodic intervals. The syntax is:

  `addPeriodicCallback` *clock script target argArray time*

- `addTimeCallback` — adds a `TimeCallback` that fires once at a specified time. The syntax is:

  `addTimeCallback` *clock script target argArray time onceOnly*

- `addRateCallback` — adds a `RateCallback` that fires when the value of the `rate` instance variable of a clock changes. The syntax is:

  `addRateCallback` *clock script target argArray onceOnly*

- `addScaleCallback` — adds a `ScaleCallback` that fires when the value of the `scale` instance variable of a clock changes. The syntax is:

  `addScaleCallback` *clock script target argArray onceOnly*

- `addTimeJumpCallback` — adds a `TimeJumpCallback` that fires when the `time` instance variable of a clock is specifically set. The syntax is:

  `addTimeJumpCallback` *clock script target argArray onceOnly*

The Arguments for the Callback-Creation Methods

The *clock* argument is the clock to which to attach the callback.

The *script* is a function, anonymous function, or method that defines the action to be invoked by the callback.

The *target* is the first argument to be passed to the script.

The *argArray* is an array of the remaining arguments. If no additional arguments are required, you can pass *argArray* as an empty array.

For a periodic callback, *time* is the time interval at which the callback fires. For a time callback, *time* is the clock's time at which the callback fires.

The *onceOnly* argument is either `true` or `false`, indicating whether or not the callback should be cancelled after it has executed the first time. You would usually pass this as `false`, so that if you restart the clock, the callback fires each time the clock reaches the appropriate time.

Callback Scripts

The script associated with a callback can be a predefined function, an anonymous function, or a method. The function or method must take at least one argument, and can have as many additional arguments as you like.

There are a couple of details to be aware of in defining callback scripts. First, calls that cause the calling thread to block shouldn't be made within a callback script. All callback scripts with the same priority run in the same thread, so any script that blocks will prevent other callbacks of the same priority from executing their scripts. See the section on "Blocking" in the "Threads" chapter for a list of functions that can cause thread blocking. The `PipeClass` methods `readNowOrFail` and `writeNowOrFail` can be used to pipe data in code that shouldn't block.

Second, if a callback script generates an exception when it is invoked, it will be called again the next time it is scheduled. This means that care must be taken within callback scripts to trap exceptions in a way that ensures the script isn't simply scheduled to fail repeatedly.

Examples of Creating Callbacks

The following examples show how to add simple callbacks to `clock1`:

```
clock1 := new clock
```

Using a Predefined Function for the Callback Script

This example adds a periodic callback to the clock `clock1`. The callback's script is a predefined function.

```
function printTime clock1 -> (
    format debug "Tick Tock. The clock's time is %*. \n" \
        clock1.time @unadorned
)

-- every 3 ticks, print the clock's time
addPeriodicCallback clock1 printTime clock1 #() 3
```

Using an Anonymous Function for the Callback Script

This example adds a time callback to the clock `clock1`. The callback's script is an anonymous function.

```
-- at time 15, print the real time
addTimeCallback clock1 \
    (a -> format debug "The global time is %* \n" \
        theCalendarClock.date @unadorned) \
        clock1 #() 15 false
```

Using a Method for the Callback Script

This example creates a class called `Slogan`, and and creates an instance of it. The example adds a periodic callback to to `clock1`. The callback's script is a method on the class `Slogan`.

```
class Slogan (RootObject)
instance variables
    mySlogans: #()
    myIndex:1
end

method pickASlogan self {Class Slogan} clock1 ->
(
    local slogans := self.mySlogans
    local myIndex := self.myIndex
    print slogans[myIndex]
    self.myIndex :=
        if myIndex >= slogans.size then 1 else myIndex + 1
)

-- create a Slogan object
slogan1 := new Slogan
append slogan1.mySlogans "Better late than never"
append slogan1.mySlogans "A stitch in time saves nine"
append slogan1.mySlogans "A watched pot never boils"
append slogan1.mySlogans "The grass is always greener on the other
side"

-- print a slogan every 6 ticks
addPeriodicCallback clock1 pickASlogan slogan1 #(c1ock1) 6
```

Start the Clock Ticking

```
-- start the clock ticking
c1ock1.rate := 1

-- after the clock has been ticking for a while,
-- set it back to the beginning, ready to start over
clock1.time := 0
```

Callback Conditions

The `Callback` class has a `condition` instance variable, that helps to determine the conditions under which a callback should be invoked.

Three kinds of callbacks are triggered by changes to the value of an instance variable on the callback's associated clock. These classes are `RateCallback`, `ScaleCallback`, and `TimeJumpedCallback`.

The `RateCallback` class has a `rate` instance variable; the `ScaleCallback` class has a `scale` variable, and the `TimeJumpedCallback` class has a `time` instance variable. These instance variables are used to compare the value belonging to the callback with the value of the corresponding instance variable of the clock itself.

The `condition` instance variable specifies how to compare the clock's instance variable with the callback's. Valid `condition` values for these callbacks are `@lessThan`, `@greaterThan`, `@equal`, `@notEqual`, `@lessThanOrEqual`, `@greaterThanOrEqual`, and `@change` (the default).

The following code demonstrates how to set a condition for a `ScaleCallback` instance:

```
function scaleScript target -> (print "Scale less than 20")
myScaleCall := addScaleCallback myClock scaleScript
    undefined #() false
myScaleCall.scale := 20
myScaleCall.condition := @lessThan
```

In this example, whenever the `scale` instance variable of `myClock` is set to a value less than 20, the `myScaleCall` callback is invoked, causing the `scaleScript` function to run.

The `PeriodicCallback` and `TimeCallback` classes also use the condition instance variable to help determine whether to invoke a callback or not at a particular time. For these two classes, the valid settings for `condition` are `@forward`, `@backward`, and `@either` (the default). The following code demonstrates how to set a condition on a `TimeCallback`:

```
function timeScript target ->
           (print "Time of 200 running backward")
myTimeCall := addTimeCallback myClock timeAction \
    undefined #() 200 false
myTimeCall.condition := @backward
```

In this case, the `timeScript` function will be called only when the clock reaches a time of 200 while the clock is running backward.

Priority and Order

The `Callback` class has two instance variables—`priority` and `order`—that determine the precedence of otherwise concurrent callbacks.

The `priority` instance variable expresses the thread priority of a callback. Valid settings for priority are `@high` and `@normal`. These settings correspond to values defined by the Threads component. For more information on thread priorities, see the chapter "Threads" in this guide.

In practice, you seldom set a callback's priority to any value other than `@normal`. Setting the priority to `@high` may slow overall performance by interfering with the ScriptX runtime environment.

Another instance variable, `order`, can be used to set the order in which callbacks of a particular priority will have their scripts invoked. Lower order callbacks have their scripts invoked first, and higher order callbacks have their scripts invoked later. By default, the order of a callback is 0. You can set the order of callbacks within a title to ensure that scripts that need to be invoked first will be.

Note – In the current release of ScriptX, callback order works correctly only for clocks with integer rates. Fractional rates can lead to inconsistences in the order in which callback actions are actually performed.

Synchronization of Periodic Callbacks

The `PeriodicCallback` class defines an instance variable, `skipIfLate`, that determines how to handle situations when a callback fails to occur within its scheduled time frame. This instance variable can help avoid a situation where late callbacks, which take precedence over others, choke overall system performance.

When `skipIfLate` is `false`, a periodic callback will have its script called regardless of whether it is on time or late. Thus, one callback may get out of sync with its scheduled time and the next one may catch up. On the other hand, one callback may fall behind and the next fall even further back. There is no guarantee that such a sequence will ever resynchronize. Instead, it may aggravate the situation further by preventing other callbacks from executing their scripts.

When `skipIfLate` is `true`, a periodic callback will have its script called only within its scheduled time period. That is, a periodic callback's script may begin execution at any time starting at its scheduled time up until the next time it is scheduled to be called. However, if the script hasn't begun execution by the next time it is scheduled to start, it will be skipped. This avoids the problem of late callbacks choking system performance.

When creating the function for a periodic callback whose `skipIfLate` value will be `true`, you should provide a mechanism for skipping. For example, rather than simply assuming the function will be called once for each specified interval, you should check the clock's time within the function, then perform an action appropriate to that time.

Cancelling a Callback

Once you've created a callback, you can let it run as scheduled, or you can cancel it explicitly using the `Callback` method `cancel`. This method allows the action function to return if it's in process, then removes the callback from its clock and returns the cancelled callback. The following code illustrates how to use the `cancel` method.

```
myCB := addTimeCallback myClock printInfo myClock #(myObject) 25 false
-- run the clock for a while
-- now it is time to cancel the callback
cancel myCB
```

The following code shows how to use the `label` instance variable of a callback to identify which callback to cancel.

```
function doStuffFn myClock myObject ->
(
    local myCB := addPeriodicCallback myClock deepenColor \
           myClock #(myObject) 25
    myCB.label := "cb1"
    -- do more things in the function if desired
)

-- write some code that creates the objects clock1 and pic1

-- create the callback
doStuffFn clock1 pic1

-- later on, cancel the callback
-- use chooseOne to find the callback with the apprioriate label
cancel (chooseOne clock1.callbacks (a b -> a.label = "cb1") 1)
```

Uncancelling a Cancelled Callback

After a `PeriodicCallback` or `TimeCallback` has been cancelled, you can reschedule it by putting the desired clock in the callback's `clock` instance variable, and putting the invocation time (or time period) in the callback's `time` instance variable.

When you cancel a callback, it retains the information about which clock it is attached to, so you do not need to reset the `clock` instance variable unless you want to attach it to a different clock. To reschedule a time callback or periodic callback however, you must set its `time` instance variable, since it is the act of setting the `time` instance variable that schedules the callback.

For example, if `cb1` is a cancelled periodic callback, you can wake it up and reschedule it so that it fires every 5 ticks as follows:

```
cb1.time := 5
```

Callbacks and Clock Behavior

Setting a clock's instance variables or calling its methods may affect the callbacks belonging to a clock. For example, if you attach a time callback to a clock, then set the clock's `time` explicitly to jump past the time of the callback, the action function won't be invoked. This means you might want to associate a `TimeCallback` with a `TimeJumpCallback` to assure that required actions take place even if the clock's time is set explicitly.

Through the timing hierararchy, other clocks may affect a clock's callbacks. For example, you might attach a rate change callback to a clock, then change the rate of a clock above it in the hierarchy. In this case, the rate callback script for the lower clock will be invoked—if the change matches the callback conditions. Similarly, if you set a time jump callback on the lower clock, then change the time of a clock above it, the lower clock's time jump action will be invoked—again, if the condition matches the change.

Callback Example

While clocks can be used for both control of models and the timed playback of media, much of the functionality needed for media playback is provided through other components, such as Media Players and Animation. See the discussions of those components for additional examples of time-based behavior.

The following example shows how to create a simulation of a kettle heating up until it boils. This example illustrates the use of callbacks to time a sequence in ScriptX, where the timing requirements might vary each time the sequence is enacted. It also demonstrates how you can use one callback to schedule another, and how to clean up callbacks to prevent callback-proliferation in an animation that creates callbacks on the fly.

This example uses callbacks as follows:

- It creates a `TimeCallback` to schedule when to start heating up the kettle, and how many cups of water to heat.
- When the time arrives to start heating the kettle, another `TimeCallback` schedules when the kettle will actually boil. The time until the boiling depends on how many cups of water the kettle is heating.
- While the kettle is heating up, a `PeriodicCallback` causes the kettle to hum continuously.

  For the sake of simplicity, in this script, the humming is simulated by printing out "hum" repeatedly. However, in a more intersting simulation, the kettle might play a humming sound over and over, or the simulation might present a visual animation of the kettle rocking back and forth gently on the stovetop as it heats up.
- When the time arrives for the kettle to actually boil, the periodic callback that causes the humming is cancelled, and the `whistle` method is called to make the kettle whistle.

As with the humming, the whistling action is simulated by printing text, in this case "Whistle." In a more interesting animation, the whistling might be simulated by playing a whistling sound, or maybe the humming sound increases in pitch when the kettle boils. Also, the kettle image might rock back and forth on the stove top more violently, and give off puffs of steam.

The `whistle` method might also schedule another callback, to make the animation continue.

Define the Class Kettle

The class `Kettle` has an instance variable that specifies how long a kettle takes to heat up one cup of water.

```
class Kettle (RootObject)
instance variables
    heatingTimePerCup:3
end
```

scheduleKettle method

The `scheduleKettle` method creates the time callback that schedules when to put the kettle on to start heating. This method simply provides a cover to the `addTimeCallback` function.

```
-- clock1 is the clock
-- cups is the number of cups of water to heat
-- t is the number of ticks from now to start the kettle
-- onceOnly is true or false depending on whether or not you
-- want the callback to cancel itself after its first invocation

method scheduleKettle self {class Kettle} \
    clock1 cups t label onceOnly ->
(   addTimeCallback clock1 putOnKettle self #(cups, clock1) \
        (clock1.time + (t as time)) onceOnly
)
```

putOnKettle method

The `putOnKettle` method prints the time when the kettle is put on to heat. This method calculates how long the kettle will take to boil, based on how many cups of water are being heated, and how long each cup takes. It schedules a callback to make the kettle boil, and calls the `heatUpKettle` method to start the kettle heating.

```
method putOnKettle self {class Kettle} cups clock1 ->
(
    local t := clock1.time
    format debug "Putting on the kettle at time %*\n" t @normal
    local timePerCup := self.heatingTimePerCup
    local timeToBoil := (timePerCup * cups) as time

    -- create a callback to schedule when the kettle boils
    -- the callback will cancel itself after it is invoked
```

```
    addTimeCallBack clock1 startBoiling self \
       #(clock1) (clock1.time + timeToBoil) true

    -- start heating the kettle
    heatUpKettle self clock1
)
```

heatUpKettle method

While the kettle is heating up, it hums. The humming is simulated by printing out "hum" repeatedly.

```
-- use the prin method as the action for the
-- callback that does the humming
method heatUpKettle self {class Kettle}  clock1 ->
(
    print "Starting to heat up the kettle "

    local cb := addPeriodicCallback clock1 \
       prin "hum \n" #(@unadorned, debug) 1

    -- give the callback a label so we can find it later
    cb.label := "hum"
)
```

startBoiling method

When the kettle starts boiling, it stops humming and starts whistling. To stop the humming, cancel the periodic callback that does the humming. To start whistling, call the `whistle` method.

```
-- when the kettle boils, it whistles
method startBoiling self {class Kettle} clock1 ->
(
    local t := clock1.time
    format debug "Starting to boil at time %*\n" t @normal

    -- cancel the callback that does the humming
    local humCB := chooseOne clock1.callbacks \
       (a b -> a.label = "hum") 1
    cancel humCB

    -- the kettle whistles
    whistle self
)
```

For simplicity, simulate whistling by printing a whistle message.

```
method whistle self {class Kettle} ->
(print "Whistle, whistle, whistle!")
```

Run the Kettle Simulation

To run the kettle simulation, create a clock and a kettle. Call the `scheduleKettle` method to schedule when to start the kettle heating up. When you call this method, specify how many cups of water to heat.

```
global clock1 := new Clock
clock1.rate := 1

-- put the kettle on in five seconds to heat 3 cups of water
-- since OnceOnly (the last argument) is true,
-- the callback will cancel itself after it is invoked

global kettle1 := new Kettle
scheduleKettle kettle1 clock1 3 5 "cb1" true

-- schedule the kettle to be put on again in 20 seconds
-- to heat 6 cups of water
-- again, the callback will cancel itself when it is finished
scheduleKettle kettle1 clock1 6 20 "cb1" true
```

C H A P T E R

Players

7

The Players component contains the abstract `Player` class, which provides facilities for classes that present data in a sequential manner. ScriptX provides three kinds of players:

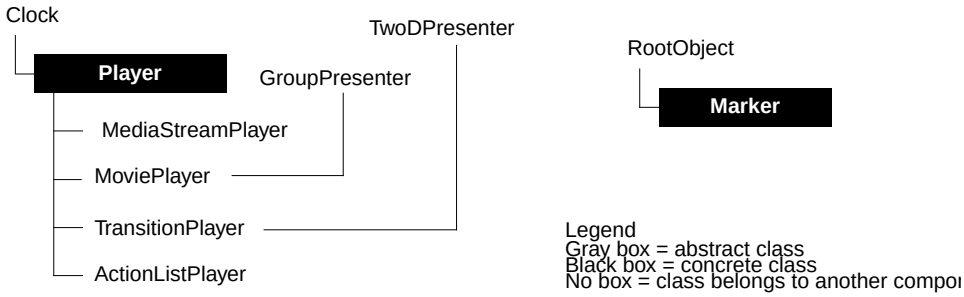
- Media player classes provide facilities for presenting media streams, such as sound and video streams.
- The `ActionListPlayer` class provides facilities for playing an animation by performing a sequence of actions in a list.
- The `TransitionPlayer` class provides facilities for presenting an image in stages so that the image appears gradually.

This chapter discusses concepts and behavior common to all kinds of players. The following chapters discuss concepts and behavior particular to each kind of player:

- Chapter 8, “Media Players” for information on the media players classes, including media stream players and movie players.
- Chapter 9, “Animation” for information on action list players.
- Chapter 10, “Transitions” for information on transition players.

Classes and Inheritance

The class inheritance hierarchy for the Players component is shown in the following figure.



The following classes form the Players component. In this list, indentation indicates inheritance.

`Player` – base class that defines the basic methods common to players.

> `MediaStreamPlayer` – specialized subclass of `Player` that presents media held in an associated media stream. `MediaStreamPlayer` has subclasses specialized to play particular types of media.

`MoviePlayer` – specialized subclass of `Player` that organizes media stream players to play together to present a movie.

`TransitionPlayer` – subclass of `Player` that performs visual transitions on a presenter. `TransitionPlayer` has subclasses specialized to present different kinds of transitions.

`ActionListPlayer` – subclass of `Player` that plays an animation by performing a series of actions.

`Marker` – class whose instances identify important time ranges for a player.

Conceptual Overview

The `Player` class defines the methods needed to control all kinds of players. These methods allow you to stop, start and pause the data, and perform other actions needed to present time-based data.

These methods include:

- `play`

  Starts a player presenting its data. The `rate` instance variable is set to 1.

- `playUntil`

  Causes the player to present its data until a given time. The `rate` instance variable is set to 1 while the player is playing, then changes to 0 when the specified time is reached and the player stops.

- `pause`

  Stops the player presenting its data by setting its `rate` instance variable to 0, and sets its `status` instance variable to `@paused`.

- `resume`

  Resumes the player playing at the same rate it was playing before it was paused. If you call the pause method on a player multiple times, you must call the `resume` method an equal number of times before it resumes playing.

- `stop`

  Stops the player presenting its data, sets the `rate` instance variable to 0, and sets the player's `status` instance variable to `@stopped`.

- `gotobegin`

  Sets the player's data back to the beginning by setting the `time` instance variable to 0. Use this method rather than rewind to set a player back to the beginning. This method does not change the rate of the player.

- `gotoend`

  Sets the player to the end of its data by setting the `time` instance variable to the value in the player's `duration` instance variable.

- `fastForward`

  Speeds up the rate at which the player presents its data to five times the normal speed.

- `rewind`

  Causes the player to present its media in the reverse direction at five times the normal rate.

- `playPrepare`

  Prepares a player for playing. Some players allocate resources and pre-roll data during this method. If a player is not prepared before playing, these operations are done when the `play` method is called, which may result in a delay before playing starts.

Some subclasses of `Player` have additional methods for controlling specific kinds of media or sequences, for example, the `DigitalAudioPlayer` class has methods for controlling the volume of sound.

Players inherit the ability to be synchronized from the `Clock` class. For example, you could synchronize an `ActionListPlayer` object that plays an animation with a `DigitalAudioPlayer` object that plays a tune so that they play, stop, and rewind together. Often it makes sense to create a completely new player whose sole task is to control other players.

A player that controls another player is a "master player."

How Players Work

Depending on the kind of player, it may work in conjunction with another object or set of objects that contain the data to be presented. For example, a digital audio player presents sound data held in an audio stream, and an action list player performs the actions contained in an action list.

Whether or not a player is presenting its data (that is, playing) depends on its rate. If its rate is 0, it is stopped. If its rate is other than 0 it presents its media at the speed and direction specified by the rate. The methods `play`, `playUntil`, `pause`, `stop`, and `rewind` change the rate of the player.

The local rate is specified by the `rate` instance variable, but the actual rate of a player depends on its effective rate, which takes its master player's rate into account, if it has one. See the Chapter 6, "Clocks" for information about how effective rates are determined.

For a player, the value of the `time` instance variable indicates the current position in the media presented by the player. You can directly set the value of the `time` instance variable to change the position of the player's media. The `time` instance variable can be set to a number of ticks or to a `Time` value. If the value is given as a number of ticks, it is interpreted in the scale of the player. For example, if a player has a scale of 30, and you set its time to 60:

```
myplayer.time := 60
=>0:0:2:0
```

The time is set to 2 seconds. (See Chapter 6, "Clocks," for more information about scale.)

You could also set the time to 2 seconds as follows:

```
myplayer.time := 2 as time
=>0:0:2:0

myplayer.time := 2 * myplayer.scale
=>0:0:2:0
```

Using Multiple Players

Multiple players of all kinds can be synchronized so that they can be controlled by a single player.

To specify one player as a master player of another, put it in the other player's `masterClock` instance variable. (This instance variable is inherited from `Clock`.)

```
player1.masterClock := masterPlayer
```

When multiple players are slaved to a single master player, you can start all the players playing in synchronization by calling the `play` method on the master player. Similarly you can simultaneously prepare for playing, pause, stop, fast forward, rewind, set to beginning and set to end all slave players by calling the appropriate method on the master player.

Setting the value of the `time` instance variable of a master player also automatically sets all its slave players to an appropriate value that takes offsets into consideration. Offsets are discussed in "Specifying Different Start Times For Slave Players" on page 171.

To make all the slave players play louder, set the value of the `globalVolumeOffset` on the master player. To change the pan of all the slave players, set the `globalPanOffset` instance variable on the master player. To mute the audio of all the slave players, set the `audioMuted` instance variable on the master player to `true`.

The effective volume for a slave player is the value of its `volume` instance variable combined with the value of its `globalVolumeOffset` instance variable. So for example, if the local volume of a player is -6 and the global volume offset is 6, the effective volume is 0. Similarly, the effective pan for a slave player is the value of its `pan` instance variable value combined with the value of its `globalPanOffset` instance variable.

The `globalVolumeOffset, globalPanOffset` and `audioMuted` instance variables only affect players that have a `volume` or `pan` instance variable, such as digital audio players. Other players are not affected by changes to these instance variables.

Note – In the current release, the `Player` class has instance variables such as `globalBrightness`, `globalContrast`, `globalHue` and `globalSaturation`. The intent of these instance variables is that the value on a master player would affect the effective values for brightness, contrast, hue, and saturation on its slave players. However, currently no subclasses of

`Player` have `brightness`, `contrast`, `hue`, and `saturation` instance variables. Thus changing the value of `globalBrightness`, `globalContrast`, `globalHue`, and `globalSaturation` of a player has no effect. These instance variables have been left on the `Player` class in anticipation of corresponding local instance variables in future releases of ScriptX.

The following script gives an example of synchronizing two slave players to a master player.

```
-- animationPlayer is a pre-defined actionListPlayer instance
-- tunePlayer is a pre-defined digitalAudioPlayer instance

-- Create a player to use as the master player
global master := new Player

-- Make animationPlayer and tunePlayer be slave players of master
animationPlayer.masterClock := master
tunePlayer.masterClock := master

-- prepare to play the sound and animation
playPrepare master 1
goToBegin master

-- start the sound and animation playing together
play master

-- stop both the sound and animation from playing
stop master

-- switch off the sound (make it mute)
master.audioMuted := true

-- start the animation playing silently
play master

-- switch the sound back on
master.audioMuted := false

-- stop the sound and animation from playing
stop master

-- set both the sound and animation to time 8
master.time := 8

-- resume playing the sound and animation from time 8
play master
```

Specifying Different Start Times For Slave Players

You can specify different start times for slave players that have the same master player. For example, you might want to play an action list player and a digital audio player together, but you want the digital audio player to play for four seconds before the animation begins.

To specify a start time delay for a slave player, set the value of its `offset` instance variable. The value of the `offset` instance variable of a slave player is the amount of time by which the player is delayed relative to its master player.

For example:

```
master := new Player
animationPlayer.masterClock := master
goToBegin master
animationPlayer.offset := 4
```

The value of the `time` instance variable of `animationPlayer` now becomes -4 seconds. When you call the `play` method on the master player, `animationPlayer` starts incrementing its clock, but does not start playing its media until the value in its `time` instance variable reaches 0, which will be when the value of the `time` instance variable of its master player is 4 seconds. (For the rest of this discussion, the time of a player means the value of its `time` instance variable.)

When you set the value of the `offset` instance variable of a slave player, the value is assumed to be a number of ticks in the scale of the player's master player. If you create a direct instance of `Player` to use as the master player, its scale will be 1.

Since the master's scale is used as the scale for offsets you can use a master player to establish the main timeline, and then specify offsets for slave players relative to the main timeline.

If you attempt to set the value of the `offset` instance variable of a player that does not have a master player (or master player) the value is ignored. The value of the `offset` instance variable of all clocks and players that do not have a master player is always zero.

Playing Slave Players at Different Rates

You can cause players that use the same master player to play at different rates by specifying values for the `rate` instance variables of the slave players, as illustrated in the following code.

```
master := new Player
slowPlayer.masterClock := master
fastPlayer.masterClock := slowPlayer.masterClock
slowPlayer.rate := 1
fastPlayer.rate := 2
goToBegin master
play master
-- slowPlayer plays at a rate of 1 while
-- fastPlayer plays at a rate of 2
```

Troubleshooting Hints – The Case of the Stopped Slave Player

If you call the `stop` method directly on a slave player instead of calling it on the master player, the slave player stops. It will not start again when you call the `play` method on its master player.

The effective rate of a slave player is the value of its `rate` instance variable times the value of its master player's `effectiveRate` instance variable. If you call the `stop` method directly on a slave player, its `rate` instance variable is set to 0. Calling `play` on its master player does not start the slave player playing, since the effective rate of the slave player remains at 0. (0 times any number is 0).

If you find yourself in this situation, you can fix it by calling `play` directly on the slave player once to start it playing, or set the `masterClock` of the slave player over again. When the `masterClock` instance variable of a player is set, the slave player's effective rate is preserved, unless its rate is 0, in which case the rate is set to 1.

Using Markers

You may sometimes want to mark certain places in the data controlled by a player. For example, you may want to mark the beginning of each verse in a song, or mark specific time points in an animation.

You can use `Marker` instances to identify interesting time points for a player. A marker is simply an object that has a starting time, a finishing time and a label. When creating a marker, you must specify a beginning time and a label and you can optionally specify an ending time.

The `Player` class has methods `gotoMarkerStart` and `gotoMarkerFinish` that advance or rewind the player to places marked by a marker.

For example, suppose you want to play two short parts of a song played by a `DigitalAudioPlayer`. You can do this by using markers to mark the two ranges.

To find the start time for the first marker, play the player, and stop it just before the first word of the first range is played. Get the time of the player:

```
start1 := myPlayer.time
```

To find the finish time for the first marker, play the player and stop it as soon as the you hear the word at the end of the first range. Check the time on the player:

```
finish1 := myPlayer.time
```

Create the first marker:

```
marker1 := new Marker start:start1 finish:finish1 \
    label:"Marker 1"
```

In a similar fashion, find the start and finish times for the second marker and create the second marker.

Add the markers to the player:

```
addMarker myPlayer marker1
addMarker myPlayer marker2
```

To play the song from the start to the end of the time marked by the first marker:

```
goToMarkerStart myPlayer marker1
playUntil myPlayer marker1.finish
```

To play the part of the song marked by the second marker:

```
goToMarker myPlayer marker2
playUntil myPlayer marker2.finish
```

Accessing Markers in the Marker List

A player's `markerList` instance variable contains an array of the player's markers sorted by start time. You can use a player's `getPreviousMarker` and `getNextMarker` method to return the marker before or after a given marker in the player's marker list.

The following code shows how to play a player from the beginning to the end of its first marker, then play it from the beginning to the end of its second marker, without explicitly referring to variables pointing to the markers.

```
-- myPlayer is a media player with at least two markers

-- The first marker is the first element in the player's marker list
m := myPlayer.markerlist[1]

-- Play the player from the start to end of the first marker
goToMarkerStart myPlayer m
playUntil myPlayer m.finish
```

```
-- Play the player from the start to the end of the next marker
m := getNextMarker myPlayer m
goToMarkerStart myPlayer m
playUntil myPlayer m.finish
```

The start and finish values for a marker are always interpreted in the scale of the player to which the marker is added. For example, if you add a marker whose start time is 20 to a player whose scale is 1, then the player reaches the start of the marker at 20 seconds. However, if the player's scale is 100, then the player reaches the start of the marker at one fifth of a second.

Using Marker Labels

A marker can have a label, which is a string stored in its `label` instance variable. For an example that illustrates the usefulness of marker labels, suppose you created an animation of the story of Little Red Riding Hood. You could use markers labelled "Wolf arrives", "Wolf eats grandma", "Red riding hood arrives" and "Woodman saves the day" to identify appropriate times in the animation as follows.

```
-- RRPlayer is an actionListPlayer
addMarker RRPlayer (new Marker start:10 label:"Wolf arrives")
addMarker RRPlayer (new Marker start:22 label:"Wolf eats grandma")
addMarker RRPlayer (new Marker start:38 label:"Red riding hood arrives")
addMarker RRPlayer (new Marker start:53 label:"Woodman saves the day")
```

The following sample script shows how to play the animation from the point at which the wolf arrives until the end of the animation.

```
-- use a path expression to find the place where the wolf arrives
global wolfComes :=
chooseOne RRPlayer.makeList(v dummy -> v.label = "wolf arrives") 0
goToMarkerStart RRPlayer wolfcomes
playUntil RRPlayer RRPlayer.duration
```

C H A P T E R

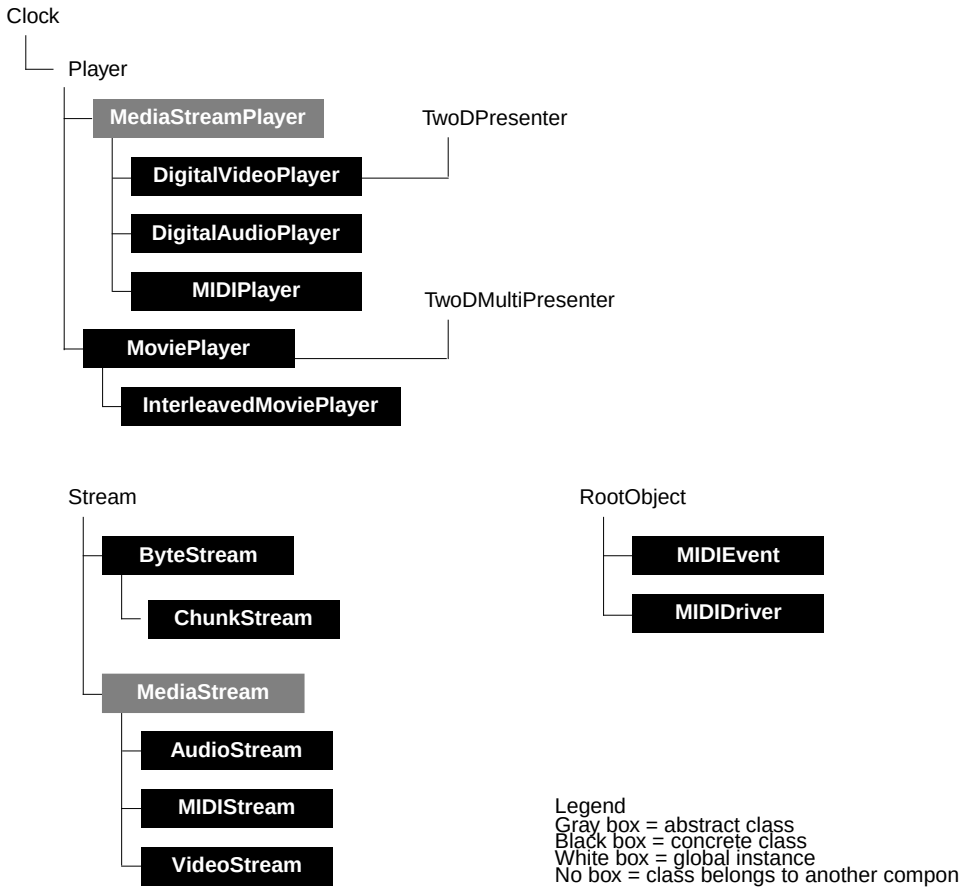
Media Players

8

The Media Players component provides classes for playing media, such as sound and video. The players in this component build on the functionality provided by the `Player` class, discussed in the previous chapter. Please read the previous chapter before reading this one.

Classes and Inheritance

The class inheritance hierarchy for the Media Players component is shown in the following figure.



The following classes form the Media Players component. In this list, indentation indicates inheritance.

`MediaStreamPlayer` – specialized subclass of `Player` that plays media held in an associated media stream.

`DigitalVideoPlayer` -- subclass of `MediaStreamPlayer` that is specialized to play video. The video is held in a stream containing digitized video.

`DigitalAudioPlayer` -- subclass of `MediaStreamPlayer` that is specialized to play sound. The sound is held in a stream containing digitized audio.

`MIDIPlayer` -- a subclass of `MediaStreamPlayer` specialized to play MIDI sound. The sound is held in a stream containing MIDI data.

`MoviePlayer` -- subclass of `Player` that is specialized to play movies. A `MoviePlayer` object controls other players that play the components of a movie.

`InterleavedMoviePlayer` -- subclass of `MoviePlayer` that is specialized to play interleaved movies.

`MediaStream` – base class that encompasses the common behavior of media streams; it is used by `MediaStreamPlayer` instances.

`AudioStream` -- subclass of `MediaStream` specialized for holding a stream of digitized sound.

`MIDIStream` -- subclass of `MediaStream` specialized for holding a stream of MIDI data.

`VideoStream` -- subclass of `MediaStream` specialized for holding a stream of digitized video data.

`ChunkStream` – subclass of `ByteStream` that is used by `InterleavedMoviePlayer` to transfer movie data in a single interleaved stream to separate audio and video streams. (See the description of `ChunkStream` in the *ScriptX Class Reference* for more information.)

`MIDIEvent` - class that holds information about MIDI events, such as the message, note and velocity for a MIDI event.

`MIDIDriver` - class of drivers for MIDI devices.

Conceptual Overview

The classes in the Media Players component can be used to present (or play) media such as movies, video, and sound. The media player classes inherit methods from the `Player` class that are common to all players, such as methods for playing, stopping, pausing, fast forwarding, and rewinding.

The class `MediaStreamPlayer` provides functionality for playing and controlling a single piece of media contained in a media stream. The classes `DigitalAudioPlayer`, `DigitalVideoPlayer` and `MIDIPlayer` play a single piece of media and inherit from the class `MediaStreamPlayer`.

The `MoviePlayer` class uses multiple digital audio players and digital video players to play a movie. The `InterleavedMoviePlayer` class inherits from `MoviePlayer` and has added functionality for playing movies whose data is interleaved in a single stream.

To use digitized media in ScriptX, you must first digitize your data in an appropriate application outside ScriptX. For example, you might digitize a movie and store it as a QuickTime file, or digitize a sound and store it as an AIFF file. After you have created the file containing the digitized media, you can import it into ScriptX using the Import/Export Engine. See the *ScriptX Tools Guide* for full details of importing sound, video and movie data into ScriptX. Note that for playing MIDI, you can either import an existing MIDI file into ScriptX, or you can generate MIDI events from within ScriptX.

After importing media into ScriptX, you can save either the media player or the media stream to a title container. If you do this, you can play the media at a later time, by opening the title container and accessing the media player or the media stream. (If you save the stream without the player, you will need to create a new player to play the stream.) The imported media can be used by media players on any platform that supports ScriptX.

As mentioned in Chapter 7, "Players", you can synchronize players so that a single player can control multiple players, and you can use markers to mark a time range in the media presented by a media player. See "Using Multiple Players" on page 170 and "Using Markers" on page 173.

How Media Players Work

Players that inherit from `MediaStreamPlayer` play a single piece of media. They have an instance variable called `mediaStream` that points to the `MediaStream` object controlled by the player. The stream must be an appropriate kind for the player.

Each media stream in turn has an `inputStream` instance variable that points to the actual stream of data for the media, as illustrated in Figure 8-1.

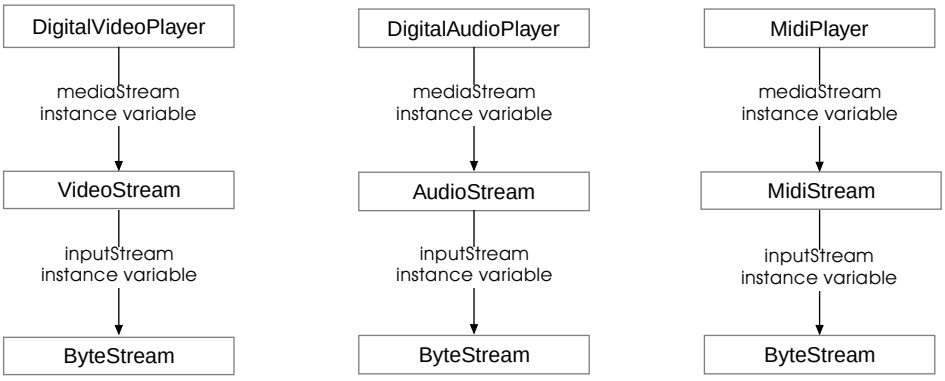


Figure 8-1: A MediaStreamPlayer has a mediaStream with an inputStream.

A `MoviePlayer` object does not play a single piece of media. Instead, it controls other players that are each responsible for playing a component of the movie. Both the `target` and `slaveClocks` instance variables of a movie player contain an array of the other players needed to play the movie.

An `InterleavedMoviePlayer` instance is a specialized `MoviePlayer` that has an `interleavedStream` instance variable that holds a stream containing interleaved data for the movie. See the `InterleavedMoviePlayer` section in

the *ScriptX Class Reference* for a detailed discussion of how interleaved movie players de-interleave their data and send it to separate digital audio and digital video players.

Start and End of the Media Data

All media players assume that their associated media starts at time zero and proceeds through the time stored in the player's `duration` instance variable . As for all players, calling the `play` method on a media player sets its rate to 1 and starts it ticking (that is, the value of the `time` instance variable begins incrementing.) A media player starts presenting its associated media when its time is zero and its rate is positive. It finishes presenting its media when its time reaches the value in its `duration` instance variable. (If the player is playing backward, the reverse is true.) When the player passes the end of its media stream, its clock continues ticking although the media is no longer being presented.

A player could be in the situation where it has a negative time value but a forward rate value. In this case, although the player's clock is ticking it is not presenting its associated media. When the time reaches 0, the media starts playing.

Importing and Saving Media in ScriptX

To get media into ScriptX, you import it from a file containing digitized media data, such as as a QuickTime or AVI file containing a digitized movie, an AIFF file containing digitized sound, or a MIDI file containing MIDI data. To import a media file into ScriptX, call the `importMedia` method on `theImportExportEngine` global instance.

The `importMedia` takes at least five arguments in addition to *self*:

- the source file to be imported
- the type of media
- the specific file type
- the output class
- an optional keyword argument for the title container in which to save the media

The actual values for these arguments depend on the kind of media being imported. The `importMedia` methods also takes extra arguments for importing some kinds of media. See the *ScriptX Tools Guide* for details and examples of importing each kind of media.

When importing digitized audio files, digitized video files, and MIDI files, you can choose whether the importing process should create just a media stream or whether it should also create an appropriate player to play the stream. For movies, the importing process always creates a master player (either a `MoviePlayer` or `InterleavedMoviePlayer` instance) along with all the necessary slave players and streams needed to play the movie.

The following code illustrates how to import an AIFF sound into ScriptX and play it:

```
theStream := getstream thestartdir "song1.aif" @readable
songplayer := importMedia theImportExportEngine theStream\
   @sound @aiff @player
play songplayer
```

If you import media as a media stream only, you also need to create a player to play the media stream. To do this, call the `new` method on the appropriate subclass of `MediaPlayer` and set the player's `mediaStream` instance variable to the media stream, as illustrated in the following code:

```
theStream := getstream thestartdir "song2.aif" @readable
songStream := importMedia theImportExportEngine theStream\
   @sound @aiff @stream
songPlayer := new DigitalAudioPlayer mediaStream:songStream
play songPlayer
```

Saving Imported Media to a Title Container

If you save the imported media to a title container, you will be able include the media in your title. The title will be able to run on any machine that supports ScriptX without needing copies of the original media files.

If you intend to save imported media to a title container, you must specify the `container` keyword to the `importMedia` method. When the media is imported, the raw data for the media goes into the title container, but no objects are saved to the title container.

In addition to specifying the `container` keyword for the importing process, you must also explicitly add the media stream object or media player object (or an object that refers either of them) to the title container. Close the title container to store the objects.

You can play the media at a later time on any machine that supports ScriptX. In future ScriptX sessions, you do not need to import the media again. To access the media stream or media player, simply open the relevant title container. You do not need to keep the original media file around, since the media data is copied directly into ScriptX.

Saving the Media Player

The following code sample illustrates how to import a media file and save the media player object to the object store. It also shows how to play the media after quitting from ScriptX and restarting ScriptX.

```
-- open a title container
tc := new titlecontainer path:"song1.sxt"

-- import the media
theStream := getstream thestartdir "song1.aif" @readable
songplayer := importMedia theImportExportEngine theStream\
   @sound @aiff @player container:tc
```

```
-- add the media player to the title container
append tc songplayer

-- define a startup action for the title container
tc.startUpAction := (tc -> songplayer := tc[1])

-- close the title container
close tc
```

Quit from ScriptX, then launch ScriptX again. If you start ScriptX from the `mysongs.sxt` icon you do not need to enter the following:

```
open titlecontainer path:"song1.sxt"
```

In this case, the variable `songplayer` was defined by the title container's startup action. To play the song, enter:

```
play songplayer
```

Saving the Media Stream

The following code sample illustrates how to import a media file and save the media stream object to the object store. It also shows how to play the media after quitting from ScriptX and restarting ScriptX.

```
-- open the title container
tc := new titlecontainer path:"song1.sxt"

-- import the media
theStream := getstream thestartdir "song1.aif" @readable
songStream := importMedia theImportExportEngine theStream\
   @sound @aiff @stream container:tc

-- add the media stream to the title container
append tc songStream

-- define a startup action for the title container
tc.startupAction := (tc -> songStream := tc[1])

-- Close the title container
close tc
```

Quit from ScriptX, then launch ScriptX again. If you start ScriptX from the `mysongs.sxt` icon you do not need to enter the following:

```
open titlecontainer path:"song1.sxt"
```

In this case, the variable `songstream` was defined by the title container's startup action. Create a digital audio player to play the song:

```
global songplayer := new digitalAudioPlayer mediaStream:songStream
play songplayer
```

Saving Multiple Batch Media to a Title Container

The following sample code illustrates how to import multiple media files and save them to a title container by saving a list of the media streams. (When you save an object to a title container, any objects referenced by that object are also saved.) The code sample also shows how to play the media after quitting from ScriptX and restarting ScriptX.

```
-- open a title container
tc := new titlecontainer path:"mysongs.sxt"

-- import several media files
s1 := importMedia theImportExportEngine \
          (getstream thestartdir "song1.aif" @readable) \
          @sound @aiff @player container:tc
s2 := importMedia theImportExportEngine \
          (getstream thestartdir "song2.aif" @readable) \
          @sound @aiff @player container:tc
s3 := importMedia theImportExportEngine \
          (getstream thestartdir "song3.aif" @readable) \
          @sound @aiff @player container:tc

songlist := #(s1, s2, s3)

-- add a list of the media streams to the title container
append tc songlist

-- define a startup action for the title container
tc.startupAction := (tc -> \
                     song1 := tc[1][1] \
                     song2 := tc[1][2] \
                     song3 := tc[1][3] )
close tc
```

Quit from ScriptX, then launch ScriptX again. If you start ScriptX from the `mysongs.sxt` icon you do not need to enter the following:

```
open titlecontainer path:"song1.sxt"
```

In this case, the variables `song1`, `song2` and `song3` are defined by the title container's startup action. Create a digital audio player to play the songs:

```
songPlayer := new digitalAudioPlayer

-- play song1
songPlayer.mediaStream := song1
play songPlayer

-- play song2
stop songPlayer
songPlayer.mediastream := song2
gotobegin songPlayer
play songPlayer

-- play song3
stop songPlayer
```

```
songPlayer.mediastream := song3
gotobegin songPlayer
play songPlayer
```

Using Media Players

To prepare a media player for playing, call its `playPrepare` method. To start it playing its associated media call its `play` method. If you call the `play` method on an unprepared player, the player's clock starts ticking immediately while the player gets prepared to play. When the player is ready, it starts playing its media from that time. Thus the very beginning of the media might not play or the media might not be perfectly synchronized with other media.

To stop the media from playing, call the player's `stop` method. A player does not stop playing until the `stop` method is called, even if it has passed the end of its media stream.

To set the media back to the beginning, call the player's `gotobegin` method. To move the media to a particular time, set the player's `time` instance variable to the desired time. To put the player in a temporarily paused state call its `pause` method.

Use the `playUntil` method to play a player until a given time. For example to play it until the end of its media stream, call `playUntil` and pass in the player's duration as the time to play until.

```
-- p is an existing media player
gotobegin p
playUntil p p.duration
```

To change the media stream that a digital audio player, digital video player or MIDI player plays, set the value of its `mediaStream` instance variable to the desired media stream object.

Playing Audio

To play sound other than MIDI sound in ScriptX, use a `DigitalAudioPlayer` instance whose `mediaStream` instance variable contains an `AudioStream` instance. Digital audio players can play sound media imported from AIFF, SND, and WAVE audio files. (MIDI sound is discussed in "Playing MIDI" on page 190.)

The following script demonstrates how to import and play a tune that has been digitized and saved to a file called `ditty.aif` in the same folder as the script.

```
-- Create a title container
tc:= new titlecontainer path:"ditty.sxt"

-- import the media
global tuneStream1 := getstream theScriptDir "ditty.aif" @readable
global tunePlayer := importMedia theImportExportEngine tuneStream1 \
    @sound @AIFF @player container:tc
```

```
-- Prepare tunePlayer
playPrepare tunePlayer 1

-- Start tunePlayer playing
play tunePlayer

-- Stop the tune from playing
stop tunePlayer

-- Start playing the tune over again from the beginning
-- This time play it once through to the end
gotobegin tunePlayer
playPrepare tunePlayer 1
playUntil tunePlayer tunePlayer.duration

-- add tuneplayer to the title container and save the title container
gotobegin tunePlayer
append tc tunePlayer
tc.startUpAction := (tc -> tunePlayer := tc[1])
close tc
```

Sound Channel Allocation

When an `AudioStream` instance becomes attached to a `DigitalAudioPlayer` instance, either during the importing process or by explicitly being put in the player's `mediaStream` instance variable, the player locates all appropriate hardware sound channels. When the `playPrepare` method is called on the player, it searches through all the appropriate sound channels looking for one that is not in use. If all appropriate sound channels are already in use then the preparing player causes a `DigitalAudioPlayer` instance that is already prepared to become unprepared, thus freeing a sound channel. This may cause another sound to stop playing.

Playing Movies

The `MoviePlayer` and `InterleavedMoviePlayer` classes provide facilities for playing movies. A`MoviePlayer` instance plays a movie whose data is separated into audio and video streams. An `InterleavedMoviePlayer` instance plays a movie whose data is held in a single interspersed stream. You use both `MoviePlayer` and `InterleavedMoviePlayer` instances just like any other players— control them with the `play`, `playUntil`, `stop`, `pause`, `gotoBegin`, `gotoEnd`, and `fastForward` methods.

A `MoviePlayer` instance controls `DigitalAudioPlayer` and `DigitalVideoPlayer` instances that play together to present the movie. An `InterleavedMoviePlayer` also controls other players, but has the added complexity that its `interleavedStream` instance variable holds a stream containing the interleaved movie data.

When an interleaved movie player plays, it uses chunk streams to pass the video data in the interleaved stream to a digital video player and to pass the audio data to a digital audio player. See the section on `InterleavedMoviePlayer` in the *ScriptX Class Reference* for more details on how interleaved movie players and chunk streams work.

When importing any movie into ScriptX, you can choose to import it as a non-interleaved movie or an interleaved movie. To separate the sound and audio data of a movie into separate streams, import it as a `MoviePlayer` instance. To preserve the existing interleaving of a movie during importing, import it as an `InterleavedMoviePlayer` instance.

If the video and audio data for a movie are held in separate streams, the video data needed for a frame may be arbitrarily distant on the storage medium from the audio data needed for the same frame, causing an increase in search time between each frame. When the audio and video data are interleaved into a single stream, 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 playing non-interleaved movies 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 from a CD. If you intend to play the imported movie from a CD, you should import it to an `InterleavedMoviePlayer` to preserve the interleaving.

To create a `MoviePlayer` or `InterleavedMoviePlayer` instance, import a file that contains a digitized movie. You must generate the digitized movie file outside of ScriptX. ScriptX can import QuickTime and AVI movies. Only QuickTime movies can be imported as interleaved movies.

Note – This release of ScriptX can import QuickTime movies whether or not they are compressed, and regardless of what kind of compression was used if they are compressed. However, only cinepak-compressed movies are guaranteed to play back successfully on any platform on which ScriptX runs.

In addition to being a player, a `MoviePlayer` or `InterleavedMoviePlayer` instance is also a presenter. To play a movie, append the movie player or interleaved movie player to a visible window, and call the `play` method on the player.

The following script demonstrates how to import and play a movie that is saved in the file "`whale`" in the same folder as the script.

```
-- create a title container
tc := new titlecontainer path:"whale.sxt"

-- Import the movie to a MoviePlayer
-- If you want to preserve the interleaving in the movie,
-- substitute @interleavedMoviePlayer for @Player

global whalestream := getstream theScriptDir "whale" @readable
global whaleplayer := importMedia theImportExportEngine whalestream \
    @movie @quicktime @Player container:tc
```

```
-- Create a window the size of the movie's screen
global w:= new window boundary:(whaleplayer.bbox)
w.x := w.y := 40
show w

-- Append the movie player to the window
append w whaleplayer

-- Play the movie
play whaleplayer

-- Stop the movie
stop whaleplayer

-- Set the movie to a particular time
-- then play it again to the end of the movie
whaleplayer.time := 10
playUntil whaleplayer whaleplayer.duration

-- set the movie back to the beginning
gotobegin whaleplayer

-- append the window to the title container
-- the startup action defines variables for the window and whaleplayer
-- close the title container
append tc w
tc.startUpAction := (tc -> w := tc[1]; whaleplayer := w[1] )
close tc
```

Note – You cannot use an `InterleavedMoviePlayer` object on a page in a document if you want the movie to change from page to page. You can use a `MoviePlayer` that dynamically updates its target when a page in a document opens. See Chapter 13, "Document Templates" for details on creating pages and documents.

Playing MIDI

ScriptX provides two ways to play MIDI sound. One way is to import an existing MIDI file, and then use a `MIDIPlayer` object to play the imported MIDI data. The other way is to create MIDI events directly in ScriptX and send them to a MIDI driver. In both cases, you must have a machine capable of playing MIDI.

This section first discusses how to import and play existing MIDI data, and then discusses how to generate and send your own MIDI events. This section does not attempt to teach the MIDI specification.

Note – This release of ScriptX supports only type 0 MIDI files.

Playing Existing MIDI Data

To play existing MIDI data in ScriptX, use a `MidiPlayer` instance whose `mediaStream` instance variable contains a `MidiStream` instance. A MIDI stream contains a series of `MIDIEvent` objects, which are initially created by importing a standard MIDI file.

You can optionally choose which MIDI device to use to play the MIDI sound by setting the `driver` instance variable of the `MIDIPlayer` object. See "Finding a MIDI Driver" on page 192 for information on how to find MIDI drivers. If you do not specify a MIDI driver, the player uses a default one.

The following script demonstrates how to import and play a hypothetical MIDI file called `harp.mid` that resides in the same folder as the script.

```
-- create a title container
tc := new titlecontainer path:"harp.sxt"
global stream1 := getStream theScriptDir harp.mid @readable
global harpPlayer := importMedia theImportExportEngine stream1 \
   @MIDI @standard @player container:tc

-- play the MIDI tune
play harpPlayer

-- append the player to the title container and close the container
stop harpPlayer
gotobegin harpPlayer
append tc harpPlayer
tc.startUpAction := (tc -> harpPlayer := tc[1])
close tc
```

Creating MIDI Events Directly

You can create MIDI events from within ScriptX. For example, you could create a piano program that displays a piano keyboard on the screen. Each time a user clicks on a key on the keyboard, the program plays an appropriate note by generating and playing a MIDI event.

The following list summarizes the steps involved in generating and playing MIDI events:

- Create and initialize a MIDI event.
- Find a MIDI driver.
- Prepare the MIDI driver.
- Send the MIDI event to the MIDI driver.
- When you have finished using the MIDI driver, unprepare it and close it.

MIDI events are represented in ScriptX as instances of the class `MIDIEvent`.

Creating a MIDI Event

To create a MIDI event, call the `new` method on the class `MIDIEvent`. Initialize the new MIDI event by setting its instance variables `statusByte`, `datatbyte1`, `databyte2` and `data`. You can either set the `statusByte`, `databyte1` and `databyte2` instance variables or set the `data` instance variable.

The `statusByte` instance variable holds a byte (that is, a number between 0 and 256 inclusive) that specifies the MIDI message. For example, 144 (which is `0x8f` + 1) indicates a note-on message to channel 1; 145 (which is `0x8f` + 2) indicates a note-on message to channel 2; 128 (which is `0x7f` + 1) indicates a note-off message to channel 1; 129 (which is `0x7f` + 2) indicates a note-off message to channel 2; and so on. Please see the external MIDI specification for a complete list of all MIDI messages.

The `databyte1` and `databyte2` instance variables hold data that depends on the value in the `statusByte` instance variable. For example, if the value in the `statusByte` instance variable represents a note-on message, then the `databyte1` instance variable should hold a byte specifying the note being sent and the `databyte2` instance variable should hold a byte specifying the velocity of the note.

The `data` instance variable holds a `ByteString` object, where the first element specifies the status byte, and the second and third bytes specify the additional data needed for that message. Additional bytes specify additional information for system-exclusive messages.

When you specify values for the `statusByte`, `databyte1` and `databyte2` instance variables of a MIDI event, the byte string in the `data` instance variable is updated, and vice versa.

Most standard MIDI devices can understand short MIDI messages, which consist of the status byte, and the first and second data bytes. Long MIDI messages, known as system-exclusive messages, have additional data bytes,

and these messages can only be understood by specific devices. To specify additional databytes for a long MIDI message in ScriptX, you must set the value of the `data` instance variable of a `MIDIEvent` object.

The following two fragments of code illustrate how to create and initialize a MIDI event.

- Initialize the MIDI event by specifying values for the `statusByte`, `databyte1` and `databyte2` instance variables:

```
midievent1 := new MidiEvent
midievent1.statusByte := 0x8f + 1 -- note on for channel 1
midievent1.databyte1 := 54 -- note
midievent1.databyte2 := 94 -- velocity
```

- Initialize the MIDI event by specifying a byte string for the `data` instance variable:

```
midievent1 := new MidiEvent
dataString := new ByteString
append dataString 0x8f + 1 -- note on for channel 1
append dataString 54 -- note
append dataString 94 -- velocity
midievent1.data := dataString
```

Finding a MIDI Driver

When ScriptX starts up on a MIDI-capable machine, it creates MIDI drivers. When ScriptX starts up on a machine that is not capable of playing MIDI, it does not create any MIDI drivers.

To find a MIDI driver, use the `getMIDIDriverList` and `openMIDIDriver` global functions. The function `getMIDIDriverList` returns a list of "MIDI driver" pairs. Each pair consists of a string of the name of the driver and a symbol for the location of the driver.

For example:

```
global Mlist := getMidiDriverList()
```

might return:

```
#(#("MIDIManager", @internal))
```

The `openMidiDriver` function opens a MIDI driver specified by a MIDI driver pair. The function returns the opened `MIDIDriver` object if it was succesful, or false if it was not.For example:

```
global Mdriver := openMidiDriver (getFirst (getMidiDriverList ()))
```

opens the first MIDI driver returned by the function `getMidiDriverList`. The following code:

```
global Mdriver := openMidiDriver #("MIDIManager", @internal)
```

opens the internal MIDI driver called MIDI Manager regardless of where it is in the list of MIDI drivers.

To open a MIDI driver regardless of what platform is being used, use the following code, which iterates over all the MIDI drivers until it successfully opens one. (If no MIDI driver is found or successfully opened, `Mdriver` ends up being undefined.)

```
global MDriver := false
for md in (getMidiDriverList ()) until Mdriver
    do Mdriver := openMidiDriver md
```

Prepare the MIDIDriver

You must do two things to prepare the MIDI Driver. You must do these things in order:

1. Set the value of the MIDI Driver's `channelPolyphony` instance variable.

   The value of this instance variable must be set to an array of 16 elements, representing the polyphony (number of voices) per channel for 16 channels. For example:

   ```
   MDriver.channelPolyphony := #(3, 1, 0, 5, 0, 0,
       2, 0, 0, 0, 0, 0, 0, 0, 0, 0)
   ```

   which means that the maximum polyphony for channel 1 is 3; for channel 2 is 1; for channel 4 is 5; and for channel 7 is 2. For all other channels, the polyphony is 0 (that is, those channels can't play.)

2. Call the `prepareDriver` method on the MIDI Driver. (Do not call `prepareDriver` before you have set the channel polyphony.) The `prepareDriver` takes a second argument, but you can pass this as `undefined` when calling the method directly. For example:

   ```
   prepareDriver Mdriver undefined
   ```

Sending the MIDIEvent to the MIDIDriver

To send a MIDI event to a MIDI device, call the `sendMIDIEvent` method on the MIDI driver for the device. For example:

```
sendMIDIEvent Mdriver midievent1
```

Closing the MIDI Driver

When you have finished using a MIDI driver, unprepare it by calling its `unprepareDriver` method, then close it by calling the `closeMidiDriver` global function. For example:

```
unprepareDriver Mdriver
closeMidiDriver Mdriver
```

MIDI Player Example

The following script illustrates how to play a sequence of MIDI notes.

The following script defines functions to send note-on, note-off, and change-program messages. The example uses the `waitTime` method on a clock to make the system wait before sending the note-off event. (If you want to send a series of MIDI events, you could create a clock and set up callbacks to send the MIDI events at appropriate times.)

```
-- get a list of all the midi drivers
global mdList := getMidiDriverList ()

global Mdriver := false

-- Find a working Midi Driver
-- openMidiDriver returns the opened driver
-- if the Midi Driver is successfully opened,
-- otherwise returns false
for md in mdlist until Mdriver do
    Mdriver := openMidiDriver md

-- if no driver was successfully opened, print a warning
if (not Mdriver) do print "This script will not work.
There is no working Midi Driver."

-- set the maximum number of voices per channel
mdriver.channelPolyphony := #(1, 0, 0, 0, 0, 0, 1, 0,
    0, 0, 0, 0, 0, 0, 0, 0)

-- prepare the Midi driver
prepareDriver  Mdriver undefined

-- Create a clock to use to time the notes
global c := new clock
c.scale := 10
c.rate := 1

-- define a function to send a note on message
fn sendNoteOn driver channel note vel ->
(
    local e := new MidiEvent
    e.statusByte := 0x8f + channel
    e.databyte1 := note
    e.databyte2 := vel
    sendMidiEvent driver e
)

-- define a function to send a note off message
fn sendNoteOff driver channel note vel ->
(
    local e := new MidiEvent
    e.statusByte := 0x7f + channel
```

```
    e.databyte1 := note
    e.databyte2 := vel
    sendMidiEvent driver e
)

-- send a program change message to determine which
-- instrument a channel should play as
-- prog is from 1 to 128
fn sendProgram driver channel prog ->
(
    local e := new MidiEvent
    e.statusByte := 0xbf + channel
    e.databyte1 := prog
    sendMidiEvent driver e
)

-- set the instrument for channel 1
-- then play a sequence of notes on channel 1
sendProgram mdriver 1 5
for i in 40 to 70 do (
    sendnoteon mdriver 1 i 200
    sendnoteoff mdriver 1 i 200
    waittime c 1)

-- set the instrument for channel 7
-- then play a sequence of notes on channel 7
sendProgram mdriver 7 120
for i in 40 to 70 do (
    sendnoteon mdriver 7 i 200
    sendnoteoff mdriver 7 i 200
    waittime c 1)

-- unprepare the midi driver then close it
unPrepareDriver Mdriver
closeMidiDriver Mdriver
```

C H A P T E R

Animation

9

The Animation component allows you to create and play sequences of actions that take place over time. These sequences can be used to create animation as well as to control or annotate other presentation elements such as video tapes and music.

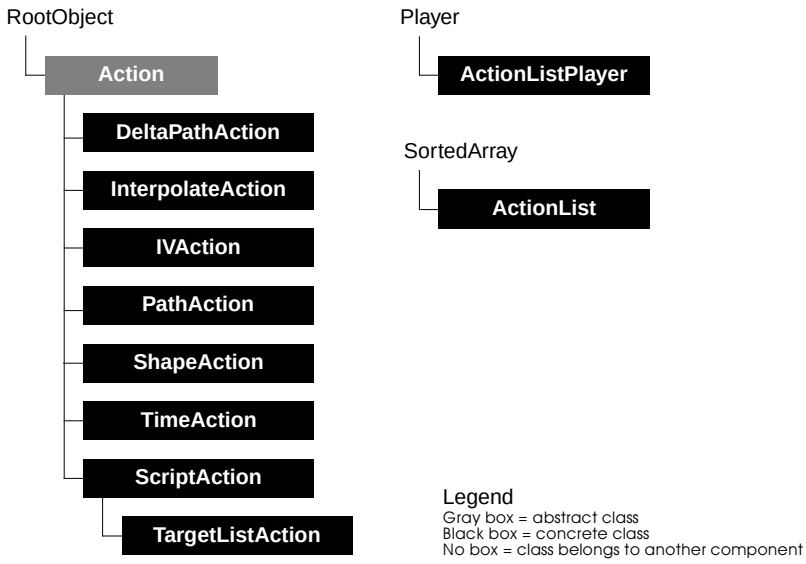
A key class of the Animation component is the `ActionListPlayer` class. It implements a non-thread-based model of action representation and processing. It is well-suited for converting sequences from other multimedia systems. Every action associated with an action list player has its own specific trigger time.

The techniques described in this chapter are not the only way to perform animation in ScriptX. You can also use controllers. For example, the Controllers chapter describes an animated

This chapter describes the Animation component and includes two examples at the end.

Classes and Inheritance

The class inheritance hierarchy for the Animation component is shown in the following figure.



The following classes form the Animation component. In this list, indentation indicates inheritance.

`Action` – an abstract superclass that defines a time and a target object.

`DeltaPathAction` – holds information about relative position change.

`InterpolateAction` – changes the destination point and time of an `Interpolate` controller object.

`IVAction` – changes a specified instance variable on the target object to a given value.

`PathAction` – holds information about absolute position change.

`ScriptAction` – holds a function to run at given time.

`ShapeAction` – holds information about shape changes.

`TargetListAction` – calls a function at a certain time and replaces a specified target object in the target list with the return value of that function.

`TimeAction` – makes an action list player go to a specified time.

`ActionList` – a list of `Action` objects, sorted by time.

`ActionListPlayer` – a subclass of `Player` that plays an `ActionList` object.

Conceptual Overview

The Animation component provides a means for creating timed sequences in ScriptX. The `ActionListPlayer` class represents a player of actions that affect a target list. Each action has a trigger time, and can operate on any or all of the targets. The action can set instance variables, call functions or methods, add or delete objects, or perform any other operation.

How Animation Works

An `ActionListPlayer` object implements a list of actions to be executed in sequence; each action has its own absolute time, target object to act on, and type of action.

Action List and Actions

As shown in Figure 9-1, action list players involve a number of related classes that must work together. When you play an action list player, it triggers a sequence of actions over time. Each action has a specific trigger time and target associated with it.

This component includes classes for a list of actions (`ActionList`), a list of target objects (the `targets` instance variable in `ActionListPlayer`), time/target pairs (`Action` objects), and a set of possible actions (the `Action` subclasses).

The subclasses of `Action` (listed previously in "Classes and Inheritance") represent the possible types of actions that can happen at a particular time: changing a bitmap (`ShapeAction`), making relative or absolute position changes (`DeltaPathAction`, `PathAction`), moving along a curve to a

specified point (`InterpolateAction`), changing a time (`TimeAction`), or performing a general script (`TargetListAction`, `ScriptAction`). The `TargetListAction` is a specialized `ScriptAction` used to add or remove objects from the `targets` list.

For example, using a `ShapeAction` object you can specify that at 10 seconds from the start of the sequence, the bitmap being shown by a `TwoDShape` object will change.

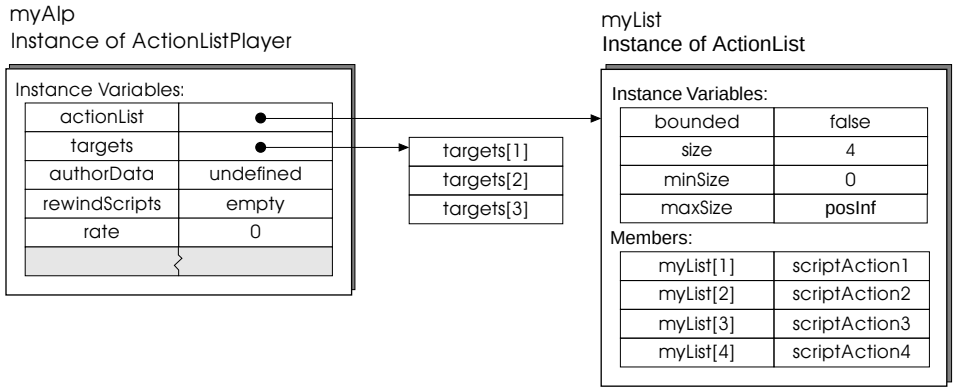


Figure 9-1: An action list player has one action list and a list of targets.

Each `Action` object contains the absolute time it will trigger. An `ActionList` object contains a list of `Action` objects, sorted by time, which is the order they will be triggered. An `ActionListPlayer` object plays back the action list and allows for operations such as pause, rewind, and speed control.

Action List Player

You can fast-forward or rewind an `ActionListPlayer` object. These correspond to setting the time on the `ActionListPlayer` object, for example, given a player `alp`, the following statement sets the time of the player to 50:

```
alp.time := 50
```

This means “go to time 50”, and performs either a fast-forward or rewind depending on where you started from—all actions are performed up to the specified time, none are skipped.

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.

Target List

The target list is specified by the `targets` instance variable of `ActionListPlayer`:

```
alp.targets
```

Put only those objects onto the target list that you want to control with actions. You can put any ScriptX objects on the list. They can be 2D presenters, stencils, controllers, collections, or anything else. If an object is never acted on by an action, then leave it off this list.

If you want several target objects to be treated as a single unit that can be easily moved or modified as a group, add them to a `GroupPresenter` object, then add this object to the target list.

There two ways to add objects to a target list: statically, where you add objects to the list before starting the action list player, and never remove them; and dynamically, where you add objects to the list as it is playing and remove them while it is rewinding. The following two sections describe these two techniques.

Setting Up an Unchanging Target List

The simplest way to use an action list player is to add objects to the target list initially, before playing it, and don't add or remove objects thereafter. To provide animation, change properties of the objects on the target list: location, shape, any instance variable, or any function that can be run. For example, to initially hide an object and then show it once the action list player begins, first add the object to the target list, then create a function to hide the object at time 0 and show it at time 1 tick:

```
alp.targets[1] := new TwoDShape target:(new Oval x2:80 y2:80)

global hideShape := new ScriptAction script:(a t p -> hide t) \
    targetNum:1 time:0

global showShape := new ScriptAction script:(a t p -> show t) \
    targetNum:1 time:1
```

These two script actions hide and then show the first object in the target list (as given by `targetNum:1`). The script is an anonymous function with three arguments `a` (the script action), `t` (the target), and `p` (the action list player), as specified in the `ScriptAction` class in *ScriptX Class Reference*.

The example "A Simple Flipbook" on page 204 shows this technique in greater detail.

Changing the Target List on the Fly

Another technique for action list players is to add targets to the target list as it plays forward and remove those targets as you rewind. If you rewind to a point in time where a target object does *not* exist, you must make sure that object is disposed of during the rewind—otherwise, the object will still be there when it shouldn't be.

You can do this by adding objects to the target list using a `TargetListAction` object and specifying a dispose function during rewind using the `rewindScript` instance variable, defined in that class. When you create a target list action, you should provide both a script to create the object,

and the `rewindScript` function for disposing the object. The rewind script has the same arguments as the `ScriptAction` script. If you want it to remove the target without adding another object, the function should return `undefined`; for example:

```
myAction.rewindScript := (action target player ->
    deleteone target.presentedBy target
    undefined
)
```

An array called `rewindScripts` (defined in `ActionListPlayer`) is an initially empty array that gets filled up with dispose functions as the action list player plays. Then, during any rewind, all the rewind scripts are executed before playing the action list player from the beginning of the action list.

The example “A Flipbook – Dynamically Changing the Target List” on page 205 shows this technique in greater detail.

Registration Points

When animating a sequence of bitmaps or other stencils, it is often useful to move their 0,0 origin of each bitmap from the default upper left corner to a more meaningful place on the bitmap. You can then line up all the bitmaps at their 0,0 points. For example, if you have a person walking across the screen, you could put the registration point at the center of their hips.

In addition, if you set up several bitmaps with their 0,0 points set to meaningful places, you could simply switch between the bitmaps and their origins would all coincide. It’s common to switch between different-sized bitmap targets of a 2D shape this way. The 2D shape that holds the bitmaps would stay in the same place, but the bitmaps would be correctly aligned.

To set a registration point of a stencil, in general, set `x1` and `y1` to negative values. To center a stencil, set those values to half the width and height.

For example, suppose `myBitmap` is a bitmap, and, like most bitmaps, has its origin at its top-left corner:

```
myBitmap := new Bitmap data:myData bBox:(new Rect x2:100 y2:100)
⇨  [0, 0, 100, 100] as Rect
```

You can change this bitmap to have its 0,0 point at some other more interesting point, such as its center, by first making a translation matrix and then transforming the bitmap with it:

```
t := translate identityMatrix -50 -50
transform myBitmap t @mutate
```

Now the origin is at the bitmap’s center, and the top-left corner is no longer 0,0:

```
myBitmap.bBox
⇨  [-50, -50, 50, 50] as Rect
```

Improving Animation by Setting Garbage Collection

If you are trying to get smooth animation in ScriptX, you should consider the effect the Garbage Collector is having. For an animation with a frame rate of 30 fps, each frame takes 33 milliseconds. The `setGCincrement` function determines the amount of time the garbage collector runs before it yields to the next thread. The default value for `setGCincrement` is 20 milliseconds. Therefore, whenever the Garbage Collector runs, you are going to lose most of a frame (because the 20 milliseconds of Garbage Collector will take up much of a frame's time). You might try setting the Garbage Collector increment lower (such as 10 ms) to get smoother animation.

Animation Examples

This section contains three examples of action list players—the first two change the shape of a 2D shape, and a more elaborate example that uses an interpolator controller to move a circle.

A Simple Flipbook

The following action list player creates a 2D shape and uses `ShapeAction` to change its shape over time, as in Figure 9-4. Compared to the next example, this script is simple in that the shape is added to the target list before the animation begins playing, and the contents of the target list does not change either during play or rewind. However, properties of objects in the target list do change; in particular, the 2D shape's target changes from an oval to a rounded rectangle, to a rectangle.

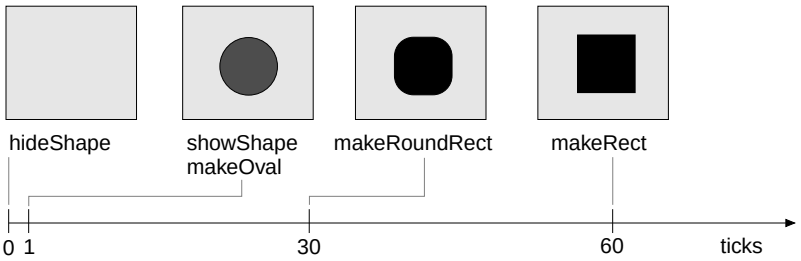


Figure 9-2: Changing a shape over time

The action list consists of six actions: `hideShape` at time 0, `showShape` and `makeOval` at time 1 tick, `makeRect` at time 30 ticks and `makeRoundRect` at time 60 ticks. The `hideShape` and `showShape` actions make the shape disappear and reappear without actually removing the target `myShape` from the target list `alp.targets`.

```
-- Filename: animshp1.sx
-- Create a window
global myWindow := new Window boundary:(new Rect x2:200 y2:200)
myWindow.y := 40
show myWindow
```

```
-- Create the ActionList and ActionListPlayer
global al := new ActionList
global alp := new ActionListPlayer actionList:al scale:30 targetCount:24

-- Create a blue circle and add it to the window and targets list
global myShape := new TwoDShape target:(new Oval x2:80 y2:80)
myShape.fill := new Brush color:blueColor
myShape.x := 60
myShape.y := 60
append myWindow myShape

-- Puts the shape on the target list
alp.targets[1] := myShape

-- Set an action to hide the shape at time 0
global hideShape := new ScriptAction script:(a t p -> hide t) targetnum:1 time:0

-- Set an action to show the shape at time 1
global showShape := new ScriptAction script:(a t p -> show t) targetnum:1 time:1

-- Set an action to set the shape to an oval
global makeOval := new ShapeAction targetNum:1 time:1 \
                                shape:(new Oval x2:80 y2:80)

-- Set an action to change its shape
global makeRoundRect := new ShapeAction targetNum:1 time:30 \
                                shape:(new RoundRect x2:80 y2:80 rx:30 ry:30)

-- Set another action to change its shape
global makeRect := new ShapeAction targetNum:1 time:60 \
                                shape:(new Rect x2:80 y2:80)

-- Append actions to the ActionList
append al hideShape
append al showShape
append al makeOval
append al makeRoundRect
append al makeRect

-- Play the ActionListPlayer
play alp

-- To play again, type: goToBegin alp
```

A Flipbook – Dynamically Changing the Target List

This example produces an animation identical to the previous example, but uses the `TargetListAction` class to add targets to the target list while it is playing, and a rewind script to remove targets from the list. These actions are illustrated in Figure 9-4. Use this technique when you want to define an action that occurs on rewind, to undo what occurred during play. This technique dynamically adds and removes objects from the target list, while the previous example keeps the target list constant. Normally, you would use this technique

only if you couldn't do what you want to do with the previous technique, since this technique is more involved, requiring you to keep track of what's on the target list over time.

The `new` method called on `TargetListAction` puts the shape on the target list. This statement is equivalent to `alp.targets[1]` of the previous example, but also defines a `rewindScript` to empty the targets.

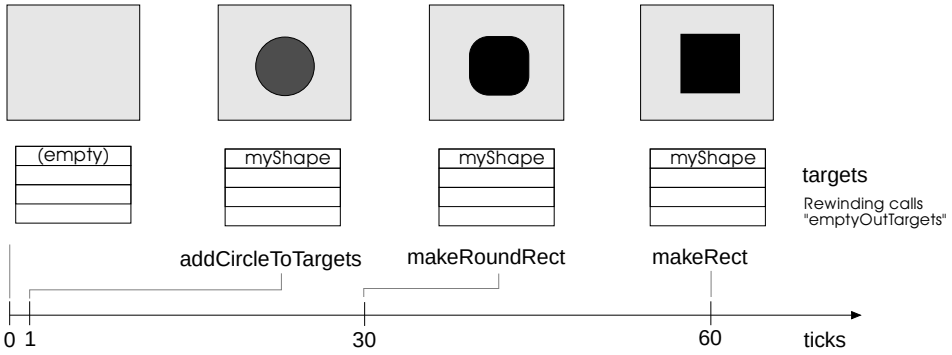


Figure 9-3: Changing a shape over time

```
-- Filename: animshp2.sx
-- Create a window
global myWindow := new Window boundary:(new Rect x2:200 y2:200)
myWindow.y := 40
show myWindow

-- Create the ActionList and ActionListPlayer
global al := new ActionList
global alp := new ActionListPlayer actionList:al scale:30 targetCount:24
alp.authorData := myWindow

-- Script to create a blue circle
function createCircle action target player ->
    (
      local myShape := new TwoDShape target:(new Oval x2:80 y2:80)
      myShape.fill := new Brush color:blueColor
      myShape.x := 60
      myShape.y := 60
      append player.authorData myShape
      myShape
    )

function emptyOutTargets action target player ->
    (
       emptyOut player.authorData
       emptyOut player.targets
    )

-- Set action to create and add the circle to the targets
global addCircleToTargets := new TargetListAction targetNum:1 time:1 \
                 script:createCircle rewindScript:emptyOutTargets

-- Set an action to change its shape
global makeRoundRect := new ShapeAction targetNum:1 time:30 \
```

```
                              shape:(new RoundRect x2:80 y2:80 rx:30 ry:30)

-- Set another action to change its shape
global makeRect := new ShapeAction targetNum:1 time:60 \
                              shape:(new Rect x2:80 y2:80)

-- Append actions to the ActionList
append al addCircleToTargets
append al makeRoundRect
append al makeRect

-- Play the ActionListPlayer
play alp

-- To play again, type: goToBegin alp
```

Animated Ball

The following action list player creates a circle and moves it three times, returning it to where it started, then loops continuously, as in Figure 9-4. This script uses the technique of keeping the target list unchanged.

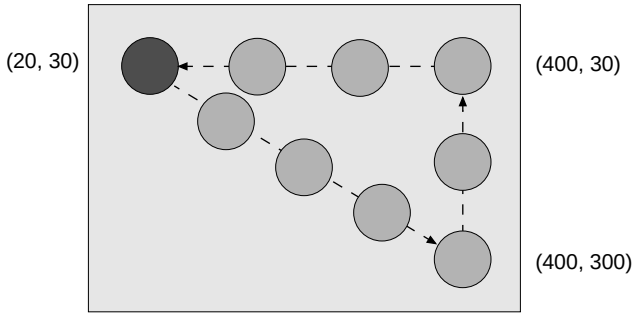


Figure 9-4: Animated ball

This example demonstrates how targets do not have to be presenters—they can be any ScriptX object. An interpolator controller appears as the only object on the target list. The shape is added to this interpolator, to enable the interpolator to move the shape. As shown in Figure 9-4, the controller is given three different destination points by adding interpolator actions to the action list. The animation loops back with an instance of `TimeAction` at time 200 which jumps back to time 0.

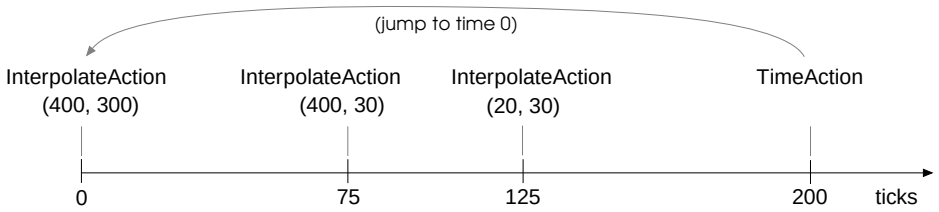


Figure 9-5: Moving a shape over time using interpolator actions

```
-- Filename: animatin.sx
-- Create the yellow window
global myWindow := new Window boundary:(new Rect x2:500 y2:400)
myWindow.fill := new Brush color:yellowColor
myWindow.y := 40
show myWindow

-- Create the ActionList and ActionListPlayer
global al := new ActionList
global alp := new ActionListPlayer actionList:al scale:30
alp.authorData := myWindow

-- Create a red circle
global myShape := new TwoDShape target:(new Oval x2:80 y2:80)
myShape.fill := new Brush color:redColor
myShape.x := 20
myShape.y := 30
prepend myWindow myShape

-- Create an interpolator controller for the window
global myInterp := new Interpolator space:myWindow clock:alp
append myInterp myShape

-- Put the interpolator on the target list
alp.targets[1] := myInterp

-- Create Interpolate actions and append them to the action list
-- Interpolate from time 0 to 75 ticks
append al (new InterpolateAction targetNum:1 time:0 \
    destPosition:(new Point x:400 y:300) destTime:75)

-- Interpolate from time 75 to 125 ticks
append al (new InterpolateAction targetNum:1 time:75 \
    destPosition:(new Point x:400 y:30) destTime:125)

-- Interpolate from time 125 to 200 ticks
append al (new InterpolateAction targetNum:1 time:125 \
    destPosition:(new Point x:20 y:30) destTime:200)

-- At time 200 jump to time 0
append al (new TimeAction time:200 destTime:0)

-- Play the action list player
play alp

-- To increase its rate, try: alp.rate := 2
```

CHAPTER

Transitions

10

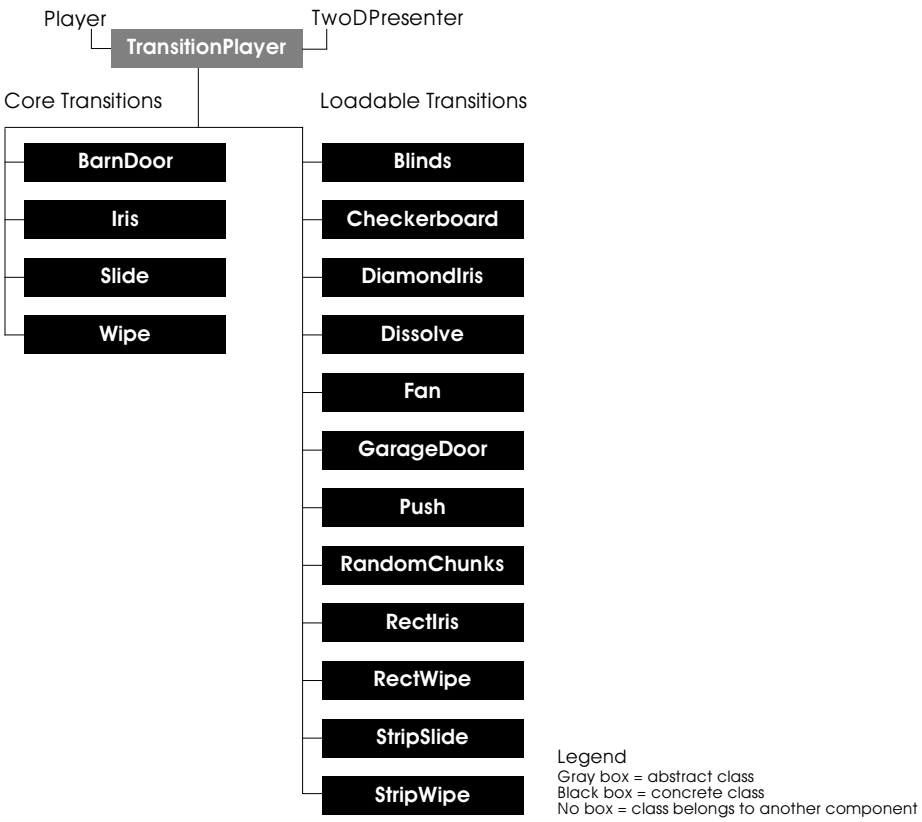
The Transitions component provides the capability for visual effects when adding a 2D presenter to a window, 2D space, or other kind of 2D multipresenter. Examples of transitions include wipe, slide, barn door, and iris. When a transition is played, it performs a series of renderings to gradually cause the target presenter to appear.

This chapter describes the classes provided with the ScriptX core classes. Other transition classes are available as loadable classes, and are described later in this chapter: blinds, checkerboard, diamond iris, dissolve, fan, garage door, push, random chunks, rect iris, rect wipe, strip slide, and strip wipe.

See the section “What The Transitions Look Like” on page 225 to see examples of each one.

Classes and Inheritance

The class inheritance hierarchy for the Transitions component is shown in the following figure.



The following classes form the Transitions component. In this list, indentation indicates inheritance.

`TransitionPlayer` – an abstract player class that sets the speed and boundary, and produces the individual frames of the transition.

`BarnDoor` (core) – reveals the target presenter by opening two vertical doors from the middle or from the edges.

`Blinds` (loadable) - reveals the target presenter by opening vertical or horizontal bands.

`Checkerboard` (loadable) - reveals the target presenter in a checkerboard pattern.

`DiamondIris` (loadable) - reveals the target presenter using a diamond opening.

`Dissolve` (loadable) - reveals the target presenter gradually as small, random dots.

`Fan` (loadable) - reveals the target presenter by sweeping in a clockwise or counter-clockwise fashion.

`GarageDoor` (loadable) - reveals the target presenter horizontally from the cente ror edges.

`Iris` (core) – reveals the target presenter with a circular iris opening from the center or from the edges.

`Push` (loadable) - reveals the target presenter by pushing it on-screen from the given direction.

`RandomChunks` (loadable) - reveals the target presenter in random-size squares, rectangles, columns or rows.

`RectIris` (loadable) - reveals the target presenter in a rectangular opening.

`RectWipe` (loadable) - reveals the target presenter in a rectangular wipe from any of four corners.

`Slide` (core) – reveals the target presenter by sliding it left, right, up, or down onto the screen.

`StripSlide` (loadable) - reveals the target presenter by sliding strips together from the side.

`StripWipe` (loadable) - reveals the target presenter by wiping it into view with either horizontal or vertical strips.

`Wipe` (core) – reveals the target presenter by wiping it onto the screen, wiping left, right, up or down.

Conceptual Overview

A common effect in multimedia titles is to gradually transition an image in or out of a scene, as shown in Figure 10-3. This visual effect adds to the continuity of a title, in contrast to the visually abruptness when an object is suddenly added or deleted. ScriptX supplies the built-in transitions as subclasses of TransitionPlayer: `Wipe`, `Slide`, `BarnDoor`, and `Iris`. ScriptX also offers many loadable transitions that are not part of the core classes, including dissolve, checkerboard, blinds, diamond iris and strip wipe.

These subclasses provide the mechanisms for transitioning from one frame to the next, producing the individual frames of the transition, and setting the speed and boundary of the transition.

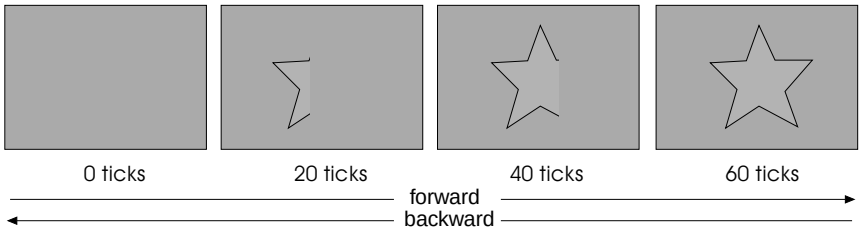


Figure 10-1: A 60-tick wipe transition from left to right

Since they operate over time, transitions are implemented as players—you can perform any player operations on a transition: play, pause, stop, rewind, fast foward, go to beginning, and go to end. In addition, transitions inherit from `TwoDPresenter` so that they can be added to any presentation collection, such as a window, group presenter or 2D multi-presenter.

Every transition operates in a window, or other 2D multipresenter, and has a target, which is the presenter that is being transitioned into that space. At each frame of the player, the transition renders a new image, revealing more of the target.

How Transitions Work

This section describes in general how transitions work. A step-by-step description of how to set up and use a transition, as well as a full, working sample script is included at the end of this chapter in the section “Using the Transitions Component.”

Revealing a Target

A transition is a visual effect that gradually adds a 2D presenter to a window or other kind of presentation collection. The 2D presenter being revealed by the transition is specified by the `target` instance variable. In the following example, `myOval` is the target being revealed.

```
global myOval := new TwoDShape target:(new Oval x2:200 y2:200) \
    stroke:blackBrush
global myWipe := new Wipe duration:30 direction:@right \
```

```
target:myOval scale:30
```

When a transition is played, it performs a series of renderings to gradually cause the target presenter to appear. Most transitions have a direction—the wipe in Figure 10-3 has its `direction` instance variable set to `@right`, which makes it travel from left to right. The transition is limited in area, occurring only within the boundary of the target presenter. The target presenter can be any 2D presenter—a text presenter, a still image, an animation, or a video image.

You place a transition in a presentation hierarchy in the position where you want its 2D presenter to appear. In other words, if you want a bitmap of a car to appear in front of a background roadway but behind a fence, you would place the transition in the hierarchy where you want the car to appear.

Performance and smoothness of transitions can be improved by setting the `direct` instance variable to `true`. This causes the transition to bypass the frame buffer and be drawn directly to the screen. The tradeoff is that the object being transitioned will appear in front of any other overlapping objects, if any. Dissolve is a direct transition, and its target will be drawn in front of any other objects in its window. All transitions except `Dissolve` have direct set to `false` by default. `Dissolve` must have `direct` set to `true` (setting `direct` to `false` causes an exception to be thrown when it is played). If the transition or its target is a direct presenter, the target should not have any other objects in front of it.

You can transition a presenter out of a space by setting the presenter as the target and setting the transition's rate to a negative value, which plays the transition backwards, as described later in "Using a Transition to Make the Target Disappear."

To perform a transition on an entire window, create a 2D multi-presenter (or other presentation container) the same size as the window and perform the transition on that.

Transitions are not automatically coupled with audio effects. If you want the audio to fade in as an image is gradually appearing, you must set up that audio transition yourself, for example, as a script that increases the volume over time.

A Transition Is a Clock

`TransitionPlayer` inherits eventually from `Clock`, which means it keeps track of its own time. Transitions are master clocks by default—when you create a transition, its `masterClock` instance variable is `undefined`, and it remains that way when you add it to a presentation (it is not made a slave of the window's clock). When you add a master clock either directly or indirectly to a title, it gets added to the title's `topClocks` instance variable. This allows the transition to pause when you pause the title.

Time and Frame of a Transition

A transition inherits the `time` and `rate` instance variables from the `Clock` class. It also inherits from `Player` methods such as `play` (sets `rate` to `1`), `stop` (sets `rate` to `0`), `goToBegin` (sets `time` to `0`) and `goToEnd` (sets `time` to `duration`).

When a transition is playing forward (normally, `rate` is set to `1`), the visual effect starts when `time` is equal to `0`, and ends when `time` is equal to `duration`. Therefore, if you wanted to insert a delay of 30 ticks before the transition, you could initially set `time` to `–30`.

The `frame` instance variable specifies what frame the transition is currently at. It ranges from a value of `0` at the start to `duration` at the end of the transition. Thus, if the `duration` is set to 30 ticks, when a transition reaches the end, the value of `frame` will be `30`.

When you first create a transition, it is stopped (`rate` is `0`), `time` is 0 and `frame` is 0. When you play a transition (set `rate` to `1`), the value of both `time` and `frame` begin incrementing—the value of `frame` stops incrementing when its value reaches `duration`, while `time` continues incrementing. For example, if `duration` is set to `30`, although the visual effect ends at frame 30, `time` keeps on incrementing. To play the transition over again, call `goToBegin` on it, which sets `time` to `0`—it will play immediately, since its rate is still `1`.

Duration and Smoothness of a Transition

A transition's duration in ticks is specified by its `duration` instance variable, where there are `scale` number of ticks in a second (if the transition's clock is a master clock, which they are by default). In other words, the length of time in seconds that a transition takes is equal to the `duration` instance variable divided by the `scale` instance variableof the transition player. For example, a transition with `duration` set to `60` and `scale` set to `30` will take 2 seconds.

The smoothness of a transition is equal to the scale times the length of the transition. For example, a wipe transition with a length of 2 seconds and a scale of 30 frames per second has 60 overall steps from start to finish. The more steps to the the transition, the smoother the transition will appear.

Using a Transition to Make the Target Disappear

In all transitions, the target appears as the transition is played forward. If, on the other hand, you want the target to *disappear*, you can play the transition backwards, as in Figure 10-3, by setting its rate to a negative number. The transition will play backward even though the rest of the title is playing forward because you are setting the rate of the transition to a negative value but leaving the rate of the window's clock set to a positive value.

Transitions that have a random element to them cannot be played backwards, including `Dissolve` and `RandomChunks`.

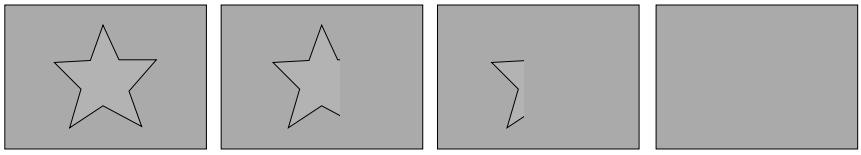


Figure 10-2: The same wipe transition shown earlier, but played backwards.

For example, to play a transition backward, first stop it, go to its end, and then set its rate to –1:

```
stop myWipe
goToEnd myWipe
myWipe.rate := -1
```

This causes the target to disappear, with the visual effect going the opposite direction—a wipe with `direction` set to `@right` would go to the left instead of the right. However, this transition does not delete the hidden presenter—this is described in the next section.

Therefore, you can create a loop that causes a presenter to alternately transition in and out by creating a callback that reverses the transition's rate when `frame` is equal to `duration`, and another callback at time 0 to reverse the rate again.

Another way to transition an existing presenter out of a space is to specify the background image as the target and transition in the background, in front of the presenter.

Deleting a Hidden Presenter

There are two cases where a presenter can become hidden during a transition:

- When transitioning forward, an object can become hidden behind the target that is being revealed, as in Figure 10-3.
- When transitioning backward, the target is no longer visible, but the transition player still exists in the presentation hierarchy, as in Figure 10-3.

In both of these cases, just because the object is hidden does not mean that it has been removed from the presentation hierarchy (that is, window). If you want the hidden object to be deleted, you must explicitly delete it yourself—the transition does not remove it for you.

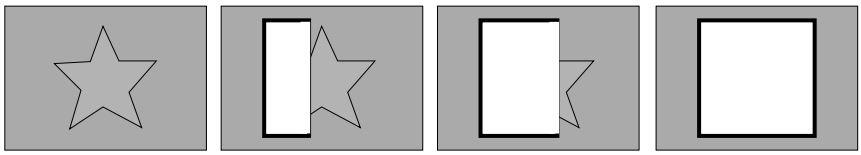


Figure 10-3: The new square covers up the star.

The following script shows how to set up a callback to delete the star after transitioning in the square. The first expression creates the wipe transition, which has the square as its target. Next the wipe is prepended to the star's

presentation collection. Then a function named `delObj` is created to delete an object. The `addTimeCallback` method sets the `delObj` function to execute when the transition is complete, at the time determined by `myTransition.duration`. (If you were playing the transition backward, you would set the function to execute when the value of `time` is `0`.)

Then `playPrepare` is called to prepare the transition, and `play` is called to start the wipe. When the transition's time reaches `wipe.duration`, it calls the function `delObj` on `star`, deleting it.

```
global myWipe := new Wipe direction:@right duration:30 target:square scale:30
prepend star.presentedBy myWipe
function delObj objToDel -> (deleteOne objToDel.presentedBy objToDel)
addTimeCallback myWipe delObj star #() myWipe.duration false
playPrepare myWipe 1
play myWipe
```

Splicing the Target into the Space

When you set up a transition in a window, the window holds the transition, which holds the 2D presenter, as shown on the left side of Figure 10-4. When you play the transition, the 2D presenter is displayed in the window. By default, when the transition is done, it stays in the window and is not deleted—this corresponds to the `autoSplice` instance variable being set to `false`. The advantage of keeping the transition is that you can play the transition again, perhaps with a new target, or play the transition backwards to remove the target.

However, when the transition is done, if you no longer want or need it, you can remove the transition from the window and "splice" the 2D presenter in its place, as shown on the right side of Figure 10-4. You can either do this yourself using `addTimeCallback`, or set `autoSplice` to `true`. Either way automatically splices in the 2D presenter at the completion of the transition.

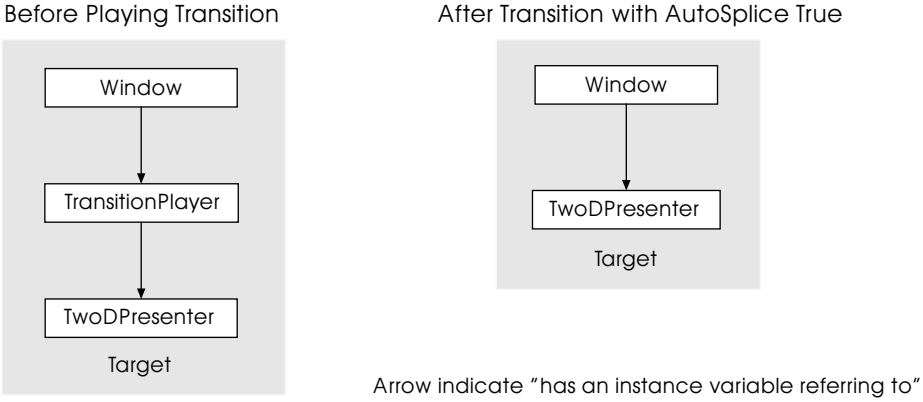


Figure 10-4: Removal of TransitionPlayer from Window when autosplice is true.

Offscreen Cached Target

The `useOffscreen` instance variable is a Boolean that allows you to control whether an offscreen bitmap copy of the target is made, which is then transitioned into the presenter. Using an offscreen bitmap can be more efficient and produce smoother transitions in certain circumstances, but uses more memory, than not using an offscreen bitmap. With `useOffscreen` set to true, the offscreen bitmap is assigned to the `cachedTarget` instance variable. A cache is an area of memory where a bitmap snapshot of the image is held. This bitmap is created either when you play the transition, or when you call `playPrepare`. Call `playPrepare` prior to calling `play`, when synchronization is important, so that the work of creating the bitmap is done ahead of time.

The `useOffscreen` instance variable is `false` by default, to conserve memory. If memory is tight and the target is large, you may want to set `useOffscreen` to `false` so it does not create a cached target.

Keep `useOffscreen` set to `false` if the target presenter is:

- Static and simple
- Large and memory is tight (not enough memory for a cached bitmap)

Set `useOffscreen` to `true` if the target presenter is:

- Static and complex, composed of many objects

Transitions with `useOffscreen` set to `false` do not work well for direct presenters, such as video (which is a direct presenter by default). For video you should either set its `direct` instance variable to `false`, or freeze its motion by setting `useOffscreen` to `true`.

The `invisibleColor` and `matteColor` instance variables may need to be set in order for the cached target to appear without a rectangular border around it; these instance variables are defined in the `Bitmap` class.

Note – Setting `useOffscreen` to `true` might yield better performance even for simple targets.

The `backgroundBrush` instance variable specifies the brush used for the background of the cached target, which is a bitmap surface. Set this brush when transitioning a white target to something other than white, and use its color for the invisible color. For targets that are not white, background brush does not need to be set. The example in the next section demonstrates `backgroundBrush`.

Moving Target

The `movingTarget` instance variable specifies whether the entire region that has been redrawn since the start 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 `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.

Using the Transitions Component

This section explains how to set up a transition and gives two examples of how you can use transitions in your scripts.

Setting up a Transition

The following steps show how to set up a transition, using `Wipe` as an example. When you play the wipe transition `myWipe`, it gradually makes the target presenter `myOval` appear in the space `myWindow`.

1. Create an instance of the transition, which can be any subclass of `TransitionPlayer`. Then set the duration and any other unique attributes that particular effect might have. The duration in seconds of this transition is 1 second (`duration` divided by `scale`).

```
global myWipe := new Wipe duration:30 \
       direction:@right \
       target:myOval \
       useOffscreen:true \
       scale:30
myWipe.backgroundBrush := redBrush
```

   Setting the value of `useOffscreen` to `true` means that the transition will make an offscreen copy of the target before playing the transition.

   At this point you could also set `myWipe.autoSplice` to `true`, if you wanted (as shown in Figure 10-4). When `autoSplice` is set `true`, the target presenter would replace the `TransitionPlayer` object at the end of the transition.

2. Add the transition to the space where you want the target to appear.

```
prepend myWindow myWipe
```

3. Since you had set `useOffscreen` to `true` in step 1, you have the option of preparing the effect by creating the offscreen bitmap (`cachedTarget`):

```
playPrepare myWipe 1
```

4. Set the color in the `cachedTarget` bitmap that you want to be invisible. The offscreen bitmap created in the previous step looks identical to the target except for transparency and matte effects. The bitmap has a white

rectangle enclosing the image, which looks fine for transitions over a whole space. However, if this rectangle is unsuitable, and you want it hidden, you can specify that the color white be made invisible by setting the `invisibleColor` instance variable (defined on the `Bitmap` class) to `whiteColor`. However, if you want your target to be white, you will need to set the background brush to some other color, and use that color for your invisible color. For targets that are not white, background brush doesn't need to be set, but can be set to produce interesting effects. (Note that this step must be done after `playPrepare`.)

```
myWipe.cachedTarget.invisibleColor := redColor
```

5. Play the transition. The target is transitioned into the space as follows: The transition player keeps track of time and tells the effect where it should be at each moment. By virtue of being in the space, the transition player is told by the compositor to draw, which in turn tells the transition effect to draw, which draws a different amount of the cached target each time.

   ```
   play myWipe
   ```

   By virtue of being a clock, a transition has a `time` instance variable. After the wipe transition plays to completion, its `time` instance variable continues incrementing.

6. After the transition is complete, since `autoSplice` is `false` (the default), the `TransitionPlayer` object remains in the space exactly as it was at the start of the transition, and can be replayed by calling:

   ```
   goToBegin myWipe
   ```

   This sets the transition's `time` instance variable to `0`, and since the transition's rate is still 1, it begins playing again immediately.

   If `autoSplice` were set to `true` in step 1, the target presenter would replace the `TransitionPlayer` object, and you would not be able to re-play the transition using `goToBegin`.

If, at the end of the transition, you want to remove a presenter from the space (perhaps because the new presenter completely covers it), you can remove it by writing a time callback with a function that deletes the object at the appropriate time. Refer to "Deleting a Hidden Presenter" on page 216 for more details.

Making the Target Disappear

In the previous example, the oval appears as the transition is played. To make the oval disappear, you can simply play the transition backwards.

For example, to play previous transition backward, stop it, go to its end, and set its rate to –1:

```
stop myWipe
goToEnd myWipe
```

```
myWipe.rate := -1
```

This causes the oval to disappear with a wipe effect from right to left, the opposite as before.

Even though the oval disappears from view, the transition is not removed from the window. To remove the transition, write a function to delete it from the window, and call `addTimeCallback` on the transition to trigger that function after the transition has played, as described in "Deleting a Hidden Presenter" on page 216.

Performing a Transition on an Existing Collection

With the technique shown so far, the target is a separate object to be added or deleted—it is either absent at the start and transitioned in (when playing forward) or present at the start and transitioned out (when playing backward). What do you do if you simply want to perform a transition on an entire collection that is not being added or removed? Using the previous technique, you would have to make a deep copy of the collection, transition it in, then delete the original collection. It is not only difficult to make a deep copy of a collection of presenters, but it also wastes time and memory when all you want to do is modify the existing objects. This technique can be useful for performing a transition on a card in a cards-and-stacks title.

The following technique shows you how to perform a transition on an existing collection by fooling the compositor. It is particularly useful when altering a complex presenter or presentation collection, such as a 2D multi-presenter, 2D space, group presenter, or group space.

In short, the technique is as follows:

1. Set up the window as you want it to appear before the transition begins. Then disable the compositor to prevent it from updating the window when the following change happens.
2. Remove the presentation collection from the window, make the collection the target of a new transition, and add the transition back into the window with `autoSplice` set `true`.
3. Set the `changed` flag of the transition to `false` to again fool the compositor, so it doesn't realize a change has been made.
4. Enable the compositor.
5. Change the contents of the presentation collection as you wish.
6. Prepare the transition, then set `direct` of the transition to `true.`
7. Play the transition.

Figure 10-5 illustrates this technique—it depicts a 2D multi-presenter that contains some bitmaps that transition from daylight to nighttime. Here, the colors in the bitmaps are changed and a car is removed and replaced by a motorcycle. The 2D multi-presenter is contained inside a window.

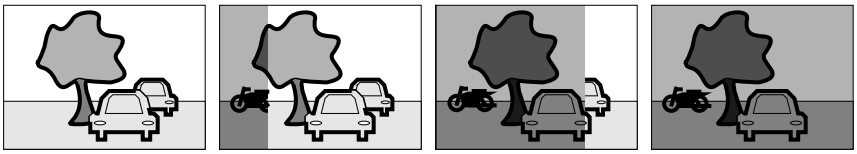


Figure 10-5: Performing a wipe transition on an existing collection.

The following example is an extremely simplified version of Figure 10-5—it has a window that contains a 2D multi-presenter and a rectangle. The transition uses a wipe to change the background fill, remove the rectangle and replace it with an oval. This technique avoids having to create a second 2D multi-presenter as the transition's target.

1. Start out with the original 2D multi-presenter in a window:

```
myWindow := new Window
show myWindow
global myMulti := new TwoDMultiPresenter \
    boundary:(new Rect x2:400 y2:300) \
    fill:(new Brush color:yellowColor)
prepend myWindow myMulti
global myBox := new TwoDShape target:(new Rect x2:300 y2:200) \
    fill:(new Brush color:greenColor)
myBox.x := 50
myBox.y := 50
prepend myMulti myBox
```

Disable the compositor. This does not delete the 2D multi-presenter from memory—it merely removes it from the window.

```
myWindow.compositor.enabled := false
```

2. Remove the 2D multi-presenter from the window, then create the transition player, set its target to `myMulti`, set `autoSplice` to `true`, and add the transition player to the window. Because the transition has not yet been played, its target will not be visible. Once the transition is played, its target will be revealed.

```
deleteOne myWindow myMulti
global myWipe := new Wipe duration:30 direction:@right scale:30
myWipe.target := myMulti
myWipe.autoSplice := true
prepend myWindow myWipe
```

3. Set the `changed` flag `false` for the transition. This prevents the compositor from knowing that you changed the transition.

```
myWipe.changed := false
```

4. Enable the compositor. Even though it is enabled, the compositor does not think the screen needs to be refreshed. Do not move any objects in the window, because that would cause `myMulti` to be erased.

```
myWindow.compositor.enabled := true
```

5. Make whatever changes you want to the 2D multi-presenter. Any changes you make will not appear until you play the transition.

```
myMulti.fill := new Brush color:redColor
deleteOne myMulti myBox
global myOval := new TwoDShape target:(new Oval x2:300 y2:200) \
    fill:blackBrush
myOval.x := 50
myOval.y := 50
prepend myMulti myOval
```

6. Prepare the transition and set `direct` to `true`. Setting `direct` after `playPrepare` causes the original 2D multi-presenter to remain visible until the transition is done. (If you want the original 2D multi-presenter to disappear at the start of the transition, you can omit that statement or move it earlier.)

```
playPrepare myWipe 1
myWipe.direct := true
```

7. Play the transition:

```
play myWipe
```

Calling `play` causes the transition to replace the old contents of the 2D multi-presenter with its new contents. In this case, the rectangle is replaced by the oval, and the background fill changes color.

Loadable Transitions

This section describes the loadable transition classes provided with ScriptX. These classes are C code extensions to ScriptX. The loadable transition classes enable you to create new instances of transitions not available in the core classes, and are alternatives to the built-in transitions.

All loadable transition classes are subclasses of `TransitionPlayer`. The loadable transitions support the instance variables and methods in `TransitionPlayer`, such as the instance variables `duration` and `direction`, and the methods `play`, `stop`, `pause` and `rewind`.

How to Load Transitions

The loadable transition classes are written in C and saved as C library files. You can load them seamlessly into the ScriptX runtime environment. To load the transition classes, execute the following expression:

```
process (new Loader) "loadable/trans"
```

As shown in Figure 10-6, the transition files resides in a machine-specific directory (`mac` or `win`) within the `trans` directory in the `loadable` directory. The `loadable` directory must be in the same directory as the ScriptX application. Each machine-specific folder contains the following three files:

- `group` is the text file with load instructions.
- `ltrans.lib` is the C library file of machine-independent loadable transitions. Machine-independent means the C source file is the same for all platforms, but is compiled for the specific platform.
- `mdltrans.lib` is the C library file of machine-dependent loadable transitions.

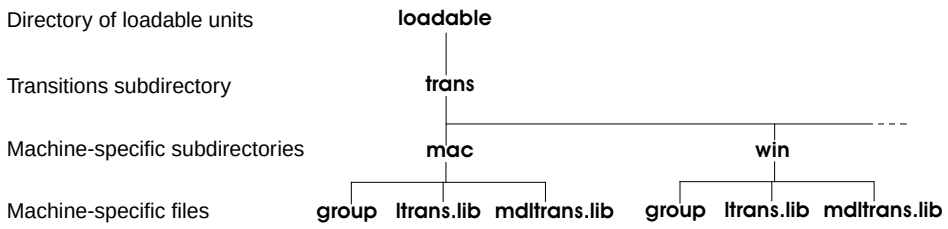


Figure 10-6: Loadable transitions are located in the "loadable" directory.

This previous expression for loading the transitions that contains the `process` method is equivalent to the following series of expressions:

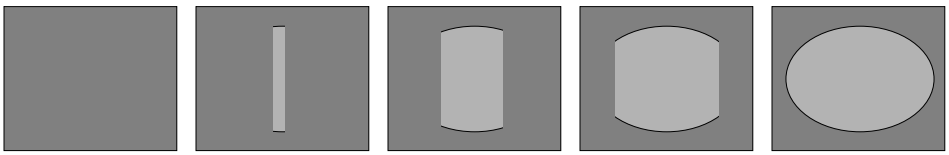
```
-- Create a new instance of Loader
global myLoader := new Loader

-- Runs the text file "group" in "loadable" directory, "trans" subdirectory
global myGroup  := getGroup myLoader "loadable/trans"

-- Gets the machine-independent loadable unit named "ltrans" from myGroup
global unit1 := getLoadableUnit myGroup "ltrans"

-- Loads unit1 from myGroup, returns ID if successful
global firstID := loadModule myLoader mygroup unit1

-- Gets the machine-dependent loadable unit named "mdltrans" from myGroup
global unit2 := getLoadableUnit myGroup "mdltrans"

-- Loads unit2 from myGroup, returns ID if successful
global secondID := loadModule myLoader mygroup unit2
```

What The Transitions Look Like

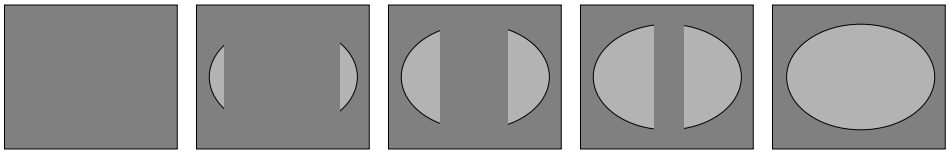
BarnDoor (Core)

The target gradually appears as vertical doors open from the center (`@open`) or from the edge (`@close`) when playing forward. To make the target disappear instead, set the transition's rate to -1.

`@open`



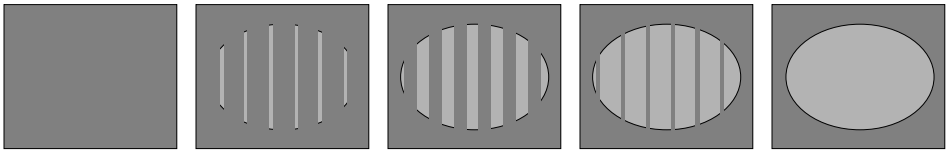
`@close`



Directions: `@open`, `@close`

Blinds (Loadable)

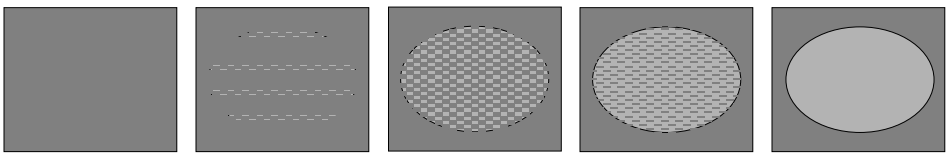
The target gradually appears as vertical (`@vertical`) or horizontal (`@venetian`) bands when playing forward. To make the target disappear instead, set the transition's rate to -1.



Directions: `@venetian`, `@vertical`

Checkerboard (Loadable)

The target gradually appears using a checkerboard effect when playing forward, as shown below. To make the target disappear instead, set the checkerboard's rate to -1.



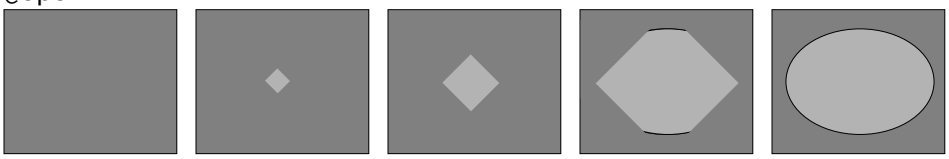
Directions: *(none)*

Rate: Can play forward or backward.

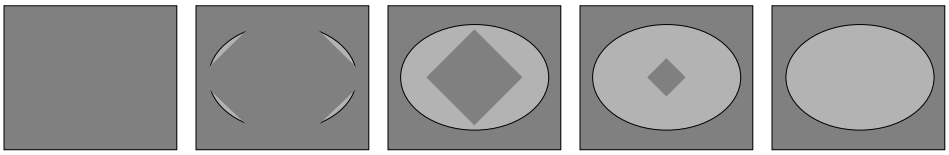
DiamondIris (Loadable)

The target gradually appears using a diamond effect. To make the target disappear instead, set the transition's rate to -1. The `@open` and `@close` directions playing forward are shown below:

`@open`



`@close`

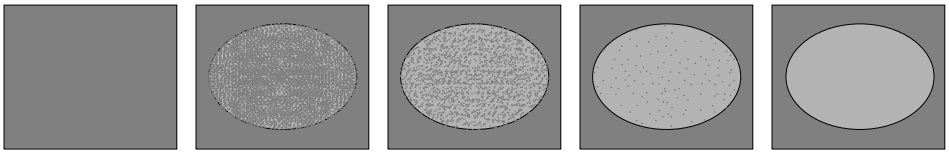


Directions: `@open`, `@close`

Rate: Can play forward or backward.

Dissolve (Loadable)

The target gradually appears as small, random dots, as shown below. Dissolve cannot be played backward.



The dissolve transition has certain restrictions not found in other transitions. During the dissolve transition, all clocks freeze. In addition, the duration of the dissolve transition is not adjustable—the `duration` instance variable is ignored; the dissolve progresses as fast as the processor allows. The duration of the transition is shorter on faster machines.

When you create an instance of `Dissolve`, you must set the `useOffscreen` keyword to `true`. A cache is made of the target presenter—the cache is a frozen image of the target used for the duration of the dissolve.

The `direct` instance variable is automatically set to `true` for dissolve. With `direct` set to `true`, the cached target is drawn directly to the window, in front of other presenters it might overlap.

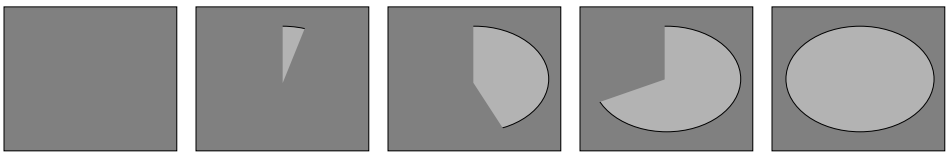
If another window can possibly overlap in front of the dissolve region, you should call `playPrepare` on the dissolve transition before playing it—this will prevent the front window from being overwritten by the dissolve.

Directions: *(none)*

Rate: Must be zero or positive. Cannot play backward.

Fan (Loadable)

The target gradually appears, swept in a clockwise or counter-clockwise fashion. To make the target disappear instead, set the fan's rate to -1. The `@clockwise` direction playing forward is shown below:



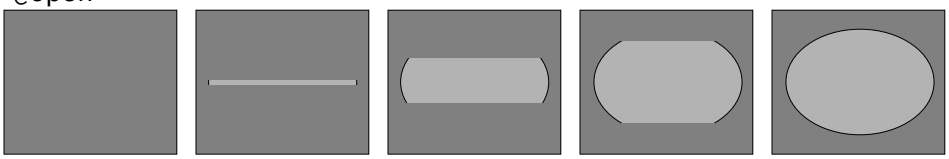
Directions: `@clockwise`, `@anticlockwise`

Rate: Can play forward or backward.

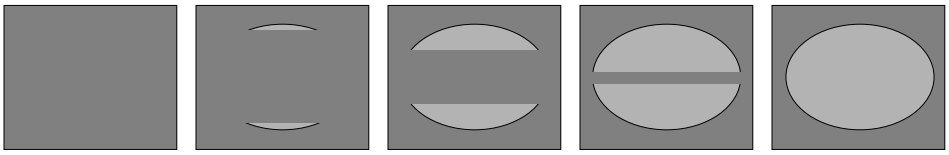
GarageDoor (Loadable)

The target gradually appears horiztonally starting either from the center or from the top and bottom edges. To make the target disappear instead, set the transition's rate to -1. The `@open` and `@close` directions playing forward are shown below:

`@open`



`@close`



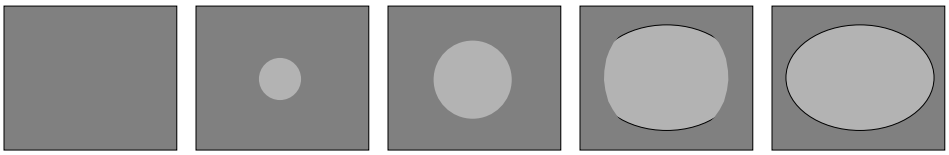
Directions: `@open`, `@close`

Rate: Can play forward or backward.

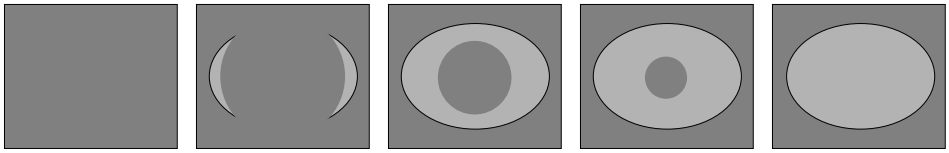
Iris (Core)

Has an effect like 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` to either `@open` or `@close`.

@open

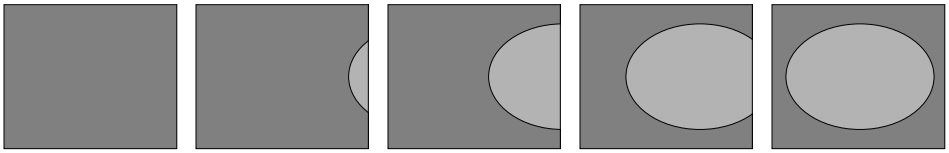


@close



Push (Loadable)

The target gradually appears, as if pushed onto the screen from the given direction. To make the target disappear instead, set the push's rate to -1. The `@left` direction playing forward is shown below:

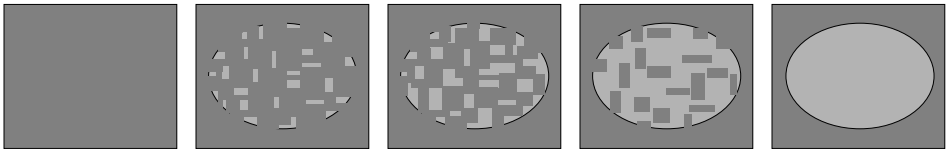


Directions: `@right`, `@left`, `@up`, `@down`

Rate: Can play forward or backward.

RandomChunks (Loadable)

The target gradually appears as random-size squares, rectangles, columns, or rows. To make the target disappear instead, set the transition's rate to -1. The `@rects` direction playing forward is shown below:



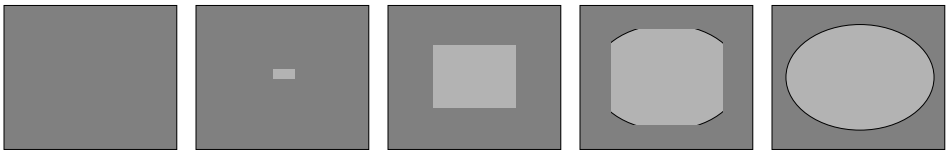
Directions: `@squares`, `@rects`, `@columns`, `@rows`

Rate: Must be zero or positive. Cannot play backward.

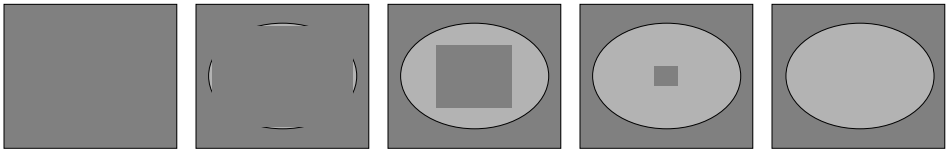
RectIris (Loadable)

The target gradually appears either from the center outward or from the edges inward. The iris's ratio of width-to-height is the same as the target's width-to-height. To make the target disappear instead, set the transition's rate to -1. The `@open` and `@close` directions playing forward are shown below:

`@open`



`@close`

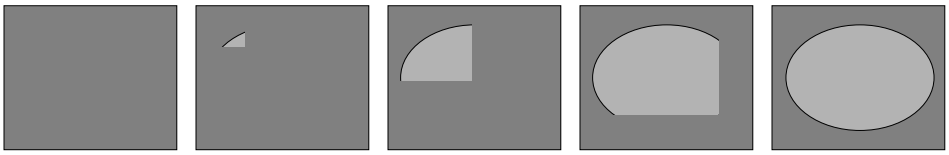


Directions: `@open`, `@close`

Rate: Can play forward or backward.

RectWipe (Loadable)

The target gradually appears, wiped into view toward the given direction. To make the target disappear instead, set the transition's rate to -1. The `@southeast` direction playing forward is shown below:

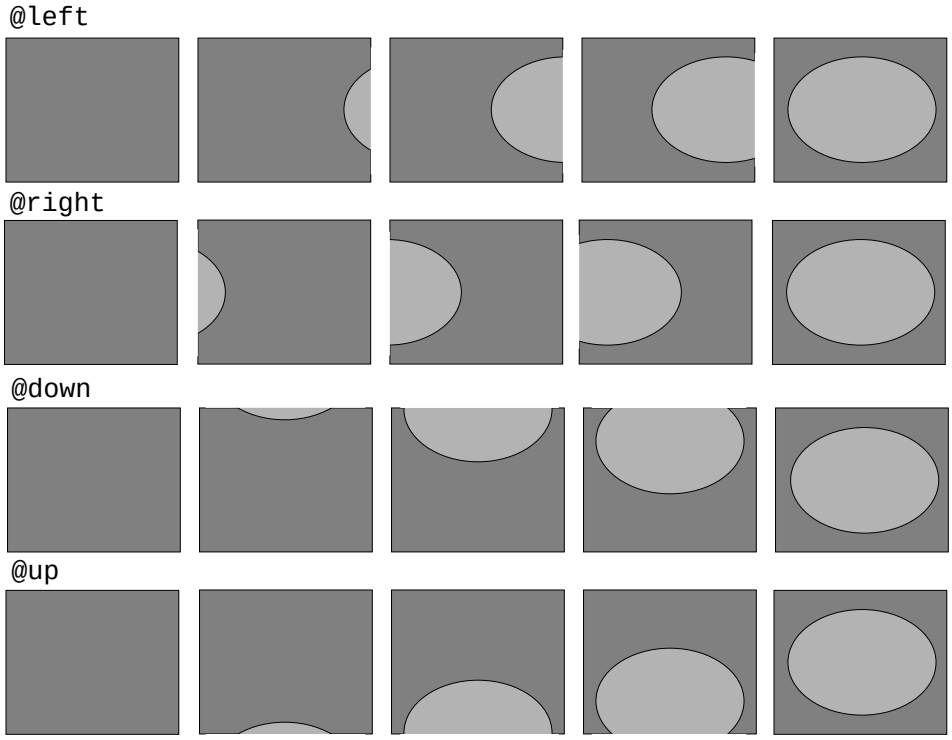


Directions: `@northeast`, `@southeast`, `@southwest`, `@southeast`

Rate: Can play forward or backward.

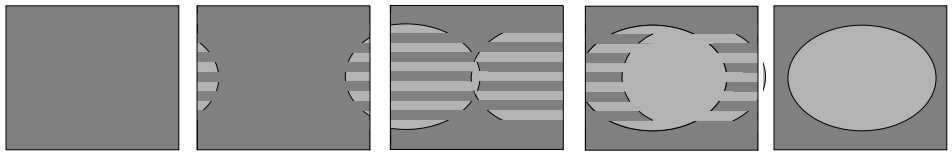
Slide (Core)

The presenter slides onto the screen. You specify the direction in which the target slides by setting the `direction` to `@left`, `@right`, `@up`, and `@down`, as shown below. For example, `@left` means the target image slides onto the screen moving to the left.



StripSlide (Loadable)

Two parts of the target gradually slide together toward the center from opposite sides. To make the target disappear instead, set the transition's rate to -1. The `@horizontal` direction playing forward is shown below:

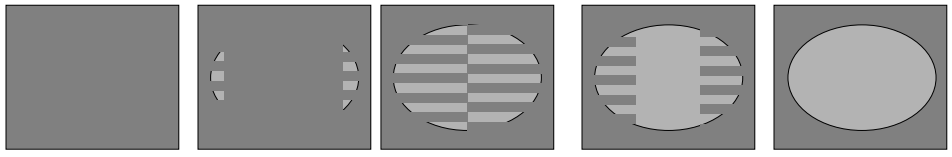


Directions: `@horizontal`, `@vertical`

Rate: Can play forward or backward.

StripWipe (Loadable)

The target gradually appears, wiped into view with either horizontal or vertical strips. To make the target disappear instead, set the transitions's rate to -1. The `@horizontal` direction playing forward is shown below:



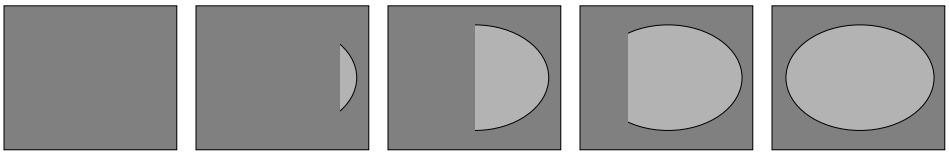
Directions: `@horizontal`, `@vertical`

Rate: Can play forward or backward.

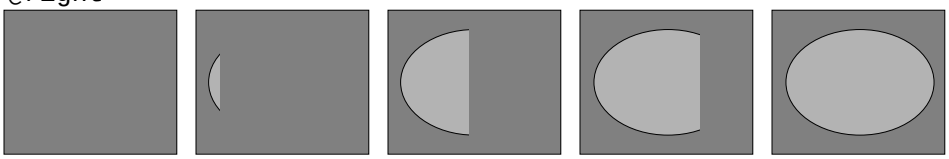
Wipe (Core)

The target is drawn incrementally onto the screen. You specify the direction in which the transition occurs by setting its `direction` instance variable to `@left`, `@right`, `@up`, or `@down`, as shown. For example, `@left` means that the target image wipes onto the screen from right to left.

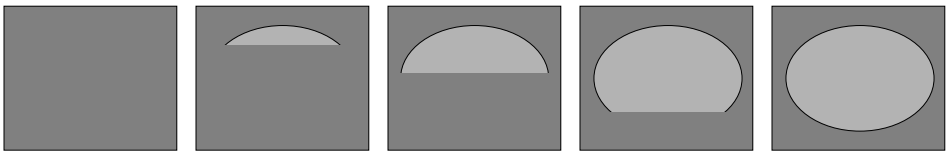
`@left`



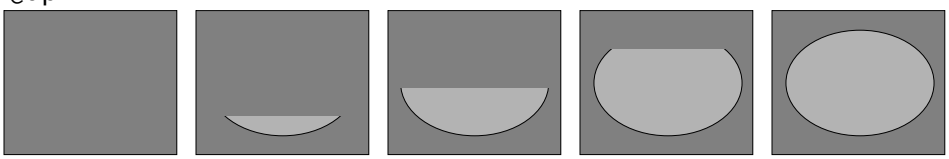
`@right`



`@down`



`@up`



Transition Example

The following is a complete, working example that demonstrates a simple transition.

A Simple Transition

The following script creates a window, displays a red rectangle, then creates an oval and wipes the oval into view from left to right in front of the rectangle, as shown in Figure 10-7. To play the transition again, type: `goToBegin myWipe`.

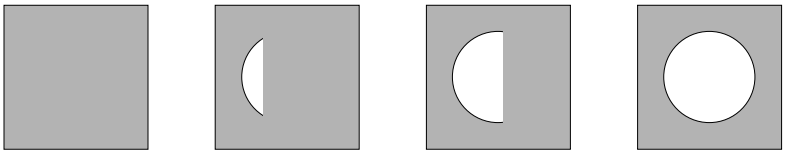


Figure 10-7: The oval is wiped into view

For demonstration purposes, this transition sets `useOffscreen` to `true`, meaning that it creates a cached target, to show how `invisibleColor` must be set to the same value as `backgroundBrush`. (In this example you would normally leave `useOffscreen` set to `false`, because the target being transitioned in is simple enough to run without a cache.)

```
-- Set up a window
global myWindow := new Window boundary:(new Rect x2:400 y2:400)
myWindow.y := 40
show myWindow

-- Create a red rectangle
global redBrush := new Brush color:redColor pattern:blackPattern
global myRect := new TwoDShape target:(new Rect x2:300 y2:300) fill:redBrush
     myRect.x := 50
     myRect.y := 50
append myWindow myRect

-- Create a white circle
global myOval := new TwoDShape target:(new Oval x2:200 y2:200) \
     fill:whiteBrush stroke:blackBrush
     myOval.x := 100
     myOval.y := 100

-- Make the transition
global myWipe := new Wipe duration:30 direction:@right target:myOval scale:30 \
     useOffscreen:true
myWipe.backgroundBrush := redBrush
prepend myWindow myWipe

-- Prepare the transition by creating the cached target
playPrepare myWipe 1

play myWipe

--To play again, type: goToBegin myWipe
```

CHAPTER

11

2D Graphics

The 2D Graphics component provides the classes that enable you to draw images to a surface, be it a visible surface such as a window's display surface, or an offscreen surface.

The ScriptX imaging system uses a stencil-and-paint brush imaging metaphor. The stencil is the image to be rendered, and the paint brush indicates the color and pattern to render it with. In addition, each stencil must be presented by a presenter.

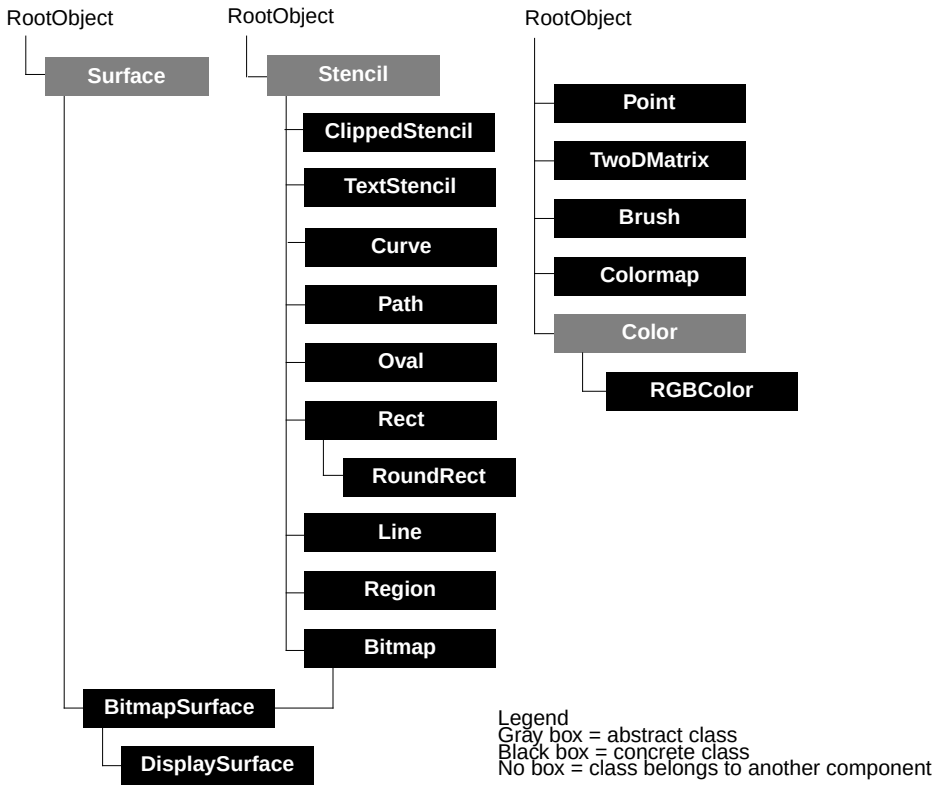
The predefined stencils provided by ScriptX included rectangles, rounded rectangles, lines, ovals, text stencils, curves, and paths. Paths can consist of any number of lines, curves, and splines, and can be open-ended or closed. The predefined stencils also include bitmaps, which are images that you generate in an external program, such as a paint, photo-editing, or scanning program. You bring bitmaps into ScriptX by using the importer.

Each kind of image is represented as a subclass of `Stencil`, such as `Rect`, `RoundRect`, `Line`, and `Bitmap`. Paint brushes are represented by `Brush` objects, that have color, pattern, and inkmode attributes. Stencils are presented by 2D presenters, such as `TwoDShape` objects, which presents a single stencil.

`Stencil` objects work in close conjunction with `TwoDPresenter` objects. This chapter talks about the `TwoDPresenter` class and the `TwoDShape` class as much as is necessary to explain how to present stencils to a surface. The `TwoDPresenter` class is discussed in detail in Chapter 3, "Spaces and Presenters."

Classes and Inheritance

The class inheritance hierarchy for the 2D Graphics component is shown in the following figure.



The following classes form the 2D Graphics component. In this list, indentation indicates inheritance.

`Surface` – An abstract rendering plane that defines the basic drawing operations: fill, stroke, and transfer.

- `BitmapSurface` – A surface representing an area of memory that can be drawn onto.
- `DisplaySurface` – A surface representing a drawing area on the screen of a particular graphic device. In windowing environments, represents a window.

`Point` – A point in a 2D coordinate system, used for positioning or rendering.

`Stencil` – An image to be rendered onto a surface.

- `Bitmap` – A stencil containing pixel values that either map to colors in a particular color space or color map, or that map to colors directly. Bitmaps can also define an invisible value and a matte value for transparent pixels.
- `ClippedStencil` – A stencil representing two stencils—one of which is used to clip the other.
- `Curve` – A stencil representing a cubic Bézier curve.
- `Line` – A stencil that represents a straight line.
- `Oval` – A stencil that represents an oval or circle.

`Path` – A stencil representing a general path made of straight lines, arcs, curves, and splines. Paths may be continuous or discontinuous, and open or closed.

`Rect` – A stencil representing a rectangle with right-angle corners.

`RoundRect` – A stencil representing a rectangle with rounded corners.

`Region` – A stencil representing an arbitrary geometric shape. Transformations performed on other kinds of stencils often return `Region` objects.

`TextStencil` – A stencil representing non-editable text defined by a font and a character string.

`Brush` – A combination of parameters and objects, such as color, pattern, transfer mode, and line-width, used in stroking or filling stencils onto a surface.

`Color` – A generalized representation of color.

`RGBColor` – A color subclass representing any RGB or grayscale color value.

`Colormap`– An array of `Color` objects that can be used as mappings from pixel values in a `Bitmap` instance to a particular color space.

`TwoDMatrix` – An affine matrix for manipulating 2D coordinates, providing both affine transformations—translation, scaling, and rotation—and non-affine transformations.

Conceptual Overview

In the ScriptX imaging model, the form or shape of an image is defined by a stencil. Like a physical stencil, a ScriptX stencil represents a *potential* image. To actually draw the image, you render the stencil to a surface. This rendering is done by a presenter, which determines what brush (paint) to use to fill the image and paint its outline. The stencil-and-paint brush model is illustrated in Figure 11-1.

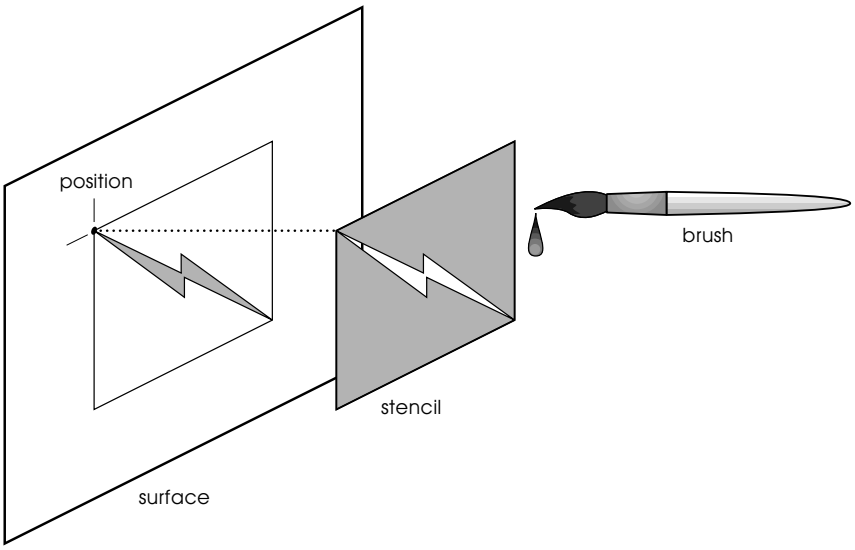


Figure 11-1: The ScriptX imaging model

How 2D Graphics Objects Work

For simple images, you use a `TwoDShape` object to present a `Stencil` object, such as a `Rect`, `Line`, or `Oval`. A `TwoDShape` object has a `boundary` instance variable that specifies the stencil to be rendered. A `TwoDShape` object also has a `fill` instance variable that holds the brush for filling the stencil, and a `stroke` instance variable that holds the brush for painting the stencil's outline.

To display a `TwoDShape` on the screen, add it to a visible space, such as a window, or any other kind of space or presenter that is ultimately embedded in a window.

The following code shows how to display an oval that is 100 pixels wide by 50 pixels high in a window. The oval is filled in white and has a black outline (or stroke). The top left corner of the rectangle that completely encloses the oval (that is, its bounding box) is positioned at the point (50, 50) in the window.

```
-- create and show the window
global w := new window
show w

-- create the oval stencil
global oval1 := new oval x2:100 y2:50

-- create a presenter to present the stencil
global shape1 := new TwoDShape boundary:oval1 \
    fill:whiteBrush stroke:blackBrush

-- position the presenter
shape1.x := 50
shape1.y := 50
```

```
-- add the presenter to the window
prepend w shape1
```

The presenter of a stencil determines where the stencil is drawn, and what brushes are used to fill it and paint its outline. The stencil itself simply determines the kind of image to be drawn, and also its position relative to its presenter's coordinate system. (See "Positioning Images" on page 239 for more information on coordinate system's.)

Each window has a compositor (specifically, an instance of the class `TwoDCompositor`) that is responsible for keeping the visible contents of the window up to date. Whenever a presenter in the window changes in any way, for example, its position or fill changes, the compositor calls the `draw` method on the presenter. The `draw` method in turn calls the `fill`, `stroke`, and/or `transfer` methods to render stencils to the surface of the window. The invocation of the `draw` method, and subsequent invocations of `fill`, `stroke`, and `transfer` all happen automatically behind the scenes as needed. You never need to call the `draw` method directly.

The predefined presenter class `TwoDShape` presents a single stencil. If you want to create a new subclass of `TwoDPresenter` that draws stencils in a special way, you need to define the `draw` method for your new class. For example, if you want a class called `ShadowedShape` that draws a shape with a shadow, then you would need to define the `draw` method to call the `fill` method to render the shadow, call the `fill` method again to render the shape, and call the `stroke` method to render the outlines.

"Creating New Classes of Graphic Presenters" on page 261 delves into details on how to define the `draw` method for new subclasses of `TwoDPresenter`.

Positioning Images

A stencil is usually presented by a presenter, which may be presented by another presenter (and so on) until finally it is presented by a window. Each stencil and each presenter has its own coordinate system, which is mapped to the presenter that is presenting it.

The origin of all coordinate systems in ScriptX is the top left corner. X values increase to the right, and y values increase going down. Coordinates in ScriptX can be specified with integer or floating point values. Units in the default ScriptX coordinate system represent screen pixels. Each point in a coordinate system actually represents an area of one pixel extending down and to the right of the specified coordinates.

Positioning TwoDPresenters

The `x` and `y` instance values of a presenter determine its position in the object that is presenting it, be it a window or another kind of presenter. The `position` instance variable also indicates the position of the presenter as a `Point` object that has `x` and `y` values. You can use either the `x` and `y` instance

variables or the `position` instance variable, to change the position of a presenter. If the `position` instance variable changes, the `x` and `y` values are changed automatically and vice versa.

The following four code samples each move the `TwoDShape` object to the position (100, 40) in the coordinate system of the object that is presenting it. Example 1 moves the shape in one step, but also creates a new object. (For optimum performance, you should try to minimize the number of unnecessary objects that get created.)

Example 2 and Example 3 each move the shape in two steps and do not create any extra objects. Example 3 is preferred just because it involves less typing.

Example 4 has the effect of moving the shape in one step, since it stops the window from updating until the move has finished. All `TwoDPresenter` objects have a `window` instance variable that indicates the window that currently contains the presenter. Setting the window's compositor's `enabled` instance variable to `false` prevents the compositor from updating the window. Setting it to true switches the compositor back on.

```
-- Example 1
myShape.position := new Point x:100 y:40

-- Example 2
myShape.position.x := 100
myShape.position.y := 40

-- Example 3
myShape.x := 100
myShape.y := 40

-- Example 4
myShape.window.compositor.enabled := false
myShape.x := 100
myShape.y := 40
myShape.window.compositor.enabled := true
```

Positioning Stencils

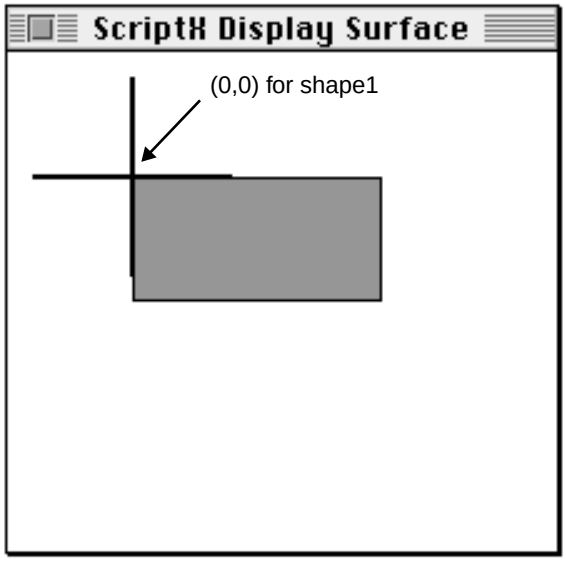
Most stencils have `x1`, `y1`, `x2`, and `y2` instance variables. The `x1`, `y1`, `x2`, and `y2` values of a stencil determine its position within the coordinate system of the presenter that is presenting it. The point `x1`, `y1` is the top left corner of the stencil in the coordinate system of its presenter. The point `x2`, `y2` is the bottom right corner.

The width of a stencil is always determined by (`x2 - x1`), and its height is always determined by (`y2 - y1`). As a general rule, when creating stencils, you can let `x1` and `y1` default to zero, and use `x2` and `y2` to indicate the width and height of the stencil. There may be times however, when you may want to set `x1` and `y1` to non-zero values to achieve results that require the stencil to be offset from the origin of its coordinate system.

Note – `Bitmap` stencils do not have `x1`, `y1`, `x2`, and `y2` instance variables. Instead, they have a `bbox` instance variable, that specifies the bounding box for the bitmap. All discussions regarding `x1`, `y1`, `x2`, and `y2` instance variables for stencils in this chapter apply to the `x1`, `y1`, `x2`, and `y2` instance variables of the bounding box of a bitmap, rather than to a bitmap itself.

Figure 11-2, Figure 11-3, Figure 11-4, and Figure 11-5 show the position of a stencil relative to its presenter, for different values of `x1`, `y1`, `x2`, and `y2` values of the stencil. In all four figures, the x-axis and y-axis lines indicate the origin of the `TwoDshape`, `shape1`, although you would not really see these in the window.

Figure 11-2 shows a `TwoDShape`, `shape1`, displayed in a window. Both the x and y values of the shape are 50. The boundary of the 2D shape is a `Rect` object whose `x1` and `y1` values are both zero, which means that the top left corner of the `Rect` is positioned at the top left corner of the presenter. The stencil is positioned at (0,0) in the presenter's coordinate system, and at (50, 50) in the window's coordinate system.

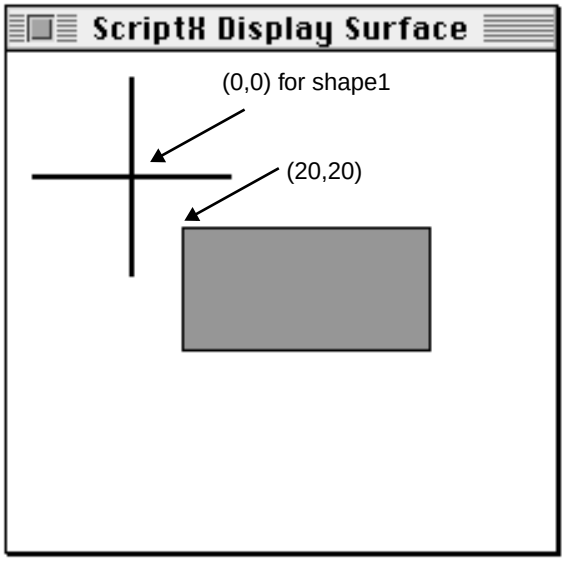


The intersection of the horizontal and vertical lines indicate the origin of the coordinate system for the TwoDShape, shape1.

```
shape1.x := 50
shape1.y := 50
shape1.boundary := new Rect \
    x1: 0 \
    y1: 0 \
    x2: 100 \
    y2: 50
```

Figure 11-2: Top left corner of the stencil and origin of its presenter coincide

In Figure 11-3, the shape is still positioned at (50, 50) in the window. However, the `x1` and `y1` values of the stencil are no longer zero. The `x1` value is 20 and the `y1` value is 20. The top left corner of the stencil is now positioned at the point (20, 20) in the presenter's coordinate system, and at the point (70, 70) in the window's coordinate system.

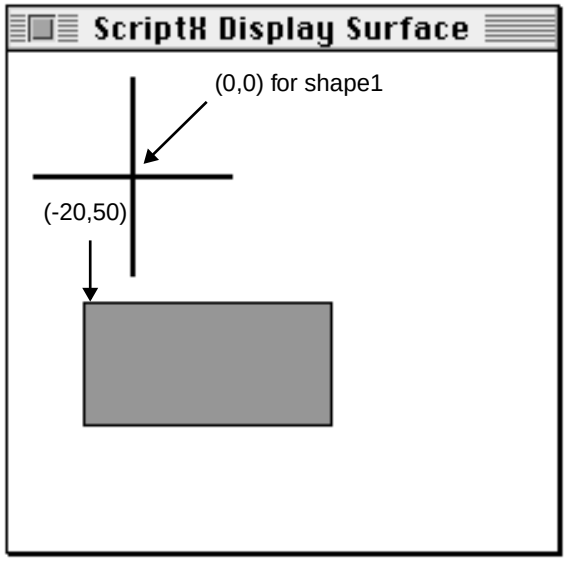


The intersection of the horizontal and vertical lines indicate the origin of the coordinate system for the TwoDShape, shape1.

```
shape1.x := 50
shape1.y := 50
shape1.boundary := new rect \
    x1: 20 \
    y1: 20 \
    x2: 120 \
    y2: 70
```

Figure 11-3: Top left corner of the stencil is right and down from its presenter's origin

In Figure 11-4, the shape is still positioned at (50, 50) in the window. Now, the `x1` value of the stencil is –20 and the `y1` value is 50. The top left corner of the stencil is positioned at the point (–20, 50) in the presenter's coordinate system, and at the point (30,100) in the window's coordinate system.



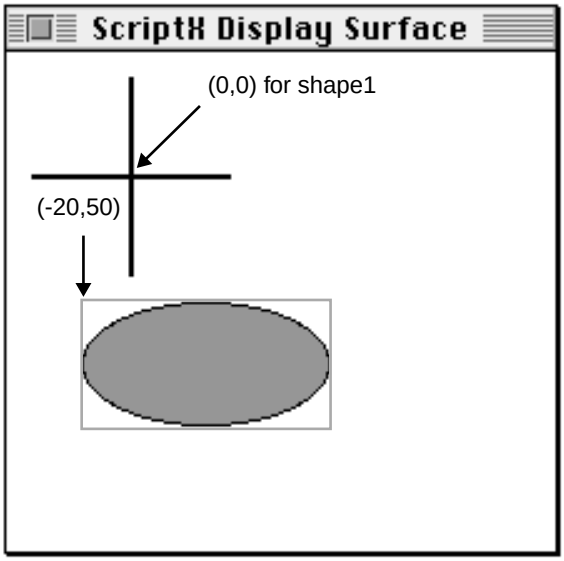
The intersection of the horizontal and vertical lines indicate the origin of the coordinate system for the TwoDShape, shape1.

```
shape1.x := 50
shape1.y := 50
shape1.boundary := new rect \
    x1: –20 \
    y1: 50 \
    x2: 80 \
    y2: 100
```

Figure 11-4: Top left corner of the stencil is left and down from its presenter's origin

For stencils other than `Rect` objects, the `x1`, `y1`, `x2`, and `y2` values determine the position and size of the bounding box, that is, the smallest rectangle that completely contains the image. Figure 11-5 shows the position of an `Oval` instead of a `Rect`. In this figure, the bounding box of the oval is shown by a

box with a dashed line, although you would not really see the bounding box. Notice that the bounding box for the `Oval` in Figure 11-5 is the same shape and position as the `Rect` in Figure 11-4.



The intersection of the horizontal and vertical lines indicate the origin of the coordinate system for the TwoDShape, shape1.

```
shape1.x := 50
shape1.y := 50
shape1.boundary := new Oval \
    x1: -20 \
    y1: 50 \
    x2: 80 \
    y2: 100
```

Figure 11-5: An oval and its bounding box

Using 2D Graphics

The use of the classes in the 2D Graphics component is closely connected with the use of the `TwoDPresenter` subclasses, since stencils need to presented by presenters.

Drawing Lines, Ovals, Rectangles and Rounded Rectangles

Use the `Line`, `Oval`, `Rect`, and `RoundRect` subclasses of `Stencil` to specify lines, ovals, rectangles, and rounded rectangles respectively.

When creating new instances of `Line`, `Oval`, `Rect` and `RoundRect`, you can specify the `x1`, `x2`, `y1`, and `y2` values. For `RoundRect`, specify the `rx` and `ry` values to indicate the `x` and `y` radius of the corners.

When using lines, it is easy to forget that the line width is not determined by an instance variable on the `Line` object. Instead, it determined by the value of the `lineWidth` instance variable of the brush in the `fill` or `stroke` instance variable of the presenter presenting the line.

The following code creates a line, rectangle, oval, and rounded rectangle and displays them in a window.

```
-- create a line
global line1 := new Line x2:100 y2:100
lineShape:= new TwoDShape boundary:line1 \
   fill:(new brush color:blackColor)
lineShape.fill.linewidth := 10
```

```
lineShape.x := 100
lineShape.y := 100

-- create a rectangle
global rect1 := new Rect x2:80 y2:60
rectShape:= new TwoDShape boundary:rect1 stroke:blackBrush \
   fill:(new brush color:(new RGBcolor red:50 blue:50 green:50))
rectShape.x := 10
rectShape.y := 10

-- create an oval
global oval1 := new Oval x2:60 y2:100
ovalShape:= new TwoDShape boundary:oval1 stroke:blackBrush \
   fill:(new brush color:(new RGBcolor red:125 blue:125 green:125))
ovalShape.y := 100
ovalShape.x := 10

-- create a rounded rectangle
global roundRect1 := new RoundRect x2:100 y2:80 rx:30 ry:30
roundRectShape := new TwoDShape boundary:roundRect1 stroke:blackbrush \
   fill:(new brush color:(new RGBcolor red:220 blue:220 green:220))
roundRectShape.x := 110
roundRectShape.y := 10

global w := new window
w.width := 220
w.height := 220
show w

prepend w lineShape
prepend w rectShape
prepend w ovalShape
prepend w roundRectShape
```

Figure 11-6 shows the results.

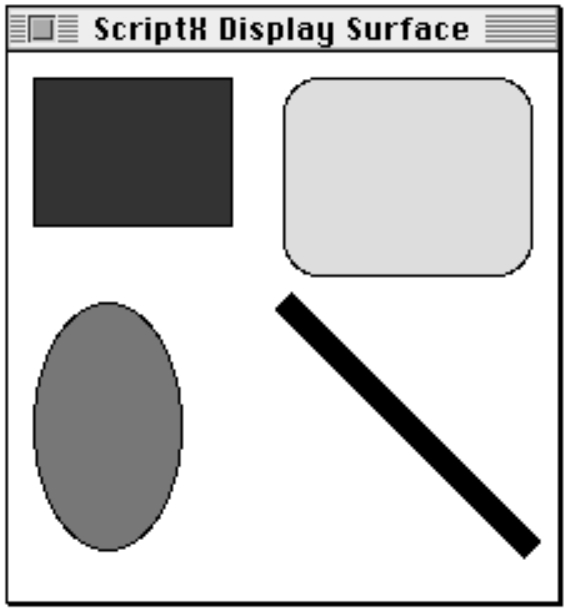


Figure 11-6: Rectangle, RoundedRectangle, Oval and Line

Drawing Curves

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.

The `Curve` class represents Bézier cubic curves. (Postscript and Adobe Illustrator use cubic Bézier 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 11-7.

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 Bézier 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 halfway along the curve, a tenth of the way along the curve, and so on.

The following code example shows how to create the curve shown in Figure 11-7.

```
curve1 := new Curve x1:0 y1:0 \
                    x2:100 y2:250 \
                    x3:200 y3:250 \
                    x4:300 y4: 0
s := new TwoDShape boundary:curve1 stroke:blackbrush
```

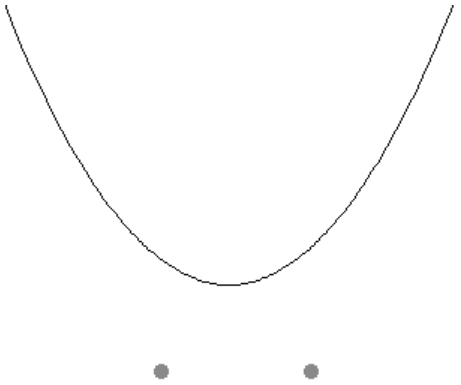


Figure 11-7: Curve1 – The dots indicate the control points

Drawing Paths

You can use the `Path` class 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 11-8 shows some sample paths.

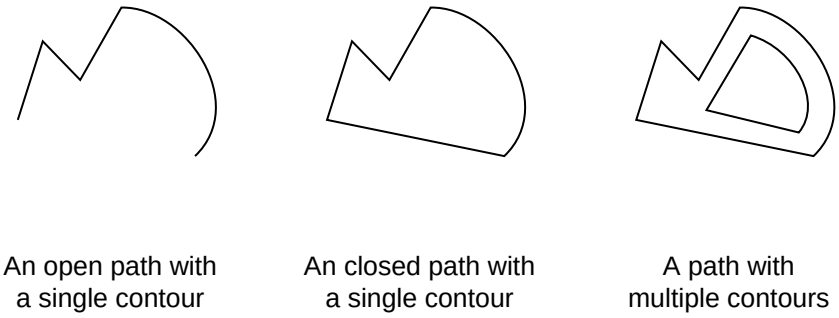


Figure 11-8: Sample paths

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 11-9, which shows an unfilled path and the same path filled.

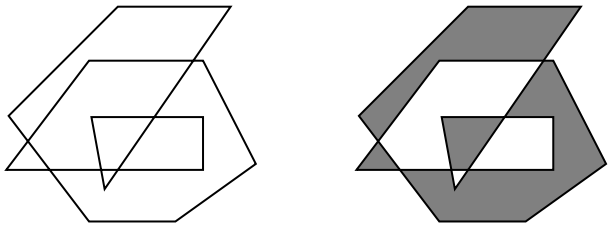


Figure 11-9: If a path crosses itself, filling alternates each side of a crossed line.

To create a new path, call the `new` method on the class `Path`. The `Path` class behaves a little differently than 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 add new segments (such as lines, curves, arcs, or splines) to it, but 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, call `notifyChanged true` on the presenter.

See the description of the `Path` class in the *ScriptX Class Reference* for full details and examples of how to add lines, arc, curves, and splines to a path.

Drawing Text Stencils

The `TextStencil` class is another subclass of `Stencil`. It provides non-editable text for use in drawings as callouts or labels.

Figure 11-10:The TextStencil class provides non-editable text

You would use a `TwoDshape` that has a `TextStencil` as its `boundary` instead of using a `TextPresenter` that has a `Text` object as its `target`, when you want the behavior of a `TwoDShape` when displaying text.

To create an instance of `TextStencil`, call `the` new method and give the font and string keyword arguments. For example:

```
textSten := new TextStencil \
       font:(new PlatformFont name:"Arial") \
       string:"Information"
```

If the string is a `Text` object that already has font and font size attributes, these characteristics will be retained if the `font` keyword is omitted. However, if the `font` keyword argument is also passed, it overrides the font setting on the `Text` object.

Note – Unlike `Text` objects, a `TextStencil` object cannot change its attributes for various character. It has only one set of attributes (font and size). Hence, the `TextStencil` will copy the attributes of a `Text` at the location 0.

Note – Text stencils can be transformed with matrices (for example, they can be rotated and scaled) but once you transform a text stencil, it is converted to a `Region` object and you cannot get it back to the original form.

Drawing Bitmaps

The `Bitmap` class represents bitmap images. Normally you create new `Bitmap` instances by importing bitmaps, instead of by calling the `new` method on the class `Bitmap`.

To import a bitmap, you need to create a `Stream` object that points to the file containing the picture, and then call the `importMedia` method on the global importer instance, `theImportExportEngine`, telling it which stream to import.

The following code shows how to import a picture saved in a file called `fluke.kic` and convert it to a `Bitmap` object in ScriptX. The data for the bitmap is saved in the title container `tc`.

```
stream1 := getStream theScriptDir "fluke.kic" @readable
flukebm := importMedia theImportExportEngine stream1 \
   @image @kic @bitmap container:tc
fluke1 := new TwoDShape boundary:flukebm
```

For details on the arguments to `importMedia`, see the “Importing” chapter in the *ScriptX Tools Guide*.

Currently, ScriptX can import images saved in pict, dib, bmp, or kic format. (KIC is Kaleida’s own Image Compression format.)

MatteColor and InvisibleColor

Bitmap objects have `matteColor` and `invisibleColor` instance variables.

The value in the `matteColor` instance variable indicates the color that is to be transparent in the bitmap's matte area, which is the area between the bitmap's bounding box and the actual image. The color in this area is usually a single color, most often white, although it could be another color such as gray or black. Any pixel in the matte area that uses the color indicated in the `matte` instance variable will be transparent, that is, images underneath will show through it.

If the `matteColor` instance variable is undefined, or is set to any color other than the one actually used in the matte area, the matte area will be solid; that is, it will not be transparent.

The value in the `invisibleColor` instance variable indicates a color that will be transparent anywhere in the region occupied by the bitmap. Any pixel in the bitmap that uses the `invisibleColor` will be transparent.

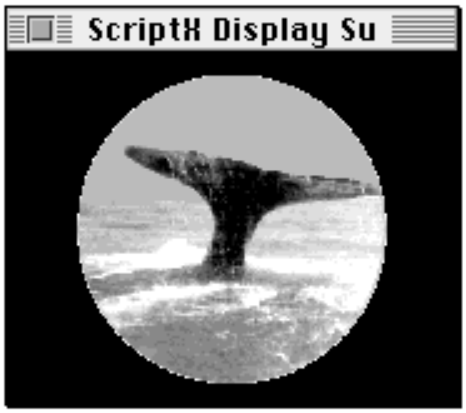
Figure 11-11 shows a bitmap of a whale's tale as it appears when the bitmap's `matteColor` and `invisibleColor` are undefined. Figure 11-12 shows the impact of setting the `matteColor` and `invisibleColor` to various values. Figure 11-12 illustrates that the invisible color only affects pixels that have exactly the specified color.

Figure 11-11:A bitmap of a whale tail with matteColor and invisibleColor undefined

Bitmap Compression

The ScriptX importers, discussed in the *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*. and the `Bitmap` description in *ScriptX Class Reference*. You can change the value of this instance variable when desired.




```
flukebm.mattecolor := whitecolor
notifyChanged fluke1 true
```

Notice that the white area surrounding the circle have become invisible, that is, you can see the background through it.


```
flukebm.invisiblecolor := whitecolor
notifyChanged fluke1 true
```

Notice that all white pixels have become invisible, both in the actual picture and in the matte area. You can see the background through them.


```
w.fill := whitebrush
flukebm.invisiblecolor := blackcolor
notifychanged fluke1 true
```

Notice that all black pixels have become invisible, both in the actual picture and in the matte area. You can see the background through them. (The background has been changed to white.)

Figure 11-12:Effects of matteColor and invisibleColor on the whale tail bitmap

Freeing Bitmap Data

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, by calling `makePurgeable` on it, 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 around 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 around to cleaning up the object, it completes the cleanup by freeing the memory occupied by the `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.

Bitmaps and Colormaps

`Bitmap` objects use `ColorMap` objects. The discussion of color maps is fairly lengthy, so it is presented in its own section, “ColorMaps” on page 259.

Clipping Shapes

If you want to draw a part of a stencil, rather than the whole stencil, you can use a `ClippedStencil` object.

`ClippedStencil` has `clip` and `outline` instance variables, that each take another stencil as their value. The outline stencil is the stencil to draw, and the clip stencil is the stencil to use as the clipping region.

For example, to draw the bottom half of an oval:

```
clipSten := new ClippedStencil \
    clip:(new rect x1:0 y1:75 x2:200 y2:150) \
    outline:(new oval x2:200 y2:150)

shape1 := new TwoDShape boundary:clipSten \
    fill:(new brush color:blueColor)
```

When `shape1` is added to a visible space such as a window, it displays the bottom half of an oval that would be 200 pixels wide by 150 pixels high if it were drawn in full.

Specifying Fill and Brush Attributes for Shapes

A `Stencil` object simply defines a shape. It says nothing about the color and pattern of the shape, or the color and pattern of its border.

To become visible, a stencil must be rendered by a 2D presenter, such as a `TwoDShape` object. Most 2D presenters have `fill` and `stroke` instance variables that each hold a `Brush` object. The value in the `fill` instance

variable indicates what brush to use to fill in the shape rendered by the presenter. The value in the `stroke` instance variable indicates what brush to use to paint the outline of the shape.

Each `Brush` object has `color`, `lineWidth`, `pattern`, and `inkMode` instance variables.

The `lineWidth` value is an integer indicating the width of the line or the outline of the shape. The default value is 1 pixel. The `pattern` instance variable indicates the brush's pattern, and the `inkMode` specifies the mixing characteristics used to apply the brush over a previous image. See the *ScriptX Class Reference* for more details of the `pattern` and `inkMode` values. By default, a brush uses a solid pattern and has an ink mode that completely covers any underlying image.

The color instance variable indicates the color of the brush. The value is an `RGBColor` object, that has `red`, `green`, and `blue` instance variables, that have values between 0 to 255 inclusive, where 255 indicates maximum saturation of that color.

ScriptX has several pre-defined colors, including `blueColor`, `cyanColor`, `yellowColor`, `whiteColor`, `blackColor`, `magentaColor`, `greenColor`, and `redColor`. You can use these when you want the colors they provide. For example, `shape1` represents a red square that has a green outline.

```
shape1 := new TwoDShape boundary:(new Rect x2:100 y2:100) \
    fill:(new brush color:redColor) \
    stroke:(new brush color:greenColor)
```

You can also create new instances of the class `RGBColor` and specify the `red`, `green`, and `blue` values as desired. For example:

```
shape2 := new TwoDShape boundary:(new Rect x2:100 y2:100) \
    fill:(new brush color:(new RGBColor red:100 blue:50 green:20)) \
    stroke:(new brush color:(new RGBColor red:200 blue:100 green:40))
```

ScriptX provides two pre-defined brushes: `blackBrush` and `whiteBrush`, which paint in black and white respectively, with a linewidth of 1 pixel. If you want to use another color, you must use one of the pre-defined `Color` objects, or create your own `RGBColor` instance.

Modifying Shapes

If you change the value of an instance variable on a presenter, such as a `TwoDShape`, the presenter immediately redraws the thing it is presenting. For example, if you change the `width`, `height`, `x`, `y`, `stroke`, or `fill` instance variables, the shape immediately updates on the screen. However, if you make an indirect change to the presenter, it does not redraw itself on the screen. An indirect change is defined as a change to an instance variable on an object that is used by the presenter.

For example, if you change the value of the `x1`, `y1`, `x2`, or `y2` instance variables of the `boundary` of a `TwoDShape`, or you change the `lineWidth` instance variable of the brush used by the `stroke` or `fill`, the shape does not change. After making an indirect change, you can call the `notifyChanged` method on the presenter, to tell the compositor to redraw the presenter. For example:

```
-- make the shape a bit redder
shape1.fill.color.red := shape1.fill.color.red + 20

-- increase the width of the shape's outline
shape1.stroke.linewidth := 5

-- now update the presenter on the screen
notifyChanged shape1 true
```

Rotating and Scaling Stencils

You can perform transformations such as rotation and scaling on stencils by transforming them using 2D matrixes. In computer graphic systems, transformations are represented by a matrix that maps one coordinate system to another. To represent transformations in a ScriptX title, you use instances of the `TwoDMatrix` class.

This chapter limits its discussion of transformations and 2D matrices to explaining how to use them to rotate and scale stencils. For more details of the class `TwoDMatrix` see the *ScriptX Class Reference*.

To create a matrix that can perform a transformation, start by making a mutable copy of the default matrix, `identityMatrix`, and perform a sequence of operations on it, such as rotating, scaling, or translating it.

After performing the desired operations on the matrix, you then use the matrix to transform the stencil.

Transforming a Stencil

To use a matrix to transform a stencil, call `transform` on the stencil, specifying the matrix and also specifying either `@create` or `@mutate`, depending on whether you want the operation to return a new stencil or destructively modify the original stencil.

You can transform any kind of stencil with a matrix, and this example uses a `Path` stencil. The following code creates the lightening bolt stencil to use in the example:

```
global lightningBolt := new Path
moveTo lightningBolt 0 0
lineTo lightningBolt 100 60
lineTo lightningBolt 40 30
lineTo lightningBolt 60 30
closePath lightningBolt
```

The following code increases the length of the lightning bolt stencil by 1.5 times, and rotates it 45 degrees in a clockwise direction about the origin, which is at its top left corner.

```
lightningTransform := mutableCopy identityMatrix
lightningTransform := scale lightningTransform 1.5 1
lightningTransform := rotate lightningTransform 45 @degrees
global newBolt := transform lightningBolt lightningTransform @create
```

This code would effectively perform a sequence of transformations on the path, resulting in the rotated image shown in Figure 11-13.

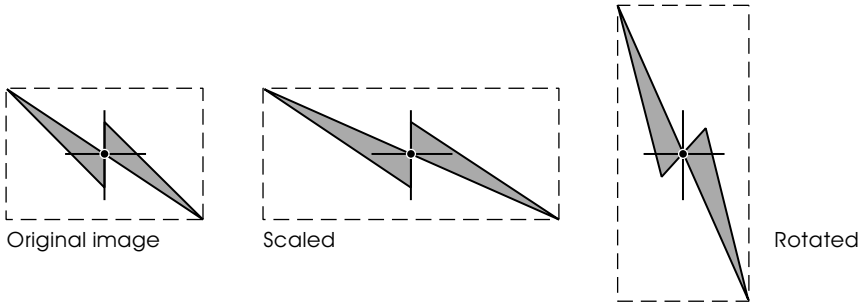


Figure 11-13:Scaling followed by rotation

There are a couple of things to note from this example. First, note that the bounding box of a stencil is redefined as the stencil is transformed. Even though an image is rotated, its bounding box remains rectangular, with sides parallel to the original coordinate system.

It's important to note that the order of matrix operations affects the results. For example, rotation followed by scaling produces different results than scaling followed by rotation. This can be demonstrated by reversing the order of transformations in the previous example:

```
lightningTransform := rotate (mutableCopy identityMatrix) 45 @degrees
lightningTransform := scale lightningTransform 1.5 1
transform lightningBolt lightningTransform @mutate
```

The result is the scaled image shown Figure 11-14.

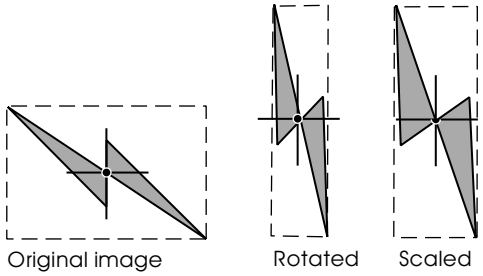


Figure 11-14:Rotation followed by scaling

Translating a Matrix

To translate a matrix, specify the amount to move in the x direction and the amount to move in the y direction. You usually move presenters by specifying values for their `x` and `y`, or their `position`, instance variables, rather than by performing translations on their stencils. However, there are times when you may need to translate a matrix after performing another operation on it, to get it back to where you want it. For example, a matrix rotates about its origin, which is in the top left corner of its coordinate system, so if you want to rotate it about its center, you need to translate the matrix as well as rotate it, as discussed in "Rotating a Matrix" on page 255.

Scaling a Matrix

When scaling a matrix, you specify the x scale factor and the y scale factor. Scaling occurs relative to the origin of the stencil. (Don't forget that (0,0) is at the top left corner of the stencil's coordinate system, not in the middle.) If you want to zoom a stencil about its center, you need to scale the matrix, and translate the scaled matrix to the left and up by an appropriate amount.

You can zoom the stencil about its center by translating the scaled matrix to the right by half the difference in width, and translating it up by half the difference in height. The code below zooms a stencil about its center:

```
-- create a matrix to use for transforming
matrix := mutableCopy identityMatrix

-- n is the scaling factor
-- to double the size of the stencil, set n to 2
-- to triple it, set to n to 3, and so on
n := 3

xoffset := (sten1.bbox.x1 + sten1.bbox.x2) * (n - 1)) / 2
yoffset := (sten1.bbox.y1 + sten1.bbox.y2) * (n - 1)) / 2

-- scale the matrix
scale matrix n n

-- translate it
translate matrix (- xoffset) (- yoffset)

-- sten1 is a Stencil object whose x1 and y1 are zero

-- transform the stencil
transform sten1 matrix @mutate
```

Rotating a Matrix

When rotating a matrix, specify the number of degrees or radians to rotate by, and also specify `@degrees` or `@radians` as appropriate. Rotation rotates a coordinate system around its origin (don't forget 0, 0 is in the top left corner), with a positive rotation angle representing clockwise rotation and a negative angle representing counterclockwise rotation.

Rotating About the Origin of a Stencil

Figure 11-15 and Figure 11-16 demonstrate that if you transform a stencil with a rotated matrix, the stencil is rotated about its origin, which is in the top left corner of the coordinate system.

In both figures, the small black dot shows the top left corner of the stencil. The black outline shows the bounding box of the `TwoDShape` that presents the stencil.

In Figure 11-15, the fluke bitmap is in its original state. In Figure 11-16, it is rotated by 45 degrees. You see that in Figure 11-16, the bitmap is rotated about the top left corner of the bitmap in its original state.

The following code creates the objects displayed in the figures:

```
global flukebm := importMedia theImportExportEngine \
    (getstream thescriptdir "fluke.pic" @readable) \
    @image @pict @bitmap

global fluke1 := new TwoDShape boundary:flukebm

global w := new window; show w
w.width := 350
w.height := 250
fluke1.x := 150
fluke1.y := 50
append w fluke1

-- add the black dot to show the origin
global p := new TwoDShape \
    boundary:(new oval x1:-5 x2:5 y1:-5 y2:5) \
    fill:blackbrush stroke:blackbrush

p.x := fluke1.x
p.y := fluke1.y

rotate1 := mutableCopy IdentityMatrix
rotate rotate1 45 @degrees

-- Figure 11-16 shows the results of changing
-- fluke1's boundary to the rotated bitmap
fluke1.boundary := transform flukebm rotate1 @create
```

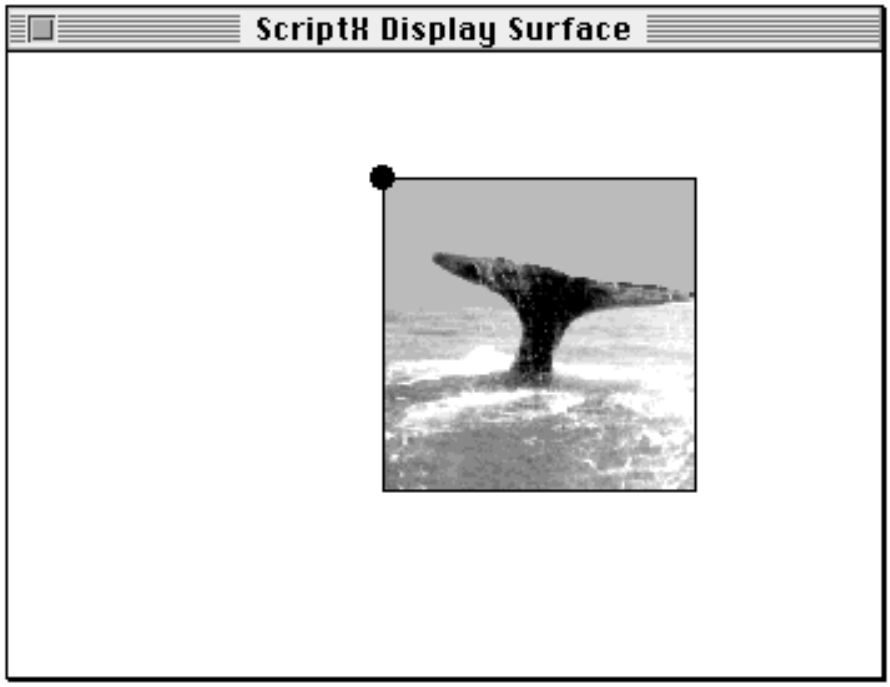


Figure 11-15:Fluke bitmap before 45 degree rotation

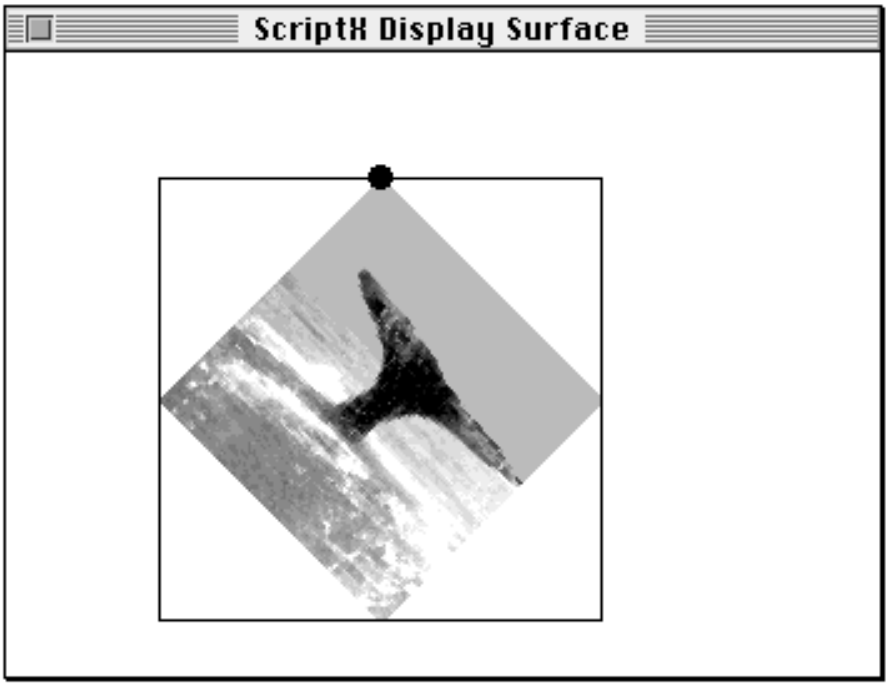


Figure 11-16:Fluke bitmap after 45 degree rotation

Rotating About the Center of a Stencil

If you want to rotate a stencil about its center, you need to translate the matrix used to do the transformation, in addition to rotating it. First translate it, then rotate it, then translate it again.

The following code shows how to rotate the fluke bitmap introduced in the previous example 45 degrees around its center. The result is shown in Figure 11-17.

```
-- Get the center of the boundary (Bitmap).
global centerPt := new Point
centerPt.x := (fluke1.bbox.x1 + fluke1.bbox.x2)/2
centerPt.y := (fluke1.bbox.y1 + fluke1.bbox.y2)/2

rotate2 := mutableCopy IdentityMatrix
translate rotate2 (-centerPt.x) (-centerPt.y)
rotate rotate2 45 @degrees
translate rotate2 centerPt.x centerPt.y
fluke1.boundary := transform flukebm rotate2 @create
```

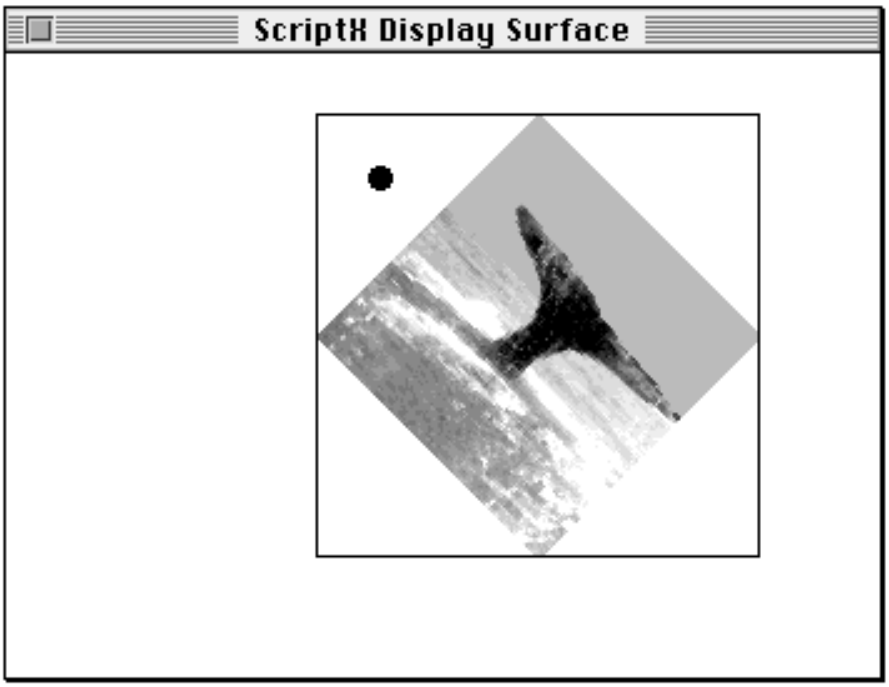


Figure 11-17:Fluke bitmap after 45 degree rotation about the center

Transformation Hints for Smoother Animation

Transforming matrices is a computation-intensive task so it can take a noticeable amount of time to perform. If you wish to achieve smooth animation by transforming a bitmap, you will often get better results by creating bitmaps for each stage in the transformation ahead of time and then swapping in the bitmaps during the animation.

Each rotation of a stencil, particularly for a bitmap, results in a small degradation of image quality. For one or two rotations, this is not usually significant, but if you rotate the same bitmap several times, the resulting loss in image quality can be significant. Therefore, it is often better to make a copy of the original bitmap at each transformation (that is, specify `@create` rather than `@mutate` when transforming the stencil).

Note however, that both these approaches create more objects and hence require more memory, so you need to weigh the pros and cons of each approach for each situation.

ColorMaps

`DisplaySurface`, `BitmapSurface`, and `Bitmap` objects use color maps.

A `DisplaySurface` is a surface that is used for drawing to the screen. Each window has a display surface. A `BitmapSurface` is an offscreen area you can render stencils to. You can build up an image on a `BitmapSurface` and then transfer it to a display surface.

Color Table Manipulation

The pixel values in a `Bitmap` object's `data` instance variable are most often used as indices into an array of color values, referred to as a color map. For example, a pixel of an 8-bit image might have a value of 100, which is an index to a particular color. Color maps are represented in the ScriptX imaging model by the `Colormap` class.

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.

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`.

Each `Bitmap`, `BitmapSurface`, and `DisplaySurface` object has a `Colormap` instance variable. This variable lets you set the `Colormap` instance associated with a particular rendering surface or bitmapped image. When you change the `Colormap` associated with a bitmap, you change the meaning of the pixels in the bitmap. The ScriptX imaging system can handle this change in one of two ways. It can try to remap the old pixel values to the corresponding colors in the new color map, or ignore the change and simply use the pixel values as indices into the new map. In the first case, the results are better, but the process takes time and can produce inconsistencies in specific colors, as when a color in the original map isn't in the new map. In the second case, unless the two maps

contain identical colors in identical positions, the results are almost always likely to be incorrect—unless, of course, this technique is being used intentionally to create an interesting visual effect.

Whenever a window is created, the value of the `Colormap` instance variable on the window's `DisplaySurface` is set by default to represent the palette installed in hardware when the display surface has focus. The definition of focus depends on the underlying windowing system. However, the display surface with focus is generally that for the window that received the last mouse-down event. Like other subclasses of `Bitmap`, `DisplaySurface` can change the content of its `Colormap` instance variable. However, note that it may not be possible to exactly match the values in a custom `Colormap` to the underlying hardware palette through this variable, because some systems reserve certain positions in their color maps for specific colors.

The remapOnDraw Flag

The `BitmapSurface` and `Bitmap` classes have an instance variable `remapOnDraw` that indicates how to handle differences between an instance's color map and the color map of a destination `DisplaySurface` instance. The value is either `true` or `false`. This variable is checked when the `transfer` operation transfers bits from one bitmap surface to another surface (such as a `BitmapSurface` or `DisplaySurface`). It is also checked when a `fill` operation transfers bits from a bitmap to a surface.

When this value is `true` (the default value), the imaging system attempts to find matches in the color map of the destination surface for the colors originally represented in the source bitmap. When the value is `false`, the rendering operation simply uses the pixel values from the bitmap as indices into the color map for the surface, regardless of the validity of the results.

The setting of the `remapOnDraw` flag would have an effect in a case where you imported a `Bitmap` instance that had a `Colormap` value different than that of the display surface to which it is rendered.

For example, suppose you import a bitmap called `MyBitmap` that has its own colormap, and display it in a window as shown by the code below:

```
w := new window
show w
mySurface := w.displaySurface

myBitmap := -- use the importer to import a bitmap
myPic := new TwoDShape boundary:myBitmap
append w myPic
```

When the bitmap is rendered to the display surface of the window, the rendering operation looks at pixel values in `myBitmap`, looks up the corresponding colors in `myBitmap.colormap`, then attempts to find color matches in `mySurface.colormap`. If it can do so, it renders each pixel using this remapping. If it can't do this directly, it finds a close match for a pixel's original color. This remapping applies only for the rendering operation—the pixel values in the source bitmap remain the same, and it retains its original color map.

The remapOnSet Flag

Another instance variable of `Bitmap` and `BitmapSurface` is `remapOnSet`. Its value, which is also a `Boolean`, is `false` by default. This variable should be set to `true` when you want to permanently change the pixels in the bitmap to match their color mapping in the new color map. The same mechanism applies as discussed for `remapOnDraw`: the rendering operation looks at pixel values in the bitmap, looks up corresponding colors in its color map, then tries to find color matches in the new color map. If it can do so, it changes the value in each pixel using this remapping. If it can't do this directly, it finds a close match for the pixel's original color. This remapping is permanently applied to the pixels in the bitmap.

If this instance variable is `false`, the bitmap's pixel data will be left unchanged, and used as indices into the new color map, regardless of the validity of the results.

Hardware Palette Changes

The `PaletteChangedEvent` class defines the mechanism to notify all `DisplaySurface` instances when the hardware palette changes. Events of the `PaletteChangedEvent` class include a `colormap` instance variable representing the new palette. A change in bit-depth is equivalent to a palette change.

Instances of `TwoDCompositor` express interests in `PaletteChangedEvent` for their display surfaces. When the palette changes for a screen, the palette changed event is broadcast to every `TwoDCompositor` for that screen. A `TwoDCompositor` instance uses an instance of `BitmapSurface` to perform cached drawing, and changes the `Colormap` instance variable for this cache whenever it receives a `PaletteChangedEvent`.

The Default Colormaps

The global `Colormap` instances `theDefault1Colormap`, `theDefault2Colormap`, `theDefault4Colormap`, and `theDefault8Colormap` are used in instances of `Bitmap`, `BitmapSurface`, and `DisplaySurface` created without a value specified for their `Colormap` instance variable. The specific global used depends on the color space supported by the underlying display hardware. The bit depth of this color map is equivalent to that of the palettized screen with the highest bits per pixel. If no such palettized color screen exists on a particular system, this value is the bit-depth of the direct color screen with the maximum depth.

Creating New Classes of Graphic Presenters

If you use the predefined subclasses of `Stencil` and `TwoDPresenter` that come with ScriptX, you do not need to worry about how the stencils are rendered to the surface of a window. The subclasses of `TwoDPresenter`, such as `TwoDShape`, take care of rendering for you. To draw an image in a window, you create a `TwoDShape`, tell it what stencil to use as its `boundary`, and use the

append or prepend method to put it in a visible space such as a window (or any 2D space that is in a presentation hierarchy in a window.) To change the characteristics of the image, you set instance variable values on the presenter, such as fill, stroke, x, y, width, and height. If you want to change the image completely, change the boundary of the TwoDShape.

Whenever you make a change to a presenter in a visible space, the compositor takes care of redrawing the presenter. If you close the window and open it again, or cover it up by another window and then uncover it, the compositor redraws the window.

Behind the scenes, the compositor calls the draw method on a presenter to make it redraw itself when necessary. Usually the draw method calls the methods fill, stroke, and transfer on the window's display surface to render the stencil to the surface.

If you confine yourself to using the predefined subclasses of TwoDPresenter, you do not need to learn about how to use the fill, stroke, and transfer methods. However, if you want to define new subclasses of TwoDPresenter that have their own special ways of drawing shapes, you need to understand how to use these methods. For example, to define a class of shape that always draws itself with a shadow, you would need to create a subclass of TwoDPresenter with a specialized draw method that draws the shadow and the stencil.

The details of how the compositor works are discussed in Chapter 3, "Spaces and Presenters." What you need to understand here, is that when you define your own class of TwoDPresenter, you need you define the draw method to draw itself as desired. The compositor takes care of calling the draw method whenever necessary. You never need to call draw, but you do not need to define it.

Boundaries and Global Boundaries

Another thing you need to know when creating your own subclass of TwoDPresenter is that the global boundary of a presenter determines the presenter's clipping region. The image drawn by the presenter is clipped by the presenter's clipping region. If you make sure that the boundary instance variable is initialized appropriately, the globalBoundary instance variable will be taken care of automatically.

The value of the boundary instance variable of a presenter is the boundary within the presenter's local coordinate system. The value of the globalBoundary instance variable is the boundary of the presenter in the window's coordinate system. For example, if a presenter's boundary is:

```
[0, 0, 200, 200] as Rect
```

and the presenter's position is:

```
[50, 50] as Point
```

then the presenter's global boundary is:

```
[50, 50, 250, 250] as Rect.
```

If the value of the `boundary` instance variable changes, the `globalBoundary` changes automatically when it is needed. Therefore you need to define your `TwoDPresenter` subclass to take care of computing its `boundary` as needed, and you don't usually need to worry about computing the `globalBoundary`.

The global boundary is updated when it needs to be used, and not before then. For example, if you create a `TwoDShape` and change its boundary before you put it in a window, the global boundary is not computed because it is not needed yet. It will be computed when you put the `TwoDShape` in a window.

If you want to force the recalculaton of the global boundary, you can call the presenter's `recalcRegion` method. For example:

```
t := new TwoDShape boundary:(new rect x2:100 y2:100)
⇨ [0, 0, 100, 100] as Rect

t.boundary
⇨ [0, 0, 100, 100] as Rect

t.globalboundary
⇨[0, 0, 0, 0] as Rect

recalcregion t
⇨ OK

t.globalboundary
⇨ [0, 0, 150, 100] as Rect
```

Arguments For the Draw Method

The `draw` method of a presenter is always called with three arguments, which are supplied automatically. These are:

self – the presenter object.

surface – the surface to draw to.

clip – the clipping region to use, which defaults to the presenter's global boundary.

The boundary may be further clipped as necessary depending on where the presenter is located and what it is presented by. For example, a `TwoDShape` in a `TwoDSpace` will be clipped if any part of the `TwoDShape` extends beyond the edge of the `TwoDSpace`.

Notice that the arguments do not include the stencil to be drawn, the brush to fill it with, or the brush to stroke it with. These things should all be computed within the `draw` method.

Fill, Stroke, and Transfer Methods

Every window has a `displaySurface` instance variable that holds the `DisplaySurface` object for the window, which is the area on screen that the window uses. The `draw` method can use the methods `fill`, `stroke` and `transfer` to render shapes on the window's display surface.

Using the Fill and Stroke Methods

The `fill` method of a surface renders a stencil to the surface by filling it in using a given brush. The `stroke` method renders the outline of a stencil to a surface using a given brush. Both the `fill` and `stroke` methods take the following arguments:

surface – the surface to draw to.

stencil – the stencil to render.

clip – the stencil to use as the clipping region.

matrix – the 2D matrix to use to position the stencil.

brush – the brush to use for the rendering operation.

The following bullets discuss how the `draw` method can arrive at each of these arguments.

- The *surface* argument that is passed to the `draw` method can be passed directly on to `fill` and `stroke`.

  (When the compositor's `useOffscreen` flag is true, *surface* is the compositor's offscreen surface. If `useOffscreen` is false, *surface* is the window's display surface.)

- The *clip* argument that is passed to the `draw` method can be passed directly on to `fill` and `stroke`, unless you want to change the clipping region.

  The *clip* defaults to the global boundary of the presenter, clipped as appropriate depending on the location of the presenter and the kind of presenter that is presenting it.

- The `draw` method must generate the value for *stencil*.

  For example, the class `TwoDShape` uses the presenter's `boundary` as the *stencil* as well as for its *clip*. A better approach for your own subclass of `TwoDPresenter` is to use the inherited `target` instance variable to hold the target stencil. This enables the presenter to use a different stencil for its clipping region than for its target.

- The `draw` method must generate the value for *matrix*.

  All `TwoDPresenter` objects have a `globalTransform` instance variable, whose value is a `TwoDMatrix` object that can be used to position the presenter's target when it is drawn. You can often use the value in the presenter's `globalTransform` matrix as the matrix to pass to `fill` and `stroke`.

- The `draw` method must generate the value for *brush*, which must be a `Brush` object.

Using the Transfer Method

Currently ScriptX defines two classes of surfaces: `DisplaySurface` and `BitmapSurface`. A `DisplaySurface` represents a drawing area on the screen. A `BitmapSurface` represents an area in memory that can be drawn to. You can use bitmap surfaces to assemble images offscreen, and then transfer them to other bitmap surfaces or to the display surface of a window.

The `transfer` method transfers the contents of one surface to another surface. Currently, you can transfer the contents of a bitmap surface to another surface, but you cannot transfer the contents of a display surface to another display surface. The `transfer` method takes the following arguments:

surface – the surface to draw to.

surfaceToCopy – the surface whose contents are to be copied.

clip – the stencil to use as the clipping region.

matrix – the 2D matrix to use to position the source image.

When calling the `transfer` method on a surface from inside a `draw` method, you can generate the values for *surface*, *clip*, and *matrix* in the same way as described for the `fill` and `stroke` methods. The `draw` method must generate the value for the *surfaceToCopy* argument.

2D Graphics Examples

Example – ShadowedShape

The following example shows the definition for a new subclass of `TwoDPresenter`, called `ShadowedShape`, which has its own definition for `draw`. Whenever a shadowed shape appears on the screen, you see both the shape and its shadow.

Notice that `ShadowedShape` is a subclass off of `TwoDPresenter`, not `TwoDShape`. `TwoDShape` uses its boundary as both its target and its clipping stencil, and `ShadowedShape` needs to have its target be different from its clipping stencil.

ShadowedShape Class (short version)

The class `ShadowedShape` uses its inherited `target` instance variable to hold the stencil defining the main shape (that is, the shape without its shadow.)

The `fill` and `stroke` instance variables hold the brushes for filling in the shape and for painting its outline. The `shadowFill` and `shadowStroke` instance variables hold the brushes for filling in the shadow and painting the shadow's outline. The `shadowXoffset` and `shadowYoffset` instance variables hold the x and y offsets for the shadow.

For simplicity, this example creates `ShadowedShape` as a direct subclass of `TwoDShape`, and uses actual instance variables. However, it would be better to use virtual instance variables for `fill`, `stroke`, `shadowFill`, and so on. Then you could override setter methods for each, so that if the user changes the values of the instance variables, the image updates automatically on the screen. For example, if the user change the fill, the image immediately redraws with the new fill.

However, you cannot override setter methods for instance variables defined on a class, but you can override setter methods for inherited instance variables. Therefore you would need to create an intermediate superclass that defines the instance variables, and create `ShadowedShape` as a subclass of that superclass. See "Improved ShadowedShape Class" on page 268 for a complete class definition of the "improved" `ShadowedShape` class.

The code below shows the simplified version of `ShadowedShape`, that does not update automatically when you make changes to its instance variables.

```
class ShadowedShape (TwoDPresenter)
instance variables
    fill:whitebrush
    stroke:blackbrush
    shadowFill:(new brush color:bluecolor)
    shadowStroke:blackbrush
    shadowXoffset:5
    shadowYoffset:-5
end
```

ShadowedShape Init Method

Notice that the `init` method (which gets called automatically immediately after the object is created) sets the value of the `boundary` instance variable to be a `Rect` object that is big enough to contain both the shape and its shadow. This is important because the boundary determines global boundary, which determines the clipping region for the shape.

```
method init self {class ShadowedShape} #rest args #key \
         target:(new Rect x2:50 y2:50) ->
(
    local boundary := new Rect \
       x2:(target.width + abs (self.shadowXoffset)) \
       y2:(target.height + abs(self.shadowYoffset))
     apply nextMethod self boundary:boundary target:target args
)
```

ShadowedShape Draw Method

The `draw` method draws a shape and its shadow (although it actually draws the shadow first, then paints the shape over the top of the shadow.) The shadow uses the same stencil as the shape, but is offset to the left or right and up or down to create a shadow effect.

The `draw` method for `ShadowedShape` does several things. It figures out which brushes to use for filling and stroking the shadow, and which to use for filling and stroking the shape. It paints the outline of the shadow on to the surface, fills in the shadow on the surface, then paints the outline of the shape to the surface, and fills in the shape on the surface. Notice that it uses the shape's `globalTransform` matrix to position the shadow and the shape. Notice also that it translates the matrix (moves it) to the right before drawing the shadow, and then moves it back to its original x position and down a bit before painting the shape. Finally, the `draw` method restores the `globalTransform` to its original state so that it is ready for next time.

The `draw` method defined here only works accurately for shadows that fall to the right and top of the shape, that is, the x offset is positive and y offset is minus. Shadows that fall in other directions are clipped since they fall outside the boundary. (The improved class definition handles shadows in all directions.)

```
method draw self {class ShadowedShape} surface clip ->
(
    -- get the offsets for the shadow
    local xval := self.shadowXoffset
    local yval := self.shadowYoffset

    -- get the transform matrix
    local matrix := self.transform

    -- translate the matrix to the right
    -- to position the shadow to the right
    translate matrix xval 0

    -- render the shadow to the surface
    fill surface self.target clip matrix self.shadowFill
    -- draw the outline of the shadow
    stroke surface self.target clip matrix self.shadowStroke

    -- translate the matrix to position the shape
    translate matrix (- xval) (- yval)

    -- render the shape to the surface
    fill surface self.target clip matrix self.fill
    stroke surface self.target clip matrix self.stroke

    -- restore the matrix back to its original state
    translate matrix 0 yval
)
```

Test the ShadowedShape Class

To use the `ShadowedShape` class, create an instance of it, specifying the target stencil to use as the main shape. Put the shape in a visible space such as a window. If you make changes to it, such as changing its fill, you must call `notifyChanged` on it to update the image on the screen.

```
global w := new window
w.height := 300
w.width := 400
show w

global shape1 := new ShadowedShape \
    target:(new oval x2:100 y2:100)

shape1.fill := new Brush color:magentaColor

shape1.x := 20
shape1.y := 20
prepend w shape1
```

Improved ShadowedShape Class

The improved version of the `BetterShadowedShape` class creates an intermediatry subclass of `TwoDShape`, called `BetterShadowedShapeSuper`. The sole purpose of this class is to define instance variables such as `fill`, `shadowedFill`, and so on, so that a subclass can override the setter methods for the instance variables.

`BetterShadowedShapeSuper` has one subclass, `BetterShadowedShape`, that defines setter and methods for instance variables that can be changed by users, such as `fill`, `shadowFill`, and so on. The setter methods call `notifyChanged` to tell the compositor that the shape has changed and needs to be redrawn.

The setter and getter methods for the `shadowXOffset` and `shadowYoffset` instance variables also recompute the boundary.

The `draw` method also handles drawing the shadow in any direction. If the shadow falls to the right, the `draw` method moves the positioning matrix to the right before drawing the shadow, and move it back to the left before drawing the shape.

If the shadow falls to the left, the `draw` method moves the matrix to the right before drawing the shape, and moves it back to the left afterward.

If the shadow falls to the top, `draw` moves the matrix down before drawing the shape, and moves it back to the center afterward.

If the shadow falls to the bottom, `draw` moves the matrix down before drawing the shadow, and move it back up before drawing the shape.

-- class definitions

```
class BetterShadowedShapeSuper (TwoDPresenter)
    instance variables
```

```
    fill
    stroke:blackbrush
    shadowFill:(new brush color:bluecolor)
    shadowStroke:blackbrush
    shadowXoffset:5
    shadowYoffset:-5
end


class BetterShadowedShape (BetterShadowedShapeSuper)
end
```

-- init method

```
method init self {class BetterShadowedShape} #rest args #key \
         target:(new Rect x2:50 y2:50) ->
(
    local boundary := new Rect \
       x2:(target.width + abs (self.shadowXoffset)) \
       y2:(target.height + abs(self.shadowYoffset))
     apply NextMethod self boundary:boundary target:target args
)
```

-- draw method

```
method draw self {class BetterShadowedShape} surface clip ->
(
    local xval := self.shadowXoffset
    local yval := self.shadowYoffset

    if (self.fill != undefined)
    do
    (   local matrix := self.transform

        --  fill the shadow and stroke its outline
        local xvala, yvala
        xvala := if xval < 0 then 0 else  xval
        yvala := if yval > 0 then yval else 0
        translate matrix xvala yvala
        fill surface self.target clip matrix self.shadowFill
        stroke surface self.target clip matrix self.shadowStroke

        -- now draw the target
        translate matrix (- xval) (- yval)
        fill surface self.target clip matrix self.fill
        stroke surface self.target clip matrix self.stroke

        -- restore the global transform
        xvala := if xval >= 0 then 0 else xvala := xval
        yvala := if yval <= 0 then yval else 0
        translate matrix xvala yvala
    )
)
```

-- updateBoundary

```
method updateBoundary self {class BetterShadowedShape} ->
```

```
(   local target := self.target
    self.boundary := new Rect \
        x2:(target.width + abs (self.shadowXoffset)) \
        y2:(target.height + abs(self.shadowYoffset))
  )
```

-- Setter and Getter Methods

```
method heightSetter self {class BetterShadowedShape} value ->
(   self.target.height := value
    updateBoundary self
    notifyChanged self true
)

method widthSetter self {class BetterShadowedShape} value ->
(   self.target.width := value
    updateBoundary self
    notifyChanged self true
)

method heightGetter self {class BetterShadowedShape} ->
(   self.target.height
)

method widthGetter self {class BetterShadowedShape} ->
(   self.target.width
)

method fillSetter self {class BetterShadowedShape} value ->
(   nextMethod self value
    notifychanged self true
    value
)


method strokeSetter self {class BetterShadowedShape} value ->
(   nextMethod self value
    notifychanged self true
    value
)


method shadowfillSetter self {class BetterShadowedShape} value ->
(   nextMethod self value
    notifychanged self true
    true
)


method shadowStrokeSetter self {class BetterShadowedShape} value ->
(   nextMethod self value
    notifychanged self true
    value
)


method shadowXoffsetSetter self {class BetterShadowedShape} value ->
(   nextMethod self value
```

```
    updateBoundary self
    notifychanged self true
    value
)

method shadowYoffsetSetter self {class BetterShadowedShape} value ->
(   nextMethod self value
    updateBoundary self
    notifychanged self true
    value
)
```

Example – Grid

This example describes the `Grid` class, which is defined as a subclass of `TwoDPresenter`. The `draw` method of the `Grid` class renders horizontal and vertical lines to make a grid.

Define The Grid Class

The `Grid` class has instance variables that users can change to specify characteristics of the grid.These include instance variables that specify the number of vertical lines (`numOfVertLines`); the number of horizontal lines (`NofHorizLines`); the brush to use for the vertical lines (`vertBrush`); the brush to use for the horizontal lines (`horizBrush`); the brush to use to fill the background (`fill`); and the brush to use to paint the outline (`stroke`).

In addition, `Grid` has four instance variables that are needed by the `draw` method, and users should not change these. They are `matrixv` and `matrixh`, which are the matrices that position the vertical and horizontal lines respectively; and `linev` and `lineh`, which are the lines to use for rendering vertical and horizontal lines respectively.

The `draw` method could create these matrices and lines on the fly, but that would result in excessive and unnecessary object creation, which would keep the garbage collector busy cleaning up the extra objects whenever the grid was drawn.

```
class grid (TwoDPresenter)
instance variables

    NofHorizLines:20
    numOfVertLines:20

    linev :(new line x1:0 y1:0 x2:0 y2:100)
    lineh: (new line x1:0 y1:0 x2:100 y2:0)

    matrixv:(mutablecopy identityMatrix)
    matrixh:(mutablecopy identityMatrix)
    horizBrush:(new Brush \
        color:(new RGBColor \
            red:150 blue:150 green:150))
```

```
    vertBrush:(new Brush \
        color:(new RGBColor \
            red:150 blue:150 green:150))

    stroke:(new brush color:blackColor)
    fill:(new brush color:whitecolor)
end
```

Define the Draw Method

The `draw` method draws the grid by rendering horizontal and vertical lines. It draws the vertical lines by drawing a single vertical line, then moving the line and drawing it again, and so on. The horizontal lines are drawn in a similar fashion.

```
method draw self {class Grid} surface clip ->
(
    -- get the matrix that determines the grid's position
    local globalTrans := self.globalTransform

    -- fill in the area
    local fillBrush := self.fill
    if isdefined fillBrush do
    fill surface self.bbox clip globalTrans fillBrush

    -- get the brushes and the lines to use
    local horizBrush := self.horizBrush
    local vertBrush := self.vertBrush

    local linev := self.linev
    linev.y2 := self.height
    local lineh := self.lineh
    lineh.x2 := self.width

    -- horizTrans is the distance between horizontal lines
    horizTrans := self.height / self.NofHorizLines

    --vertTrans is the distance between vertical lines
    vertTrans := self.width / self.numOfVertLines

    -- initialize the matrices that position the lines
    local matrixv := self.matrixv
    setTo matrixv globalTrans

    local matrixh := self.matrixh
    setTo matrixh globalTrans

    -- draw the vertical lines
    -- move the positioning matrix to the right
    -- after drawing each line
    for i in 1 to (self.numOfVertLines - 1)
    do
    (   translate matrixv vertTrans 0
        fill surface linev clip matrixv vertBrush
    )
```

```
    -- draw the horizontal lines
    -- move the positioning matrix to down
    -- after drawing each line
    for i in 1 to (self.NofHorizLines - 1) do
    (   translate matrixh 0 horizTrans
        fill surface lineh clip matrixh horizBrush
    )

    -- draw the outline of the grid
    local strokeBrush := self.stroke
    if isdefined strokeBrush do
        stroke surface self.bbox clip globalTrans strokeBrush
)
```

Create and Display a Grid

To display a grid, simply create an instance of `Grid` and put it in a visible space such as a window.

```
w := new Window
w.width := 500
w.height := 400

show w

global grid1 := new Grid boundary:(new Rect x2:200 y2:100)

append w grid1

grid1.x := 50
grid1.y := 50

-- change the width and height
grid1.width := 200
grid1.height := 200

-- change the number of vertical lines
grid1.numOfVertLines := 10
grid1.NofHorizLines := 10
notifyChanged grid1 true
```

Improving the Grid Class

To improve the `Grid` class, you could define setter and getter methods for all the instance variables that can be changed by users (such as `fill` and `numOfVertLines` and so on), so that the grid updates immediately when they are changed. See the “2D Graphics Examples” on page 265 for examples of defining such setter and getter methods.

Since each `Grid` object uses its default boundary (which you specify when creating an instance of `Grid`) the boundary is recalculated automatically and correctly whenever the width and height of a grid change. Thus you would not need to define setters methods for the `width` and `height` instance variables.

Example – Stencilizer

The `Stencilizer` class that is discussed in this section acts like a paint program. If you put a `Stencilizer` object in a window, you can paint in the area of the stencilizer by pressing and dragging the mouse.

When you press the mouse down in the area of the stencilizer, or you move the mouse around in the stencilizer, the stencilizer draws a stencil to the stencilizer's offscreen drawing cached, which is a `BitmapSurface`, and then immediately notifies the compositor that the stencilizer has changed. Behind the scenes, the compositor calls the stencilizer's `draw` method, which transfers the contents of the offscreen bitmap surface back to the window's display surface.

This level of indirection is needed so that the stencilizer will draw itself properly whenever needed. If the `mouseEvent` objects drew directly to the window instead of to the stencilizer's drawing cache, the results of the drawing would be lost if the window was closed or covered, or if the stencilizer was moved about in or removed from the window.

Stencilizer Class Definition

A `Stencilizer` instance paints by drawing stencils to a `BitmapSurface` called the drawing cache. The `draw` method of `Stencilizer` first transfers the contents of the drawing cache to the screen.

In addition to demonstrating 2D graphics and 2D presenters, the `Stencilizer` class demonstrates interactive use of mouse events. A `Stencilizer` instance creates event interests when it is initialized to catch mouse events in its area. It defines instance methods to receive mouse events—these methods create stencils along the brush stroke as the user drags with the mouse.

The `Stencilizer` class defines several instance variables representing 2D graphics objects. The `drawingCache` instance variable is a `BitmapSurface` instance used to cache previous drawings. The `strokeClass` instance variable provides a way to set the shape of the brush stroke; it can be one of the `Stencil` subclasses `Oval`, `Rect`, or `Line`. The `currentBrush` is the brush to use for this stroke.

The `Stencilizer` class defines the `mouseDown`, `mouseMoved`, and `MouseUp` instance variables that hold the `MouseEvent` objects that are needed to detect mouse actions. In the stencilizer's initial state, the `MouseDown` event is waiting for the mouse to be pressed down. When the mouse is pressed down, the `MouseMoved` and `MouseUp` events are brought into action, to detect when the mouse is moved or when it is released. When you let the mouse up, the `MouseMoved` and `MouseUp` events are temporarily relieved from duty, since they will not be needed until the mouse is pressed down again.

The `mouseDown`, `mouseMoved`, and `MouseUp` event objects call the `fill` method to render a stencil (determined by the `strokeClass`) to the stencilizer's drawing cache.

(For more on event handling, see Chapter 18, "Events and Input Devices in this manual).

```
class Stencilizer (TwoDPresenter)
    instance variables
        drawingCache -- bitmap for caching previous strokes
        strokeClass -- class of Stencil used in the brushStroke
        currentBrush -- Brush instance now used to fill thebrushStroke
        strokeWidth -- Half the width of the brush stroke
        mouseDown -- event interests managed by theStencilizer
        mouseMoved
        mouseUp
end
```

RenderStroke Method

The `Stencilizer` class's `renderStroke` method creates a stencil at the current mouse point. The brush and kind of stencil are determined by values in the `currentBrush` and `strokeClass` instance variables.

After rendering the stencil to the offscreen drawing cache, the `renderStroke` method calls `notifyChanged` on the stencilizer, to tell the compositor that the stencilizer has changed. This causes the compositor to call the stencilizer's `draw` method behind the scenes.

-- renderStroke

```
method renderStroke self {class Stencilizer} theInterest theEvent ->
(
    -- get the current mouse point
    local thisPoint := theEvent.localCoords
    local pointx := thisPoint.x
    local pointy := thisPoint.y
    local strokeWidth := self.strokeWidth

    -- create a stencil
    local thisStencil := (new self.strokeClass \
           x1:(pointx - strokeWidth)\
           y1:(pointy - strokeWidth) \
           x2:(pointx + strokeWidth) \
           y2:(pointy + strokeWidth))

    -- render the stencil to the bitmap surface
    fill self.drawingcache \
        thisStencil \
        self.boundary \
        identityMatrix \
        self.currentBrush

    -- call notifyChanged to tell the compositor
    -- that the stencilizer has changed
    notifyChanged self true

    -- accept the event
    @accept
)
```

BeginStroke and EndStroke Methods

The `Stencilizer` class defines methods that are invoked by mouse events. The `beginStroke` method is invoked when the mouse is pressed down in the area of the stencilizer. The `renderStroke` method is invoked after the mouse has been pressed down and is moved. The `endStroke` method is invoked when the mouse is released.

The `mouseDown` event invokes the `beginStroke` method, which puts the `mouseMove` and `mouseUp` events on duty, and then calls `renderStroke` to render the stroke. The `mouseUp` event invokes the `endStroke` method, which takes the `mouseMove` and `mouseUp` events off duty, and then calls `renderStroke`. The `mouseMove` event simply renders the stroke; it does not need to manage events, so it calls `renderStroke` directly.

-- beginStroke

```
-- this method handles mouse down events
method beginStroke self {class Stencilizer} theInterest theEvent ->
(
    -- put the mouseMoved and mouseUp events on duty
    addEventInterest self.mouseMoved
    addEventInterest self.mouseUp

    -- render the stroke
    renderStroke self theInterest theEvent
)
```

-- endStroke

```
-- this method handles mouse up events
method endStroke self {class Stencilizer} theInterest theEvent ->
(
    -- remove the mouseMoved and mouseUp events from duty
    removeEventInterest self.mouseMoved
    removeEventInterest self.mouseUp

    -- render the stroke
    renderStroke self theInterest theEvent
)
```

Initializing a Stencilizer Instance

Initializing a `Stencilizer` instance consists largely of setting default values for instance variables defining the brush stroke, setting up the drawing cache, and defining mouse-event interests for interactive painting. Each event interest identifies the presenter (`self`) interested in the event, a method for receiving the event, and the author data to pass to the function (again, `self`). The `init` method for `Stencilizer` is as follows:

```
-- this method initializes the Stencilizer
```

```
method init self {class Stencilizer} #rest args ->
(
    apply nextMethod self args

    -- define the brush characteristics
    self.currentBrush :=  new Brush color:redColor
    self.strokeWidth := 4
    self.strokeClass := Oval
    -- make the drawing cache be fairly big
    self.drawingCache := new BitmapSurface \
           bbox:(new rect x2:600 y2:400)

    local theMouseDevice := new MouseDevice

    -- set up the mouse event interests
    self.mouseDown := new MouseDownEvent
    self.mouseDown.eventReceiver := beginStroke
    self.mouseDown.authorData := self
    self.mouseDown.device := theMouseDevice
    self.mouseDown.presenter := self

    self.mouseMoved := new MouseMoveEvent
    self.mouseMoved.eventReceiver := renderStroke
    self.mouseMoved.authorData := self
    self.mouseMoved.device := theMouseDevice
    self.mouseMoved.presenter := self

    self.mouseUp := new MouseUpEvent
    self.mouseUp.eventReceiver := endStroke
    self.mouseUp.authorData := self
    self.mouseUp.device := theMouseDevice
    self.mouseUp.presenter := self

    -- make sure stencilizer gets mouse up, even outside its boundary
    self.mouseUp.matchedInterest := self.mouseDown

    -- at the beginning, put the mouseDown event on duty
    addEventInterest self.mouseDown
)
```

The Stencilizer Draw Method

The `Stencilizer` class's `draw method` simply transfers the contents of the drawing cache to the specified surface, and then paints the outline of the stencilizer.

The `renderStroke` method discussed previously uses the `identityMatrix` to position the stencil on the drawing cache. The `draw` method, however, uses the stencilizer's `globalTransform` value as the matrix to position the transferred image. This ensures that if the stencilizer has been moved to a location other than (0,0) in the window, it will be drawn correctly at the new position.

The `draw` method transfers the bitmap surface used as the drawing cache to a destination surface, rather than directly filling and stroking stencils to the surface. Thus the `draw` method needs to use the stencilizer's

globalBoundary value as the clipping region, rather than using the clip value that is passed in to it. Since the draw method does not use the clipping region that is passed to it, it will not be clipped properly if it is put in a TwoDMultipresenter, such as a TwoDSpace.

Thus you should not embed a Stencilizer object inside another presenter that does clipping. You can however put a Stencilizer object inside another presenter that does not do clipping, such as a GroupSpace.

```
-- draws the contents of the Stencilizer's drawing cache to a surface
-- this method is called automatically by the compositor
method draw self {class Stencilizer} theSurface clip ->
(
    -- transfers the drawing cache to the surface
    transfer theSurface \
        self.drawingCache \
        self.globalBoundary \
        self.globalTransform

    -- draw the outline of the stencilizer's area
    stroke theSurface \
        self.boundary \
        self.globalBoundary \
        self.globalTransform \
        blackbrush)
```

Testing the Stencilizer

The following script creates and displays an instance of Stencilizer in a window.

```
-- set up a simple example of the stencilizer class
global myWindow := new Window \
    boundary:(new Rect x2:400 y2:300) \
    name:"ScriptX Stencilizer"

global myStencilizer := new Stencilizer \
    boundary:(new Rect x2:300 y2:250)
append myWindow myStencilizer
myWindow.x := 50
myWindow.y := 50
show myWindow
```

When the stencilizer window appears, you can set instance variables of myStencilizer to change the results of painting. To change the color of the brush stroke, set myStencilizer.currentBrush to blackBrush or some other Brush instance. To change the shape of the brush, set myStencilizer.strokeClass to either the Rect or Line class (it's Oval by default). To change the width of the brush, set myStencilizer.strokeWidth to an integer value such as 1, 5, or 10. (Note that the strokeWidth value is actually half the stroke width.)

You can change the position of the stencilizer, and you can change its width and height. The stencilizer acts as a viewing frame for an underlying canvas (the drawing cache). If you increase the size of the stencilizer, you will see more of the canvas. If you decrease the size of the stencilizer, you will see less of the canvas.

Resizing the stencilizer does not resize the drawing cache, since it is a separate `BitmapSurface` object. Also, if you make the stencilizer bigger than its drawing cache, you will not be able to paint in the region of the stencilizer that extends beyond the drawing cache.

You can clean the stencilizer's slate by giving it a new `BitmapSurface` to use as its drawing cache.

If you remove the stencilizer from its current window, and put it in another one, it will draw itself correctly.

CHAPTER

12

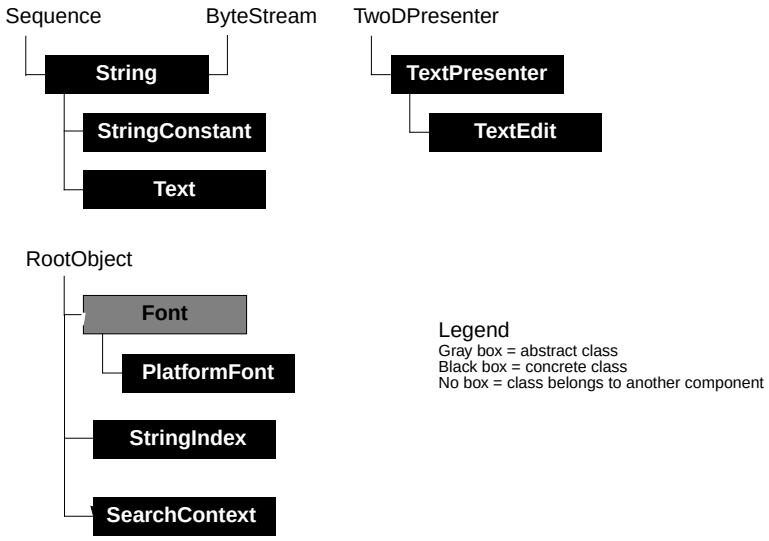
Text and Fonts

This chapter describes the Text and Fonts components. Text provides for the display, editing, and formatting of text, including paragraph formatting. The Fonts component provides access to the font glyphs and metrics.

To present text, ScriptX defines two classes: `TextPresenter` and `TextEdit`. Both inherit from the class `TwoDPresenter`, part of the ScriptX spaces and presenters hierarchy.

Classes and Inheritance

The class inheritance hierarchy for the Text and Fonts components is shown in the following figure.



The following classes form the Text and Fonts components. In this list, indentation indicates inheritance.

`String` – a sequence of characters stored with variable-length encoding.

`StringConstant` – an immutable string.

`Text` – a searchable string of characters and a set of style attributes.

`TextPresenter` – a presenter for the display of `String`, `StringConstant`, and `Text` objects, which inherits from the class `TwoDPresenter.`

`TextEdit` – a subclass of `TextPresenter` that provides selection and editing services through the mouse and keyboard.

`Font` – an abstract class providing access to a typeface.

`PlatformFont` – a concrete subclass of `Font` providing access to the font technology of the underlying system.

`StringIndex` – a signature index providing fast searching of text

`SearchContext` – an object with information about where to search in a `StringIndex` object

Conceptual Overview

The Text component provides facilities for the creation, manipulation, and presentation of text. These include string encoding, text formatting, text editing, and string searching capabilities. Text can be used to create specialized multimedia objects such as annotations and hypertext links. There are no restrictions on the length of a string in ScriptX.

The Fonts component provides a mechanism that allows ScriptX classes, such as those in the Text component, to locate and process font data provided by the underlying system.

The core classes of strings include three concrete classes—`String`, `StringConstant`, and `Text`—that represent a series of characters. `String` provides the basic behavior for character storage and encoding, `StringConstant` is a class of strings that cannot be modified, and `Text` is a string class with additional methods for managing character attributes and selections.

Theses three string classes are not presenters. In order to be presented on screen, they must be the target of a `TextPresenter` or `TextEdit` object. An ASCII representation of a string can be printed to a debugging console or stream. See Chapter 3, “Spaces and Presenters” for more information on how presentation takes place.

Strings are both collections and streams. Although individual characters are not themselves objects, a string is a collection in which each character is an element. In the methods that `String` inherits as a collection, the focus is on access to individual elements. `String` also inherits from `Stream`, from which it derives many methods for processing a linear stream of data. Because of its dual heritage as a collection and a stream, there are often several ways to perform an operation on a string.

How Text Works

String Encoding and Unicode

The `String` class, which is the superclass of both `StringConstant` and `Text`, allows strings to be stored as a sequence of 31-bit unsigned integers, each of which represents a character in the Unicode or ISO 10646 standards. Unicode conformance allows ScriptX to display text in every major written language, and to accommodate variations in how different languages display and order individual characters.

Unicode is a standard for encoding characters and symbols used in all major writing systems, including languages that are written with non-Roman alphabets or ideographic symbols (Arabic, Chinese, Hebrew, Japanese, Korean, and others). In standard Unicode, the lower 16 bits encode character data. Integers from 1 to 127 represent the ASCII character set. (The ASCII character 0 cannot appear within a string.) Integers above 128 encode special characters and characters in non-Roman alphabets.

The original Unicode standard imposed a rigid ordering on characters, to which some nations objected. For example, the Chinese and Kanji writing systems share many ideographic symbols, but these symbols have different meanings and are ordered differently in Chinese and Japanese. ISO 10646 extends Unicode beyond 16 bits to allow for different national standards on the ordering of characters. National committees will be allocated 16-bit subspaces, which they can partition individually.

ISO 10646 permits the definition of thousands of additional character sets, should the need arise. As new standards are released, ScriptX may accommodate an additional 15 bits of information for each Unicode character. In practice, the value of the 15 high bits will usually be 0.

Since 31-bit integers are demanding of memory and storage, ScriptX implements variable-length encoding through the File System Safe Unicode Translation Format (UTF). UTF encoding is transparent to other scripts and objects. Each character is stored as a sequence of 1 to 6 bytes. Every byte in a sequence is either an initial byte or a trailing byte. The high-order bits of the initial byte indicate the number of bytes in the sequence that follows. All trailing bytes begin with binary 10 in the two high-order bits. Table 12-1 shows the format of UTF characters of varying length.

Table 12-1: UTF encoding of strings

Length	Initial Byte	Trailing Bytes…				
1 byte	0xxxxxxx					
2 bytes	110xxxxx	10xxxxxx				
3 bytes	1110xxxx	10xxxxxx	10xxxxxx			
4 bytes	11110xxx	10xxxxxx	10xxxxxx	10xxxxxx		
5 bytes	111110xx	10xxxxxx	10xxxxxx	10xxxxxx	10xxxxxx	
6 bytes	1111110x	10xxxxxx	10xxxxxx	10xxxxxx	10xxxxxx	10xxxxxx

Any byte in the sequence can be easily identified as part of a single byte or multibyte sequence. If the high-order bit is 0, the sequence is one byte in length and represents an ASCII character. Thus, a ScriptX `String` object stores text from languages written in Roman alphabets very efficiently, with all ASCII characters encoded in a single byte.

Special characters such as accents and ligatures, and characters from non-Roman alphabets such as Greek, Cyrillic, Hebrew, and Arabic are encoded in 2-byte sequences. Ideographic symbols that are used in written languages such as Japanese, Chinese, and Korean, can be encoded mostly in 3-byte sequences.

Strings as Collections

`String` inherits from `Sequence`, one of the classes that provides the collection protocol. Because strings are collections, many methods that operate on individual elements of a collection can be used to access and modify individual characters of a string. (It should be noted, however, that elements of a string are not themselves objects.)

A ScriptX expression must return an object; therefore, when an element of a string is assigned to a variable, the variable is actually assigned an integer which represents the Unicode value of the character. The result is that a string behaves as if it were a collection of 31-bit unsigned integers.

The following code demonstrates that when working with strings as collections, each character is considered an element in that collection, and each of those elements is represented as an integer value corresponding to a Unicode value.

```
getNth "foo" 2
⇨  111 -- the Unicode integer value for the character "o"
str := "remove the spaces" as String
deleteAll str 32 -- 32 is the space character
print str
⇨  "removethespaces"
```

An alternate method is to use the element access construct:

```
str[2]
⇨  101 -- the Unicode integer value for the character "e"
```

You can find the integer value of any single character using the element access expression with a string of that character:

```
"r"[1]
⇨  114
" "[1]
⇨  32
```

To coerce an integer into a string of one character, use one of the following functions. The first version of `intToString` is based on the collection behavior of a string. The second version uses the `writeByte` method defined by `Stream`, described in the next section. The two versions are almost identical in performance.

```
-- a collection version of intToString
function intToString val -> (
    local str := "" as String
    append str val -- a Sequence method
    return str
)
-- a stream version of intToString
function intToString val -> (
    local str := "" as String
    writeByte str val
```

```
    return str
)
intToString 32
⇨  " "
intToString 114
⇨  "r"
```

Since strings inherit from `Collection`, you can access the `size` instance variable to find the length of a string:

```
str := "this is a sample string" as String
str.size
⇨  23 -- str contains 23 elements
```

Methods which work on collections also work on strings. See the section on concatenating and modifying strings for more examples.

Strings as Streams

Because strings also inherit from streams, they can be treated as write-only, non-seekable streams.

Table 12-2: Behavior of String classes as streams

String Class	isReadable	isWritable	isSeekable
`String`	`false`	`true`	`false`
`StringConstant`	`false`	`false`	`false`
`Text`	`false`	`true`	`false`

Since `String` objects and `Text` objects are writable streams, methods that write to streams can operate on them. For example, the `prin` method, used to display the ASCII representation of an object, expects a stream as its last argument. By substituting a string for that argument, the result of the `prin` method is placed into that string:

```
str := "Rectangles look like: " as String
prin (new Rect) @normal str
print str
⇨  "Rectangles look like: [0, 0, 0, 0] as Rect"
```

The most common stream method used on strings is the `writeByte` method, which can append a single character to a string (the collection method `append` can perform this same operation):

```
str2 := "foo" as String
writeByte str2 102; print str2
⇨  "foof"
```

One conceptual difference between a stream and a collection is that with a stream, there is the concept of current position. This current position is called a *cursor*, and is continually updated as a script processes data in a stream.

Strings and Iterators

Iterators, which are defined in the Collections component, are streams that a script uses in order to perform an operation on each member of a collection. Since `String` inherits from `Sequence`, a script can iterate over the elements of a string by creating an instance of `SequenceIterator`, which is readable, writable, and seekable. (`String` and `Text` objects are only writable without an iterator.) The following script adds a new method to the `String` class, using an iterator to pass through each element in the stream:

```
method getUpperCase self {class String} -> (
    local lowers := new NumberRange lowerBound:97 upperBound:122
    local upperCaser := getFirst "a" - getFirst "A"
    -- make it handle string constants by coercing them to strings
    if self.mutable = false do self := self as String
    local myIterator := iterate self -- creates an iterator
    repeat while (next myIterator) do (
        if (withinRange lowers myIterator.value) do (
            myIterator.value := myIterator.value - upperCaser
        )
    )
    return self
)
getUpperCase ("GHoBi" as String)
⇨ "GHOBI"
```

Iterators are useful for functions and methods that process text. A string is not readable or seekable as a stream. By using an iterator, a script can perform additional stream operations, such as `previous` and `seekFromCursor`. For more information on iterators, see page 470 of the “Collections” chapter.

Cursor Positions

Ranges of characters in strings, as used in the string methods, selections, and text attributes, are expressed as cursor positions. Cursor positions occur between characters, with position 0 indicating the beginning of the string (before the first character). This is different from the *ordinal position* of each character (as used in collections), which occurs directly on the character itself. Figure 12-1 illustrates the difference between cursor and ordinal positions. Note that a single character is also a range of two cursor positions.

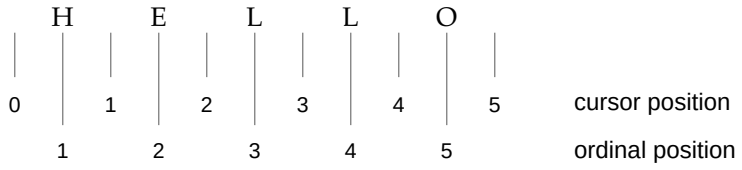


Figure 12-1: Cursor And Ordinal Positions

How Fonts Work

Fonts

Font scaling and rendering in ScriptX takes place by accessing the font technology of the underlying platform and the fonts that have been installed on that system. ScriptX provides two classes, `Font` and `PlatformFont`, that allow other ScriptX objects to access and use the available installed fonts and font technology.

To get access to a particular font family such as Helvetica, Times New Roman, Caslon Antique, etc., use an instance of the concrete class `PlatformFont`.

The `PlatformFont` object provides access to the font family specified by the `name` keyword. To use variations of that font family (for example: bold, italic, or condensed) use the attributes defined by the `TextPresenter` or `Text` classes. Text attributes are described in "Overview of Text Attributes" on page 293. The use of fonts in ScriptX is also described in more detail in that section.

Using Text and Fonts

Creating Strings

Strings can be created in various ways. The simplest way to create a string is through the use of a string literal, that is, by simply typing the string in a script surrounded by double quotes:

```
"this is a string"
```

Special Unicode characters outside the standard ASCII character set can be specified in a string literal using the `\<nnnn>` notation. The Hexadecimal number represented by `nnnn` is any valid Unicode/ISO 10646 character.

```
global myStringConstant := "ScriptX\<2122>" -- 0x2122 is "trademark" in
      -- Unicode
⇨  "ScriptX\<2122> -- ASCII representation in the debugging stream
```

Note that in the ASCII printed representation of the string (as in the ScriptX listener window), special Unicode characters may not appear or may look like garbage. The special characters appear in their proper form when the string is displayed by a presenter. Which special characters are available depends on the underlying platform on which ScriptX is running. A ScriptX text presenter displays non-ASCII characters correctly, provided that the underlying operating system interprets them correctly and matches them with an appropriate font glyph. The following script displays the `StringConstant` object `myStringConstant` in a new window, with the trademark symbol.

```
global myWindow := new Window boundary:(new Rect x2:200 y2:200)
show myWindow; myWindow.y := 40
object myBoundary (Rect) x2:100, y2:25 end
```

```
global myTextPresenter := new TextPresenter boundary:myBoundary \
    target:myStringConstant fill:whiteBrush stroke:blackBrush
-- now the non-ASCII trademark symbol shows up correctly
append myWindow myTextPresenter
show myWindow
```

String literals actually create instances of the class `StringConstant`. You can then coerce those objects into instances of `String` or `Text` using the `as` expression:

```
global myString := "This is another string" as String
```

You can also create instances of `String`, `StringConstant`, and `Text` using the `new` method with the `string` keyword:

```
global theText := new Text string:"this is a text string"
```

Since `StringConstant` objects are immutable, a method that tries to modify an instance of `StringConstant` will produce an exception. The easiest way to avoid an exception is to coerce the `StringConstant` object into a `String` object.

Creating Fonts

Since `Font` is an abstract class, you do not create `Font` objects directly. It is possible to use the method `new` with `Font`, but that will create a `PlatformFont` object which is set to the default system font. If you want to specify a font, you must create an instance of `PlatformFont`, using the `new` method and the appropriate keywords. The possible keywords are `name`, `macintoshName`, `windowsName`, and `os2Name`. If you use the `name` keyword, the use of any of the other keywords is optional. In other words, you can use one, two, three, or four keywords in any combination as long as `name` is one of them. If you do not use the `name` keyword, you must use all of the platform-specific keywords.

If you supply the name of a platform-specific font and that font is available on the platform being used, it takes precedence over the font supplied for the `name` keyword. If you did not specify platform-specific fonts, or the ones you specified are not available, ScriptX will try to use the font you supplied for the `name` keyword, if you supplied one. Otherwise, it will use the default font for that platform.

When ScriptX is attempting to use the font supplied for `name`, it searches for a matching string and uses the first one it finds. For example, if you give “Times” for `name`, the font used could be “Times”, “Times Roman”, “Times New Roman”, or any other font beginning with “Times” that is available on that platform, depending on which one was encountered first. For this reason, it is advisable to supply a platform-specific font if you want a specific font on a particular platform. If the font supplied for `name` is not found, ScriptX uses the default font for that platform.

```
fnt := new PlatformFont macintoshName:"Times" \
    windowsName:"Times New Roman" os2Name: "Times"
```

If both the generic `name` and a platform-specific name are used in the same `new` statement, the platform-specific name is used if ScriptX is being run on that platform. Otherwise, the generic name is used. For example, if ScriptX is being run on the Macintosh, this `new` method returns an instance of the font named Optima:

```
fnt := new PlatformFont name:"Times" macintoshName:"Optima"
```

However, the following expression on the Macintosh version of ScriptX returns an instance of the font named Times (`windowsName` is ignored)

```
fnt := new PlatformFont name:"Times" windowsName:"Optima"
```

If the named font is not found, ScriptX uses the default system font. You can also get access to the default system font by using the `default` class variable, defined on the `Font` or `PlatformFont` classes:

```
PlatformFont.default
```

⇨ PlatformFont@0x11337a8 -- the address of a PlatformFont object

Presenting Strings

To present a string on screen, you need a text presenter. The ScriptX Core Classes define two presenters for text, `TextPresenter` and `TextEdit`. Generally, the `TextPresenter` class displays static text, while the `TextEdit` class displays text that the user can modify. The `TextEdit` class provides additional behavior for setting insertion points and selections within the displayed text using a mouse, and for inserting and deleting text using the keyboard.

Both the `TextPresenter` and `TextEdit` classes use an instance of the `String` class (generally a `Text` object) as their target. The target of a text presenter is the text that the presenter is actually presenting. The overall appearance of that text is defined by the presenter through the use of text attributes, such as font, color, amount of space between lines, etc. “Overview of Text Attributes” on page 293 describes text attributes and how to use them.

Any text which will be used as the target of a text presenter should always end with a carriage return. If it does not, the method `calculate` will not operate on it correctly. The description of `calculate` included in the definition of the class `TextPresenter` in the *ScriptX Class Reference* gives examples and more information.

Although there is no limit to the length of a `String` object, there is a limit to the amount of text that can be displayed in a text presenter at any one time. `TwoDPresenter` objects are limited to 32K in height (and width), and since `TextPresenter` and `TextEdit` objects inherit from `TwoDPresenter`, they

also have a 32K limit. For example, if you have some 12 point text with a leading of 1 (1 point of space between lines), you can display up to 2520 lines worth of that text (32K / (12 +1)).

Creating New Instances of TextPresenter and TextEdit

The `TextPresenter` and `TextEdit` classes are subclasses of `TwoDPresenter` and are created in much the same way that other presenters are. Text presenters, in particular, require two initialization keywords, `boundary` and `target`.

```
myTP := new TextPresenter \
    boundary:(new Rect x2:300 y2:300)\
    target:"This is a string"
```

The `boundary` keyword requires a `Stencil` object which defines the shape and size of the text presenter. In the current ScriptX release, that boundary must be an instance of the class `Rect`.

The `target` keyword specifies the actual characters that are to be presented, that is, a `String`, `StringConstant`, or `Text` object. Whereas a `TextPresenter` object can have any of the three string types as its target, a `TextEdit` object requires that its target be a `Text` object. This is the case because a `TextEdit` object needs a string with attributes in order to do any editing. Since `String` and `StringConstant` objects do not have attributes, you cannot change the appearance of individual characters within either of them.

The `TextPresenter` class defines two other initialization keywords: `fill` and `stroke`. Both require a `Brush` object. Through inheritance, those keywords are available to the `TextEdit` class as well. The `fill` keyword defines the color and pattern of the text presenter's background; the `stroke` keyword defines the color, pattern, and width of the boundary of the text presenter.

Setting Other Instance Variables

The `offset` instance variable is a cursor position (as opposed to an ordinal position), and the text which follows it will appear starting at the top left corner of the presenter. By default, the `offset` is 0, meaning the presenter displays the text starting at the first character. Changing the presenter's `offset` has the effect of scrolling the text in the presenter such that the character after the given offset is at the top left corner.

The `inset` instance variable is used to set off text from the edges of the text presenter. If, for example, the stroke of the boundary is several points thick, you would need to set the inset so that the text is not covered by the boundary. The value of the `inset` instance variable is a `Point` object, in which the `x` value of that object determines the horizontal (left to right) inset in pixels, and the `y` value determines the vertical (upper to lower) inset.

The `selectionForeground` and `selectionBackground` instance variables define the appearance of selected portions of text. Note that both `TextPresenter` and `TextEdit` objects can have selections, but only `TextEdit` objects can make a selection using the mouse. The values of both `selectionForeground` and `selectionBackground` are instances of the `Brush` class. See "Setting Selections, Insertion Points, and Cursors" on page 303 for more details.

The `cursor` and `cursorBrush` instance variables define the shape and appearance of the cursor marking the current insertion point. See "Setting Selections, Insertion Points, and Cursors" on page 303 for more details on defining a cursor.

Overview of Text Attributes

Text attributes define how text is to be displayed and formatted in a text presenter. Text attributes are contained in a keyed linked list of key-value pairs, where the key is the name of an attribute (such as `@style` or `@size`) and the value is the value of that attribute (such as `@italic` or `14`).

When you create a `TextPresenter` object, its attributes are automatically initialized to a set of default values. These default values can subsequently be set to new values if you want to change them. The values to which the attributes of the text presenter are set (either automatically at initialization or explicitly by the programmer) become the default values for the string being presented. Because of this, you can think of the attributes of a text presenter as a template for the appearance of the text it presents. These attributes apply to the target text globally, meaning that if the `@size` attribute is `12`, then all characters in the target string will have a point size of 12. If the presenters's `@weight` attribute is `@bold`, then all charcters are bold, and so on. If the target is a `String` object or a `StringConstant` object, the appearance is determined completely by the attributes of the presenter.

When the target of a text presenter is a `Text` object, the appearance of the text can be changed because `Text` objects, unlike `String` and `StringConstant` objects, have attributes. A `Text` object is created with its default attributes all set to `undefined`; as a result, its attributes are set to those of its presenter at initialization. Initially, then, all three types of strings are displayed with the attributes of their presenters, which is why the attributes of a text presenter can be thought of as the default attributes for the string it is presenting. With `Text` objects, however, you can change the values of attributes, and any change made in a `Text` object attribute overrides the attribute setting of its presenter. Additionally, in a `Text` object you can specify a particular character or range of characters where the change takes effect. This differs from text presenters, where attributes affect all characters and cannot be set to affect just certain ones.

A `TextPresenter` object is the appropriate presenter when you want to display textual material that will not be changed by the user. A `String` or `StringConstant` object is the appropriate target for a `TextPresenter` object when you want all of the characters being presented to share the same attributes (those of the text presenter). You need to use a `Text` object as the

target of a presenter if you want some text indented or underlined or a different size or a different style, etc. And if you want the user to be able to select text and modify it using the keyboard or a mouse, then you need a `TextEdit` object as the presenter with a `Text` object as its target text.

In summary, text attributes are available in a text presenter (a `TextPresenter` object or a `TextEdit` object) and in a `Text` object serving as a presenter's target. The attributes of the text presenter determine the overall look of the text; individual attributes defined by a `Text` object override the attributes of the text presenter for specific characters (for example, to make a particular word bold or to center a title). Because `String` and `StringConstant` objects do not have attributes, the appearance of individual characters within them cannot be changed. Also, `String` and `StringConstant` objects cannot be used as the target of a `TextEdit` object; the target must be a `Text` object.

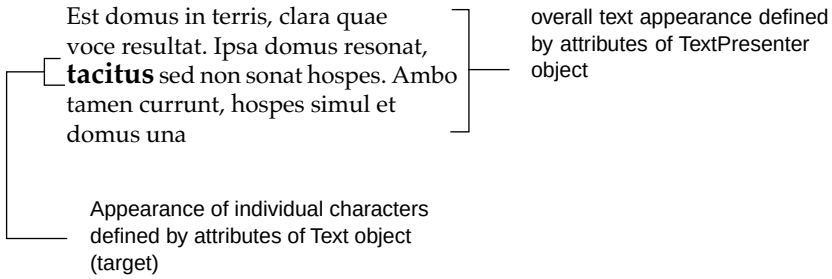


Figure 12-2: Attributes of TextPresenter and Text

New instances of `TextPresenter` or its subclasses are created with a set of default attributes to determine a simple look for text. You can see what those default attributes are by calling the method `getDefaultAttrs` on a newly-created instance of `TextPresenter` and then supplying the result as an argument to `prin`.

```
tp := new TextPresenter target:"foo" boundary:(new Rect)
prin (getDefaultAttrs tp) @complete debug -- print them all
⇨  #(@font:PlatformFont@0x12b6b28, @size:12, @weight:@regular,
@width:@normal, @style:@roman, @leading:13, @alignment:@fill,
@indent:0, @indentFromEnd:0, @paraIndent:0, @brush:Brush@0x1290548,
@underline:0)
```

Attributes defined by a `Text` object must be set explicitly for a range of characters using the `setAttr` or `setAttrFromTo` methods. You can also query the attributes for a specific character or character range by using the methods `getAttr` and `getAttrs`.

```
theText := "this is some Text" as Text
-- turn the first word bold
setAttrFromTo theText @weight 0 4 @bold
-- make the last word 18 points
setAttrFromTo theText @size 13 17 18
-- set the first three words to a new font
setAttrFromTo theText @font 0 13 (new PlatformFont name:"Times")
-- query stuff
getAttrs theText 3
```

```
⇨  #(@weight:@bold, @font:PlatformFont@0x12e51a8)
getAttrs theText 8
⇨  #(@font:PlatformFont@0x12e51a8)
getAttrs theText 14
⇨  #(@size:18)
```

The `Text` class also defines a `getAttrRange` method that, given a text object, an attribute key, and a cursor position, returns the range of text for which that attribute applies:

```
getAttrRange theText @size 15
⇨  13 to 15 -- the point size is 15 from cursor position 13 to 15
```

The default attributes of `Text` objects are defined by the `defaultAttributes` class variable (initialized to `undefined` by default). You can modify that class variable and create a template for new instances of `Text`, but doing so is not normally recommended. Setting `defaultAttributes` affects all new `Text` objects, so if another title is running, you will change the default attributes for any new `Text` objects in that title in addition to those in your own title.

Text and TextStencil Objects

A `TextStencil` object is not part of the Text and Fonts Component, but it is related because it must be supplied with a `String` object for its `string` instance variable. Instances of the class `TextStencil` can be used to display text as an image (for example, to put labels and callouts on illustrations and user interface controls). To do this, you need to create an instance of `TwoDShape` using a `TextStencil` object as the *stencil* argument. If the `TextStencil` object's string is a `Text` object, and if its font and size attributes have been set, then whatever those attributes are set to at cursor position 0 will be carried over to the `TextStencil` object. In this case, you do not need to specify a font or size when you create a `TextStencil` object. If you do specify a font or size, each one will override that of the `Text` object.

`TextStencil` objects are similar to `TextPresenter` objects in that their attributes affect all their text and cannot be set to affect only certain characters. They are different, however, in many ways. `TextPresenter` objects can have a multitude of attributes, whereas `TextStencil` objects have only font and size as attributes. Also, `TextStencil` objects are not presenters; in order to be displayed, they must be the *stencil* argument for a `TwoDShape` object.

List of Attributes

There are several attributes available for `Text` and `TextPresenter` objects. The attributes for both classes are the same except that `TextPresenter` does not use the `@action` attribute. Table 12-3 summarizes the attributes available to both, and each attribute is described in greater detail on the following pages.

Table 12-3: Text Attributes

Attribute	Possible values
`@action`	`AbstractFunction` object (that is, a function or method)
`@brush`	`Brush` object
`@font`	`Font` object
`@size`	`Number` object
`@weight`	`@extraLight`, `@light`, `@regular`, `@medium`, `@demiBold`, `@bold`, `@extraBold`, `@heavy` (currently, only `@light`, `@medium`, and `@bold` are available)
`@width`	`@condensed`, `@normal`, `@expanded`
`@style`	`@roman`, `@italic`, `@oblique`
`@underline`	`0`, `1` (for future compatibility with numeric values)
`@leading`	`Number` object
`@paraLeading`	`Number` object
`@firstLineLead -ing`	`Number` object
`@alignment`	`@flush, @flushLeft, @flushToEnd, @flushRight, @fill, @center, @tty`
`@paraIndent`	`Number` object
`@indent`	`Number` object
`@indentFromEnd`	`Number` object

Getting and Setting Attributes

`TextPresenter` attributes govern the appearance of the presenter's target string. The values for all attributes can be accessed using the method `getDefaultAttrs`. By using this method on a newly-created instance of `TextPresenter`, you can see a listing of the default values with which the `attributes` instance variable was automatically initialized:

```
global tp := new TextPresenter target:"test" boundary: (new Rect)
getDefaultAttrs tp
⇨  #(@font:PlatformFont@0x12b6b28, @size:12, @weight:@regular,
@width:@normal, @style:@roman, @leading:13, @alignment:@fill,
@indent:0, @indentFromEnd:0, @paraIndent:0, @brush:Brush@0x1290548,
@underline:0)
```

You can change the values for attributes by using the method `setDefaultAttr`, which takes the arguments *self*, the name of the attribute, and the value to which the attribute is to be set.

```
setDefaultAttr tp @size 18
```

To see what the attribute `@size` is now set to, you can use the method `getDefaultAttr`, as demonstrated in the following code:

```
getDefaultAttr tp @size
⇨  18
```

Since `TextEdit` inherits from `TextPresenter`, `TextEdit` objects are initialized with the same default attributes as `TextPresenter` objects. This is demonstrated in the following code:

```
global myText := new Text string:"testing"
global myTE := new TextEdit target:myText boundary: (new Rect)
prin (getDefaultAttrs myTE) @complete debug -- to print them all
⇨  #(@font:PlatformFont@0x12b6b28, @size:12, @weight:@regular,
@width:@normal, @style:@roman, @leading:13, @alignment:@fill,
@indent:0, @indentFromEnd:0, @paraIndent:0, @brush:Brush@0x1290548,
@underline:0)
```

Setting the Font, Size, Weight, Width, and Style

The `@font`, `@size`, `@weight`, `@width` and `@style` attributes determine the family name, size, and variation of the font with which this text is displayed.

The `@font` attribute specifies the font family this text uses. The value for the `@font` attribute is an instance of the class `PlatformFont`, and the specified font must be installed on the system for it to be available. “Overview of Text Attributes” on page 293 describes how to create new instances of the `PlatformFont` class.

```
setDefaultAttr thePresenter @font \
    (new PlatformFont macintoshName:"Avant Garde" \
        windowsName:"Arial" os2Name:"Times Roman")
```

The `@size` attribute specifies the size of the current font in points (one point is 1/72 of an inch), measured from the baseline of the font to the baseline of the line of text above it. Note that the actual size of font glyphs (characters) may vary from font to font even though the point size may be the same (for example, an x in 12 point Avant Garde is different from an x in Times Roman).

```
setDefaultAttr thePresenter @size 18
```

The `@weight`, `@width` and `@style` attributes are used to access variations of a particular font, for example, bold, italic, or condensed. ScriptX attempts to use an installed variation of a font family before constructing one.

The `@weight` attribute specifies the "boldness" of the text and can have one of eight values: `@extraLight`, `@light`, `@regular`, `@medium`, `@demiBold`, `@bold`, `@extraBold`, and `@heavy`.

```
setDefaultAttr thePresenter @weight @bold
```

Note – In the current version of ScriptX, `@extraLight` is equivalent to `@light`, `@regular` is equivalent to `@medium`, and `@demiBold`, `@extraBold`, and `@heavy` are all equivalent to `@bold`.

The `@width` attribute specifies how closely the letters in the font are spaced (as defined by the font itself). The `@width` attribute can have one of three values: `@condensed`, `@normal`, and `@expanded`.

```
setDefaultAttr thePresenter @width @normal
```

The `@style` attribute determines the obliqueness (italics) of the font, and can have one of three values: `@roman`, `@italic`, or `@oblique`.

```
setDefaultAttr thePresenter @style @italic
```

Note – In the current version of ScriptX, `@italic` is equivalent to `@oblique`.

Setting the Color

The `@brush` attribute determines the color of the text which it affects. The `@brush` attribute contains an instance of the `Brush` class. Note that although `Brush` objects can contain a pattern, that pattern is ignored.

```
setDefaultAttr thePresenter @brush \
    (new Brush color:blueColor)
setAttrFromTo theText @brush 0 50 \
    (new Brush color:blueCOlor)
```

Setting Underline

The `@underline` attribute can have one of two values: 0 or 1. The 0 value specifies that the text does not have an underline; 1 specifies that it does.

```
setDefaultAttr thePresenter @underline 1
```

Note – Future versions of ScriptX will use the `@underline` attribute to specify the distance away from the text the underline should appear, in points. This is why the values for `@underline` are numeric rather than `true` and `false`.

Setting Leading

Text leading determines the spacing between lines of text. There are three forms of leading attributes: `@leading`, `@paraLeading`, and `@firstLineLeading`. All three contain a number representing points.

The `@leading` attribute determines the spacing between lines of text within a paragraph, from baseline to baseline. Text leading in ScriptX is independent of the size of the font, and, in fact, if you set the `@leading` attribute to be less than the `@size` attribute, the lines of text will run together. Typically, the value of leading is set slightly larger than the size of the text.

```
setDefaultAttr thePresenter @size 12
setDefaultAttr thePresenter @leading 14
```

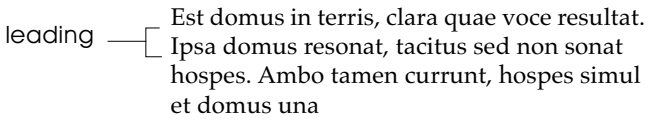


Figure 12-3: Leading

The `@paraLeading` attribute determines the baseline to baseline spacing between paragraphs in the text presenter (where the end of a paragraph is defined as a linefeed or carriage return). If `@paraLeading` is smaller than the `@leading` attribute (or `empty`), `@leading` is used instead.

```
setDefaultAttr thePresenter @leading 12
setDefaultAttr thePresenter @paraLeading 14
```

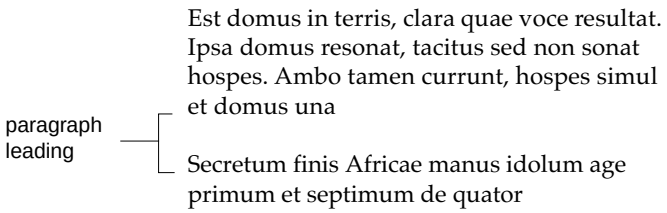


Figure 12-4: Paragraph Leading

The `@firstLineLeading` attribute determines the spacing between the first line of text in a text presenter and the top of the text presenter. Without first line leading, parts of the first line of text may be cut off at the top of the presenter. If the `@firstLineLeading` attribute is smaller than `@leading`, or `undefined`, the value of `@leading` is used instead. If the line at the top of the text presenter is also the beginning of a paragraph, and the value of `@firstLineLeading` is smaller than `@paraLeading`, the value of `@paraLeading` is used instead.

```
setDefaultAttr thePresenter @leading 12
setDefaultAttr thePresenter @firstLineLeading 14
```

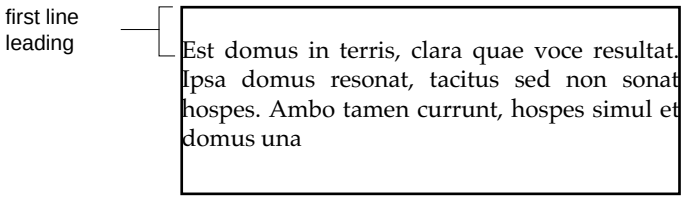


Figure 12-5: First Line Leading

The distance beween baselines in text is actually determined by testing the following values, in the order given below:

`@firstLineLeading` – if it is the first line in the text presenter and the value is set and non-zero

`@paraLeading` – if it is the first line in a paragraph and the value is set and non-zero

`@leading` – if the value is set and non-zero

`@size` – if the value is set and non-zero

The first of these four tests to succeed determines the distance between baselines. (A value of 0 for any of these properties is the same as the property not being set.)

Setting Alignment

Whereas leading determines the spacing between lines in the vertical direction, alignment determines the arrangement of those lines in the horizontal direction. The `@alignment` attribute can have one of seven values: `@flush`, `@flushLeft`, `@flushToEnd`, `@flushRight`, `@fill`, `@center`, and `@tty`.

The `@flush` and `@flushLeft` values for the `@alignment` attribute, which are equivalent, arrange the lines of text so that they are aligned along the left edge:

```
setDefaultAttr thePresenter @alignment @flush
```

Est domus in terris, clara quae voce resultat.
Ipsa domus resonat, tacitus sed non sonat
hospes. Ambo tamen currunt, hospes simul et
domus una

Figure 12-6: Flush Left Alignment of Text

The `@flushToEnd` and `@flushRight` values are equivalent and arrange the lines of text so that they are aligned along the right edge:

```
setDefaultAttr thePresenter @alignment @flushRight
```

Est domus in terris, clara quae voce resultat.
Ipsa domus resonat, tacitus sed non sonat
hospes. Ambo tamen currunt, hospes simul et
domus una

Figure 12-7: Flush Right Alignment of Text

The `@fill` value adjusts the whitespace between words in the lines so that both the left and right margins are aligned:

```
setDefaultAttr thePresenter @alignment @fill
```

Est domus in terris, clara quae voce resultat. Ipsa domus resonat, tacitus sed non sonat hospes. Ambo tamen currunt, hospes simul et domus una

Figure 12-8: Fill Alignment of Text

The `@center` value adjusts the white space around either side to center the line within the presenter:

```
setDefaultAttr thePresenter @alignment @center
```

Est domus in terris, clara quae voce resultat. Ipsa domus resonat, tacitus sed non sonat hospes. Ambo tamen currunt, hospes simul et domus una

Figure 12-9: Center Alignment of Text

And, finally, the `@tty` alignment, which is intended to be used for code samples in a monospaced font such as Courier, aligns the text along the left margin, does not wrap at the right margin without a carriage return, and allows tab characters to be represented as four spaces (tabs are ignored in all other forms of text presentation).

```
setDefaultAttr thePresenter @alignment @tty
```

```
class SampleClass (RootObject)
    instance vars a, b, c
    instance methods
    method fluff self -> (
        print "fluff!"
    )
end
```

Figure 12-10:TTY Alignment of Text

Setting Indentation

Indentation determines the marginal indents of the text from both sides of the presenter. There are three indentation attributes: `@indent`, `@indentFromEnd`, and `@paraIndent`. All three indentation attributes contain a number representing points.

Note that all the indentation attributes are independent of the `inset` instance variable, a property of `TextPresenter`. The inset specifies the distance of the text from the presenter's boundary. The three indentation attributes indent the text away from that inset.

The `@indent` attribute specifies the indentation of the left side of the text from the edge of the text presenter, but does not affect the first line of text:

```
setDefaultAttr thePresenter @indent 4
```

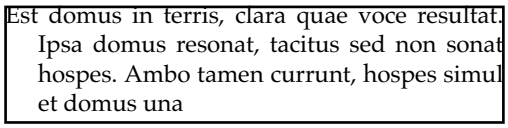


Figure 12-11:Left Indent

The `@indentFromEnd` attribute specifies the indentation of the right side of the text from the edge of the text presenter:

```
setDefaultAttr thePresenter @indentFromEnd 4
```

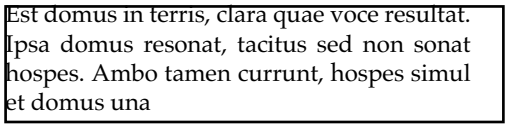


Figure 12-12:Right indent

Finally, the `@paraIndent` attribute specifies the indentation of the first line of text in each paragraph. The `@paraIndent` attribute can be greater than `@indent` for an indented first line, or less than the `@indent` value for an "out-dented" first line.

```
setDefaultAttr thePresenter @paraIndent 4
```

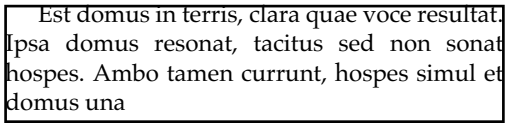


Figure 12-13:Paragraph Indent

Setting Actions

The `@action` attribute is available only on ranges of characters in `Text` objects, and contains a function or method object which is executed when a point in this range of text is selected in a text presenter. Examples might be a hypertext link or an annotation that plays a sound.

The function that this attribute holds must be defined with three arguments, which are passed by the text presenter to the function. You do not need to use those arguments in the body of that function, but the function must be defined to include them:

- The `TextPresenter` (or `TextEdit`) object that was clicked on
- The range of characters with the same `@action` attribute
- The offset from the beginning of the text that received the mouse click

For example, the following function turns the text presenter that receives the mouse click to "inverse video": the fill of the presenter is turned black and the text itself is turned white.

```
function inverse thePresenter theRange theOffset -> (
```

```
    thePresenter.fill := blackBrush
    setDefaultAttr thePresenter @brush whiteBrush
)
```

"Using Text Actions" on page 312 provides more detail on text actions.

Setting Selections, Insertion Points, and Cursors

A *selection* is a character or range of characters that has been highlighted for editing (`cutSelection`, `copySelection`, `pasteToSelection`) or for other operations. The `selectionForeground` and `selectionBackground` instance variables, defined on `TextPresenter` (and therefore available to `TextEdit` and other subclasses of `TextPresenter`) hold an instance of the `Brush` class and determine the color of the selection. The value of the `selectionForeground` variable determines the appearance of the selected text itself, and the value of `selectionBackground` determines the appearance of the space around the selection. If `selectionBackground` is `undefined` (or is the same color as the fill of the presenter), the whitespace between characters does not appear to be selected, even if it is.

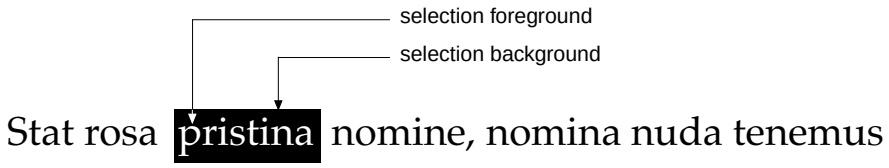


Figure 12-14:Selection Foreground and Background

The selection itself, that is, which characters within the target are selected, is defined by the presenter's target. The `Text` class has a `selection` instance variable which can hold one of three values: `empty`, representing no selection; a single integer, representing a single cursor position; or a range (a `NumberRange` object) representing a range of cursor positions. Only `Text` objects can hold selections; `TextPresenter` objects that use instances of `String` or `StringConstant` as their target cannot display selected text.

In `TextEdit` objects, selecting text with the mouse changes the values of the `selection` instance variable on the target. You can also set the value of a target's `selection` in a script, and the selection appears in the presenters that present that target.

Selections have an appearance only when the `selection` instance variable holds a range of cursor positions. If the `selection` instance variable holds a single cursor position, that selection is considered to be a single insertion point at which additional text can be inserted into the target. Insertion points can be graphically represented in a presenter by defining the shape and appearance of a *cursor*.

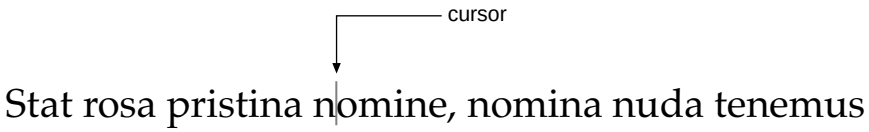


Figure 12-15:A cursor

In `TextPresenter` or `TextEdit` objects, the appearance of the cursor is determined by the `cursor` and `cursorBrush` instance variables. The `cursor` variable holds a `Stencil` object representing the shape and location of the cursor (typically an instance of `Rect` or `Line`); the `cursorBrush` variable determines the color and pattern of the cursor. Both must be defined, and the selection in the target must be set, for the cursor to appear.

Once a cursor has been defined, clicking the mouse at a given point in the text in a `TextEdit` object causes that cursor to appear and assigns the offset of that insertion point to the `selection` instance variable in the target. You can also set the insertion point of a target's `selection` instance variable in a script, and the cursor appears at that position.

The registration point of the cursor (that is, the place it is drawn), is at the baseline of the line of text, and at the midpoint between two characters. For this reason, it is common to define the cursor's stencil to draw some points back from the center (the negative x direction) or below the baseline (the positive y direction), depending on the effect needed.

```
theTextPresenter.cursor := \
    (new Rect x1:-1 y1:-10 x2:5 y2:1)
```

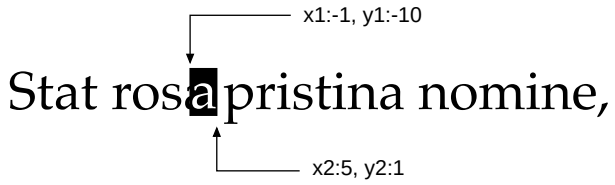


Figure 12-16:Cursor Position

When both the appearance of the selection and a cursor are defined, and a range of text is selected, the cursor appears at the end of the selection.

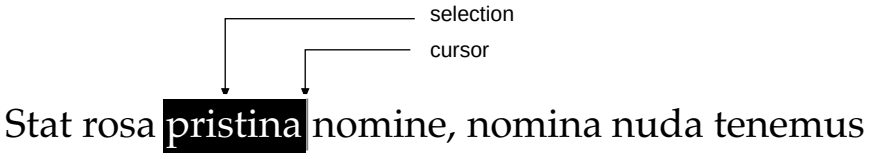


Figure 12-17:Selections and Cursors

Changing Default Values for Selections and Cursors

New instances of `TextPresenter` and `TextEdit` are automatically initialized with default values for the instance variables `cursor`, `cursorBrush`, `selectionBackground`, and `selectionForeground`. (Default values for the instance variable `attributes` are discussed under "Getting and Setting Attributes".) The default for `cursor` is `undefined` (which means that no cursor is displayed), the default for `cursorBrush` is `blackBrush`, the default for `selectionBackground` is `blackBrush`, and the default for `selectionForeground` is a `Brush` object with `color: redColor`. These can be accessed and set to new values using the standard getter and setter methods. The following code shows that the default color of the cursor for the newly-created `TextEdit` object is black (`Brush@0x119edb0`) and then changes the color of the cursor to `cyanColor`.

```
global te := new TextEdit target: ("text string" as Text) \
   boundary: (new Rect)
te.cursorBrush
⇨  Brush@0x119edb0

te.cursorBrush := (new Brush color: cyanColor)
```

Concatenating and Modifying Strings

There are many ways to add to a string or to delete from a string because you can use `Collection` and `Stream` methods as well as methods which are defined in `String`. The `Collection` and `Stream` methods used below are generic functions, which means that their implementation can be specialized for subclasses. Since strings inherit from `LinearCollection` and `Sequence`, generic functions called on strings will use the `LinearCollection` and `Sequence` implementations. For instance, a method like `forEach` will process the elements of a string in sequential order.

In general, the methods which are the most useful and the easiest to use are those which have been defined for `String`. These include the operators `+` and `–` (for string arithmetic) and the method `insertAt`. In addition, the global function `format` allows you to print text using variables.

Adding a String to Another String

The following methods and functions add a string to another string:

- Operator +

Perhaps the easiest way to concatenate strings is to use the language operator + (addition). The + operator combines its two operands into a single string:

```
"this " + "and that"
⇨  "this and that"
```

The result of both the + and - operations is an instance of the class `String`, regardless of the class of either of the operands. (The - operator is described later, in the section on deleting characters and strings.)

- `insertAt` allows you to add a `String` object to another `String` object at a particular insertion point. One of the features of this method is that if you use it to insert a `Text` object into another `Text` object, it will insert both the characters and the attributes associated with those characters. Note that both the text being inserted and the text into which it is being inserted must be `Text` objects in order to preserve the attributes of the inserted text. No other method will preserve attributes. In other words, if you want to add formatted text to another string, you must use `insertAt`, and both strings must be `Text` objects.

The following code creates a `Text` object with all characters in bold typeface and adds it to the end of a string:

```
global myText := new Text string: "is bold text."
setAttr myText @weight 0 @bold
global myFirstText := "What gets added here " as Text
insertAt myFirstText myText myString.size
⇨  "What gets added here is bold text."
```

In order to see that "is bold text" is indeed bold, you need to create a text presenter and put it in a window to display the newly modified string:

```
global tp := new TextPresenter \
    boundary:(new Rect x2:320 y2:480) \
    target:myString \
    stroke:blackBrush
tp.x := 20
tp.y := 40
global win := new Window boundary:(new Rect x2:640 y2:480)
append win tp
show win
```

- `format` is a global function which allows you to append text using variables.

```
global i := "1000"
global myString:= "The results are: " as String
format myString "%* rounded to the nearest 10" i @unadorned
⇨  "The results are: 1000 rounded to the nearest 10"
-- the style @unadorned removes the quotation marks from "1000"
```

- `addMany` appends one collection (in this case a string) to another. It modifies the first argument instead of returning a value, so in the following example, `print` is called to show the resulting string.

```
global str := "this is a sample string" as String
addMany str " thing"; print str
⇨  "this is a sample string thing"
```

- `writeString` appends one string to another string

```
global txt := "Some words" as String
global str := " with some more added"
writeString txt str--adds str to the end of txt
⇨  "Some words with some more added "
```

Adding One Character to a String

The following `Collection` and `Stream` methods add one character at a time. Note that the character to be added must be given as an integer which represents the Unicode value of the character.

- `writeByte` adds one character to the end of a string.

```
global txt := "Some word" as String
writeByte txt "s"[1] --adds "s" at the end of txt
⇨  "Some words"
writeByte txt 32 --adds a space to the end of txt
⇨  "Some words "
```

- `append` adds the given value to the end of a sequence and returns the key (in the case of a string, the key is the ordinal position) at which the value was inserted.

```
global s := new String string:"This is incomplet"
append s 101
⇨  18 -- the appended "e" is the eighteenth character
s
⇨  "This is incomplete"
```

- `appendNew` adds the given value to the end of a sequence only if that value does not already appear in the sequence. It returns `empty` if a string already contains the given Unicode value; otherwise, it returns the ordinal position where value was added.

```
appendNew s "x"[1]
⇨  19
s
⇨  "This is incompletex"
appendNew s "x"[1]
⇨  empty
```

- `prepend` adds one Unicode value (character) to the beginning of a sequence and returns the key (in this case, the ordinal position) at which the value was inserted.

```
global myString := "mall is better" as String
prepend myString 115
⇨  1
myString
⇨  "small is better"
```

- `prependNew` adds one Unicode value (character) to the beginning of a sequence only if that value does not already appear in the sequence. It returns `empty` if a string already contains the given value; otherwise, it returns the ordinal position where value was added.

```
prependNew myString "x"[1]
⇨  1
myString
⇨  "xsmall is better"
prependNew myString "x"[1]
⇨  empty -- returns empty because there is already an "x" in myString
```

- `add` inserts the given character at the ordinal position specified. (Note that `insertAt` will do the same thing, and since it is a method specifically for strings, you can supply one or more characters as a string instead of the Unicode value for a character.) The methods `addNth`, `addFirst`, `addSecond`, `addThird`, `addFourth`, and `addFifth` are similar to `add` and can also be used, but, as stated before, using `insertAt` is easier.

```
global anotherString := "This was complete yesterday." as String
add anotherString 18 "d"[1]
⇨  18 -- the return value is the ordinal position (key)
anotherString -- see what anotherString contains now
⇨  "This was completed yesterday."
```

Deleting a Character or a String

There are many ways to delete a character or a string. The following are those that are most useful:

- – operator. Using the language operator – (subtraction) is an easy way to delete a string. The – operator removes the second string from the first. If the second string is not present in the first, the first string is returned unchanged.

```
"bacon, lettuce, tomato, and sprouts" - ", and sprouts"
⇨  "bacon, lettuce, tomato"
"bacon, lettuce, tomato, and sprouts" - "with mayo"
⇨  "bacon, lettuce, tomato, and sprouts"
```

- `deleteOne` removes the first occurrence of the given character (which must be expressed as the integer representing its Unicode value), returning `true` if successful and `false` otherwise.

```
global ourString := "bats, balls, gloves, mitts" as String
deleteOne ourString "s"[1]
⇨  true
ourString
"bat, balls, gloves, mitts"
```

- `deleteAll` removes all occurrences of the given character (which must be expressed as the integer representing its Unicode value) and returns the number of characters deleted.

```
global myString := "girls, boys, cats, dogs"
deleteAll myString "s"[1] --removes all four occurrences of "s"
⇨ 4
myString
⇨ "girl, boy, cat, dog"
```

- `deleteNth` removes the *n*th character in a string, returning `true` if it was successful and `false` otherwise.

```
deleteNth myString 12
⇨ true
myString
"girl, boy, at, dog"
```

- `deleteFirst`, `deleteSecond`, `deleteThird`, `deleteFourth`, `deleteFifth`, and `deleteLast` all remove the character in the designated position.

```
deleteLast myString
⇨ "girl, boy, at, do"
deleteThird myString
⇨ true
myString
⇨ "gil, boy, at, do"
```

- `deleteRange` removes the string supplied, returning `true` if the string was successfully deleted and `false` if it was not found.

```
global yourString := "Do you like my sample text?" as String
deleteRange yourString " sample"
⇨ true
yourString
⇨ "Do you like my text?"
```

- `emptyOut` clears out the entire string, leaving an empty string

```
emptyOut yourString
⇨ OK
yourString
"" -- yourString is now an empty string
```

Searching Strings

Searching with Global Functions

You can search strings using two global functions, `findNthContext` and `searchIndex`, and also with various collection methods.

- `findNthContext` global function allows you to parse strings.

With `findNthContext` you can search for the nth word, sentence, or paragraph. You can also define your own delimiter and search for the nth set of characters bounded by that delimiter.

The following code segment illustrates searching for the fifth word in a string:

```
global str := "first, second, third, fourth, fifth"
global args := #(str, 5, 0, str.size) as Quad
findNthContext args @word
⇨  true
copyFromTo args[1] args[3] args[4] -- show the result
⇨  "fifth"
```

You set your own delimiter by supplying an anonymous function with the Unicode value of the character you want to be the delimiter. The following code searches for the second set of characters bounded by a colon (“:”) Notice that spaces are included.

```
--first find out the Unicode value for ":"
":"[1]
⇨  58
global myStr := "name: year or date:age:occupation:address"
args := #(myStr, 2, 0, myStr.size) as Quad
findNthContext args (r -> r == 58)
⇨  true
copyFromTo args[1] args[3] args[4]
⇨  " year or date"
```

- `searchIndex` allows you to search for a match to a string

The global function `searchIndex` is very fast because it actually searches a signature index, which is created when you create an instance of the class `StringIndex`. It conserves memory if the text is saved to disk because it brings only the (smaller) index into memory while searching. In addition to a `StringIndex` object, you must also create a `SearchContext` object, which will be used as one of the parameters to `searchIndex`, telling it where in the index to begin searching. You create the first instance of `SearchContext`, which is used to search for the first occurrence of a given string, by calling the global function `initialSearchContext`. (You never call `new` to create a `SearchContext` object.) The function `searchIndex` uses this `SearchContext` object as a parameter and also creates one as a return value. If you want to search for the second occurrence of the same string, you use the

`SearchContext` object returned by the first call to `searchIndex` as a parameter. To search for the next occurrence, you supply `searchIndex` with the `SearchContext` object it returned on the previous call.

```
-- create a StringIndex object
global strIndex := new StringIndex \
    string:"Her thermal underwear is here in her bag."

-- create a SearchContext object. The second argument indicates where
-- in the string to begin the search (as a cursor position)
global sc0 := initialSearchContext strIndex 0


-- search for the first occurrence of "her". "true" means that the
-- match must be a whole word ("her" embedded within another word will
-- not be considered a match)
global sc1 := searchIndex strIndex "her" sc0 true
-- verify the return value
copyFromTo sc.string sc.startOffset sc.endOffset
⇨  "Her"

-- search for the second occurrence of "her" as a whole word
global sc2 := searchIndex strIndex "her" sc1 true
```

See the entry in *ScriptX Class Reference* for examples of how to search for multiple occurrences of a match in one string or in many strings. The string you want to match must be at least three characters long and cannot contain white space. This means that `searchIndex` will search for only one word.

Searching with Collection Methods

- `findRange` allows you to search for any number of words or characters as a string, but you can search for only the first occurrence of that string. The return value is the ordinal position of the first character of the match, or 0 if there is no match.

```
global myString := "You might as well smile."
findRange myString "might as well"
⇨  5
findRange myString "might as well frown"
⇨  0
```

- Various collection methods can be used to access the nth character in a string. The simplest is the collection access construct ([ ]). Since a string is really a collection of Unicode values, the return value is an integer. The following code accesses the fifth element in `myString` and converts the resulting integer to a string by using the function `intToString` defined in the section “Strings as Collections” on page 286.

```
myString[5]
⇨  109
intToString 109
⇨  "m"
```

The following methods return the character at the position indicated in their names: `getFirst`, `getSecond`, `getThird`, `getFourth`, `getFifth`, `getNth`, `getMiddle`, `getLast`.

```
-- find the middle character (Unicode value) and convert it to a string
getMiddle myString
⇨  115
intToString 115
"s"
```

- `getKeyOne` and `getOrdOne` allow you to get the ordinal position of the first occurrence of a character. The first occurrence is returned because strings inherit from `LinearCollection`, which means that `getKeyOne` and `getOrdOne` process items in order. If the character does not occur in the string, the return value for `getKeyOne` is `empty`; `getOrdOne` returns `0` if the character does not occur.

```
global s := "Where have all the flowers gone?"
getKeyOne s "e"[1]
⇨  3
getKeyOne s "x"[1]
⇨  empty

getOrdOne s "e"[1]
⇨  3
getOrdOne s "z"[1]
⇨  0
```

Using Text Actions

A text action is defined as a "hot spot" within presented text that, when clicked with the mouse, performs some operation. The ability to set hot spots and execute a particular function when they are clicked is integrated into the `Text`, `TextPresenter`, and `TextEdit` classes.

The `TextPresenter` class defines an instance variable, `enabled`, which determines whether or not the presenter accepts text actions. If `enabled` is `true`, when the presenter receives a mouse down event, it checks for a value of the `@action` attribute at that cursor position within the target text.

The `@action` attribute, defined only on instances of `Text`, contains the actual function or method that is executed when the action hot spot is selected. A target can have as many actions over as many ranges of text as are necessary. Because only instances of class `Text` can have attributes, presenters that use `String` or `StringConstant` objects as their targets cannot execute actions.

The function or method the `@action` attribute contains is passed three arguments when the action is activated (and therefore, that function must have been defined to take these three arguments):

- The `TextPresenter` (or `TextEdit`) object that was clicked on
- The range of characters currently being presented by the presenter

- The offset from the beginning of the text that received the mouse click

You do not have to use any of these arguments in the body of your function.

Here is a simple example of how to create a text action. It assumes that you already have a window available on which to display this text presenter (see Chapter 3, "Spaces and Presenters" for more information).

First, define the function that determines what happens when the action is clicked. In this example, the function `playSnd` causes an imported sound file contained in the global variable `theSound` to start playing:

```
function playsnd textPres rng offset -> (
    play theSound
)
```

Note that although the function did not use any of its three arguments, it has to be defined to include them.

Now, create the text presenter that will display the target text (contained in the global variable `theText`). Also, set the `enabled` instance variable to `true` in the text presenter, so that that presenter can accept mouse clicks:

```
global tp := new TextPresenter \
    boundary:(new Rect x2:300 y2:300) \
    target:theText
tp.enabled := true
```

Now that you have a text presenter, append it to the space so that it can be displayed. Here, the space is contained in the variable `theSpace`:

```
append theSpace tp
```

Finally, create the actual "hot spot" in the target text and link the `playSnd` function to it by using the `setAttrFromTo` method. This method assigns the `@action` attribute to a range of text (here, cursor positions 50 to 59):

```
setAttrFromTo theText @action 50 59 playSnd
```

When you click at the appropriate spot in the text, the `playSnd` function is activated and the sound is played.

CHAPTER

Document Templates

13

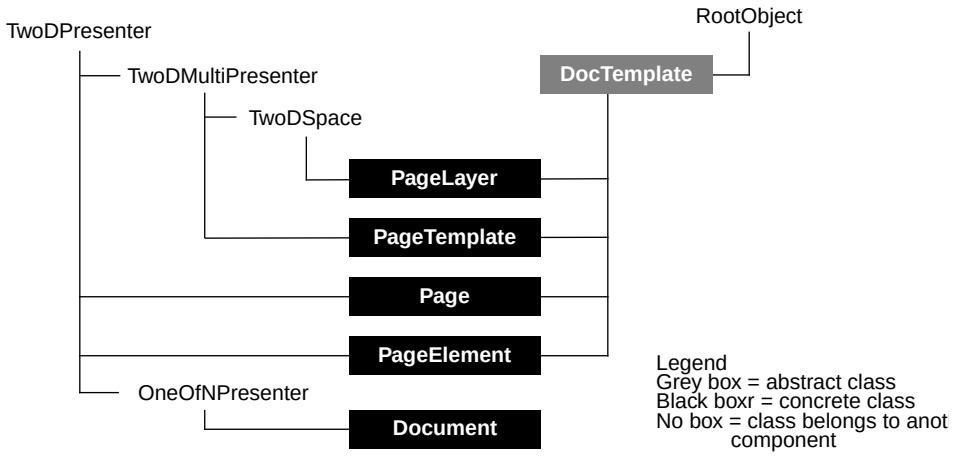
The Document Templates component is a set of classes for implementing on-screen, multi-media documents. These documents consist of one or more pages, where each page can use one or more layouts to present information. The presented information itself can be dynamically determined each time a page is displayed. Documents can incorporate all types of media in their layouts—text, images, animation, audio, and video.

The document template classes facilitate the construction of titles (or parts of titles) using a document metaphor. Some titles constructed using this metaphor may be similar to traditional paper documents, consisting of a sequence of pages, where each page displays certain information. You could also create "virtual documents" where the contents of a page change each time the page is displayed, depending, for example, on choices you made in a previous page.

One example of the use the Document Templates component is to create titles that use a card metaphor, consisting of a stack of cards that you can navigate through by using pushbuttons, menus, and active links. Each card in the stack would be a page in a document, and could display text, pictures, movies, pushbuttons, and animations.

Classes and Inheritance

The class inheritance hierarchy for the Document Templates component is shown in the following figure.



The following classes form the Document Templates component. In this list, indentation indicates inheritance.

`Document` – a collection of pages.

`DocTemplate` – an abstract class that is a superclass of `Page`, `PageElement`, `PageLayer`, and `PageTemplate`, allowing them all to inherit the `getParentData` and `findParent` methods.

- `PageElement` – represents a single visual element within the final, rendered page.
- `PageLayer` – a collection of `PageElement` objects and presenter objects. A page layer defines the design of a layer of information for a page.
- `PageTemplate` – a collection of `PageLayer` objects. A page template contains one or more design layers for a page.
- `Page` – a page in a document. The appearance of the page is determined by the page template or page layers that it uses. A document displays one `Page` at a time, much like one page in a book is open at a time.

Conceptual Overview

The Document Templates component provides classes that allow you to build documents that contain pages. The layout or design of a page can be separated from the data or objects to be laid out on the page. This separation of data and design allows you to use the same design for multiple pages that present different information but have a common layout. Alternatively, the same data can be presented with several different layouts, giving the user different views of the data.

The separation of data and design makes it possible to revise either the data or the layout independently. In a well-designed title, data can be maintained separately, and it can be modified without requiring changes in coding or design of the page. Revise the layout and you can present the same old data with a brand new look.

Figure 13-1: Three pages using the same design to present different information.

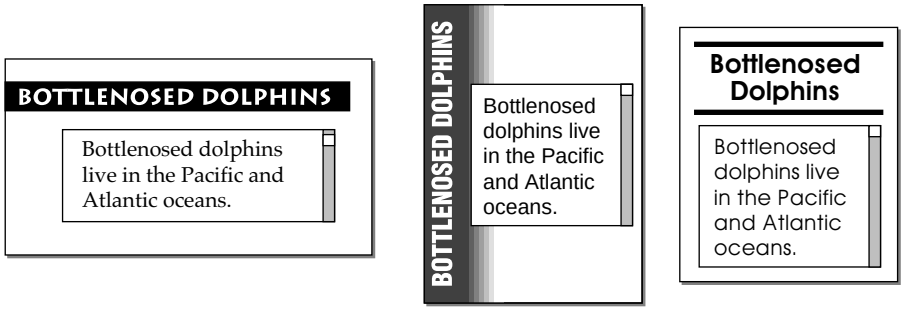


Figure 13-2: Three pages using different designs to present the same information.

Each page in a document can have multiple layers of design. Each layer can contain multiple design elements.

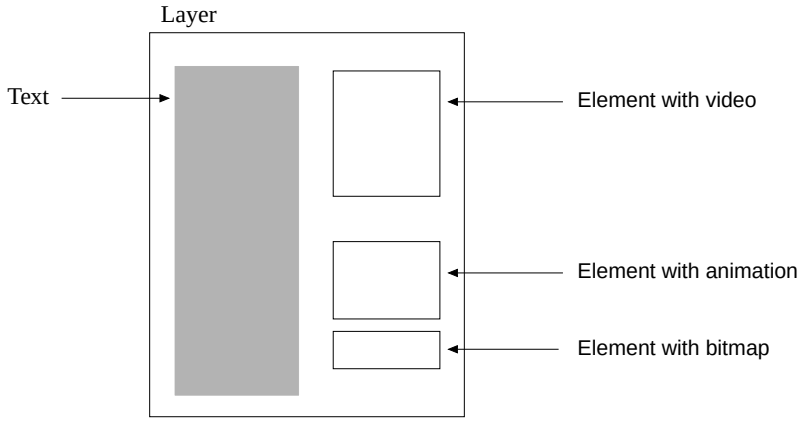


Figure 13-3: A page layer

How Document Templates Work

The Document Templates component provides classes for building documents consisting of pages that contain layers of design elements. These classes are `Document`, `Page`, `PageTemplate`, `PageLayer`, and `PageElement`.

Documents

A document is a container for pages. When a `Document` object is first created, it is an empty array. After creating a document, you can append `Page` objects to it.

A `Document` object can be thought of as containing and organizing the pages of the document. It can have a label and a bitmap that can be used to represent the document. The `Document` object can also contain references to the raw data that the document will present.

To display a document, it must be appended to a visible space, such as a window.

Pages

A `Document` object can display one `Page` object at a time. A page uses one or more layers that define its layout. Each page layer contributes elements to the design of the page.

A very simple page might have a single page layer that specifies a black rectangular border for the page, and a rectangular text presenter that displays text.

A more complicated page might use several page layers—one that defines static background design, one that defines header and footer design, one that defines the page number design, and one that defines the design for the text and images to be displayed on that page.

Each `Page` object has a `frame` instance variable, whose value determines the design of the page. The value is either a single `PageLayer` object, or a `PageTemplate` object, which is a collection of `PageLayer` objects. If a page uses multiple layers, its frame is a `PageTemplate` that contains the layers. If the page uses a single layer, its frame is a single `PageLayer` object.

A page can contain references to the information it displays. See “How Does a Page Element Know What to Present?” on page 323 for more information on storing data on a page.

PageTemplates and PageLayers

A page template holds one or more page layers in front-to-back order. A page layer acts as a layer of design containing the elements that a user actually sees on the page.

Pages can share the same set of design elements, such as background art or a set of controls, by using the same set of page layers. For example, in an electronic version of a paper document, each page would use a page template containing multiple page layers. Each layer would contain one or more design elements that define some aspect of the page. Certain layers, such as running heads, folios, or chapter titles, might be common to many pages, while others might be unique to a single page. A page template is very similar to a single master page in Quark XPress® or FrameMaker®, and to a layer in Adobe Illustrator®.

Page Elements

A page layer contains one or more design elements, represented by `PageElement` and `Presenter` objects. If the content or appearance of the design element varies from one page to another, (such as a variable footer) the element can be represented by a presenter embedded in a `PageElement`

object. If the content of the design element never varies from one page to another (such as a company logo), it can be represented directly by a `Presenter` object.

`PageElement` objects contain layout and content information for text boxes, image boxes, and other presenters that render information on a page. Each page element defines a particular presenter that appears on the page.

Each `PageElement` object has a `presenter` instance variable that specifies the presenter for that element, and a `target` instance variable that determines the target (or data) for the presenter. For example, if a page element's presenter is a `TextPresenter` object, its target would be the string (or an expression that returns the string) to be presented by the text presenter.

When creating a `PageElement` instance, you specify its presenter. This presenter is then permanently associated with that page element. For example, when creating a `PageElement` to display text, you would specify its presenter as a `TextPresenter` object. When creating a `PageElement` to display a bitmap, you would specify its presenter as a `TwoDShape` object.

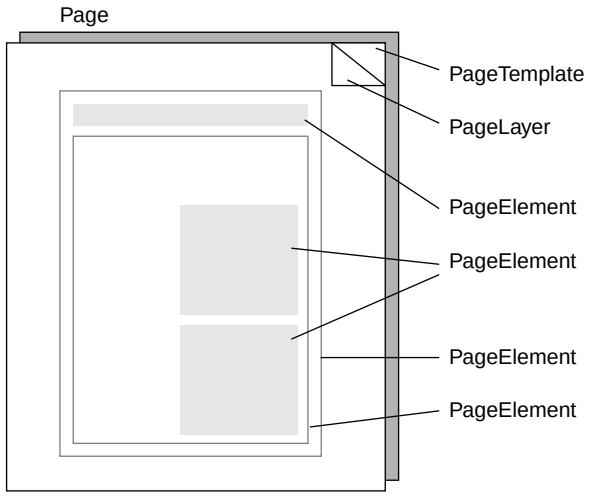


Figure 13-4: A sample page.

When creating a page element, you can supply either a specific target for the presenter, or an expression that gets evaluated when the page is displayed. For example, a presenter that shows the page number of a page must re-evaluate its `target` instance variable each time the page is displayed. In this case, the value in the page element's `presenter` instance variable would be a text presenter, and the value in its its `target` instance variable would be an expression that evaluates to the current page.

You can place presenters directly on a page layer when the the presenter's target does not change from page to page. For example, if you want a company logo to appear on each page, add a `TwoDShape` object whose target is a bitmap of the logo to the page layer. The logo does not change, no matter what page it is on. However, if you want a page to display an icon whose image depends on the page being displayed, add a page element containing a `TwoDShape` object to the page layer. When a page is displayed, the target for the `TwoDShape` object is dynamically evaluated.

Note – This discussion simplifies the hierarchy of templates that make up the page. It describes how the `PageTemplate`, `PageLayer`, and `PageElement` classes can be used to create an electronic counterpart of a conventional document. But ScriptX actually has few restrictions. It is possible for a `PageTemplate` to contain another `PageTemplate`, for a `PageLayer` to hold `PageTemplate` objects or other `PageLayer` objects. A title developer is free to combine these templates in unconventional ways—to reuse elements or even to extend the document metaphor.

Pages Share Templates

For multiple pages that use the same template (stored in the `frame` instance variable of the page) , multiple `Page` objects are created, one for each page in the document. In this case, only a single template object (be it a `PageTemplate` or `PageLayer`) is used, and hence only one instance of each page layer, page element and presenter used in the template is needed.

For example, suppose twenty pages in a document use the same template. Each page is represented by a separate `Page` instance, so there are twenty `Page` objects. However, there is only one `PageTemplate` instance, and there is only one instance of each presenter and page element used by the template.

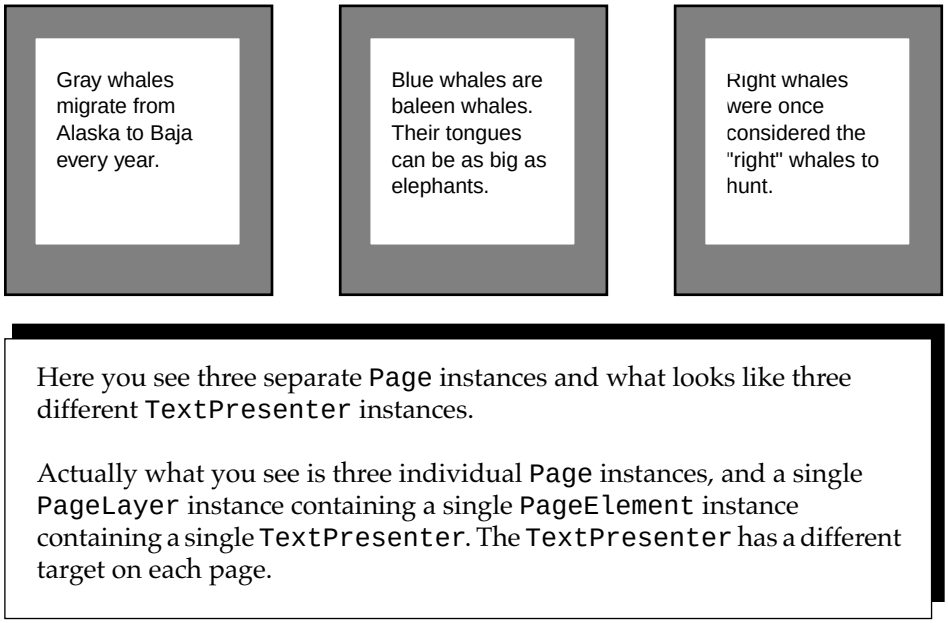


Figure 13-5: Multiple Page objects can share a single template

How Does a Page Element Know What to Present?

When a page is displayed, all the presenters (whether they are embedded inside page elements or not) on all the layers used by the page are displayed.

Each time a page is displayed, the system evaluates the expression in the `target` instance variable of each page element on that page to determine the target (or data or content). The resulting data is presented by the presenter associated with the page element.

The value of the `target` instance variable can be a hardwired value (such as a string constant, a number, a specific bitmap and so on) or it can be an expression that references data elsewhere.

For example, in a document that displays glossary definitions, the target of a page element could be an expression that evaluates to the text contained in the `definition` instance variable of the *n*th `Glossary` instance in a list, where *n* is the current page number.

Where Should the Data be Stored?

Sometimes it makes sense to store data, or expressions that generate the data, for a document on the document itself. For example, suppose you want every page to show the date and time the document was last opened. In this case, you could have an instance variable on the document, such as `DateLastOpened`, that contained the date and time the document was most recently opened.

In other cases it makes sense to store the data, or expression that generate the data, on the page. For example, if a document displays different text on each page, then each page could store the text that it displays.

And finally, in cases where a design element always displays the same data (such as company logos or string constants) it makes sense to store the data directly on the page element. However, in these cases, it makes more sense to use the presenter directly instead of embedding it inside a page element.

You can use the `target` instance variable on a `Document`, `Page`, `PageTemplate`, `PageLayer`, or `PageElement` object to store raw data or to store an expression that retrieves or generates data. Depending on the structure of the document, a page element might figure out what its target is by referring to the `target` instance variable of the page or document that contains it. (A page element could also refer to data stored on the page layer or page template that contains it but such situations are less common.)

The `target` instance variable of a page or a document (or also a page layer or page template) is evaluated if a page element referring to it is rendered when a page opens. If the `target` instance variable of a page or a document is not referred to by any page elements, it is never evaluated (in which case there's no point in giving it a value.)

What Should the Target Instance Variable Contain?

The value of a page element's `target` instance variable should be a suitable target for the page element's presenter, or an expression that returns such a target. For example, if the page element's presenter is a `TwoDShape`, then the value of the `target` instance variable should be a stencil or an expression that returns a stencil. For example:

```
element1.target := (new rect x2:200 y2:200)
element1.target := (self -> getNextStencil (stencilList))
```

The value of the `target` instance variable of a document or page can be a single value, an array of values, or an expression that returns a single value or an array of values.

```
page1.target := "Spam"
page2.target := (self - > pickRandomSlogan(55))
page3.target := #("red", "green", blue")
page4.target := (self -> #(whitefn(self), pinkfn(self), yellowfn(self))
```

If the value of the `target` instance variable of a page element, page layer, page template, page or document is an anonymous function, it takes a single argument of the object in question (often referred to as *self*).

getParentData

All classes that inherit from `DocTemplate`, including `Page`, `PageElement`, `PageLayer` and `PageTemplate` have a method called `getParentData`, that can be extremely useful for evaluating the target data for a page element. The `getParentData` method gets the value returned by the expression in the `target` instance variable on the specified parent class, which can be `PageLayer`, `PageTemplate`, `Page` or `Document`.

For example:

```
page1.target := "First Wonderful Page"
page2.target := "Second Excellent Page"
page3.target := "Third Incredible Page"

labelElement := new PageElement \
          presenter:(new TextPresenter \
                          boundary:(new rect x2:200 y2:200)
                          target:" " -- required here but ignored
          target:(self -> getParentData self Page)
```

When a page containing `labelElement` is displayed, `labelElement` gets the value returned by the expression in the `target` instance variable on the current page. Thus when `page1` is displayed, `labelElement`'s target is "First Wonderful Page"; when `page2` is displayed, it is "Second Excellent Page" and for `page3` it is "Third Incredible Page."

For another example of the use of `getParentData`:

```
doc1 := new Document
doc1.target := "Tales of Sinbad"

bookTitleElement := new PageElement \
          presenter:(new TextPresenter \
                              boundary:(new rect x2:200 y2:200)
                              target:" " -- required here but ignored
          target:(self -> getParentData self Document)
```

In this case, when a page containing the the element `bookTitleElement` is displayed, `bookTitleElement` gets the value returned by the expression in the `target` instance variable of the document. For document `doc1`, it will be "Tales of Sinbad."

Multiple Page Elements can get Data from a Page or Document

If the `target` instance variable of a document or page holds data for multiple page elements, then its value should be a collection of values or an expression that returns such a collection. In this case, each page element should know which element in the collection it is interested in. For example:

```
page3.target := #("Mark", "Plumber", 4155556666)
page4.target := (self -> #( self.person.name, \
                            self.person.occupation, \
                            self.person.phone))

nameElement.target := (self - > (getFirst getParentData self Page))
occuElement.target := (self -> (getSecond getParentData self Page))
phonElement.target := (self -> (getThird getParentData self Page))
```

Storing Data Independently from the Document

Often it is a good idea to store the data to be displayed in a document on a completely different set of objects than the objects that make up the document. For example, if you had a clothing database, you could use a set of objects to represent the items in the database, and then create a document to display information about the clothing objects. If the information in the database changes, each page automatically gets the latest information when it opens.

The following code shows an example of creating objects to hold data about clothing items, and creating pages that get data from the clothing items. Each page can display four page elements: one that displays the code for the clothing item, one that displays the price, one the color and one the description. For a more complete example, see “An Extended Document” on page 337.

```
class ClothingItem (RootObject)
instance variables
    code
    price
    color
    description
end
```

```
class ClothingPage (Page)
instance variables
    clothingItem
end

dress1 := new clothingItem
dress1.code := "DressA1"
dress1.price := 20
dress1.color := "Red"
dress1.description := "Flowing knee-length cotton dress"

dress2 := new clothingItem
dress2.code := "DressB2"
dress2.price := 25
dress2.color := "Blue"
dress2.description := "Easy care crepe dress with belt"

-- clothingTemplate is a predefined PageTemplate
page1 := new ClothingPage \
              frame:clothingTemplate \
              boundary:clothingTemplate.boundary
page1.clothingItem := dress1
page1.target := (self -> #( self.clothingItem.code, \
                            self.clothingItem.price, \
                            self.clothingItem.color, \
                            self.clothingItem.description)

page2 := new ClothingPage \
              frame:clothingTemplate \
              boundary:clothingTemplate.boundary
page2.clothingItem := dress2
page2.target := (self -> #(self.clothingItem.code, \
                            self.clothingItem.price, \
                            self.clothingItem.color, \
                            self.clothingItem.description)
```

Boundaries of Documents, Templates, Layers, and Pages

The pages within a document do not need to all be the same size and shape. Each page determines its own size and shape through the value in its `boundary` instance variable. When creating a new page, you should supply the `boundary` keyword argument to the `new` method. If a page does not have a suitable value in its `boundary` instance variable, that page may appear blank when displayed.

The boundary of a document is always the boundary of the current page. The `new` method for the class `Document` optionally takes a `boundary` keyword argument by virtue of the fact that `Document` is a subclass of `TwoDPresenter`. However, for the class `Document`, the `boundary` keyword argument is neither necessary nor useful. If it is supplied it has no effect.

For the `new` methods for the classes `PageTemplate` and `PageLayer`, the `boundary` keyword argument is necessary. If you do not supply a boundary for a page layer, the page layer will never be visible. If you do not supply a boundary for a page template, pages that use that template may not be visible.

If the boundary of a page layer extends beyond the boundary of the template containing the layer or the page displaying the layer, the presenters in the excluded area will be clipped (that is, they will not be visible).

A good rule of thumb for simple documents is to ensure that a page has exactly the same boundary as the page layer or page template in its `frame` instance variable. Also ensure that all layers in the page are either the same size as, or are completely contained by, the boundary of the page.

The following code creates `page1`, which uses the same boundary as its frame.

```
-- template1 is a predefined page template
page1 := new page frame:template1 boundary:template1.boundary
```

Fills and Outlines for Pages and Page Layers

Page layers and page templates have `fill` and `stroke` instance variables, whose values determine the background color and outline color respectively of the layer or the template. Pages do not have `fill` and `stroke` instance variables.

To give a page layer a visible border or fill, set its `stroke` or `fill` instance variable to the desired brush, such as `blackBrush`. To give a page template a visible border or fill, set its `stroke` or `fill` instance variable to the desired brush. To give a page a visible border or fill, ensure that the page's template, or one of the layers in its template, has the same boundary as the page, and has a value in its `stroke` or `fill` instance variable as desired. Note that the fill of the template will only be visible if none of the layers have a fill; and the stroke of the templatew will only be visible if none of the layers has a stroke.

When a page opens, the fill and stroke of a page template are drawn before the fill and stroke of its page layers. The layers are drawn starting with the last one and ending with the first one. If a layer does not have a fill or stroke, the fill or stroke of the layer underneath is visible.

Using the Document Templates Component

This section discusses how to create a Document, and explores some of the issues of creating a document, such as how to add movies to a page.

Creating a Document

The following steps summarize the tasks involved in creating and displaying a document.

1. Create one `Presenter` or `PageElement` object for each design element to be displayed on a page. Create a `Presenter` object if the content of the element is to be the same on every page, and create a `PageElement` object if the content is to vary from page to page.

   Also set the `x` and `y` instance variables of each `Presenter` and `PageElement` objects to set its position within its page layer.

When creating a new page element, supply the `presenter` and a `target` keyword arguments. The `presenter` is the presenter that renders the displayed information, and the `target` is an expression that evaluates to the object to be rendered by the presenter when the page is displayed.

```
element1 := new PageElement \
    presenter :(new TwoDShape) \
    target: (e -> getfirst (getParentData e Page)) \
    x: 50
    y: 150

element2 := new PageElement \
    presenter :(new TextPresenter \
               boundary:(new rect x2:100 y2:50) \
               -- target is a required keyword arg for
               -- text presenter, although here it will be
               -- ignored since the target of the element
               -- overrides it
               target:"") \
    target: (e -> getsecond (getParentData e Page)) \
        x: 50
        y: 50
element3 := new TwoDshape target:LogoBitmap
```

2. Create one or more `PageLayer` objects. Each `PageLayer` represents a layer of design for a page.

   Append the `PageElement` and `Presenter` objects to a `PageLayer` object. (Each page element can be appended to only one page layer.)

   ```
   boundaryRect := new rect x2:200 y2:400
   layer1 := new PageLayer boundary:boundaryRect
   append layer1 element1
   append layer1 element2
   append layer1 element3
   ```

3. Optionally, create `PageTemplate` objects to contain multiple `PageLayer` objects. Note that a single `PageLayer` can be appended to multiple `PageTemplate` objects.

   ```
   template1 := new PageTemplate boundary:boundaryRect
   append template1 layer1
   append template1 layer2
   ```

4. Create a `Document` object.

   Depending on the needs of the pages and page elements used in the document, you may need to supply an object or expression for the `target` instance variable on the document.

   ```
   doc1 := new Document
   ```

5. Append `Page` objects to the document.

For each page, specify a value for the `frame` keyword argument, which can be a `PageTemplate` or a `PageLayer` object. Also give a value for the `boundary` keyword argument that determines the shape and size of the page.

Depending on the needs of the page elements for the page, you may need to supply an object or expression for the `target` instance variable on the page. The following example uses a subclass of `Page`, called `MyPageSubclass`, that has an instance variable `DataObject` which points to an object that stores data.

```
--MyPageSubclass is a subclass of Page
-- that has a dataObject instance variable
append doc1 (new MyPageSubclass boundary:boundaryRect \
              frame:template1 \
              target:(p -> #(p.DataObject.Picture,\
                             p.DataObject.Heading))
```

6. Append the document to an open window to view it. Use the `goto` method to view a page.

```
w := new window
show w
append w doc1
-- Display the first page of the document
goto doc1 1
```

Navigating Through a Document

After a document has been created and opened in a visible space, you can navigate through it using its `goto`, `forward` and `backward` methods.

To display the next or previous `Page` object in a document, call the document's `forward` or `backward` method respectively. To display a specific page, call the document's `goto` method.

Specifying Things to Happen when a Page Changes

When you call the `goto`, `forward` or `backward` method on a document, the current page closes and another page opens. Each time a page opens, the document calls the `changePage` method on the page to be opened, which calls `changePage` on its page template, which calls `changePage` on all its page layers, which each call `changePage` on all its page elements.

As part of the `changePage` process, each page element evaluates the expression in its `target` instance variable to find what it should be presenting. If the `target` instance variable contains a call to the `getParentData` method, then the expression in the `target` instance variable of the appropriate parent is evaluated.

Finally the page opens, and each page element on the page presents the object returned by the expression in its `target` instance variable.

You may want to modify the actions that occur when a page opens. For example, you might want a sound or movie to start playing automatically when a page opens.

To modify the behavior when a page opens, create a subclass of `Page` and specialize its `changePage` method. Be sure that the customized definition for `changePage` includes a call to `nextmethod`, since the inherited definition takes care of evaluating and updating the targets for all the page element objects on the page. The `changePage` method takes two arguments: *self* and *newpage*.

For example, the following code specializes the `changePage` method for a class called `MusicalPage` so that when the page opens, a tune starts playing.

```
class MusicalPage (Page)
instance variables
    tune
end

method changePage self {class MusicalPage} newpage ->
(   -- call nextmethod to get default behavior
    nextMethod self newpage

    local mytune := self.tune
    gotobegin mytune
    playPrepare mytune 1.0
    playUntil mytune mytune.duration
)
```

Similarly, to customize the behavior of individual `PageElement` objects when a page opens, create subclasses of `PageElement` and customize their `changePage` method, being sure to include a call to `nextmethod`.

Finding the Presenter of Objects in a Document

When defining the expression to determine the target data for a `PageElement`, it can be useful to refer directly to the `Page` or `Document` containing the `PageElement`.

You can use the `findParent` method of an instance of `DocTemplate` to find which instance of a specified class is ultimately presenting it. For example, you can find which document is presenting a page element; which document is presenting a page; which page is presenting a page element, and so on. In this context, the "parent" of an object in a document hierarchy is the object that is presenting it (or contains it).

For example, to find the `Page` of a `PageElement`, call the `findParent` method on the `pageElement` instance and specify `Page` as the presenter class to look for.

```
p := findParent pageElement Page
```

To find the document presenting a page element:

```
d := findParent pageElement Document
```

To find the document presenting a page:

```
d := findParent page Document
```

To find the page containing a pushbutton:

```
p := findParent pushbutton.presentedBy Page
```

Since a `PushButton` object does not inherit from `DocTemplate`, it does not have a `findParent` method. However, if the pushbutton is embedded in a page element, its `presentedBy` instance variable points to the page element, which does have a `findParent` method. If the pushbutton has been directly added to a page layer, its `presentedBy` instance variable points to a page layer object (which also has a `presentedBy` method).

For another example, suppose a page element is intended to display the current page. In this case, the value in the `presenter` instance variable of the page element would be a text presenter. The expression in the `target` instance variable of the page element would be an anonymous function that finds the value of the `cursor` instance variable of the document that contains the page element. This anonymous function could be:

```
( self ->
    local thisdoc := findParent self Document
    thisdoc.cursor
)
```

Using Controllers in a Document

You may want to add pushbuttons or other actuators to a page to control an element of the page. For example, you might want a page containing an `InterleavedMoviePlayer` object to have a pushbutton that can be used to start and stop the movie.

When using a pushbutton in a document, if the button's appearance does not change from page to page, you can simply add the button to a page layer. If the button's appearance changes from page to page, put it in a page element and add that element to the page layer.

Whether you add a button (or any other actuator) directly to a page layer, or first put it inside a page element, you also need to create an `ActuatorController` to control the layer. To do this, create an actuator controller, and specify the page layer as the controller's space. (See Chapter 4, "Controllers" for a discussion of using controllers to control actuators.) The following code gives an example of using a button on a page layer.

```
-- create a page layer
pageLayer1 := new PageLayer boundary:(new rect x2:200 y2:400)

-- create an actuator controller for the page layer
controller1 := new ActuatorController space:pageLayer1

-- create a pushbutton
```

```
pushbutton1 := new pushbutton releasedpresenter:upIcon \
                                pressedpresenter:downIcon

-- Place the pushbutton directly on the page layer
append pageLayer1 pushbutton1

-- Alternatively put the pushbutton inside a page element
-- on the page layer
pageElement1 := new pageElement presenter:pushbutton1
append pageLayer1 pageElement1
```

You can call a method on a presenter embedded in a page element by calling the method on the page element itself. So if you want a button's `activateAction` method to call a method on another presenter that is embedded in a page element on the same page, you can define the `activateAction` to call the method on the page element, which will pass it on to its presenter.

For details on how to make a pushbutton change its appearance from page to page, see "Dynamically Updating Presenters that Don't Use Targets" on page 334.

Displaying Movies on a Page

This section gives some hints on playing movies on a page in a document.

Pages in a document can contain either `MoviePlayer` or an `InterleavedMoviePlayer` instances. However, it is better to use interleaved movie players, since they play back much more smoothly from a CD than do movie players. (The use of `MoviePlayer` objects instead of `InterleavedMoviePlayer` objects is very much discouraged in all circumstances.)

However, an `InterleavedMoviePlayer` cannot change which movie it plays from page to page. An `InterleavedMoviePlayer` always plays the same movie, due to the complexity of passing deinterleaved data from the player's interleaved stream to the player's slave players.

Therefore, to create page layers that contain movies that can change from page to page, you need to take a couple of extra steps. You can use a `GroupSpace` as a page element's presenter, and add the movie to the group space. Define the `target` setter method for the group space to append the given value to the group space.

When the page changes, the page element figures out which movie to play, and tries to put that value in the group space' s `target` instance variable. However, the `targetSetter` method intercepts the target data, and adds it to the group space instead.

This specific steps are:

1. Create a subclass of `GroupSpace`, called say, `MovieGroupSpace`.

```
class MovieGroupSpace (GroupSpace)
end
```

2. Define `target` setter and getter methods for the subclass. The setter method empties out the group space, and adds the given value to it. The getter method returns the first item in the space, which will be the interleaved movie player.

```
method targetSetter self {class MovieGroupSpace} value ->
(   emptyout self
    prepend self value
)

method targetGetter self {class MovieGroupSpace} ->
(self[1] )
```

3. Create a page element whose `presenter` is an instance of the `MovieGroupSpace` subclass. Define the `target` expression to return an `InterleavedMoviePlayer` appropriate to the current page.

```
movieElement := new pageElement \
    presenter: (new MovieGroupSpace) \
    target: (e -> getMovieForThisPageFn e)
```

The advanced example later in this chapter gives a complete example of how to use movies in a document.

Playing the Movie

The purpose of displaying a movie on a page is to be able to play it when the page is open. The movie could perhaps start playing automatically as soon as the page opens, or it could be played under user control through the use of buttons on the page.

See “Displaying Movies on a Page” on page 332 for a coded example of how to use pushbuttons to play and stop a movie on a page.

To make the movie start playing automatically when the page opens, create a subclass of `Page` and modify the `changePage` method on the new subclass. Don’t forget to call `nextMethod` in the definition for `changePage`, since the inherited definition takes care of updating the information for all the page elements on the page.

The following code defines a `changePage` method for a subclass of `PageElement`, so that when the page opens, the movie starts playing automatically. This example assumes that the movie is the first item in a group space which has been embedded in a page element, as discussed above.

```
class MoviePageElement (PageElement)
end

method changePage self {class MoviePageElement} newPage->
(
    -- do the default behavior
    nextMethod self newPage

    -- the page element’s presenter is a group space
```

```
    -- containing the interleaved movie player
    local movie := self.presenter[1]
    -- make sure the movie is stopped and rewound
    stop movie
    gotoBegin movie

    -- prepare the movie, then play it
    playPrepare movie 1.0
    playUnti movie movie.duration
)
```

Dynamically Updating Presenters that Don't Use Targets

This section discusses how to add presenters to a page so that they update each time a page opens, in situations where the presenter does not usually use its `target` instance variable.

For example, the classes `GroupSpace`, `ScrollingPresenter`, and `PushButton` inherit the `target` instance variable from the `Presenter` superclass, but they ignore it when deciding what to present. The previous section discussed how you can use a `GroupSpace` to present a movie in a page element. This section here looks at how you can use `ScrollingPresenter` and `PushButton` objects in a document, in such a way that they can change appearance from page to page. Although these classes are discussed as examples, the general theory discussed here applies to all kinds of objects.

This section does not explain how to use scroling presenters and pushbuttons. See Chapter 5, "User Interface" for information on these presenters.

Create a subclass of the desired presenter class and define a setter method for the `target` instance variable. This method should do something with the `target` value so that it helps determine the appearance of the presenter. Remember that the presenter's `target` instance variable will always be dynamically evaluated when the page element is displayed. (See the *ScriptX Language Guide* for information on using and defining setter and getter methods.)

Changing the Target of a Scrolling Presenter

Suppose you want to use a scrolling text presenter to display different text on different pages in a document. In this case, the `target` instance variable of each page in the document should return the desired text for that page.

Here you would create a subclass of `ScrollingPresenter` and define a `targetSetter` method that passes the input value on to the `target` instance variable of the scrolling presenter's `targetPresenter`. (Scrolling presenters let you scroll the presenter in their `targetPresenter` instance variable.)

You would then embed an instance of the specialized scrolling presenter class in a page element. When the page element is displayed on a page, it will automatically find the target data, and will try to put that data in the `target` instance variable of its presenter. However, the `targetSetter` method will intercept the target data, and pass it on to the text presenter instead. For example:

```
page1.target := ("Springer Spaniels are very friendly dogs although \
    they need a lot of exercise. They are very intelligent and \
    highly trainable, and usually get on very well with other \
    dogs. They come in colors of liver and white, or black and white. \
    They can also be tri-colored with liver, black and white. They \
    come in two main varieties: field and show. The field dogs tend \
    to be shorter and sturdier, and the show dogs tend to have \
    more feathery hair. One famous Springer Spaniel is Millie, a \
    previous resident of the White House.")

-- Make a subclass of scrollingpresenter

class ScrollingPresenterForPage (ScrollingPresenter)
end

-- Define the setter method for the scrollingpresenter's target iv
-- to pass on the text to the targetpresenter.

method targetsetter self {class ScrollingPresenterForPage} value -> \
    (self.targetPresenter.target := value)

-- Create an instance of scrollingPresenter
-- assume that scrollbar1 has already been defined as a scrollbar

ScrollingPresenter1 := new ScrollingPresenterForPage \
    vertScrollBar:scrollbar1 \
    boundary:(new Rect x2:450 y2:300) \
    targetPresenter:(new textPresenter \
              boundary:(new rect x2:400 y2:900))

-- Put the scrolling presenter in a page element
global scrollingElement := new pageElement \
           presenter:ScrollingPresenter1 \
           target:(self -> getParentData self Page)

-- add scrollingElement to a page layer and so on.
```

Changing a PushButton's Appearance from Page to Page

For another example, consider the case of pushbuttons. When a pushbutton is released, its appearance is determined by the presenter in its `releasedPresenter` instance variable. When it is pressed, its appearance is determined by the presenter in its `pressedPresenter` instance variable.

The following example illustrates how you would set up a pushbutton whose appearance can change from page to page in a document.

```
-- page1 and page2 are instances of a customized subclass of Page
-- that has upPresenter and downPresenter instance variables

page1.target := #(self.upPresenter, self.downPresenter)
page2.target := #(self.upPresenter, self.downPresenter)

class PushButtonForPage (PushButton)
end
```

```
method targetsetter self {class PushButtonForPage} valueList -> \
    (
        self.releasedPresenter := valueList[1]
        self.pressedPresenter := valueList[2]
    )


button1 := new PushbuttonForPage
button1.activateAction := (authordata button -> doSomething())

pushElement1 := new pageElement \
                  presenter:button1
                  target:(self getParentData self page)

-- add pushElement to a page layer and so on
```

Document Template Examples

This section presents two example scripts, one for creating a simple document and one for creating a document that uses several different layers of design.

A Simple Document

The following script demonstrates how to create a very simple document.

The document in this example has two pages, that both simply display their page number, which is "hardwired" into the `target` instance variable for the page. The frame for each page is a single page layer.

(The next example shows how to dynamically calculate the page number on the fly.)

```
-- Create the boundary for the pages
pageRect := new Rect x2:300 y2:360

-- Create a line to appear across the top of each page.
theLine := new twoDshape boundary:(new line x1:0 x2:300 y1:0 y2:0)
theLine.stroke := new brush color:redcolor

-- Position the line near the top of the page
theLine.y := 50

-- Create a text box to hold the page number appear on each page.
textBounds := new Rect x2:50 y2:20
pageNumberBox := new TextPresenter boundary:textBounds \
    target:("" as Text)
pageNumberElement := new PageElement \
    presenter:pageNumberBox target:(e -> getParentData e Page)

-- Center the page number element near the bottom of the page
pageNumberElement.x := 250
pageNumberElement.y := 30
```

```
-- Create a page layer and make its outline be blue
layer1 := new PageLayer boundary:pageRect
layer1.stroke := new brush color:bluecolor

-- append the line and the page number box to the page layer
append layer1 pageNumberElement
append layer1 theLine

-- Create a document and append two pages to it
theDocument := new Document

page1 := new Page boundary:pageRect \
    frame:layer1 target:("Page 1" as text)

page2 := new Page boundary:pageRect \
    frame:layer1 target:("Page 2" as text)

append theDocument page1
append theDocument page2

-- Create a window that is slightly bigger than the first
-- page in the document.

w := new window
w.fill := whitebrush
w.height := page1.boundary.height + 50
w.width := page1.boundary.width + 50

show w

-- Append the document to the window.
-- Center the document in the window.
append w theDocument
theDocument.x := 25
theDocument.y := 25

-- Open the second page
goto theDocument 2
```

An Extended Document

The following example code creates a document with two alternative page layouts, one for displaying text and a picture, and one for displaying text and playing a movie. These two layouts, referred to as the Definition PageTemplate and the Movie PageTemplate, are illustrated in Figure 13-6. Code for this example is provided on the ScriptX CD in the `doctempl` folder.

This example shows how to combine `PageLayer` objects into `PageTemplate` objects, how to play movies in a document, and how to place active buttons on a page. It illustrates how to separate the data displayed in the document from the document design. The data is stored in other objects, so the document layout is completely separate from the data.

The document can be used to display the data stored in any object that conforms to the following criteria:

- It has a `heading` instance variable whose value is a `Text` object with a shortish string.
- It has a `description` instance variable whose value is a `Text` object.
- It has either a `picture` instance variable whose value is a `Bitmap` object, or a `movie` instance variable whose value is a suitable target for a movie player.
- It has a `copyright` instance variable whose value can be an empty string or can contain copyright information.

The example code shows how to create the page elements, page layers, and page templates needed for the document. It also provides a function that creates a document. When calling the function, you must supply a list of objects containing text and pictures, and a list of objects containing text and a movie target. The function creates an appropriate page in the document for each object in the input lists.

The PageLayers Used in the Document

The Definition page template and the Movie page template share page layers. The layers are described below.

The **background layer** contains:

- A page element for the heading.
- A page element for the bar on the right which is a pushbutton that moves the document forward a page.
- A page element for the bar on the left which is a pushbutton that moves the document back a page.
- A page element for the colored background box for the page number.
- A page element for the page number.

The **definition layer** contains:

- A page element for the text description.

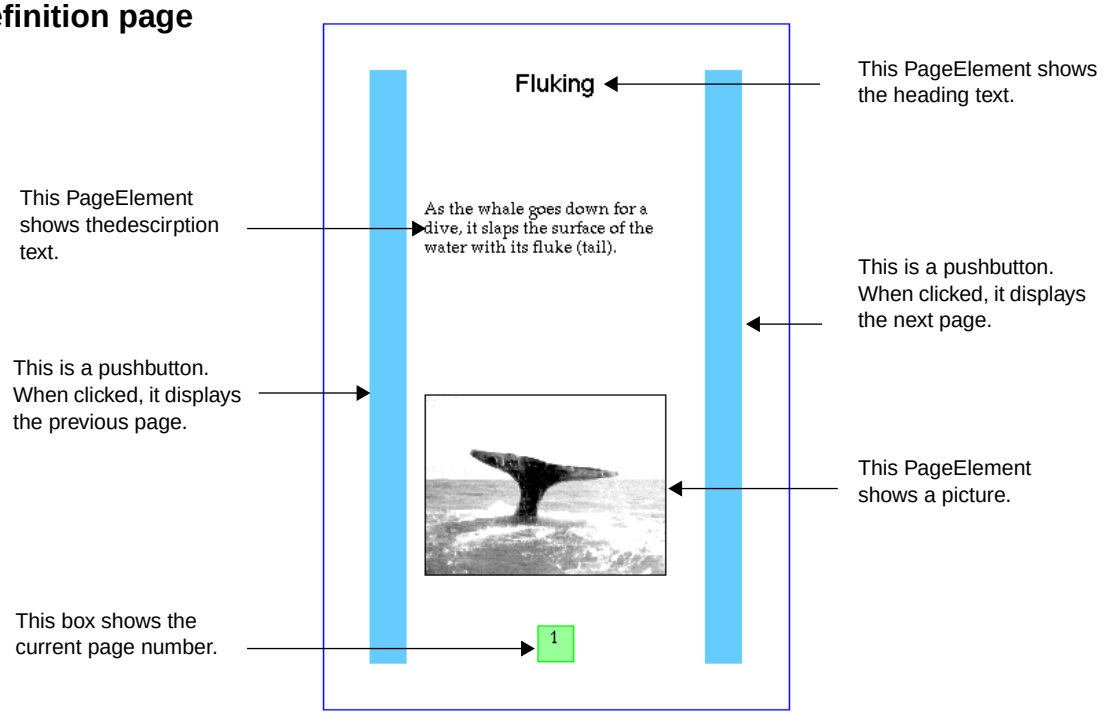
The **picture layer** contains:

- A page element for the picture.

The **movie layer** contains:

- `A` page element for the movie player that manages the movie.
- Pushbuttons that can be used to play and stop the movie.
- Two `TwoDShape` objects for two colored boxes to use as shadows for the buttons that control the movie.
- A page element containing a text presenter that displays copyright information.

Definition page



Movie Page

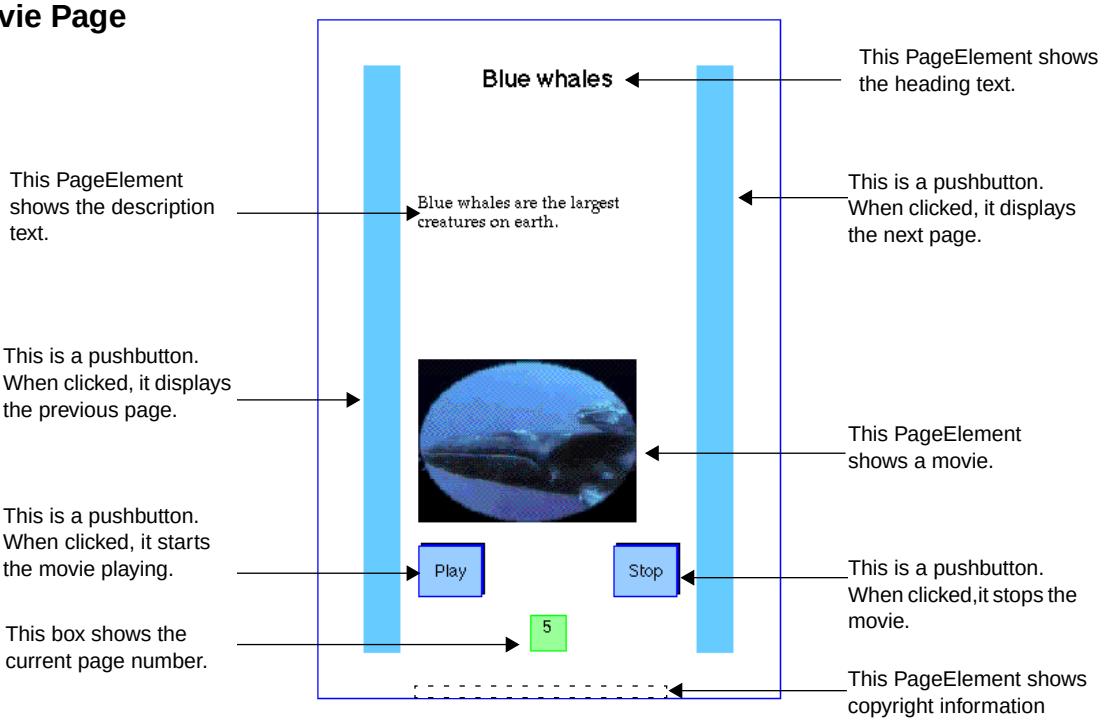


Figure 13-6: A sample Definiton page and Movie page

The Code for the Example

The remainder of this chapter discusses the code for the example. You can find the code in the `doctempl` folder on the ScriptX CD.

Define Some Customized Colors

The document uses some customized colors, so define them at the start of the script.

```
---<<<
-- Document Templates Example


-- Define some custom colors

global constant blueBrush := new brush color: blueColor
global constant skyBlueBrush := new brush \
    color: (new RGBColor red:80 green:180 blue:250)
global constant darkerBlueBrush := new brush \
    color:(new RGBColor red:20 green:120 blue:250)
global constant otherBlueBrush := new brush \
    color:(new RGBColor red:140 green:180 blue:250)
global constant greenBrush := new brush color: greenColor
global constant middleGreenBrush:= new brush \
    color:(new RGBColor red:160 green:250 blue:160)
```

Make the Background Layer

The background layer contains the elements that appear on every page. These include the pushbuttons at the left and right that move the document to the next and previous page, a text presenter for the page number, and a text presenter for the heading.

The pushbuttons that move the document backward or forward a page do not really need to be put inside `PageElement` instances. Although the `activateAction` function for each pushbutton is evaluated each time the pushbutton is pushed, the target of the pushbutton is always the same, (that is, the images used to display the button do not change from page to page.)

However, the pushbuttons for moving through the pages are put inside `PageElement` instances in this example, to show you how to do it. When adding `PageElement` instances containing pushbuttons or other objects controlled by a controller, you have to take additional steps to ensure that the controller works with the page element. The example illustrates these steps.

Further on in the example, you will see how to add pushbuttons to a page without putting the pushbuttons inside `PageElement` instances.

```
-- Make the background layer.
-- This layer contains the outline of the page, the bars
-- on the left and right that let you move through the document,
-- and also text presenters for the page heading and the page number.
-- pageWidth and pageHeight are the width and height of the page.

-- the margin is the distance between the edge of the page
-- and the back page/forward page buttons
-- the innerMargin is the margin at which contents other
-- than the back/forward buttons start

function makeMargin pageWidth -> (pageWidth * 0.1)
function makeInnerMargin pageWidth -> (pageWidth * 0.22)

function makeBackgroundLayer pageWidth pageHeight ->
(
    local margin       := makeMargin pageWidth
    local innerMargin := makeInnerMargin pageWidth

    -- the smallwidth is the width of the back and forward
    -- buttons and the pageNumber element
    local smallwidth  := (innermargin - margin) * 0.66

    local smallRect    := new rect x1:0 y1:0 x2:smallwidth y2:smallwidth

    -- Create a Text Presenter that holds the pagenumber
    local pageNumberText := (new TextPresenter \
           boundary: smallRect stroke:greenBrush \
           fill: middleGreenBrush \
           target: ("" as Text))
    pageNumberText.inset := new point x:0 y:2
    setDefaultAttr pageNumberText @font \
           (new PlatformFont name:"Palatino")
    setDefaultAttr pageNumberText @size 12
    setDefaultAttr pageNumberText @alignment  @center


    -- Find the page number.
    -- The document's cursor indicates
    -- the current position in the doc
    local pageNumberElement := new pageElement\
       presenter: pageNumberText \
       target: (e -> (findParent e Document).cursor as Text )
    pageNumberElement.x := (pageWidth - smallWidth) / 2
    pageNumberElement.y := pageHeight - (margin + smallWidth)


    -- Create the Text Presenter for the heading
    local headingText := (new TextPresenter \
           boundary: (new rect \
              x2:(pageWidth - (2 * innerMargin)) y2:35) \
           stroke: blackBrush target: ("" as Text) \
           fill: skyBlueBrush)
    headingText.inset := new point x:10 y:10
    setDefaultAttr headingText @font    \
           (new PlatformFont name:"Helvetica")
    setDefaultAttr headingText @size 18
```

```
setDefaultAttr headingText @alignment @center

local headingElement := new pageElement presenter: headingText \
       target: (e -> getfirst (getParentData e SpecialPage))
headingElement's x := innerMargin
headingElement's y := margin

-- Create the forward and backward page buttons
local buttonrect := new rect x2:smallwidth \
   y2:(pageHeight - (2 * margin))

local forwardUp := new twoDshape boundary: buttonRect \
       fill: skyBlueBrush stroke: otherBlueBrush
local forwardDown := new twoDshape boundary: buttonRect \
       fill: darkerBlueBrush

local backwardUp := new twoDshape boundary: buttonRect \
       fill: skyBlueBrush stroke: otherBlueBrush
local backwardDown := new twoDshape boundary: buttonRect \
       fill: darkerBlueBrush

local forwardButton := new pushbutton \
       releasedPresenter:forwardUp \
       pressedPresenter: forwardDown
local backwardButton := new pushbutton \
       releasedPresenter:backwardup \
       pressedPresenter: backwarddown

-- When a pushbutton is pressed, its activateAction function is called
-- with the arguments pushbutton.authordata and pushbutton.
-- Here we want to find the document containing the pushButton,
-- so call findParent on the object that is presenting the pushbutton

forwardButton.activateAction := \
       (a b -> forward (findParent (b.presentedBy) Document))

backwardButton.activateaction := \
       (a b -> backward (findParent (b.presentedBy) Document))

-- put the push buttons inside page elements
local forwardButtonElement := new pageelement \
       presenter: forwardButton
forwardButtonElement.x := (pageWidth - (margin + smallwidth))
forwardButtonElement.y := margin

local backwardButtonElement := new pageelement \
       presenter:backwardButton
backwardButtonElement.x := margin
backwardButtonElement.y := margin

-- Create a pagelayer that holds background information
local bkgdLayer := new pagelayer \
       boundary:(new rect x2:pageWidth y2:pageHeight)

bkgdLayer.stroke := blueBrush
append bkgdLayer pageNumberElement
append bkgdLayer forwardButtonElement
append bkgdLayer backwardButtonElement
append bkgdLayer headingElement
```

```
    -- Make an actuator controller to control
    -- the buttons in the background layer
    local docController := new actuatorController space:bkgdLayer

    append doccontroller forwardButtonElement
    append doccontroller backwardButtonElement

    -- return the background layer
    bkgdLayer
)
```

Make the Definition Layer

The definition layer contains a text presenter that displays a description that varies depending on the page displayed.

```
-- Make the definition layer, which contains text definition

function makeDefinitionLayer pageWidth pageHeight->
(
    local margin := makeMargin pageWidth
    local innerMargin := makeInnerMargin pageWidth

    -- Create the Text Presenter that presents the info
    local infoText := (new TextPresenter \
       boundary: (new rect x2: (pageWidth - (2 * innerMargin)) y2: 80) \
       stroke:blackbrush target:("" as Text))

    infoText.inset := new point x:10 y:10
    setDefaultAttr  infoText @font \
           (new PlatformFont name:"Palatino")
    setDefaultAttr infoText @size    12
    setDefaultAttr infoText @leading  14
    setDefaultAttr infoText @alignment  @flush

    -- put the info text presenter into a page element
    local infoElement := new pageElement presenter : infoText \
           target: (e -> getsecond (getParentData e SpecialPage))
    infoElement.x := innerMargin
    infoElement.y := innermargin + 15

    -- create a page layer
    local definitionLayer := new pagelayer \
       boundary:(new rect x2:pageWidth y2:pageHeight)
    append definitionLayer infoElement

    -- return the definition page layer
    definitionLayer
)
```

Make the Picture Layer

The picture layer displays a picture that varies from page to page.

```
-- Make the picture layer
function makePictureLayer pageWidth pageHeight ->
(
    local margin       := makeMargin pageWidth
    local innerMargin := makeInnerMargin pageWidth

    -- Create the box that displays the picture
    local picPresenter := \
           new TwoDShape fill:blackbrush stroke:blackbrush

    local picElement := \
        new pageElement presenter: picPresenter \
            target: (e -> getThird (getParentdata e SpecialPage))
    picElement.x := innerMargin
    picElement.y := 160 + margin

    -- Make a page layer for the picture
    local pictureLayer := new pagelayer \
           boundary:(new rect x2:pageWidth y2:pageHeight)

    -- add the picture element to the picture layer
    append pictureLayer picElement

    -- return the picture layer
    pictureLayer
)
```

Make Supporting Classes for the the Movies

This example uses `InterleavedMoviePlayer` objects to play movies. Since interleaved movie players cannot change their target, we use a group space to hold the interleaved movie player. However, the `GroupSpace` class does not use its `target` instance variable, so we need to do a little more work.This involves creating a subclass of `GroupSpace` (called `MovieGroupSpace`). Define the `target` setter and getter methods on the subclass to make the group space behave as if it switches its target from one interleaved movie player to another when a page changes.

```
class MovieGroupSpace (GroupSpace)
end

-- define the target setter for MovieGroupSpace
-- so that when you add a value to the target iv
-- it effectively adds the target to the group

method targetSetter self {class MovieGroupSpace} value ->
(   emptyout self
```

```
    prepend self value
 )


-- define the getter to return the
-- first element in the group space, which should always
-- be an interleaved movie player
method targetGetter self {class MovieGroupSpace} ->
(self[1] )
```

If you change a page while a movie on the page is playing, the movie should stop playing immediately. To implement this behaviour, the document needs to keep track of what movie is playing at any time. To this end, we define a subclass of `Document`, called `MyDocument`, that has a `currentMovie` instance variable.

```
class MyDocument (Document)
instance variables
    currentMovie
end
```

The value of the document's `currentMovie` instance variable always indicates which movie, if any, is currently playing.When a movie starts or stops playing, the value of this instance variable is updated as appropriate. When a page changes, if the document's `currentMovie` instance variable has a value, the movie is stopped and the value of `currentMovie` becomes undefined. (See page 349 for a definition of the appropriate `changePage` method.)

The user starts and stops the movie on a page by using pushbuttons. This example creats a subclass of `PushButton` that has a `movieElement` instance variable. Each pushbutton needs to know what movie player it is controlling, so its `movieElement` instance variable keeps track of that information. The `movieElement` instance variable holds the `PageElement` containing the movie group space. The interleaved movie player is always the first element in the movie group space.

The `playMovie` method plays the movie on the page, and the `stopMovie` method stops the movie. Both these methods illustrate how the page element acts as a proxy for its presenter. In the method definitions, `thisMovieElement` is a page element. However, the code:

```
thisMovieElement[1]
```

returns the first item in the page element's presenter, not the first item in the page element itself. (The page element is not even a collection.)

```
-- Make a class for the Pushbutton that controls the movie
class MoviePushbutton (PushButton)
    instance variables
        movieElement
end

-- Define the methods used by the movie button
method playMovie button {class MoviePushButton} ->
(
    -- the button's movieElement is a page element
    -- whose presenter is a movie group space

    -- the movie is the first item in the group space
    -- notice that we access the group space by accessing the page element
    local movie := thisMovieElement[1]

    stop movie
    gotobegin movie
    playprepare movie 1
    play movie

    -- keep track of which movie is currently playing
    local doc := findparent thisMovieElement.presentedBy Document
    doc.currentMovie := movie
)


method stopMovie button {class MoviePushButton} ->
(
    -- the button's movieElement is a page element
    -- whose presenter is a movie group space
    local thisMovieElement := button.movieElement

    -- the movie is the first element in the group space
    -- notice that we access the group space by accessing the page element
    local movie := thisMovieElement[1]

    stop movie
    gotobegin movie
    playprepare movie 1

    -- no movie is currently playing
    local doc := findparent thisMovieElement.presentedBy Document
    doc.currentMovie := undefined
)
```

Make the Movie Layer

The movie layer contains the elements and presenters for displaying a movie, and buttons for controlling the movie.

The user starts and stops the movie on a page by using pushbuttons. In this example, each pushbutton is added directly to a page layer without being embedded in a `PageElement` instance. (However, if the appearance of the

pushbutton depends on the page being displayed, you would need to put it in a `PageElement`.) Each button is an instance of `MoviePushButton`, which has an instance variable `movieElement` that contains the page element that provides access to the movie.

```
function makeMovieLayer pageWidth pageHeight ->
(
    local margin       := makeMargin pageWidth
    local innerMargin := makeInnerMargin pageWidth
    local buttonWidth  := 35
    local buttonHeight := buttonWidth * 0.75

    local movieElement := new pageElement \
       presenter: (new MovieGroupSpace) \
       target: (e -> (getNth (getParentdata e SpecialPage) 3))
    movieElement.x := innerMargin
    movieElement.y := 160 + margin

    -- Create the Movie Control Buttons
    local movieButtonRect := new rect x2: buttonWidth y2: buttonHeight

    local playUp := new textPresenter boundary: movieButtonRect \
       fill: otherBlueBrush stroke: BlueBrush \
       target: ("Play" as Text)
    playUp.inset := new point x:0 y:4
    setDefaultAttr playUp @font (new PlatformFont name:"Helvetica")
    setDefaultAttr playUp @alignment @center

    local playdown := new twoDshape \
           boundary:movieButtonRect fill:darkerBlueBrush

    -- Create the pushbutton to play the movie.
    local playButton:= new MoviePushbutton \
           releasedPresenter: playUp pressedPresenter: playDown
    playButton.movieElement := movieElement

    -- When a push button is pressed,
    -- its activateAction function is called with the arguments
    -- pushbutton.authorData and pushbutton.
    -- Here we want to call the playMovie function on the pushbutton

    playButton.activateAction := (a b -> playMovie b)

    playButton.x := innerMargin
    playButton.y := (pageHeight -  margin - playbutton.height )

    -- make the shadow for the playButton
    local playShadow := new twodshape fill: darkerBlueBrush \
           stroke: darkerBlueBrush target: movieButtonRect
    playShadow.x := playButton.x + 2
    playShadow.y := playButton.y - 2

    -- make the stop button
    local stopUp := new textPresenter boundary:movieButtonRect \
           fill: otherBlueBrush  stroke: BlueBrush  target: ("Stop" as Text)
```

```
    setDefaultAttr stopUp @font   (new PlatformFont name:"Helvetica")
    setDefaultAttr stopUp @alignment  @center
    stopUp.inset := new point x:0 y:4

    local stopDown := new twoDshape boundary: movieButtonRect \
           fill: darkerBlueBrush

    local stopButton := new MoviePushbutton \
           releasedPresenter: stopUp  pressedPresenter: stopDown
    stopButton.movieElement   := movieElement
    stopButton.activateaction := (a b -> stopMovie b)
    stopButton.x := (pageWidth - (innerMargin + buttonWidth))
    stopButton.y := pageHeight - stopbutton.height - margin

    -- make the shadow for the stop button
    local stopShadowButton := new twoDshape fill: darkerBlueBrush \
           stroke: darkerBlueBrushh  target: movieButtonRect
    stopShadowButton.x := stopButton.x + 2
    stopShadowButton.y := stopButton.y - 2

    -- make a box to show any required copyright notices
    local copyRightBox := new TextPresenter \
       boundary:(new rect \
           x2: (pagewidth -  innermargin) y2:15) \
           target:""
    setdefaultAttr copyRightBox @font (new PlatformFont name:"Times")
    setdefaultAttr copyRightBox @size 11

    -- put the copy right text presenter into a page element
    local copyrightElement := new pageElement presenter: copyRightBox \
           target: (e -> (getNth (getParentdata e SpecialPage) 4))

    copyrightElement.x := innermargin
    copyrightElement.y := pageheight - 20

    -- Make a page layer for the movie page
    local movieLayer := new PageLayer \
       boundary:(new rect x2:pageWidth y2:pageHeight)
    append movieLayer movieElement
    append movieLayer playButton
    append movieLayer stopButton
    append movieLayer playShadow
    append movieLayer stopShadowButton
    prepend movieLayer copyrightElement

    -- Make an actuator controller to control the buttons on the movieLayer

    local movieController := new actuatorController space:movieLayer
    append movieController playButton
    append movieController stopButton

    -- return the movie layer
    movieLayer
)
```

Make a Page Subclass that Points to Data Objects

The new subclass of `Page`, called `SpecialPage`, has an instance variable `dataObject` whose value is an object containing data for the page.

Define the `changePage` method on `SpecialPage` to stop the current movie, if a movie is playing.

```
-- Make a new Page subclass that points to the object
-- holding data for the page
class SpecialPage (Page)
    instance variables
        dataObject
end

-- define the changePage method for specialPage so
-- that whenever the page changes, if a movie is
-- playing, it stops automatically
method changePage self {class SpecialPage} newpage ->
(
    -- find the document containing the page
    local doc := findparent self document

    -- if the document's currentMovie instance variable
    -- has a value, stop the currentMovie and set the
    -- current movie to undefined.
        if (doc.currentMovie != undefined)
        do (stop doc.currentmovie
            doc.currentMovie := undefined
            )
    -- Do the default changePage behavior
    nextMethod self newpage
)
```

Make a Page Template

The function `makeTemplate` makes a template containing each of the page layers in a list of layers that is passed to the function.

This function is used later to create the definition template and the movie template.

```
-- makeTemplate makes a page template from a list of page layers
-- Call this function with a list of the layers that make up the template
-- It uses the boundary of the first layer in the list as the boundary
-- for the page template

function makeTemplate layerList ->
(
    local template1 := new pageTemplate boundary:layerlist[1].boundary
```

```
    for i in layerList do append template1 i
    template1
)
```

Make Movie Pages and Definition Pages

A movie page displays a heading, description and movie, and has buttons to control the movie.

The `appendMoviePage` function adds a movie page to a document. The `doc` argument must be an existing document. The `dataObject` argument must be an object that has `Text` values in its `heading` and `description` instance variables, and whose `movie` instance variable contains an interleaved movie player. The `copyRightInfo` instance variable of the `dataObject` contains copyright information, or an empty string if there is no appropriate copyright information.

The `template` argument must be a "movie page template", that is, an object returned by the function `makeTemplate`, whose input collection is a background layer, a definition layer and a movie layer.

A definition page displays a heading, description and bitmap.

The `appendDefinitionPage` function adds a definition page to a document. The `doc` argument must be an existing document. The `dataobject` argument must be an object that has `Text` values in its heading and description instance variables, and a bitmap in its picture instance variable. The `template` argument must be a "definition template", that is, an object returned by the `makeTemplate` function whose input collection is a background layer, a definition layer and a picture layer.

The necessary layers can be created by the functions `makeBackgroundLayer`, `makeDefinitionLayer` and `makeMovieLayer` as appropriate.

Note that you can use a single template object for multiple pages, you do not need to create a new template for each page.

```
function appendMoviePage doc dataobject template ->
(
    -- The new page has the same boundary as its template
    local newPage := new specialPage frame:template \
           boundary:(template.boundary)

    -- Set the value of the page's dataObject instance variable
    newPage.dataObject := dataObject

    -- Set the target of the page to be an anonymous function that
    -- returns an array of the heading, description, movie, and
    -- copyright info which are found on the dataObject
    newPage.target := (p -> #(p.dataObject.heading,
                              p.dataObject.description,
                              p.dataObject.movie,
```

```
                            p.dataObject.copyrightInfo))
    append doc newPage
    -- return the new page
    newPage
)

function appendDefinitionPage doc dataObject template ->
(
    -- The new page has the same boundary as its template
    local newPage := new specialPage frame:template \
           boundary:(template.boundary)
    newPage.dataObject := dataObject

    -- Set the target of the page to be an anonymous function that
    -- returns an array containing the heading, description
    -- and picture, which are found on the dataObject
    newPage.target:= (p   -> #(p.dataObject.heading,
                            p.dataObject.description,
                            p.dataObject.picture))
    append doc newPage
    -- return the new page
    newPage
)
```

Make a Document

The function `makeDocument` makes a document containing movie pages and/or definition pages.

The arguments to the function `makeDocument` are the page width, page height, a list of objects containing data for definition pages and a list of objects containing data for movie pages.

You need to create the `definitionObjectList` and the `movieObjectList` collections yourself.

The `makeDocument` function is an example function that illustrates how to create a document containing both definition pages and movie pages. You should modify this function to suit your needs.

The `makeDocument` function uses the `makeBackgroundLayer`, `makeDefinitionLayer`, `makeMovieLayer` and `makePictureLayer` functions already defined to make the page layers. It uses the `makeTemplate` function to make page templates containing the page layers. Then it creates a document and adds one definition page for every object in the `definitionObjectList`, and adds one movie page for every object in the `movieObjectList`.

After creating a document with function `makeDocument` you would need to append the returned document to an open window to make it visible.

See the file `MakeDoc.sx` in the folder `doctempl` in the folder that contains the sample code for this manual, for an example of creating the `definitionObjectList` and `movieObjectList` collections by importing media, and for calling `makeDocument` to create a document. The file `MakeDoc.sx` also shows how to save the document to a title container.

```
-- Example function that creates a document
-- You would modify this to suit your needs

function makeDocument pageWidth pageHeight \
              definitionObjectList movieObjectList ->
(
    -- Make the page layers
    local bkgdLayer := makeBackgroundLayer (pageWidth, pageHeight)
    local definitionLayer := makeDefinitionLayer (pageWidth, pageHeight)
    local movieLayer := makeMovieLayer (pageWidth, pageHeight)
    local pictureLayer := makePictureLayer (pageWidth, pageHeight)

    -- Make the page templates
    local glossaryTemplate := \
           makeTemplate #(pictureLayer, definitionLayer, bkgdLayer)
    local movieTemplate := \
           makeTemplate #(movieLayer, definitionLayer, bkgdLayer)

    -- Create the document and append pages to it.
    -- Note that we create an instance of the MyDocument subclass of Document.
    -- Instances of MyDocument know what movie they are currently playing.
    local doc := new MyDocument
    for i in definitionObjectList do
        appendDefinitionPage doc i glossaryTemplate

    for i in movieObjectList do
        appendMoviePage doc i movieTemplate

    -- return the document
    doc
))
```

C H A P T E R

14

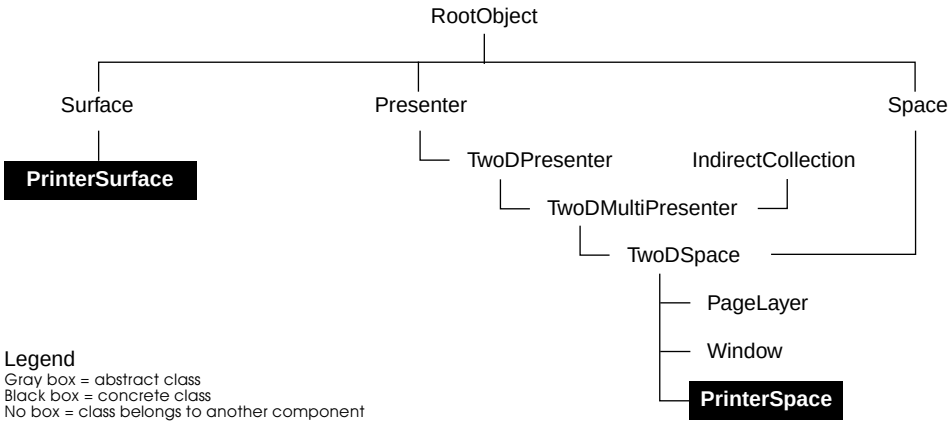
Printing

The Printing component is a separate loadable extension that 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
- Print a `TextPresenter` object to a series of pages
- Print a `OneOfNPresenter` (and therefore a `Document`) to a series of pages

Classes and Inheritance

The class inheritance hierarchy for the Printing component is shown in the following figure.



The Printing component consists of the following classes.

`PrinterSurface` – provides the low-level mechanism for rendering stencils

`PrinterSpace` – prints out the state of a presentation hierarchy

For information on the other classes shown in this figure, see Chapter 3, “Spaces and Presenters,” and Chapter 11, “2D Graphics,” in this guide.

Conceptual Overview

Printing is implemented similar to the way the 2D graphics subsystem is implemented. In 2D graphics you usually have a `Window` instance with various presenters attached to it and an underlying `DisplaySurface`. In 2D graphics, a `draw` method on a `TwoDPresenter` instance calls a `fill` or `stroke` method on a stencil, and all of these methods are passed the `DisplaySurface` object, on which the stencil is rendered.

The PrinterSpace and PrinterSurface classes in the Printing component are analogous to the Window and DisplaySurface classes used for screen display. Each PrinterSpace object has a PrinterSurface object associated with it. As with the screen-based classes, 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.

The PrinterSurface class, like its screen counterpart, DisplaySurface, implements fill, stroke, and transfer methods to render stencils to the printer. The printing loadable extension adds methods to the individual Stencil classes to actually perform the rendering.

A PrinterSpace instance, like a Window instance, can be made the parent space of a presentation hierarchy. The printFrame method on PrinterSpace 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 subpresenters. Since a title may use a Window that is a different size from the PrinterSpace page, you have to be careful when you print the presentation hierarchy associated with a Window. The section “Printing a Window View” on page 357 describes issues and options associated with this task.

Putting Print Capability in Your Title

The mechanism you use to put printing into your ScriptX title is similar to the mechanism used for cut, copy, and paste: You have the option of enabling or disabling the feature in the title’s menu, and you can provide an implementation of the feature at the TitleContainer or Window level.

The standard mechanism for a user to choose to print from an application is the **Print** menu item. The SystemMenuBar class (an instance of which is associated with each TitleContainer instance) provides a mechanism for enabling and disabling individual menu items. The identifiers that the SystemMenuBar class accepts for enabling and disabling items are @cut, @copy, @paste, @clear, @open, @close, and @print. The @print identifier enables or disables the printing-related menu options. On the Macintosh, this includes the **Print** and **Page Setup** menu items. On Windows, this includes **Print** and **Print Setup**. See Chapter 3, “Title Management,” in this guide for information about the system menu bar.

The printing-related menu options are disabled by default in a ScriptX title (the value of the SystemMenuBar key @print is false). To enable the printing-related menu options, enter, for example:

```
enableitem thetitlecontainer.systemmenubar @print
```

If a user chooses the **Print** menu item, the printTitle method is invoked on the title container that has user focus. The default implementation of printTitle invokes the printWindow method on the topmost window of the title. The default implementation of printWindow does nothing. You can create a subclass of TitleContainer and specialize the printTitle method, or you can create a subclass of Window and specialize the printWindow

method to provide an implementation of printing for your title. (Recall that the `print` global function has different functionality and is not part of the Printing component.)

The classes needed for printing are part of a loadable extension. You can load the `printing` extension dynamically in your implementation of the `printTitle` or `printWindow` method or while the title is starting up. The following is an example of how to load the printing extension while your title is starting up:

```
if (not isdefined PrinterSpace) do \
    (process (new loader) "loadable/printing")
```

The `if` condition ensures that you do not load this extension more than once, since loading any extension more than once causes ScriptX to throw an exception. You may not need to do this check during title startup, but you need to do it if you load the printing extension in either the `printTitle` or `printWindow` method.

For more information on working with loadable extensions, see Chapter 12, "The ScriptX Loader," in the *ScriptX Tools Guide*.

In summary, do the following to incorporate printing into your title:

1. Enable the printing menu options using the `TitleContainer` class's menu bar.
2. Provide an implementation of `printTitle` or `printWindow` that uses the printing classes `PrinterSpace` or `PrinterSurface`.

Printing a Window View

A window view is made up of all the presenters it contains, as arranged in their presentation hierarchy. A window view does not include what is clipped or hidden.

Printing a window view can be accomplished in one of two ways:

1. Print the offscreen bitmap used by the window's compositor. This involves a two-step process (see the following section):
   a. `TwoDPresenter` instances render themselves to the offscreen bitmap
   b. The bitmap is transferred to the printer, possibly with scaling
2. Have the individual `TwoDPresenter` instances in the presentation hierarchy (and, thereby, the `Stencil` instances they contain) render themselves directly to the printer. This may provide better results, especially if scaling is involved.

Printing a Window's Bitmap Image

This is a good example of how you might use the `PrinterSurface` class. In most cases, you should use the `PrinterSpace` class.

This code is an excerpt from the example file:
DOCEXMP/acguide/printing/winprin1/mywin.sx

```
class MyWindowClass (Window)
end

method printWindow self {class MyWindowClass} -> (
     local surface, scaleFactor, myTransform, oldRate

     -- Create PrinterSurface.
     surface := new PrinterSurface

     -- Calculate the ratio of our width and height to
     -- that of the surface. The minimum of the two ratios
     -- is what we will use to scale our bitmap.
     scaleFactor := min (surface.boundary.width / self.boundary.width) \
                          (surface.boundary.height / self.boundary.height)

     -- Create a transformation matrix that represents the scaling.
     myTransform := mutableCopy identityMatrix
     scale myTransform scaleFactor scaleFactor

     -- Display the printer dialog box and see if we should continue.
     if (printerDialog surface) do (
          -- Stop the window's clock to halt the presentation
          oldRate := self.clock.rate
          self.clock.rate := 0

          -- Transfer the window's offscreen representation
          -- to the PrinterSurface with the appropriate scaling.
          transfer surface (snapshot self undefined) \
                    surface.boundary myTransform

          -- Restart the window's clock.
          self.clock.rate := oldRate

          -- Flush the document.
          flushDocument surface
     )
)
```

This example starts by creating a new `PrinterSurface`. Invoking the `new` method on the `PrinterSurface` class without passing any keyword arguments gives you a `PrinterSurface` instance that represents the default printer device. By using the `deviceName` keyword argument, you can create a `PrinterSurface` that represents other available printer devices on the system. You can obtain a list of device names by calling the global function `getPrinterNameList`; note that on some platforms, printer selection is carried out outside the scope of an application and only the currently chosen printer is listed.

In general, the `PrinterSurface` and `PrinterSpace` classes should only be instantiated in a `printTitle` or `printWindow` method; that is, instances should not be stored or maintained across invocations of these methods.

Next, this example obtains the factor by which you need to scale the compositor's bitmap to display it on the printer with the best possible fit between the bitmap and the printer surface. This is done by calculating the ratio of the width of the window to the width of the printer surface and the height of the window to the height of the printer surface. The minimum of the two ratios is the amount used to scale the bitmap (keeping its aspect ratio) to best fit on the printed page. A `TwoDMatrix` instance (`myTransform`) is created to represent this transformation.

Then this example invokes the `printerDialog` method on `PrinterSurface`. This displays a machine-specific dialog box to get printer job settings, such as the number of copies and the range of pages to print. The dialog box also confirms whether printing should proceed. A Boolean value returned from this method determines whether to continue the print operation. Alternatively, you could display a custom dialog box for the application.

If the printer dialog box returns `true`, this example stops the window's clock and, hence, freezes the state of the presentation. Note that this assumes there are no clocks outside the window's hierarchy that are used to change the presentation. In other words, all clocks used in the presentation should be slaved off the window's clock.

Finally, the `transfer` method on the printer surface is used to print a bitmap representation of the window to the printer. Note that the example makes use of the `snapshot` method on the window, which returns an offscreen representation of the window. The clipping region represents the entire printable area of the printed page, and the transformation matrix is used to do the scaling.

The `flushDocument` method on `PrinterSurface` is used to indicate the end of the printing job. At this point, any spooled printer commands are sent to the printer.

This procedure for printing a window's view is not always the ideal solution. This is often most obvious when dealing with text. If the offscreen bitmap needs to be scaled (as is often the case), the bitmap representation of the text also gets scaled, often resulting in undesirable artifacts. The procedure described above should be used when:

1. Your presentation hierarchy uses features not supported by the printing engine. For example, if you use presenters that have non-rectangular clip regions or invisible colors (neither of which is supported by the printing system), this procedure might be your best option.

2. Your application is sensitive to the variability in the rendering of stencils by different printer drivers.

In most other cases, a better option is to incorporate the scaling operation into the actual rendering of the stencils, as shown in the next section.

Printing Each Presenter

Every `TwoDPresenter` instance has a `transform` instance variable — a `TwoDMatrix` instance that specifies a transformation for the contents of the `TwoDPresenter` object. For a `TwoDMultipresenter` object, the transformation applies to all the `TwoDPresenter` objects contained by the `TwoDMultipresenter` object. By modifying the `transform` instance variable of the top presenter of a presentation hierarchy, you can apply a transformation to the entire hierarchy while it renders itself to a printer. The following is an example of how that might be done.

This code is an excerpt from the example file:
`DOCEXMP/acguide/printing/winprin2/mywin.sx`

```
class MyWindowClass (Window)
end

method printWindow self {class MyWindowClass} -> (
     local p, scaleFactor, oldRate, tmpArray, tmpSpace

     -- Create a PrinterSpace object.
     p := new PrinterSpace

     -- Calculate the ratio of the width and height of the PrinterSpace to
     -- the width and height of the Window. The minimum of the two
     -- ratios will be used to scale the bitmap.
     scaleFactor := min (p.boundary.width / self.boundary.width)  \
                          (p.boundary.height / self.boundary.height)

     -- Create a GroupPresenter to hold the items in the Window.
     -- The transform goes on the GroupPresenter.
     tmpSpace := new GroupPresenter
     scale tmpSpace.transform scaleFactor scaleFactor
     prepend p tmpSpace

     -- This array holds all the items in the Window.
     -- It is needed because the following cannot be done:
     --         for i in self do
     --             prepend p i
     -- This is because we cannot iterate over the items
     -- in the Window while we're deleting them from it.
     tmpArray := new Array
     for i in self do
          append tmpArray i

     -- Display the printer dialog box to determine whether to continue.
     if (printerDialog p) do (
          -- Stop the Window's clock so nothing will change.
          oldRate := self.clock.rate
          self.clock.rate := 0
          self.compositor.enabled := false

          -- Move everything over to the PrinterSpace.
          for i in tmpArray do
               prepend tmpSpace i
```

```
            -- Take a snapshot of the presentation.
            printFrame p

            -- Put everything back.
            for i in tmpArray do
                 prepend self i

            -- Restart the Window's clock and re-enable the compositor.
            self.compositor.enabled := true
            self.clock.rate := oldRate

            -- Flush the document.
            flushDocument p
        )
)
```

This script starts out looking similar to the previous example, except that it creates a `PrinterSpace` instance. As with the `PrinterSurface` class, creating a new `PrinterSpace` without specifying a `deviceName` gives you a representation of the default printer. Besides the `deviceName` keyword, you can use the `surface` keyword to specify a `PrinterSurface` for the `PrinterSpace` instance to use.

Note that several of the instance variables and methods of `PrinterSpace` used in this example (`printerDialog` and `flushDocument`, for example) are similar to those that exist in the `PrinterSurface` class. This obviates the need to use the `PrinterSurface` class when you are dealing with a `PrinterSpace` object.

After calculating the factor by which you need to scale to best fit into the `PrinterSpace` object's boundary, this example creates a `GroupPresenter` instance to hold the presentation hierarchy. The boundary of the `GroupPresenter` instance accommodates all presenters in the hierarchy. The transform is added to the `GroupPresenter` object rather than to the `PrinterSpace` object, since you want to clip to the boundary of the `PrinterSpace` object, not to the scaled version of the boundary.

After confirming whether printing should continue, the example moves the presentation hierarchy (actually, all the direct subpresenters of the `Window`) to the `GroupPresenter` and takes a snapshot of the presentation using the `printFrame` method. Note that the compositor is disabled while the hierarchy is moved to the `PrinterSpace`, so this process is not visible on the screen.

Note – This operation is computationally expensive: Moving a presentation hierarchy involves losing cached information used by the compositor, rescaling clocks in the presentation hierarchy, and recalculating other presentation information.

The same method can be used to build presentation hierarchies on-the-fly, specifically for printing. In the Autofinder demo, for example, you can print the entire contents of the Map scene containing the location of the cars if the

Detail window is not displayed. If the Detail window is displayed, you can print a presentation hierarchy consisting of a bitmap for the car, a short description of the car, and user notes about the car.

Printing a TextPresenter to Multiple Pages

Another common use for a printing engine is to print several pages of formatted text.

In ScriptX, you can print text using the `TextPresenter` and `PrinterSpace` classes. The procedure to print text is summarized as follows:

1. Create a `PrinterSpace` instance and sets its margins.
2. Append the `TextPresenter` object to the `PrinterSpace` object, and set the `boundary` of the `TextPresenter` object to the `boundary` of the `PrinterSpace` object. The `TextPresenter` object lays out its text based on this boundary.
3. Query for the offset of the last completely visible character on the page, using the `getLastVisibleOffset` method on the `TextPresenter` object. This offset defines where to start printing the next page.
4. Modify the height of the `TextPresenter` object to display only completely visible lines. The height is obtained by using the `getPointForOffset` method on the `TextPresenter` object, using the offset from the previous step.
5. Print the page using the `printFrame` method on the `PrinterSpace` object.
6. Reset the height of the `TextPresenter` object and modify its `offset` instance variable to lay out the next page.
7. Continue until all the pages are printed.

The previous two examples printed out only a single page. To print multiple pages, use the `flushPage` method as shown in this section.

There are two ways to print a subrange of a set of text pages:

- Send all the pages in the set to the print engine, and allow the print engine to print the correct subrange of pages according to what the user selected in the print dialog box. In this case, the print engine still processes all the pages for printing, even though it actually prints only the correct subrange.
- Send only the subrange of pages that the end user specified in the print dialog box to the print engine. Your application has better performance in this case since you avoid unnecessary processing by the print engine. The example in this section shows this kind of implementation.

Consider the following example. This code is an excerpt from the example file: `DOCEXMP/acguide/printing/textprn/textprin.sx`

```
method printText self {class TextPresenter} margin units -> (
     local p, oldX, oldY, oldBoundary

     -- Create a new PrinterSpace and set its margins.
     p := new PrinterSpace
     setMargin p margin units

     -- Record the old dimensions and location and attach
     -- the TextPresenter to the printer
     oldBoundary := self.boundary
     self.boundary := p.boundary
     oldX := self.x; oldY := self.y
     self.x := self.y := 0
     append p self
     recalcRegion p

     -- Set the context to be the PrinterSurface. The TextPresenter
     -- might be queried for metric information before it's told to draw.
     setContext self p.surface p.globalBoundary

     if (printerDialog p) do (
          local firstPage, lastPage, \
                 newOffset, pageNum
          local defaultPageHeight := self.height

          -- Record the page range from printerDialog in local variables.
          -- Set the printerDialog variables back to their default values.
          -- The script will send only the specified subrange of pages to
          -- the print engine, telling the print engine to print all of
          -- those pages.
          firstPage := p.firstPage
          p.firstPage := 1
          lastPage := p.lastPage
          p.lastPage := @all

          self.offset := newOffset := 0
          pageNum := 1

          -- While there is still text to be printed out:
          repeat while (newOffset < self.target.size) do (

            self.offset := newOffset

             -- Get the offset of the last visibly complete character on this page
            -- The text for the new page will start right after it.
            newOffset := (getLastVisibleOffset self)

             -- Get the position of this character and change the height
             -- of the presenter to match that. Note that the point information
             -- is returned in global coordinates
             local lastXY := getPointForOffset self newOffset
             self.height := (lastXY.y - self.globalboundary.bbox.y1) /
                         self.globaltransform.d

             -- If the current page is within the correct
             -- range, then print it
             if ((pageNum >= firstPage) and \
```

```
                ((lastPage = @all) or (pageNum <= lastPage))) do (

                -- Take a snapshot of the current page.
                printFrame p

                -- Flush the current page and move on to a new one.
                flushPage p
            )

            pageNum := pageNum + 1
            newOffset := newOffset + 1
            -- Restore to standard height
            self.height := defaultPageHeight
        )

        flushDocument p
    )

    emptyout p
    self.offset := 0
    self.x := oldX; self.y := oldY
    self.boundary := oldBoundary
)
```

This example first creates a `PrinterSpace` instance and sets its margins. The `setMargin` method accepts a collection of four values and a name that represents the units of those values. For example,

```
setMargin p #(1, 1, 1, 1) @inches
```

sets the margins of `p` to one inch on each side. Setting the margins effectively changes the `boundary` instance variable of the `PrinterSpace` instance so that it is correctly offset and clipped on the printed page. A list of accepted units names is provided in the description of `PrinterSpace` in the *ScriptX Class Reference*.

This example uses the `setContext` method on `TextPresenter`. This is used to provide the `TextPresenter` with a context (a target surface and a clip) for text metric calculations if the `TextPresenter` object is not drawn. This method enables you to call methods like `getOffsetForXY` and `getLastVisibleOffset` if you skip pages in the text.

This script looks at the `firstPage` and `lastPage` instance variables of the `PrinterSpace` instance before printing. The default values of these variables are `1` and `@all`, which tell the print engine to print all pages in the set sent to it. These instance variables may be set to other values by the end user during the `printerDialog` method. If the script does nothing with these values, then the print engine uses them to print the subrange specified by the end user, if any. This example instead saves these values, sets them back to their defaults, and then uses the saved values to send only the user-defined subrange of pages to the print engine. The script (rather than the print engine) accepts the responsibility for printing out the correct page range. This increases the printing performance of the application.

This example uses the `flushPage` method to indicate a page break. Call `flushPage` for every page of the text that is actually printed.

Another issue to address is the resolution (dot-pitch) of the printer device. Although the underlying device could be anything from a 72dpi dot-matrix printer to a 300dpi laser printer, it is often important that the results of a print job look relatively similar on each device, at least in terms of layout if not quality. To accomplish this, by default the `transform` instance variable of a `PrinterSpace` object is always set in such a way as to simulate a 72dpi device. In other words, the default coordinate space in which you deal with a `PrinterSpace` is a 72dpi point space, irrespective of the dot-pitch of the underlying device. The `transform` instance variable does the necessary scaling to the pixel space of the underlying `PrinterSurface`.

In the first two examples, resolution was not much of an issue. Those examples scaled the presentation space to best fit into the boundary of a printed page and applied their own transform to do the scaling. Different sized pages would yield different results in terms of the scaling of the presenters. For example, the fonts in a block of text in the presentation space would be scaled to different sizes on differently sized pages.

In the third example, resolution is more important. The layout, but *not* the size of the font, needs to change on different pages. The default `transform` on the `PrinterSpace` object accomplishes this by ensuring that you are always dealing with the same coordinate space. In other words, you do not have to worry about the resolution of the device — just be aware that scaling may be occurring. “Resolution and Scaling” on page 367 discusses dealing with printer devices in their own coordinate spaces.

Printing a OneOfNPresenter to a Series of Pages

Titles that probably benefit most from printing capabilities are online documents. The `Document` classes in ScriptX provide a framework for building such documents. The following example shows how the concepts from the previous sections can be used to print out pages of a document. It defines new subclasses of `Document` and `Window` and provides implementations of `printDocument` and `printWindow` to render them to a printer. This code is an excerpt from the example file:
`DOCEXMP/acguide/printing/docprn/docprint.sx`

```
in module docModule

class MyDocumentClass (Document)
end

method printDocument self {class MyDocumentClass} -> (
     local p, scaleFactor, myTransform, oldTransform, \
            oldPosition, oldPresentedBy

     -- Create a PrinterSpace
     p := new PrinterSpace
     myTransform := mutableCopy identityMatrix
```

```
        oldPresentedBy := self.presentedBy
        prepend p self

        -- Start the document and see if we should continue
        if (printerDialog p) do (
             local firstPage, lastPage, i, n

             -- Record the firstPage and lastPage
             firstPage := p.firstPage
             p.firstPage := 1
             lastPage := p.lastPage
             p.lastPage := @all

           -- Clip the page range to the extent of the document
            if ((lastPage = @all) or (lastPage > self.size)) do
                  lastPage := self.size

             -- Record the old state of the Document, so that we can reset it
             oldPosition := self.cursor
             oldTransform := self.transform

             for i := firstPage to lastPage do (
                  -- If this page is in the correct page range
                  -- Go to that page and find out how best to
                  -- scale it. Since different pages can have
                  -- different boundaries, we have to do it everytime.
                  goto self i
                  scaleFactor := min (p.boundary.width / \
                                      self.boundary.width)   \
                                      (p.boundary.height /   \
                                      self.boundary.height)

                  -- Set the transform based on the scaling factor
                  reset myTransform
                  scale myTransform scaleFactor scaleFactor
                  self.transform := myTransform

                  -- Take a snapshot of the page
                  printFrame p

                  -- Move on to the next page
                  flushPage p
             )

             -- Flush the document
             flushDocument p

             -- Reset the state of the document
             goto self oldPosition
             self.transform := oldTransform
        )

        prepend oldPresentedBy self
    )

    class MyWindowClass (Window)
    end
```

```
method printWindow self {class MyWindowClass} -> (
      local doc := self[1]
      self.compositor.enabled := false

      printDocument doc

      self.compositor.enabled := true
)
```

This example incorporates concepts from the previous two examples. The `Document` is printed out by attaching it to a `PrinterSpace` instance and iteratively going to and taking a snapshot of each page. Note that the scaling is carried out differently for each page, since the pages in a `Document` can have different boundaries. The `firstPage` and `lastPage` instance variables are used to determine the range of pages to print.

These examples are meant to serve as just that — examples. They provide some of the basic functionality you might need to put printing into a title. The expectation is, however, that you will eventually be able to build more sophisticated implementations that better serve the needs of your title.

Other Printing Issues

Resolution and Scaling

As mentioned earlier in this chapter, a `PrinterSpace` object is set up, by default, to operate in a 72dpi point scale, irrespective of the resolution of the underlying device. This is accomplished by applying a scaling transform (stored in the `transform` instance variable of the `PrinterSpace`) to do the coordinate space conversion. The `transform` instance variable can be modified, however. For example, changing it to an identity matrix allows you to deal with the `PrinterSpace` in the pixel space of the device. If you want to perform some other type of affine transformation, however, be aware that you could disturb the scaling components in the matrix.

You can query the physical resolution of the print device by using the `physicalResolution` instance variable on a `PrinterSurface`. This returns a `Pair` consisting of two numbers that represent the horizontal and vertical resolution in dots per inch. The `PrinterSurface` class also has an `availableResolutions` instance variable that consists of an array of available resolutions for the printer. Many Apple printers, for example, can work at different resolutions. The `availableResolutions` instance variable tells you which resolutions the `physicalResolution` instance variable can be set to.

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.

Bitmaps need to be at the printer resolution to look good. A scale factor of 4X often works better than other scale factors.

Graphics Features That Don't Print Well

Several features that can be used for screen graphics do not translate well to a printer. For example, non-rectangular clip regions and invisible colors do not always work well for a printer. Transfer modes other than `srcCopy` are not recommended.

C H A P T E R

15

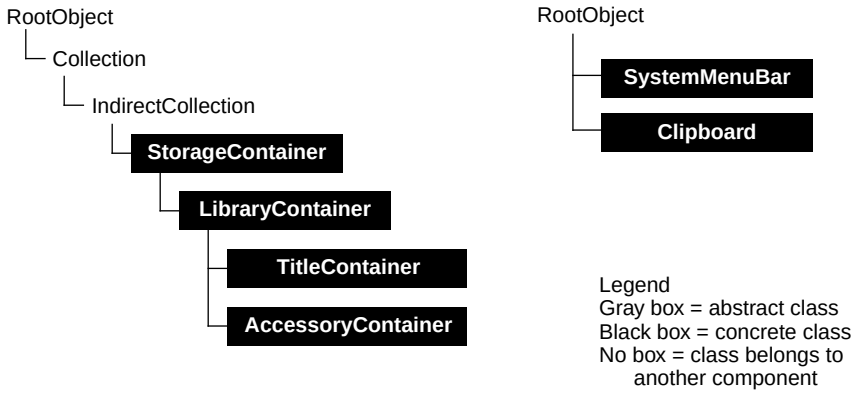
Title Management

The Title Management component is a title's functional interface to the underlying file system and desktop interface. It provides the capability to start and quit the ScriptX Player, and open and close ScriptX titles. It provides access to system resources such as a menu bar, the clipboard, and the open and save dialog boxes. It also provides for distinct ScriptX libraries and accessories to support titles.

This chapter begins with a conceptual overview. It then describes opening and closing titles, creating title containers, managing objects in a title, managing windows, pausing a title, using the system menu bar and clipboard, using libraries and accessories, and quitting a title and ScriptX. It describes the storage container classes `StorageContainer`, `TitleContainer`, `LibraryContainer`, and `AccessoryContainer`, and the supporting classes `SystemMenuBar` and `Clipboard`. Finally, this chapter shows several examples of using the Title Management component.

Classes and Inheritance

The class inheritance hierarchy for the Title Management component is shown in the following figure.



The following classes form the Title Management component. In this list, indentation indicates inheritance.

`StorageContainer` – represents a general purpose file for the storage of objects. You do not need to work directly with `StorageContainer` instances to manage your titles. Instead, work with title containers, library containers, and accessory containers, which are storage containers by inheritance.

`LibraryContainer` – represents a collection or library of loadable objects and has prestartup, startup, and terminate actions. A library can be identified by name, version, and copyright. An instance of this class can hold media objects, ScriptX modules or classes, and other objects to be used by titles and other libraries.

`TitleContainer` – represents an interactive multimedia title or tool that a user can open and close independent of other ScriptX applications.

`AccessoryContainer` – represents a set of classes and instances that can be dynamically added to a running title to supplement the title with data or behavior.

`SystemMenuBar` – provides the ability to control the appearance of the ScriptX Player menu bar presented by the native operating system.

`Clipboard` – represents an area of memory where text can be copied to and pasted from, for moving text from one title or application to another.

Conceptual Overview

ScriptX titles and tools manage three basic kinds of container files: titles, libraries, and accessories. The three kinds of containers are represented by the `TitleContainer`, `LibraryContainer`, and `AccessoryContainer` classes, as shown in Figure 15-1. Each of these container objects can hold a loadable set of objects and classes, but each kind of container has a different purpose. A complete ScriptX title typically comprises one title container and several library containers, and it may include a number of accessory files.

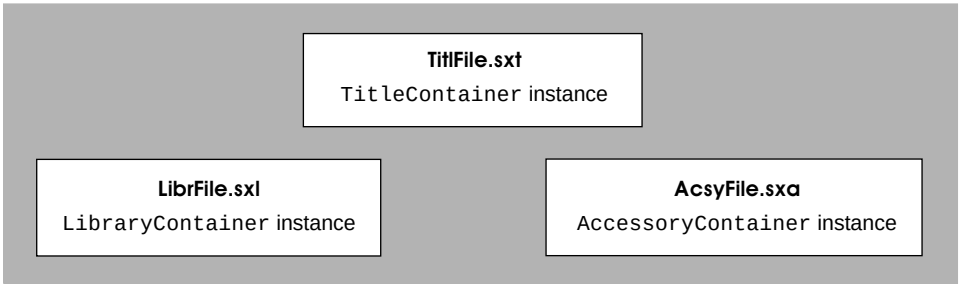


Figure 15-1: Titles, libraries, and accessories are represented by their corresponding containers.

Note that the organization of files in a title differs from the inheritance hierarchy of the classes that represent those files. In the inheritance hierarchy, `TitleContainer` and `AccessoryContainer` both inherit from `LibraryContainer`, which inherits from `StorageContainer`. In a title that runs with the ScriptX Player, the `TitleContainer` file is the top-level file, the user's entry point. This title file draws on resources from library and accessory files, and an end-user product generally does not include any `StorageContainer` files.

`TitleContainer` and `AccessoryContainer` specialize `LibraryContainer` for different purposes. This specialization is summarized in "Classes and Inheritance" on page 371, and also in the discussion of "Title" on page 374. The two subclasses inherit much of their functionality from

LibraryContainer, and they share a similar mode of operation. For example, each of these container classes implements an open, close, update, and terminate method and defines the instance variables preStartupAction, startupAction, and terminateAction.

Note – Even though TitleContainer and AccessoryContainer objects are LibraryContainer objects by inheritance, the term “library” by convention is used in the narrower sense shown in Figure 15-1 to refer to files that contain code or media objects used by a title.

The following terminology is used in the remainder of this chapter:

title – the complete multimedia application that the end user experiences

title file – a file that holds a TitleContainer object, from the user launches the title application

title container – a TitleContainer object

library – a set of code or media objects that is used by a title without any direct action by the end user

accessory – a container object that the end user can add to a running title

The title file is the focal point for the user, since the user starts a ScriptX title by opening the title file in the ScriptX Player. The title should automatically load any libraries it needs. If a required library is not present, the developer can define the preStartupAction script that either continues or stops the title. As the title runs, the user can open accessories that supplement the title.

In addition to title, library, and accessory files, you can create a general purpose storage file. Storage files are instances of the StorageContainer class. They have more limited capabilities than the other three types of container files. For example, they lack the startup and shutdown mechanisms defined by the LibraryContainer protocol. In the title development process, you normally do not create storage containers directly.

Storage Container

An instance of StorageContainer represents a general purpose storage file. This kind of storage file has more limited capabilities than the other three types of container files. The StorageContainer class defines important functionality that library containers inherit, such as the open class method and the close, update, and requestPurgeForAllObjects methods.

Unlike a TitleContainer file, a StorageContainer file cannot be opened by the user at the system level, and it cannot have user focus. It has no list of players and clocks, and it has no explicit relationship with other containers, such as libraries and accessories.

The StorageContainer class is sealed and cannot be subclassed. You may want to create an instance of a StorageContainer if you need a file that does not require the additional functionality of LibraryContainer or

`TitleContainer`. For example, you may want to use `StorageContainer` objects in a ScriptX tool. (A tool is a program that assists a ScriptX developer in creating a title.)

Title

A *title* is a complete, stand-alone, interactive ScriptX program. From the perspective of a user, a title is composed of objects interacting over time in multiple, concurrent spaces, in which a user can participate. As described in Chapter 1, "ScriptX Features," most titles implement one or more design metaphors such as modular compositions, virtual spaces, conversational interactions, constructive experiences, and multitrack sequencing.

A *title container* is the ScriptX object (an instance of `TitleContainer`) that maintains the high-level organization of a title, including the container file in which objects are stored. In particular, a title container is a collection of ScriptX media, code, and data objects that make up a title. A title container also defines startup and shutdown activities for a title through its prestartup, startup, and terminate actions, and it provides access to system resources such as the system menu bar and clipboard. A title container maintains a list of the libraries that it uses in its `libraries` instance variable. A title container also maintains a list of its windows, top clocks, and top players so that the title as a whole can be paused, resumed, muted, opened, and closed.

A title runs in the ScriptX Player. A title container can hold either a title or a stand-alone tool. A title can use libraries and accessories, but in general a title does not use other titles.

Much of the remainder of this chapter describes how a ScriptX title works: how it opens and closes and how it gains access to system resources, including the system menu bar and the clipboard.

Library

A *library* is a ScriptX resource file that can provide any kind of code, media, or data objects to a title. Although a library is a separate file, it cannot run alone in the ScriptX Player—it must be accessed from a title. One library can be shared by more than one title. When a title uses a library, the title automatically opens and closes the library. A library can use accessories and other libraries.

There are several ways to use libraries, depending on their contents:

- Code libraries can contain importers, exporters, transitions, classes, or functions that extend the capabilities of ScriptX. Libraries might supply fixes or version upgrades to a title. Another possible use of a code library is to supply a set of user interface controls, such as pushbuttons and popup menus, that share a common look and feel.
- Media libraries can contain video streams, audio streams, graphic images, animation, and text. A large title that has natural, metaphorical divisions could organize its media objects into libraries. For example, detailed models of the Earth and Moon might be saved as two separate libraries, since a user would be at only one place at a given time.

- Data libraries can supply updated numerical information to be presented by the title. A library could contain user preferences and user-created data or media. For example, a game might save the user's name, score, and position to a data library.

There are many different reasons why you might provide a library rather than include the library's contents directly in a title. Both titles and libraries allow developers to group, manage, and load stored objects effectively. Libraries also allow developers to update and distribute content separately from titles. In addition, a library can be reused by different titles.

A *library container* is the ScriptX object (a `LibraryContainer` instance) that maintains the high-level organization of a library, including the container file to which objects are stored. Like a title container, a library container is a collection of objects that defines its own startup and shutdown activities through a startup action and a terminate action. A library container also maintains a list of the titles or libraries that use it in its `users` instance variable and a list of other libraries that it uses, in its `libraries` instance variable.

Accessory

An *accessory* is a ScriptX resource file that a user can add dynamically to a running title. It is intended for incrementally adding data or behavior to titles. Like a library, it may be usable in one or many titles. The main difference between an accessory and a library is that a user can directly open an accessory from the ScriptX Player File menu using **Open Accessory**. Examples of accessories include a tape measure that can be used to measure the size of objects, or an inspector that can analyze the state of objects.

An *accessory container* is the ScriptX object (an `AccessoryContainer` instance) that maintains the high-level organization of an accessory, including the container file to which objects are stored. It is a collection of objects, and has all of the capabilities of a library container, but is uniquely identified as an accessory so that a user can open or close it from within a title.

Storing ScriptX Code in Storage Containers

When ScriptX source code is compiled, the results of compilation are ScriptX objects: instances of `RootClass` or `ByteCodeMethod` and global instances of other objects. These code objects can be saved in storage containers. Although it is possible to store individual classes, methods, functions, and so forth to storage containers, the process would be very tedious, and prone to error.

A much easier technique exists for storing ScriptX code. ScriptX code is always compiled in a module. (If you do not declare that your program is being compiled in a particular module, it is compiled in the `Scratch` module, a default module which cannot be saved.)

Modules are really collections of name bindings. (Indeed, the `ModuleClass` class, although it does not inherit from `Collection`, implements several generic functions from the Collection protocol.) In effect, every code object that is compiled in a module is a subobject of the module itself.

In this respect, a module acts as a container for all the code objects that are compiled in it. Adding a module to a title container is a convenient way to add all of the classes, instances, global variables, and global functions that are defined in that module to a title.

Note – ScriptX variables are saved with the module that provides their definition, not the module in which they are declared. When you add a ScriptX module to any library container, it also adds all the other modules it uses and any modules that also use it. By storing a module, you are effectively also storing everything that touches or is touched by that module. See the "Modules" chapter of the *ScriptX Language Guide* for more information.

Only a variable that is owned by a module is saved in a storage container when you save the module to the container. A variable is owned by the module that defines it. Only one module owns each variable. A variable that is owned by the `Scratch` module is transient, and is not saved when the module is saved. Only a variable that is owned by a module is saved when the module is saved and can be exported (shared). Variables owned by a module are global to only that module and any modules to which that variable is exported. For more information about modules, see the *ScriptX Language Guide*.

One possible structure for a ScriptX title and its associated files is to store only modules in the title container itself, using library containers to store data and media objects.

Unlike other persistent objects, modules must be explicitly loaded from containers. Typically, all modules in a container are loaded by the container's startup action.

Startup Actions

Library containers (`LibraryContainer`, `TitleContainer`, and `AccessoryContainer` objects) implement a mechanism that allows a developer to specify a function that runs when the container is opened. The `preStartupAction` and `startupAction` instance variables, which are inherited from `LibraryContainer`, store these functions.

A startup action can be used to preload objects for performance reasons. For example, a title could preload a large bitmap into memory so that there is minimal latency when the user requests that it be displayed. Or a title could preload objects that are used in its first scene to minimize the time the user waits for the first window to be populated. As a rule, objects do not need to be preloaded. Objects load automatically when a method is called on them.

Modules are a special case. It is common for a title, library, or accessory to contain a module, and these modules are usually loaded in the container's startup action. Since modules act as containers for compiled code, to save a module to a container is to save all classes and functions that are compiled in that module. Modules need to be loaded when the container is opened, so that the title's code objects are all accessible, as in the following example:

```
-- myTC stores a module with the explicit key @myModule:
myTC.startupAction := (tc -> load myTC[@myModule])
```

For more information on modules, see the "Modules" chapter of the *ScriptX Language Guide*. Startup actions are described in more detail in the sections "The Title Startup Sequence" on page 394, "Opening and Closing a Library" on page 412, and"Opening an Accessory File" on page 418.

How Storage Containers Work

From the point of view of memory management, there are two kinds of objects in a ScriptX program: stored objects and allocated objects. Objects that are saved in containers are called *stored* objects or *persistent* objects because they live on disk even when the title is not running. Objects that live only in memory and do not exist when the title is not running are called *allocated* objects or *transient* objects.

Life Cycle of a Stored Object

The purpose of any storage container (including titles, libraries, and accessories) is to store objects so that they can be used again. The life cycle of a stored object has two phases: When you create an object, it is a transient object. You make it persistent by putting it in a storage container. After you save it to disk, it can be purged from memory. Thereafter, each time you use the object, its most recently updated version is reloaded into memory automatically.

Storage containers are collections. Since `StorageContainer` inherits from `IndirectCollection`, any objects that you put in a storage container by calling a collection method (such as `add`) are added to the target collection of that storage container. The target collection itself is accessible through the storage container's `targetCollection` instance variable.

```
global tc := new TitleContainer path:"myTitle.sxt" name:"myTitle"
global win := new Window name:"Welcome to My Title"
prepend tc win -- Window win is a target collection object in tc.
```

A storage container contains (stores) all the objects that are elements of its target collection, and also any objects that are reachable from those target collection objects. Objects that are reachable from target collection objects are called subobjects. Subobjects include elements of collections and instance variables of a target collection object. For example, if you store an array and a pushbutton in a storage container, the array's elements and the pushbutton's subpresenters are all subobjects of the container's target collection.

An object can only be stored in one container. If you attempt to store an object in a second container, only a reference to the first container is actually stored in the second container. At any give time, the system contains either zero or one instance of any stored object.

When you open a storage container, you can reference any objects in the container. However, objects do not automatically load into memory just because you opened the container that stores them. A stored object is loaded into memory when you call a method on it.

Objects can only be accessed by calling a function or method on them. Note that this can happen even if you do not explicitly call a method on the object. Whenever an object is used, explicitly or implicitly, in any expression, the object is loaded automatically if it is not already in memory. For example, suppose that an object is referenced by a ScriptX variable. If you enter the variable's lexical name in the Listener, the Listener to displays its value. To display the value, it calls the global function `print` on the object, and the `print` function calls that object's method for `prin`.

Not every operation on an object requires that the object be brought into memory. The identity test (`==`) test does not load an object into memory, since comparing storage locations does not require knowledge of the object's contents. (Two objects that have the same location in memory or in the object store must be the same object.) The identity test really does not examine an object—it only examines its pointer.

Persistent Versus Transient Objects

Not all objects used in a ScriptX need to be stored objects. If an object requires more time and resources to load than to create, you might want to create it dynamically when you need it rather than storing and then reloading it. If your title is delivered through a medium with low bandwidth, such as a CD–ROM or the Internet, it may be much faster to create objects on the fly. Buttons, menus, and other small user interface objects are good candidates for this.

Make sure the state you are saving is part of the run-time state and is not associated with authoring-time. The ScriptX language defines qualifiers for instance variables that override the normal storage behavior of objects. See the discussion of the `readOnly`, `transient`, and `reference` qualifiers in the *ScriptX Language Guide*.

References Across Containers

An object can only be stored in one container. When you attempt to store it in another container, only a reference to the first container is stored in the second container. Thus, when you use the object from the second container, the first container (where the object is actually stored) is inflated to memory automatically, an effect you may not have intended.

Be careful that you do not add an object that was already added to another container. This can easily happen if the object you are adding is a subobject of an object that has already been added to another container, since adding an object to a container automatically adds all of its subobjects to that container.

Object Store Protocol

Every ScriptX class defines or inherits a set of methods that allow instances to be stored and retrieved from the object store. Together these generic functions comprise the Object Store protocol. The default implementation for each of these generics, which include `inflate`, `deflate`, `update`, `load`, and

`afterLoading`, is inherited from `RootObject`, the root system class. With the exception of `afterLoading`, it is unusual for developers to specialize the generic functions in the Object Store protocol in scripted classes.

A protocol is an informal group of properties and behaviors that work together to accomplish some set of tasks. The Object Store protocol is one of the fundamental "sets of behaviors," inherited from `RootObject`, that are common to all objects.

This section describes these Object Store generic functions, for which all objects in the system define or inherit a method. The Object Store protocol is supplemented by a set of helper functions and system globals that provide additional information about objects and the state of the system as a whole. These global functions and variables are defined in sections that follow.

The load Method

The `load` method basically calls an object and does nothing. It touches the object, and thus causes it to be loaded automatically into memory. The process of loading an object is often referred to as *object inflation*.

It is not usually necessary to call `load`. A stored object is loaded into memory automatically, unless it is already in memory, when it is supplied as the argument of a function or method. In certain cases you might want to preload a group of objects, such as the set of objects required to open a scene.

In general you should avoid preloading objects. The exception to this rule is modules, which act as containers for stored code objects.

The update method

If you modify a persistent object, whether it is a member of the container's target collection or a subobject, call `update` on the object to store the modified version back to the storage container. If the object is a subobject of some other object in the same container, you can call `update` on the parent object. Alternatively, you can call `update` on the container file itself. For a code example, see the section, "Saving Objects to a Container File" on page 385.

It is a good habit to save objects each time you modify them. If you modify an object without updating it, your modifications are lost and the original object is loaded the next time the object is loaded from storage. When an object is loaded from storage, ScriptX loads the most recently updated version.

When you close a storage container, all objects in the container are automatically saved to storage if the container was opened in `@update` mode, which is the default. This includes all target collection objects and any subobjects that are reachable through the target collection.

When a title container uses a library or accessory, objects in the library or accessory continue to be saved and stored as persistent objects in their original container. When you save the title, it saves pointers to any libraries or accessories it uses, and it reports an error if those files are not present. Saving a

title does not automatically save the libraries and accessories that the title is using. You must save those library and accessory containers separately if you want to save changes to them.

The inflate and deflate Methods

Whenever a persistent object is needed from the object store, `inflate` is called automatically to load the object. When an object is saved, the `update` method calls `deflate` automatically.

Since `inflate` is called each time an object is brought in from storage, it is analogous to `init`. You can specialize `inflate` and `deflate` to perform custom object storage management, but this is rarely necessary and is not encouraged. Contact Apple Developer Services if you feel you need to specialize `inflate` and `deflate`. To perform some action each time an object is retrieved from the object store, a better strategy is to specialize `afterLoading`.

The canStore Method

The `canStore` method is called automatically by the system when it traces through a network of objects, to determine which objects should be stored. Although it is possible to specialize `canStore`, the ScriptX language provides higher-level hooks for overriding the normal storage characteristics of objects that are far easier to implement. See the `transient` qualifier in the *ScriptX Language Guide*.

The afterLoading Method

After an object is loaded into memory, the system calls `afterLoading` on the object automatically. Your script should never call `afterLoading` explicitly; it is visible to the scripter so that it can be specialized to perform some task that is required whenever an object is loaded into memory. The default implementation of `afterLoading`, which is defined by `RootObject`, does nothing.

The answer to a popular analogy question on the ScriptX Aptitude Test (SAT) goes, "`afterLoading` is to `inflate` as `afterInit` is to `init`." In this respect, `afterLoading` is analogous to `afterInit`, which is called automatically after initialization. A typical application of `afterLoading` would be to load member objects of a collection or subpresenters of a multipresenter such as `PushButton`.

Object inflation takes place in two steps: The object's `inflate` method initializes transient objects referred to by the inflated object's instance variables. Its `afterLoading` method provides a place to access and manipulate those instance variables. The `afterLoading` method is invoked automatically on an object after it has been loaded from a storage container by the `inflate` method. Override `afterLoading` to perform custom initialization on the object being inflated (loaded) and its subobjects.

When you close a storage container, `close` calls `update` on the container automatically. All persistent objects in the container that have been loaded in memory become purgeable.

Shared Objects

Shared objects are an exception to several object storage rules. Shared objects define their own instance methods for `inflate`, rather than inheriting the default version of `inflate` from `RootObject`. ScriptX has two categories of shared objects: immediate objects and interned objects.

Immediate objects are numbers (`ImmediateInteger` and `ImmediateFloat`) that are not full-fledged objects. Since immediate objects "collapse" into their own pointers, they are not allocated separate storage either in the ScriptX heap or in the object store. For more information, see "Immediate Objects" on page 486 of Chapter 17, "Numerics."

Interned objects are stored in a system table and referenced by value. Two interned objects that have the same value are the same object. ScriptX defines one class of objects that can be interned, the `NameClass` class. For more information on `NameClass` objects, see the definition of the `NameClass` class in the *ScriptX Class Reference*.

Unlike other stored objects, which can be stored in only one container, shared objects can be stored in more than one container. Other persistent objects do not load until you call a method on them. Shared objects that are persistent are loaded into memory as soon as they are referenced. Shared objects want to live in a transient space even though they are stored.

Object Store Helper Functions

The following global functions are useful for managing storage containers. They are described in detail in the *ScriptX Class Reference*.

`canRequestPurge` – queries whether an object is a persistent object, so that you can legally call `requestPurge` on it. (An object that is transient cannot be purged.)

`getStorageContainer` – returns the `StorageContainer` instance in which a given object is stored, or will be stored the next time the container is updated. It returns `undefined` if the object is not persistent.

`isInMemory` – indicates whether or not an object is in memory. `isInMemory` can be called on any object, either stored or allocated.

`isPurgeRequested` – confirms whether you have called `requestPurge` on an object.

`requestPurge` – purges a stored object from memory. The function `requestPurge` can only be called on persistent objects.

Object Store System Globals

The following global variables provide useful state information for managing storage containers. They are described in detail in the *ScriptX Class Reference*.

`objectStoreMessages` – prints messages to the stream specified by `objectStoreMessagesStream` whenever a storage container is loaded from or saved to. Which messages are printed depends on which flags are set. See the *ScriptX Class Reference* for more information.

`objectStoreMessagesStream` – specifies which stream object store messages are sent to. By default, these messages are sent to the Listener.

`rarelyInflatedClasses` – Is a collection of classes whose instances are rarely loaded. These classes and their children should be moved to the end of a storage file to improve loading performance.

`theContainerSearchList` – specifies a collection of `DirRep` instances that represent directory search paths for resolving references between objects stored in `StorageContainer` files. When any title, library, or accessory container is opened, its directory is automatically added to this list. A title can add items to and remove items from this collection.

`theOpenContainers` – specifies all open storage containers, including library, title, and accessory containers. Containers are listed in the order in which they were opened, with the most recently opened container listed first in the collection.

Performance and Optimization

To preload an object is to bring it into memory from the object store before it is actually needed in the title. ScriptX allows a title to perform load management, to determine the timing with which objects are brought into memory. Load management techniques can be used to make smoother transitions in a program, for example, by bringing in a large media object before it is actually needed.

Performance is partly perception. Scene transitions can be carefully staged, so that the screen appears busy while other objects are being loaded. Putting up a splash screen can hide some of the time taken to preload objects.

You can preload objects used in your title's first scene in its startup action. A class or object's `afterLoading` method can be specialized to preload its subobjects. Preloading can run in the background—the title can activate a load operation in another thread.

Never assume that preloading objects improves performance. Preloading also takes time, and the net result may be worse. In general you get the best overall performance by not loading objects until they are actually needed, which means allowing them to load automatically as you use them. The payoff for preloading is greatest when you can switch active load management tasks into idle time.

Modules, which are really collections of code objects, are a special case in that they should always be preloaded. Generally, a title container should load all of its modules in its startup action.

```
-- the title container myTitle is a collection of modules
myTitle.startupAction := (myTitle -> for i in myTitle do load i)
```

Storage Reorganizer

Reorganization of objects on the storage medium can also improve performance. On some media, especially on CD–ROM, seek time is quite large. To minimize latency, use the Storage Reorganizer to make sure that objects are loaded in a consistent order.

The ScriptX Storage Reorganizer can improve performance if objects tend to have some locality in their loading order. For example, a pushbutton and its subpresenters, each of which is stored separately, are all needed at once when the pushbutton appears in a scene. By reorganizing the disk so that objects that are used together can be loaded together, you can minimize the number of seeks required to load a scene.

The loading order of individual objects does not have to be exactly the same. It is more important that groups of objects that are used together, such as all the objects needed to open one scene, are consistently loaded at the same time. See the "Title Analysis Tools" chapter of the *Development Tools User's Guide* for information about using the Storage Reorganizer.

Using Storage Containers

The examples in this section demonstrate how to manage objects in a container. They apply equally to all storage containers (including title, library, and accessory containers), although each of the examples in these features only one of the container classes.

Adding Objects to a Container

A storage container is a collection. Its target collection, by default, is an array, but other collection classes are commonly used, especially the explicitly keyed collections. Use collection methods such as `add`, `append`, and `prepend`, to add objects to a container. (`append` and `prepend` are defined by `Sequence`, and apply to sequences such as arrays and linked lists.)

```
global tc := new TitleContainer \
       path:"myTitle.sxt" \
       name:"myTitle"
global win := new Window name:"Welcome to My Title"
prepend tc win -- Window win is a target collection object in tc.
```

After creating a new container, you should explicitly add to the title container all the objects that you want saved in it that are not subobjects of objects you have already added to it. You can add subobjects of target collection objects,

but it is not necessary to do so to save those objects in the container. You may want to add some subobjects explicitly so that they can be accessed directly as members of the container's target collection.

In the following example, library container `myLibe` contains two target collection objects, both of which are arrays. Elements of those arrays are subobjects. Only the two arrays were explicitly added to the title container, but all the elements of each target collection array are also saved in the title container. Notice that there is only one of each object in memory; the object is not copied when it is added to the title container.

This example is extremely simplified to demonstrate adding objects and their subobjects to a container. In usual practice, title containers should consist only of modules, library containers consist of class definitions and media, and accessory containers consist of stand-alone tools and other title add-ons.

```
global myLibe := new LibraryContainer \
       path:"myTitle.sxt" \
       name:"My Title"
⇨ <LibraryContainer named "My Title">
global a := #("ScriptX", "is", "so", "cool")
⇨ #("ScriptX", "is", "so", "cool")
append myLibe a
⇨ 1
myLibe[1] == a -- there is only one version of this array in memory
⇨ true
a := #(7,28,95) -- reuse the global object
⇨ #(7, 28, 95)
prepend myLibe a -- add more objects to the title container
⇨ 1
myLibe.targetcollection
⇨ #(#(7, 28, 95), #("ScriptX", "is", "so", "cool"))
```

In the previous example, `append` and `prepend` are used to add objects to a container whose target collection is `Array` by default. The following example demonstrates use of the `add` method in a container that has been defined to have an explicitly keyed target collection.

Choosing a Target Collection

A storage container's target collection can be any ScriptX collection, but explicitly keyed collections are the usual choice, since they allow objects to be "named." The core classes supply several explicitly keyed collections: `KeyedLinkedList`, `SortedKeyedArray`, `BTree`, and `HashTable`. Each of these classes places some restrictions on what objects can be used as keys. For example, hash tables can only be used with keys that implement a hashing function, such as names and strings.

In practice, it is best to restrict the choice of keys to objects of a single class. The best choice is usually a `NameClass` object (a ScriptX interned name such as `@Maydene` or `@Sandra`). `NameClass` objects can be sorted and compared more efficiently than strings.

This example creates a title container with a hash table as its target collection:

```
global tch := new TitleContainer \
       path:"myTitleH.sxt" \
       name:"HashTitle" \
       targetCollection:(new HashTable)
⇨  <TitleContainer named "HashTitle">
add tch @car "sedan"
⇨  @car
add tch @pet "cat"
⇨  @pet
tch.targetcollection -- examine the target collection
⇨  #(@car:"sedan", @pet:"cat") as HashTable
tch[@car] -- retrieve an individual object
⇨  "sedan"
```

Saving Objects to a Container File

To save an added object to the container's underlying system file, call `update` on the object or, to save all of the container's objects (its target collection objects and all of their subobjects), call the `StorageContainer` method `update` on the container. To save all the container's objects and close the container, call `close` on the container. It is unnecessary to call `update` on the container before you close it, since `close` calls `update` automatically if the container was opened in `@update` mode.

When you save a single object, that object and all of its subobjects are saved to the file. The object on which you are calling `update` must be a persistent object. You can only call `update` on an object that has previously been added to a storage container.

```
update myLibe[1] -- update target collection object
⇨  #(7, 28, 95)
update myLibe[2][4] -- update only a subobject
⇨  "cool"
```

When you save a storage container, all of its objects (all of its target collection objects plus all subobjects of those objects) are saved to the file.

```
update tc -- update the entire storage container
```

Note – Saving an entire storage container can be very time-consuming, depending on the number and kinds of objects that are in the container. You may want to get into the habit of saving individual objects instead, especially when only a few of the container's objects have changed.

See also "Modifying and Deleting Objects in a Container" on page 386 for more examples of `update`.

Loading Stored Objects into Memory

Objects are loaded into memory on demand, when a method is called on them. Calling a method on a object in a container's target collection does not automatically load subobjects of that object. Calling a method on a subobject of some target collection object brings only the subobject into memory, not the entire target collection object.

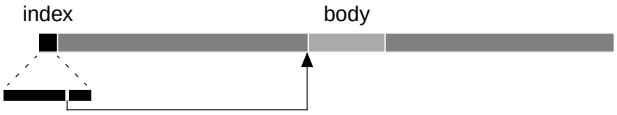
An object can be the parent of an entire tree of subobjects. For example, an instance of `PushButton` has several subpresenters, including a pressed, released, and disabled presenter. Each of these subobjects can in turn have subobjects. For example, the released presenter of a button may be a text presenter that presents a `Text` object.

The following example uses the library container that was defined in the section "Adding Objects to a Container" on page 383, to demonstrate which target collection objects and subobjects are in memory as various objects in the container are retrieved from the object store.

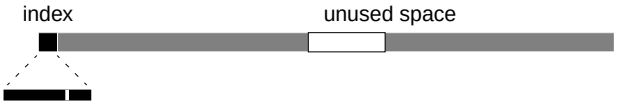
```
global myLibe := open LibraryContainer \
       path:"myLibe.sxt" \
       mode:@update
⇨  <LibraryContainer value of theTitleContainer named "My Title">
isInMemory myLibe[2]
⇨  false
myLibe[2][1]
⇨  "ScriptX"
isInMemory myLibe[2][1]
⇨  true
isInMemory myLibe[2][4]
⇨  false
myLibe[2][4]
⇨  "cool"
isInMemory myLibe[2][4]
⇨  true
isInMemory myLibe[2][2]
⇨  false
myLibe[2]
⇨  #("ScriptX", "is", "so", "cool")
isInMemory myLibe[2][2]
⇨  true
```

Modifying and Deleting Objects in a Container

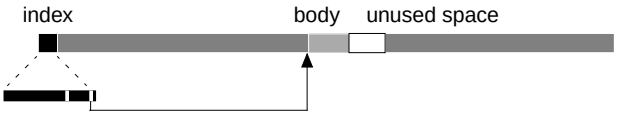
ScriptX objects are stored in blocks of space in storage containers. Large objects may be chained over several non-contiguous blocks. Each object in a container has an entry in the container's data index.



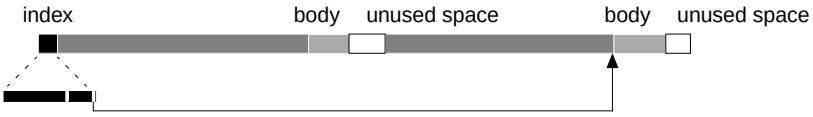
When an object is deleted from a container, the space it occupied can be used for another object of the same size or smaller. The size of a container file never decreases. Although unused space in a container can be reused by another object, it cannot be recovered.



When a new object is added to a container, the file does not necessarily increase in size. A newly created or modified object may be stored in space that has been reclaimed. However, the logical size of the container's data index always increases. (The physical size of the data index increases in arbitrary blocks.) The space for an index entry (which requires 4 bytes) cannot be recovered.



If no space that is large enough can be reclaimed, the container grows in increments of 4096 bytes. (The size of a physical block in a container is arbitrary and may change in future versions of ScriptX.)



To recover unused space while setting to null all index entries that are no longer valid, use the Storage Reorganizer. The Storage Reorganizer is a post-production tool that rebuilds storage containers, and rearranges objects in the container for optimal retrieval. If you provide no profile file, the Storage Reorganizer can serve as a simple compactor. See the "Title Analysis Tools" chapter of the *Development Tools User's Guide* for more information.

Modifying Persistent Objects

The following example, a continuation of the previous example, demonstrates the use of `update`.

```
tc[2] -- load a container object from storage
⇨ #("ScriptX", "is", "so", "cool")
tc[2][3] := "way" -- modify a subobject
⇨ "way"
tc[2] -- Show that the subobject is modified in the container.
```

```
⇨ #("ScriptX", "is", "way", "cool")
requestPurge tc[2] -- Purge without saving (updating).
⇨ undefined
isInMemory tc[2] -- Show that the object has been purged from memory.
⇨ false
tc[2] -- Reload the container object from storage.
⇨ #("ScriptX", "is", "so", "cool")
-- The modification made above is lost because
-- no update was done before the changed object was purged.
tc[1] -- Load another container object from storage.
⇨ #(7, 28, 95)
tc[1] := #(10,3,95) -- Modify the object.
⇨ #(10, 3, 95)
close tc -- Closing the container also updates the container.
⇨ true
tc := undefined -- Make sure the container is purged from memory.
⇨ undefined
-- Re-open the container.
tc := open TitleContainer path:"myTitle.sxt" mode:@update
⇨ <TitleContainer value of theTitleContainer named "My Title">
tc.targetcollection -- Confirm that its contents were saved.
⇨ #(#(10, 3, 95), #("ScriptX", "is", "so", "cool"))
```

Deleting Persistent Objects

Just as you use collection methods to add objects to a container, you can use the various forms of the delete collection methods to delete objects from a container's target collection. If you cannot delete a subobject directly, try creating a lower level binding to that object as shown in the example below.

```
deleteOne tc tc[1]
⇨ true
update tc
⇨ OK
close tc
⇨ true
tc := undefined
⇨ undefined
tc := open TitleContainer path:"myTitle.sxt" mode:@update
⇨ <TitleContainer value of theTitleContainer named "My Title">
tc.targetcollection
⇨ #(#("ScriptX", "is", "so", "cool"))
tc[2]
⇨ empty
deleteOne tc tc[1][3]
⇨ false
tc[1]
⇨ #(#("ScriptX", "is", "so", "cool"))
global a := tc[1]
⇨ #("ScriptX", "is", "so", "cool")
deleteOne a a[3]
⇨ true
x
⇨ #("ScriptX", "is", "cool")
tc[1]
⇨ #(#("ScriptX", "is", "cool"))
```

Note – Be careful when you delete a target collection object, since all of its subobjects are deleted also.

These delete methods have the same effect on the container's target collection as they have on other collections: Deleted objects are no longer members of the collection and cannot be referenced through the collection, but they are still held in storage. Thus, it is still valid for other objects to reference these deleted objects, even though the specified storage container collection cannot.

Removing Objects from Memory

When ScriptX runs there are two separate worlds of objects: the "persistent" (stored) world and the "transient" (non-stored) world. When you are finished using an object, or if you are temporarily not using it, you may want to remove it from memory to free up that memory for other objects to use. How you do this with stored and transient objects is quite different:

- With a stored object, you can "purge" it from memory, and recover it later from storage. A small part of the object, its handle, remains in memory so that all references to the object can be maintained.
- However, with transient objects, you can free them for garbage collection, but then must create new instances if you need them again.

Transient objects are fairly easy to understand and use—you free them by simply dropping all references to them. They will be freed in the next cycle of the garbage collector. Be aware that transient objects cannot be recovered once they have been removed from memory since they have not been stored in any container.

However, if you were to drop all references to a stored object, there would be absolutely no way to load it back into memory from storage, because you would not have a reference to it. So, instead of dropping references, you can free a stored object by "purging" it, using the `requestPurge` function.

Presenters are a special case of object, in that the process of displaying them prevents them from being garbage collected, even if they are not assigned to any variable or otherwise explicitly referenced. For example, if you have an displayed, empty, transient window with no other references to it, when you hide it, it will automatically be garbage collected. See the memory management section of this document for more information.

In previous releases of ScriptX, you needed to call `requestPurge` (formerly named `makePurgeable`) on a stored object in order to free it from memory. You no longer need to do this. In fact, you should be cautious in your use of `requestPurge` for two important reasons:

- The `requestPurge` global function, like the `update` method, operates on a tree of objects: the named object and all of its subobjects. Be sure you know what you are calling `requestPurge` on.
- You may use the object again between the time you call `requestPurge` on it and the time it is actually removed from memory. In this case, you may be using the object in an unknown state.

An example of when to use `requestPurge` is to free a large, static object that has no subobjects but is itself a subobject of an object that cannot be freed from memory yet because it is still being used. In most cases you should simply allow the system to remove the object from memory automatically when the object is no longer being used.

Calling `requestPurge` on an object registers a request to remove that object from memory, but an object will not be removed from memory while it is still in use. The object that you called `requestPurge` on, and any subobjects that are not being used, will be freed from memory in the first garbage collection cycle that runs after the object you called `requestPurge` on is no longer being used.

If a subobject of the object you called `requestPurge` on is still being used by some object other than the object you called `requestPurge` on, then that subobject and all of its subobjects will remain in memory and will not be garbage collected.

If you call `requestPurge` on an object and then use the object again before the object is actually removed from memory, then the request to remove that object from memory is cancelled. In this case you will be using the object in whatever state it was in when you called `requestPurge` on it, which may be different from the state that is stored in the container. So you need to be very careful in your use of `requestPurge`. For example, do not make a habit of purging all objects in a scene when you transition to a new scene. This can cause problems if your user transitions to a new scene and then quickly changes back to the previous scene before the garbage collector has had a chance to remove the scene's objects from memory. If you allowed your user to modify some objects in the scene, but your intention is to re-load the unmodified scene from storage each time the user transitions into it, then you may not get the results you want; the user may transition into a scene and find some old, changed objects and some freshly loaded objects. Forcing a full garbage collection cycle between scenes is not a good solution to this for performance reasons.

Note – Know what you are calling `requestPurge` on. In general, you should only call `requestPurge` on static objects (objects that do not change state during the title). Be extremely cautious about calling `requestPurge` on a tree of objects such as a scene or a window.

Note that `requestPurge` only applies to persistent objects (objects that have been added to some storage container). Use the global function `canRequestPurge` to test whether you can use `requestPurge` on a particular object.

Freeing of Persistent and Transient Objects

The following gives more detail into how objects can be stored and loaded from storage. All objects have two parts: a handle which points to its body. The handle is a kind of pointer to the body and is always 32 bits; the body is the rest of the object. When you drop all references to the handle of any object, the handle and body are automatically garbage collected.

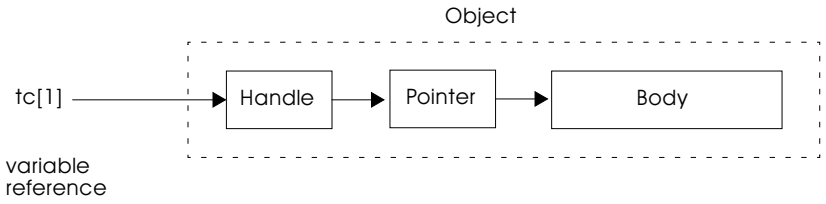


Figure 15-2: All objects have a handle and a body. The body can be freed when you call requestPurge. The handle and body can both be freed only by dropped all references to both.

Essentially there are two separate worlds of objects: the "persistent" (stored) world and the "transient" (non-stored) world:

- A transient object is created when calling `new`. If there are no references to it, it is automatically garbage collected. Because the object is not stored, the handle and body are always garbage collected together, as a pair.
- Stored objects are brought into memory from a storage container. A *handle* to a top-level object loaded from a container cannot be garbage collected, since the container holds onto it. (If you remove the object from the top level, then its handle can be garbage collected if there are still no other references to it.) A subobject can be garbage-collected if it is no longer reachable from core objects.

If you want to keep the handle of a stored object in memory (to maintain all its current references), but free its body, call `requestPurge` on the object. Requesting purge marks the object for the garbage collector to remove it from memory during the next cycle. It will likely be purged by the end of the next complete garbage collection cycle. (When all references are dropped from an object's handle, the entire object is garbage-collected at the next cycle; when an object is purged, its body is freed but not its handle.)

When you have a complete network of the persistent objects that can reach each other, and call `requestPurge` on all those objects in the network, both the bodies *and* handles for all those objects will be garbage collected.

Any method call on a stored object after it has been marked as purgeable will have one of two results (the one chosen is determined by garbage collector latency and substrate reference optimization):

- If the object was actually removed from memory, it will be reloaded from the storage container before the method is called on it
- If the object has *not* been removed from memory, the purge "bit" will be cleared and the method will be called on the original object still in memory

You must be prepared to deal with either case. The first case can happen if the method call happened before the complete garbage collector cycle ran, or if in the substrate some code has a direct reference to the object's body.

To reiterate, the purpose of `requestPurge` is to maintain references to the object so that you can again pull it into memory (otherwise, you could just drop all references to it).

Saving and Closing a Container

ScriptX provides two ways to save a container: the `update` method, which saves objects and keeps the container open, and the `close` method, which saves objects and closes the container. Libraries and accessories typically are closed automatically when the title that is using them closes, so you do not need to call `close` on them explicitly.

When you close or update the container, it saves (writes to storage) all objects that belong to the specified container (both target collection objects and subobjects of those target collection objects).

The `update` method saves a container without closing it. After you have created a container and added objects to it, you can update it (save it), add, delete, and change objects in it, and then update again.

```
update tc -- Saves the title container without closing it.
```

If what you are adding uses a lot of memory, you may want to purge it from memory before you add the next object. For this procedure, see “Removing Objects from Memory” on page 389 or “Adding Media to a Library” on page 413. The procedure is the same for a title, library, or accessory.

To save and close a title container, call `close` on it:

```
close tc -- Saves and closes the title container.
```

Calling `close` on a storage container automatically calls `update` on that container; you do not need to call both.

How Title Containers Work

A title container provides the startup activities for a title through a startup action, close activities through a terminate action, and access to system resources such as the ScriptX menu bar and clipboard. A title container also maintains a list of its windows, top clocks, and top players so the title as a whole can be paused, resumed, muted, opened, and closed.

When you create an instance of `TitleContainer`, it automatically creates a file for the title to be stored in. The user can open this file from the operating system (by double-clicking on its icon, for example) or from the ScriptX Player.

A title container is a collection that holds objects that can be loaded into memory and holds the startup action that is run when the user opens the title container file. A title container can include any kind of objects: visible objects, such as a window and pushbuttons, or nonvisible objects, such as clocks, controllers, classes, and functions. A title must display objects in windows (which are top presenters). The startup action can load these objects into memory and then initiate some action, such as opening a window or starting a player. See “The Title Startup Sequence” on page 394.

Once a title is started, the global variable `theTitleContainer` holds the instance of the title container that has user focus.

Useful Title Variables

The following global variables are useful for managing titles and are described in more detail in the *ScriptX Class Reference*.

`theContainerSearchList` – This global variable is a collection of `DirRep` instances that represent directory search paths for resolving references between objects stored in separate `StorageContainer` files. When any title, library, or accessory container is opened, its directory is automatically added to this list. A title can add items to and remove items from this collection.

`theOpenTitles` – This global variable is a collection of all open title containers, generally ordered by focus priority, which is the same order in which they appear visually on-screen.

`theTitleContainer` – This global variable represents the title container that currently has user focus. This means the system menu bar for this title container is visible (if the value of `isVisible`, an instance variable it defines, is set to `true`), and the title's frontmost window, if any, has user focus. This value is updated when the user (or a script) selects a title, making it active.

Opening and Closing ScriptX Titles

When your customers obtain a ScriptX Player title, they receive a title file on disk (often a CD-ROM). When they open this title file, it is important for ScriptX to have a systematic way to load objects from disk storage into memory and get the title running. Objects are automatically loaded into memory as they are used. You may want to preload some objects (such as those for the first scene) to make the display of your title's first screen as smooth as possible (see "Loading Stored Objects into Memory" on page 386).

The `startupAction` instance variable holds a function the developer writes that specifies which objects to load at startup. This instance variable is implemented by `LibraryContainer` class and inherited by `TitleContainer`.

In general, a title container holds all the objects and classes defined in the title, as shown in Figure 15-1. (Likewise, a library container holds the objects in a library, and an accessory container holds the objects in an accessory.) A title container is central to the startup sequence when a title is opened. Notice that a title is a kind of storage container, which is automatically and transparently managed as you manage your title container. In general, you can ignore the storage container and work only with the title container.

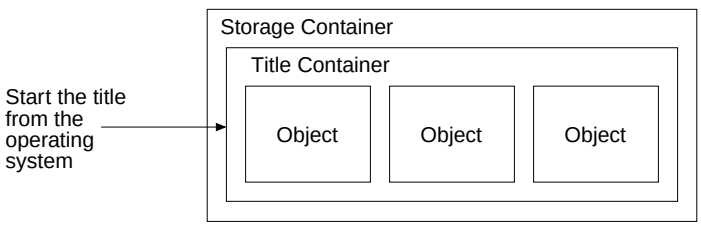


Figure 15-3: The title container contains the objects that make up the title

The Title Startup Sequence

The title startup sequence is the process of loading a ScriptX title and any associated libraries and accessories into memory from storage.

Before a title can open, the ScriptX Player must be running. If the user tries to open a ScriptX title when the ScriptX Player is not running, the operating system determines whether or not to automatically start the ScriptX Player. Currently, that is the case in all supported platforms.

A user can open a ScriptX title potentially several different ways, each of which invokes the class method `open`:

- From the operating system, by selecting the title file and choosing **Open** from the system or file menu, or by double-clicking on the title file.
- From the ScriptX Player, by choosing **Open Title** from the **File** menu and selecting the title file.
- From within another ScriptX title, by some user action that invokes the `open` method on the title container

When a title opens, the following steps occur:

1. The ScriptX Player loads into memory, if it is not already loaded. The mouse pointer changes to the `@wait` icon (a wrist watch on MacOS, an hourglass on Windows and OS/2) while the ScriptX Player is loading.
2. The ScriptX Player opens the title file, appends the value supplied with the `dir` keyword to `theContainerSearchList`, and then loads into memory the `TitleContainer` instance but none of its contents. The mouse pointer changes to an arrow.
3. After the container is loaded, the function specified by the instance variable `preStartupAction` runs. This step is skipped if `open` was called on a title that was already loaded.

   The prestartup action function, which is supplied by the developer, can check the environment for minimum system requirements, such as color depth and available memory. It can add paths to the container search list (which is stored in the global variable `theContainerSearchList`) for locating libraries that will be loaded in the next step (this global variable is a list of directories where the library files can be found).

   The prestartup function must return `true` or `false`.

   - If it returns `false`, an exception is reported, the title is purged, and the startup sequence aborts.
   - If it returns `true`, the startup sequence continues.

   If the prestartup function requires more than a few seconds, change the mouse pointer to the wait icon for the duration of the function:

   ```
   (new MouseDevice).pointerType := @wait
   ```

4. All libraries in the title container's `libraries` instance variable are loaded into memory in the order in which they appear in this list. The system calls `addUser` on the title for each library container to add that title to the library's `users` list and add the library to the title's `libraries` list.

   The `libraries` instance variable is the list of libraries that the developer has determined the title needs A developer can add a library to a title by specifying a `user` for the library when calling `new` or `open`, or by calling `addUser` directly. Libraries can contain any loadable classes or instances that support the title. The title searches automatically for libraries in directories specified in `theContainerSearchList`, which contains at least the title's directory and `theStartDir`. As each library is loaded, its directory is added to the container search list, if it is not already there, making it easier to find related files.

   Each library has its own startup sequence that it goes through when it is loaded, including adding the title to its `users` instance variable and running its own prestartup and startup actions. The `users` list ensures that the library won't be purged from memory while it is still being used.

5. The directory in which the container was found is copied to the `directory` instance variable. The value is either the value that was supplied with the `dir` keyword or, if the container was not found there, the container's directory taken from `theContainerSearchList` (the container's directory was added to `theContainerSearchList` when the container was opened).

6. The title is prepended to the global variable `theOpenContainers`.

7. The script stored in the `startupAction` instance variable runs.

   This function performs any other tasks you want to occur at startup. If the title's `targetCollection` is an explicitly keyed collection, you can define the startup function to access items in the title container by their item number (`tc[1]`), or key (`tc[@text]`). You can define global functions, variables, and constants used throughout the life of the title, determine the order in which other objects or files are loaded, and open windows, start a player, or perform any other startup task. You can also open one or more libraries that were previously loaded in step 4.

   You may want to prevent the startup action from being run more than once if `open` is called on the library by more than one title. To do this, define a startup flag that is initially `false`, enclose the startup statements in a conditional, and set the flag to `true` once the startup function has been run.

   When loading related files, you can use `theStartDir` to refer to the directory where the currently running ScriptX Player is located, or the `directory` instance variable to refer to the directory where an open title, library, or accessory is located. (For development only, you can use `theScriptDir` to refer to the directory where the last opened script is located.)

   In some cases, you may want to use the startup script to preload objects into memory (explicitly call the `load` method on them). You may want to do this to set up a scene to avoid waiting for each object to be loaded as a method is

called on it. Keep in mind that subobjects of objects that you call `load` on are not loaded into memory unless you call `load` or some other method on them. Objects are loaded into memory when a method is called on them. In general you get best performance when you do not preload objects but instead allow them to load automatically as they are needed. One case when you need to preload objects is to load the modules in a title container (see "Loading Stored Objects into Memory" on page 386).

8. The title container is given user focus by setting its `hasUserFocus` instance variable to `true`. This means its system menu bar is made visible, and its frontmost window is made active.
9. The global variable `theTitleContainer` is assigned this instance of `TitleContainer`.
10. The title container is prepended to the global variable `theOpenTitles`.

Once a ScriptX title is open, if the user goes back to the operating system and tries to open the title again, what happens depends on the operating system and is no different for ScriptX than for other programs. In Microsoft Windows, this action does nothing, since Windows will try to start another instance of ScriptX from the executable file, which is not possible. On the Macintosh, this action brings the currently running version of that title forward and gives it user focus.

The Title Close Sequence

The title close sequence is the process of closing a ScriptX title (and its associated libraries and accessories, if any), saving them to storage, and purging them from memory. Again, the title container is the focal point of the close sequence, in that closing a title container closes the libraries and accessories that it is using (if no other titles are using them). A title is closed ultimately by invoking the `close` method on the title container.

ose (menu
nmand)

A user can close a title two ways, both of which invoke the `close` method:

- From the ScriptX Player, by choosing the **Close** menu item from the File menu. This works on all title containers except the Scratch title, where this menu item closes only the single window that is currently active. (The Scratch title is described later.)
- From within a ScriptX title, by a user interface action that invokes the `close` method on the title container.

Note – Closing the windows of a title does not close the title.

When a title is closed, the following steps occur:

1. The `close` method is called. If the `user` keyword is supplied with a library, then `removeUser` is called to notify the library that the title is no longer using it.
2. If the title has any users (that is, if its `users` list is not empty), then the `close` method aborts.

3. Otherwise, the `close` method makes the title and all of its contained objects purgeable, and then calls the title's `terminate` method.
4. The `terminate` method does the following:
    a. Calls the function specified by the title's `terminateAction` instance variable.
    b. Removes the title from the global variable `theOpenContainers` list.
    c. Calls `close` on each of the libraries in the title's `libraries` list.
    d. Calls `hide` on all of the title's windows listed in `windows`, calls `playUnprepare` on all players in `topPlayers`, calls `pause` on all clocks in `topClocks`, and calls `close` on each accessory in its `accessories` instance variable.
    e. Calls `update` on itself.
5. The `close` method returns `true` to indicate the title did close down.

When you open a library container and use its objects in a title container, even though the objects become an integral working part of the title, they continue to be saved to their own library container, not to the title container.

For more details about saving and closing, refer to the section "Saving and Closing a Container" on page 392.

The Scratch Title

During development and playback of a title, a developer may want to temporarily create windows, clocks, and players that are not associated with any explicit title. For this purpose, the ScriptX development environment uses a scratch title that is automatically created at startup and cannot be saved. This scratch title is held in the global constant `theScratchTitle`. This scratch title is not present in the ScriptX Player.

When you create a new window, clock, or player and omit the `title` keyword, the object is added to the scratch title. You cannot add objects explicitly to the scratch title using collection methods as you can do with other storage containers. Although in general you cannot have the same object in two different title containers, you can put an object from the scratch title into a developer-defined title by calling a collection method on the object and the developer-defined title; you cannot *save* an object in two different storage containers (objects in the scratch container are not saved).

Using Title Containers

The following sections describe how to create, save, and close a title container, how titles manage their windows, how to pause, resume and mute a title. They also describe the menu bar, clipboard, managing libraries and opening multiple title containers.

Creating a Title Container

To create a title container, call `new` on the `TitleContainer` class, supplying the filename and directory where you want it saved.

Note – The extension `.sxt` is the convention for naming a title container file.

The default target collection of any container is an `Array`. To make it easier to retrieve objects, set the target collection to an explicitly keyed collection, such as `HashTable` or `KeyedLinkedList`. “Choosing a Target Collection” on page 384 demonstrates use of a `HashTable`.

The following script creates a title container, `tc`, and saves it to the filename `myTitle.sxt` in the directory `theStartDir`. Note that `theStartDir` represents the directory where the ScriptX Player is running. The global variable `theStartDir` is defined in the *ScriptX Class Reference*.

If this is a read-only device, such as a CD-ROM drive, the title container cannot be created unless you set the `dir` keyword to specify a read-write directory. (For a more complete sample script that demonstrates the `TitleContainer` class, see page 421.)

This example script creates a window, shows it, and prepends it to the title container. Finally, it creates a startup action—an anonymous function that runs when the title container is opened. At startup, this function loads the first item in the container, which is the window. This window is displayed automatically because it was showing when the title container was saved.

```
global tc := new TitleContainer \
       dir:theStartDir \
       path:"myTitle.sxt"
global myWindow := new Window
show myWindow
prepend tc myWindow
tc.startupAction := (t -> load t[1])
close tc          -- Closes and saves the title container
```

Calling `new` on `TitleContainer` creates the file named `myTitle.sxt` in the directory `theStartDir` on disk, visible to the user through the native operating system. The user can double-click on the file icon named `myTitle.sxt` to start the title.

Figure 15-4 demonstrates the relationships between the title container, its startup action, and the window `myWindow`. Other instance variables are not shown.

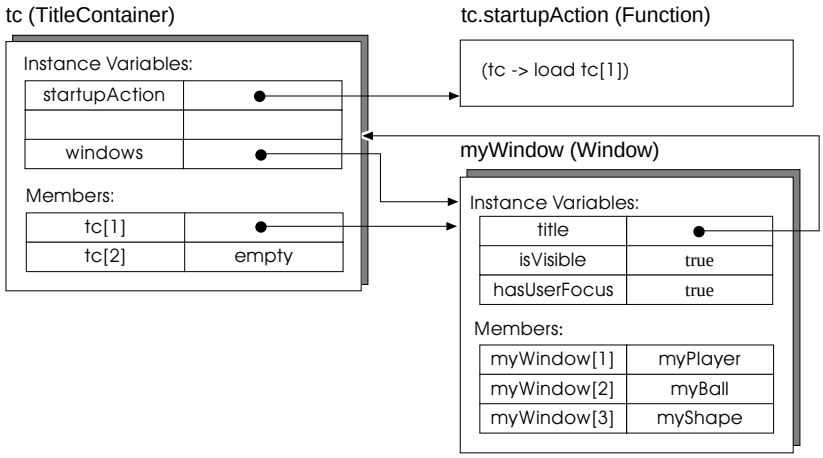


Figure 15-4: The title container holds a startup script and a window.

Saving and Closing a Title Container

`TitleContainer` implements two methods that save the title: `update`, which saves objects and leaves the container open, and `close`, which saves objects and closes the container. When you close or update a title container, it saves (writes to the object store) all objects that belong to the specified title, including target collection objects and subobjects of those target collection objects.

To update a title container without closing it, call `update` on it.

```
update tc -- saves tc without closing it
```

Once you have created a title container and added objects to it, you can update it repeatedly, adding, deleting, and modifying the objects it contains as you go. If you are adding a large object to a container, you might want to update the container and purge the object from memory before you add the next object. For more discussion of this procedure, see the sections “Removing Objects from Memory” on page 389” and “Adding Media to a Library” on page 413. The procedure is the same for a title, library, or accessory.

To save and close a title container, call `close` on it:

```
close tc -- save tc and close it
```

To confirm that your user really wants to close the title, create a subclass or specialized instance of `TitleContainer` and override `close` as follows. Pass a new Boolean argument (`withDialog` in this example) into the method.

- If `withDialog` is `true`, the `showCloseDialog` method opens a dialog box and asks “Okay to close title?”
- If `withDialog` is `false`, `showCloseDialog` closes the title without the dialog box.

The nextMethod expression calls the default implementation of close, defined by TitleContainer. The function showCloseDialog, not implemented here, should create a dialog box that allows the user to confirm that she wants to close the title.

```
class MyTitleClass (TitleContainer) end

method close self {class MyTitleClass} #rest args \
    #key withDialog:(true) -> (
        if (withDialog) do (
            if not showCloseDialog() do
                return false
        )
        return (apply nextMethod self args)
)
```

Each time close is canceled, it should return the title to the same state it was in when the user selected **Close**. Therefore, a close method should not release memory or perform any nonrecoverable operations. Use the terminate method and terminateAction instance variable for actions that release memory and are nonrecoverable.

Managing Windows, Clocks, and Players in a Title

A ScriptX title container can manage windows, clocks, and players. The following section demonstrates window management in a title. Most of this discussion (except for showing, hiding, and closing) applies similarly to clocks and players in a title.

Managing Windows in a Title

After a window is loaded into memory from a title container, it can be managed by that title. It can also be reassigned so that it is managed by a different title. A title manages windows as follows:

- Windows close automatically when the title is closed.
- Windows are not automatically opened when the title is opened. They can be opened in the title's startup action.
- A window shares user focus with the title that manages it. That is, if the title has user focus, its frontmost window has user focus; if the title does not have user focus, none of its windows can have user focus. When the user clicks on a window, the window's title container, including its system menu bar, gets user focus.
- The cutSelection, copySelection, pasteToSelection, printTitle, and clearSelection methods, when called on a title container, are passed on to its frontmost window.
- When the title is paused or resumed, the compositors for its windows are paused or resumed.

TitleContainer defines a windows instance variable that determines which windows it is managing. Similarly, Window defines a title instance variable that indicates which title is managing the window. A window is managed by a title if that title is the value of the window's title instance variable.

A title container's windows instance variable and a window's title instance variable are interrelated. They provide a cross-reference between a title and the windows that are managed by that title. A window's title list always contains one and only one item. The default value of title is the scratch title, which can be referenced by the global constant theScratchTitle.

```
myWindow.title -- returns the title that manages myWindow
```

A title container's windows list contains as many windows as are managed by that title. The value of windows can be empty. A window can be on many windows lists in one ScriptX session, but it can be on only one windows list at any given time. (Only one title at a time manages any given window).

```
myTitle.window -- returns list of windows managed by myTitle
```

A window's title instance variable is set automatically when the window is added to a title (see "Adding Objects to a Container" on page 383). The value of title can also be assigned explicitly. Operations on windows such as show, hide, bringToFront, and sendToBack cause the window's title to gain user focus. (The global variable theTitleContainer stores a reference to the title that has user focus.) Similarly, operations such as pause and resume on the window's clock cause the window's title to gain user focus. (Window inherits from TwoDSpace, which defines a clock instance variable.)

The title's windows list is transient. It is not saved with the title. It is automatically maintained by the system and should not be set programmatically. A window gets added to a title's windows list automatically when the window's title instance variable is set to that title.

A window's title instance variable can be set in one of the following ways:

- By adding the window to the title's target collection.
- By explicitly setting the window's title instance variable:

```
myWindow := new Window title:myTitle
-- or
myWindow.title := myOtherTitle
```

The title container's windows list is sorted by focus priority—by the order in which the user would see the windows on the screen if they were all showing. Therefore, to access the frontmost window of the frontmost title container, enter the following:

```
theTitleContainer.windows[1]
```

Do not add or delete windows directly in the `windows` list of the title. Allow the title to manage this list automatically. You can rearrange the title's windows by calling `bringToFront` or `sendToBack`, methods defined by `Window`.

A window is not necessarily saved to the title that manages it. A window can be on a title's `windows` list but not on the title's `targetCollection` list (see the example below). Windows that are stored in the title container's target collection, either directly or as subobjects, are saved automatically with the title.

Note – Assigning the value of `title` for a window does not cause that title to store the window—it does not make that window part of that title container. A window must be explicitly added to a title container.

Managing Clocks in a Title

Clocks and players, like windows, can be managed by a title. (Note that `Player` inherits from `Clock`.) Clocks can be "slaved" in a hierarchy, so that all clocks in the hierarchy are synchronized with the top clock. A ScriptX title can manage the top clocks and players in such hierarchies. The mechanisms by which a title manages clocks and players are the same as for windows. `TitleContainer` defines the instance variables `topClocks` and `topPlayers`, which it uses to maintain a list of the clocks and players the title is managing. `Clock` defines a `title` instance variable, which indicates which title is managing the clock if it is a top clock. For more information, see the section "Pausing, Resuming and Muting a Title" on page 405.

To save a clock to a title container, you need to add the clock to the title container (see "Adding Objects to a Container" on page 383). If you set the `title` instance variable on the clock, but do not add the clock to the title container, the clock will *not* be put in the title container.

After you add the clock to the title container, the clock's `title` instance variable shows which title the clock is in, and the title's `topClocks` instance variable lists the clock.

If you reassign a clock's `title` instance variable to a different title container object, the clock's `title` instance variable changes to the new title container and the title container's `topClocks` instance variable adds the clock to its list, but the clock is *not* added to the new title container's target collection and it is not removed from the original title's target collection.

```
global tc := new TitleContainer path:"clktest.sxt" name:"Clock Test"
⇨ <TitleContainer named "Clock Test">
global c1 := new Clock
⇨ Clock@0x1133668
tc.targetcollection -- The clock has not yet been added.
⇨ #()
tc.topClocks
⇨ #()
c1.title -- By default, the clock is in the Scratch title.
⇨ <TitleContainer named "Scratch Title">
```

```
append tc c1
⇨ 1
c1.title -- The append changes the title instance variable.
⇨ <TitleContainer named "Clock Test">
tc.targetcollection
⇨ #(Clock@0x1133668)
tc.topClocks
⇨ #(Clock@0x1133668)
tc.startupAction := ( tc ->
    c1 := tc[1]
    print "The clock's time is now"
    print c1.time
    c1.rate := 1)
⇨ #<ByteCodeMethod anonymous@0x113403c of 1 argument>
close tc
⇨ true
tc := undefined
⇨ undefined
tc := open TitleContainer path:"clktest.sxt"
⇨ "The clock's time is now"
⇨ 0:0:0:0 as Time
⇨ <TitleContainer named "Clock Test">
```

Window Management Example

This example shows the two cases when a window and a title are related (when a window is added to and operated on by a title, and when *window*`.title` is explicitly set), and it shows that a window can be managed by a title and still not be a member of that title.

```
global win1 := new Window
⇨ #<Window over #()>
win1.title
⇨ <TitleContainer value of theScratchTitle named "Scratch Title">
global tc1 := new TitleContainer path:"Title1.sxt" name:"Title1"
⇨ <TitleContainer named "Title1">
tc1.windows
⇨ #()
show win1
⇨ #<Window over #()>
tc1.windows
⇨ #()
prepend tc1 win1
⇨ 1
win1.title
⇨ <TitleContainer named "Title1">
tc1.windows
⇨ #(#<Window over #()>)
global win2 := new Window title:tc1
⇨ #<Window over #()>
win2.title
⇨ <TitleContainer named "Title1">
tc1.windows
⇨ #(#<Window over #()>, #<Window over #()>)
global tc2 := new TitleContainer path:"Title2.sxt" name:"Title2"
⇨ <TitleContainer named "Title2">
```

```
tc2.windows
⇨ #()
tc2.targetCollection
⇨ #()
-- Note the difference between windows and targetCollection:
tc1.windows
⇨ #(#<Window over #()>, #<Window over #()>)
tc1.targetCollection
⇨ #(#<Window over #()>)
prepend tc2 win2
⇨ 1
tc2.windows
⇨ #(#<Window over #()>)
tc1.windows
⇨ #(#<Window over #()>)
win2.title
⇨ <TitleContainer named "Title2">
win1.title
⇨ <TitleContainer value of theTitleContainer named "Title1">
```

Closing a Window

A user can close a window by clicking in the close box in the window frame or by choosing **Close** from the system menu (not from the ScriptX or ScriptX Player File menu). This calls `hide` on the window. Although people commonly speak of "closing a window," these operations call `hide`, not `close` on the window (`close` is for library containers).

The `hide` method, when called on a window, removes the window from the screen, sets the window's `hasUserFocus` to `false`, gives user focus to the next window onscreen, and disables the window's compositor.

Be sure not to confuse closing a window with closing a title. Notice that clicking the close box is not equivalent to choosing the **Close** command on the File menu. The **Close** command on the File menu closes the title and all its windows.

Freeing a Window from Memory

A window that is hidden (closed, from an end user point of view) is not freed from memory. To recover the memory being used by the window, stop using the window (make sure no methods are being called on the window or any of its subobjects). In general, you need to empty the window, hide the window, and undefine (or set to some other value) any variables whose value is the window. The window will then be freed from memory in the next garbage collection cycle. See "Removing Objects from Memory" on page 389 for more information.

Pausing, Resuming and Muting a Title

A title's `topClocks` instance variable is a list of all of the top clocks currently in memory that the title is managing. If a clock or player is "managed" by a title, then the clock's or player's `title` instance variable is set to that title, and the title's `topClocks` list contains that clock or player. Compare "Managing Windows in a Title" on page 400.

A top clock is a clock or player at the top of a timing hierarchy: its `masterClock` is `undefined`. When you call `pause` or `resume` on the title container, it in turn calls that method on the clocks and players in `topClocks`, causing their slave clocks and players to pause or resume.

A title's `topPlayers` instance variable also contains a list of all of its top players currently in memory. When you set `audioMuted` to `true` for the title container, it sets `audioMuted` to `true` for all `topPlayers` and all players slaved off of the players in `topPlayers`.

System Menu Bar

Each title has its own menu bar that appears at the top of the ScriptX Player, as shown in Figure 15-5. The menu bar is specified by the `systemMenuBar` instance variable in `TitleContainer`. This menu bar is created automatically as an instance of `SystemMenuBar` when you create a new instance of `TitleContainer`. When a title has user focus, its menu bar is displayed, replacing the previously displayed menu bar.

This menu bar has a default appearance, which you can modify slightly. You can hide or show the menu bar and enable or disable its menu items. You cannot add or delete options from the menus. The following shows how to disable a menu option:

```
disableItem myTitle.systemMenuBar @open
```

If you want two titles to have exactly the same menu bar, you can assign the same menu bar to both of them. Then, for example, whatever menu item is disabled for one title is also disabled for the other title:

```
myTitle2.systemMenuBar := myTitle1.systemMenuBar
```

The location and appearance of the system menu bar is platform-dependent, as shown in Figure 15-5. On the Macintosh, 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 is at the top of the screen only when the ScriptX window is "maximized").

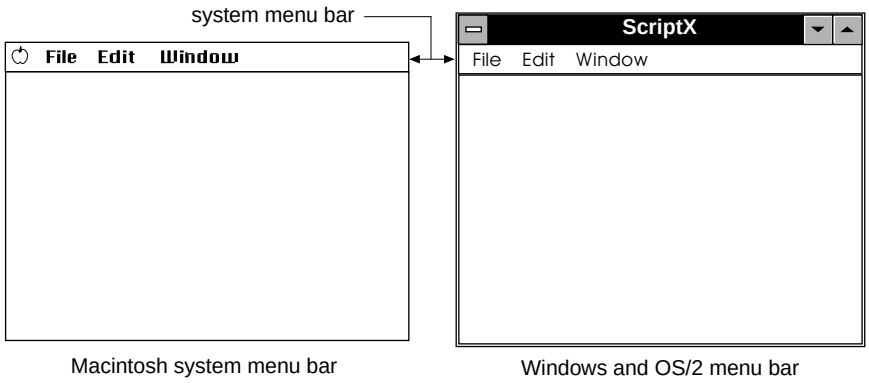


Figure 15-5: Menu bars appear in different places in different operating systems

Clipboard

The `Clipboard` class represents an area in memory that can hold text, so that text can be copied or pasted from one location to another. ScriptX defines just one instance of `Clipboard`, represented by the global constant `theClipboard`. Because the ScriptX Player automatically creates this clipboard at startup, there is no need to create another instance.

The clipboard allows the exchange of text between any two places—within a ScriptX title, between ScriptX titles, or between a ScriptX title and another application.

To put data on the clipboard, use `setClipboard`:

```
setClipboard theClipboard "I am on the clipboard."
⇨  "I am on the clipboard."
```

The clipboard supports the transfer of any ScriptX objects within a title or between titles, as the `@native` type. The current version of ScriptX supports only text from other applications, specified by the `@text` type.

To find out what kind of data is available from the clipboard, use the `typeList` instance variable, which is defined by `Clipboard`:

```
theClipboard.typeList
⇨  #(@native)
```

Once you know what kind of data is on the clipboard, you can ask for it by calling `getClipboard` on the clipboard:

```
getClipboard theClipboard @native
⇨  "I am on the clipboard."
```

The **Cut**, **Copy**, and **Paste** menu commands are standard user operations associated with the clipboard. These correspond to methods defined in `TitleContainer` and `Window`. When the user chooses **Cut**, for example, ScriptX calls `cutSelection` on the current title container, which calls

`cutSelection` on its frontmost window. You must provide an implementation for these methods for each particular window. For example, the `copySelection` method would eventually call `setClipboard` and `pasteToSelection` would call `getClipboard`. The implementation must include how 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 in the *ScriptX Class Reference*.

The operating system has its own clipboard, and every application also has its own clipboard. When using `cutSelection`, `copySelection`, and `pasteToSelection` within an application, such as the ScriptX Player, the local clipboard is used. However, when you switch applications, the system clipboard is used to transfer data between applications.

When the ScriptX Player is switched out, the ScriptX clipboard attempts to coerce its contents to text and place the text on the system clipboard. Likewise, when the ScriptX Player switches in, the ScriptX clipboard attempts to coerce the contents of the operating system clipboard to text and place it on the ScriptX clipboard.

The following scenario demonstrates how data types are typically handled with the clipboard between applications. If you were to copy data from another application, such as Macromedia Director®, to a ScriptX title, it would work as shown in Figure 15-6:

1. In Director, for example, copy the data from a document to its clipboard. The data remains in native Director data type.
2. Switch out of Director into the ScriptX Player (or ScriptX). The data is coerced to a system-standard media type, such as text, picture, or sound, and placed on the system clipboard (which overrides any data previously on the system clipboard).

   From the ScriptX Player, if you query the clipboard `typeList`, you get `@text`, because that's the only data type on the system clipboard that ScriptX currently recognizes.
3. In the ScriptX Player, paste the data from its clipboard into the title. At this time, the data is transferred from the system clipboard into the ScriptX clipboard and title.

Similarly, when copying data from the ScriptX Player to another application, the data is coerced to system data types when switching out of the ScriptX Player.

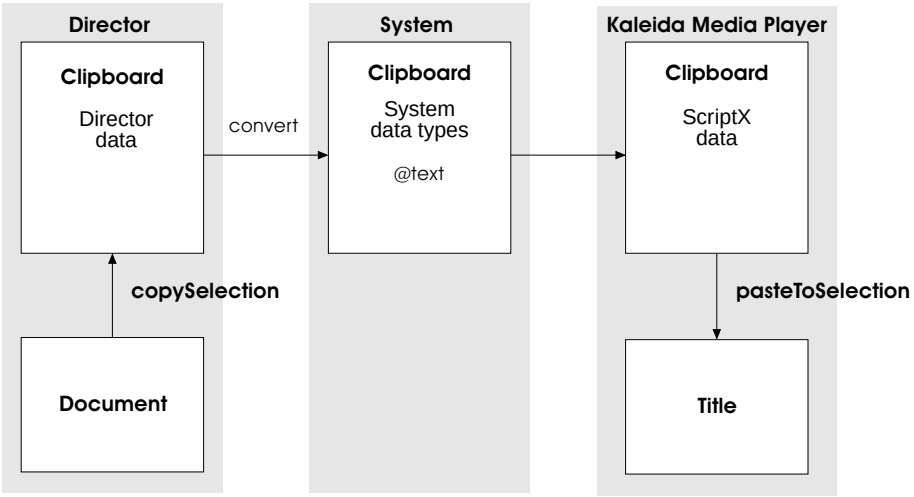


Figure 15-6: Copying data from another application to a ScriptX title

When switching out of an application (as in step 2 above), it is the responsibility of that application to make its clipboard data (text, graphics, sound, video) available to other applications by converting it to system-standard types. The system clipboard can simultaneously hold different types of media. The instance variable `typeList` lists the media types for the objects currently on the system clipboard that ScriptX recognizes (currently `@text`).

Clipboard media can come from only two places: the system clipboard or the ScriptX clipboard. If it comes from the ScriptX clipboard, the media is native to ScriptX, causing `typeList` to return `@native`. If it comes from the system clipboard, the media is not native ScriptX media (because any ScriptX object put onto the system clipboard is coerced to system text), causing `typeList` to return the appropriate type, such as `@text` (the only data type that ScriptX currently supports).

Managing Libraries in a Title

Use a library container's `users` instance variable to manage the relationship between a title and a library that the title needs to use. You can modify the library's `users` list by using the `user` keyword to `new`, `open`, or `close` or by calling the `addUser` or `removeUser` methods.

When you specify that a title is a user of a library, the title is prepended to the library's `users` instance variable, and the library is prepended to the title's `libraries` instance variable, as shown in the following example:

```
global myTitle := new TitleContainer path:"myTitle.sxt" name:"myTitle"
⇨ <TitleContainer named "myTitle">
myTitle.users
⇨ #()
myTitle.libraries
⇨ #()
global myLib := new LibraryContainer path:"myLib.sxl" user:myTitle
⇨ <LibraryContainer named undefined>
myLib.users
```

```
⇨ #(<TitleContainer named "myTitle">)
myLib.libraries
⇨ #()
myTitle.users
⇨ #()
myTitle.libraries
⇨ #(<LibraryContainer named undefined>)
```

Making a title container be a user of a library has two important effects:

- Since the library is on the title's `libraries` list, the library is automatically opened when the title is opened, if it is not already open, and closed when the title is closed, if no other title is still using it. See "The Title Startup Sequence" on page 394 and "The Title Close Sequence" on page 396.
- Since the title is on the library's `users` list, the library will stay open as long as the title is open (if you don't intervene by calling `removeUser` or `close` on the library explicitly while the title is still open). This feature is to protect against the library being closed by another title while your title is still using it. For example, suppose two titles are running simultaneously and both titles use the same library. When the first title is closed, it will attempt to close all the libraries on its `libraries` list, but it won't be able to close the library that is still being used by the second title because that library's `users` list is not yet empty.

The `users` instance variable is transient: It is not saved with the library, since a library cannot close until its `users` list is empty anyway. Only titles that are currently using the library are on that list; they are added to the `users` list when the title opens and removed from the list when the title closes.

The `libraries` instance variable is persistent. It is saved with the title, and the libraries it refers to are opened and closed automatically when the title is opened and closed.

If for some reason you want to open and close a library manually in the middle of a title and do not want the system to do it automatically, you must be careful to manage the `users` and `libraries` lists correctly to get the correct behavior described above. When you open the library, either specify the title as `user` in `open` or call `addUser` right after you open the library. Then be sure to call `removeUser` right before you close the library, or specify the title as `user` in `close`.

Figure 15-7 shows the relationship between titles and the libraries they use, between `users` and `libraries`.

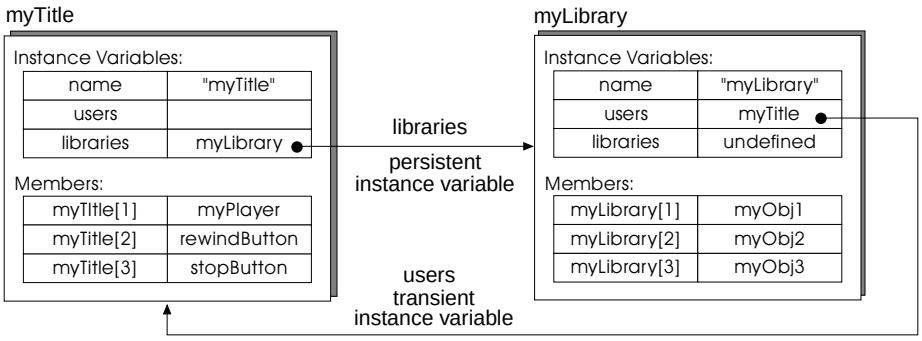


Figure 15-7: The title container myTitle depends on the library myLibrary.

Opening Multiple Title Containers

The ScriptX Player allows multiple titles to be open at the same time. Figure 15-8 shows this schematically. Each title has its own visual and audio portion. The visual portion of a title is composed of windows, each with its own presentation hierarchy. Each presentation hierarchy is composed of all the objects displayed in that window. Windows and presentation hierarchies are described in greater detail in Chapter 3, “Spaces and Presenters.”

The global variable `theTitleContainer` holds a reference to the title that has user focus. The system menu bar and windows of `theTitleContainer` title are in front of those of other titles. All open title containers, library containers, and accessory containers have a reference listed in the global variable `theOpenContainers`. All windows that are managed by a title container have a reference in the `windows` instance variable of that title container (see “Managing Windows in a Title” on page 400). All objects in a window are held in the `subpresenters` instance variable of that window.

When you create a new title container, it does not have user focus (its system menu bar does not appear) until you add a window to the title and display it. In contrast, when you open an existing title container, it automatically gains user focus.

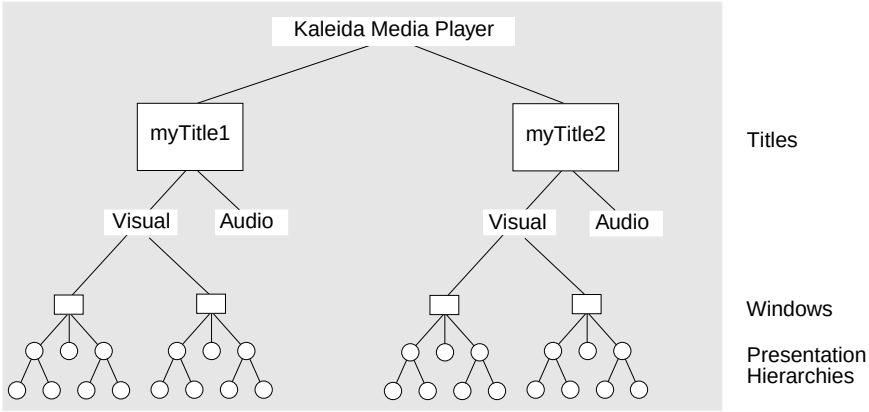


Figure 15-8: Each title has its own visual and audio portions.

Using Library Containers

In contrast to a title container, a library container has no menu bar or clipboard and cannot be paused, resumed, or muted. Instead, a library container supports a title container by providing media or code objects for the title. A library can be defined to automatically open and close when a title or library that uses it opens and closes (see “Managing Libraries in a Title” on page 408).

Like a title container, a library container is a collection that holds classes and instances that can be loaded into memory and has both a prestartup action and startup action that are run when the file is opened. A library container can include any kinds of classes or instances—visible objects, such as a window and pushbuttons, or nonvisible objects, such as clocks, controllers, modules, classes and functions.

A library container cannot have user focus. Any windows or presenters it contains must be managed by some title. If you do not specify a title container for a library container, the default title container `theScratchTitle` is used.

When deciding whether a library container should contain an instance or a class, you should take into account whether or not the library will be shared. For example, it would not normally make sense for the same instance of a 2D presenter to be used by multiple titles, since it can be in only one presentation hierarchy at a time. It would be better for the library to contain the class, and for each title to make its own instance. (Note that instances of `Bitmap` and other graphics primitives are not 2D presenters, and so have no such restriction and can be shared.)

Creating a Library Container

Creating a library container is similar to creating a title container. Call `new` on the `LibraryContainer` class and supply at least the filename to which you want it saved. You may also want to provide a directory and specify a title as a user of the library (see “Managing Libraries in a Title” on page 408). To make it easier to retrieve objects, set the target collection to a keyed collection, such as `HashTable` or `KeyedLinkedList` (the default target collection is an `Array`). The following example creates a library container, creates a new class called `Room`, prepends the new class to the library container and closes the library container.

Note – The extension `.sxl` is the filename convention for a library container.

```
global myLib := new LibraryContainer \
       path:"myLib.sxl"
class Room (Array) end      -- create a new class called "Room"
prepend myLib Room
close myLib
```

In the first statement, calling `new` on `LibraryContainer` creates the file named `myLib.sxl` in the `theStartDir` directory on disk, visible to the user in the operating system. Unlike title files, users cannot double-click on the file icon named `myLib.sxl` to open the library. A library can be opened only from within a title. The `close` method saves and closes the library container.

Using an explicitly keyed collection for the target collection allows the items in the library to be accessed by name. The following example is the same as the previous example except that it uses a hash table for the target collection:

```
global myLib := new LibraryContainer \
       path:"myLib.sxl \
       targetCollection:(new HashTable)
class Room (Array) end
add myLib @RoomClass Room
close myLib
```

When you want to get the `Room` class from the library, you can access it with `myLib[@RoomClass]`.

If a library container supplies a window, clock, or player to a title, you need to take an extra step to relate those objects to the title that manages them (see “Managing Windows, Clocks, and Players in a Title” on page 400). These objects all have a `title` instance variable that determines which title manages them. In general, you should set the `title` instance variable of windows to the title. If the clock or player is a top clock or player, you should set its `title` instance variable to the title, as described in “Pausing, Resuming and Muting a Title” on page 405. It is not necessary to set the `title` instance variable if the clock or player is slaved off a top clock or player.

You may not know all the titles that might want to use a library, so to avoid variable name conflicts between containers, it is generally good programming practice to ensure the library has its own namespace by giving the library its own module. Its module can also provide a convenient container for storing and retrieving code objects that are compiled in that module. For more information about modules, see the *ScriptX Language Guide*.

Opening and Closing a Library

You can define a library to open and close along with a title that uses it, or you can open and close a library independent of when a title opens and closes. There are some advantages to allowing the system to automatically open and close a library along with a title that uses it, so you will want to take steps to preserve these advantages if you open and close a library manually.

Recall that a title is a library (`TitleContainer` inherits from `LibraryContainer`), and see “Managing Libraries in a Title” on page 408 for details on opening and closing libraries automatically and manually.

A `TitleContainer` object has some specialized behavior that a `LibraryContainer` object does not have. A library cannot have user focus and `theTitleContainer` and `theOpenTitles` global variables do not change when you open a library. A library is not intended to be opened directly but rather to be used within a title.

Adding Media to a Library

Libraries are useful for storing media, such as video streams, audio streams, graphic images, and text. You can store any kind of objects in a library, but they are especially useful for storing large objects, and media objects usually are large. If the objects you are loading are very large, you probably want to purge them from memory whenever you are not using them to avoid running out of memory. You may want to preload some objects or keep some objects loaded that you are not currently using, but will be using again shortly, to avoid latency times.

You can import media from many standard data types and append the data to a library container, title container, or accessory container. For more detailed information on importing media, see the *ScriptX Tools Guide*.

Importing Media

To import media, use `importMedia`, a generic function defined by the `ImportExportEngine` class and specialized by each importer. Many of the importers for media classes define a `container` keyword argument, which is required for saving certain kinds of media to a container. Media falls into two groups:

- Non-compressed bitmaps. These do not require a `container` keyword, and must fit entirely into memory.
- Compressed bitmaps, audio, and video. If you want to save this media to a container, you must specify the `container` keyword when you call `importMedia`. Omit the keyword if you are not saving the media. The size of these files can be greater than will fit into memory—the importer can import the data in pieces and save them to the container you specify.

The `container` keyword specifies the library, title, or accessory container where you want to save the data. Whether the `container` keyword is supplied or not, you must separately append the media to the library, title, or accessory container.

For example, to import into memory a black-and-white `PICT` image from a file named `bitmap.pict`, call `getStream` and `importMedia`:

```
global myPictStream := getStream theStartDir "bitmap.pict" @readable
global myPict := importMedia theImportExportEngine myPictStream \
       @image @pict @bitmap
```

The `getStream` method opens a stream from the file `bitmap.pict`. The `importMedia` method imports the stream into memory. The next section shows how to save the `PICT` image to disk and purge it from memory.

Saving and Purging Media

If you import a series of large files to a container, you may need to release each one from memory after it has been saved to storage and whenever you are not using it to make room for the next file in memory. The following sample script shows how to do that, using the media `myPict` imported from the previous script:

```
global myPictStream := getStream theStartDir "bitmap.pict" @readable
global myPict := importMedia theImportExportEngine myPictStream \
       @image @pict @bitmap

global myLib := new LibraryContainer \
       path:"myPict.sxl" \
       targetCollection:(new HashTable)
add myLib "myPict" myPict
update myPict
-- use myPict here
requestPurge myPict
```

The following describes the steps shown in this example:

1. The `importMedia` method imports a copy of the bitmap.
2. The `new` method creates the library container as a hash table.
3. The `add` method adds the bitmap with the key `"myPict"` to the library container.
4. The `update` method saves a copy of the bitmap to the library file.

   At this point the bitmap is saved, and we want to release the bitmap from memory so that we can import the next bitmap.
5. The global function `requestPurge` prepares the bitmap for garbage collection.

Using Accessory Containers

An accessory is a set of code, data, and media objects that can be dynamically added to a running title. An instance of `AccessoryContainer` is meant to be opened from within a title and its objects added to the title. It is intended to incrementally add data or functionality to titles.

Like a title and a library, an accessory has both a prestartup action and startup action that run when the file is opened. Like a library, an accessory may be used in only one title or in many titles.

A library container cannot have user focus. Any windows or presenters it contains must be managed by some title. If you do not specify a title container for a library container, the default title container `theScratchTitle` is used.

Recall that an accessory is a library (`AccessoryContainer` inherits from `LibraryContainer`). An `AccessoryContainer` object has some specialized behavior that a `LibraryContainer` object does not have. An accessory

cannot have user focus and values of the global variables `theTitleContainer` and `theOpenTitles` do not change when you open an accessory. An accessory is not intended to be opened as a stand-alone program but rather to be used within a title. A user can directly open an accessory from the ScriptX **File** menu, but not a library. An accessory is a program that a user chooses to add to a title, while a library is something a title itself opens.

Examples of accessory containers include a tape measure that can be used to measure the size of objects, or an object inspector that can analyze the state of objects in a title. The accessory container is an ideal vehicle for adding value incrementally to an existing multimedia title.

In addition to what it inherits from `LibraryContainer`, the `AccessoryContainer` class defines one additional method, `getAccessory`, that returns a collection of objects that the title can use. The `TitleContainer` class defines three methods for handling accessories: `isAppropriateAccessory`, `addAccessory`, and `removeAccessory`.

The default implementation of `addAccessory` does nothing. The scratch title, the default title that is always open and cannot be opened or closed, cannot use accessory objects. It is only useful to open an accessory inside a developer-defined title. It is up to the developer to specialize `addAccessory` at the class or instance level to make use of accessory objects.

The responsibility for determining which accessories are suitable in a title rests with the title. Specialize `isAppropriateAccessory` to determine which accessories to accept or reject.

An accessory can be written to interact in various ways with titles. It might work with only one title at a time or with any number of titles. It might remove itself from its current title if another title tries to add it, or it might create a new instance of its objects for that second title.

As with a library container, when you decide whether an accessory container should contain a particular instance or class, you should consider whether or not the accessory will be shared. Presenters cannot be Instances of `Bitmap` and other graphics primitives are not presenters, so they can be shared.)

As with a library container, you must take special care if an accessory container supplies a window, clock, or player to a title. These objects define a `title` instance variable, which determines which title manages them. The `TitleContainer` class in turn defines the instance variables `windows`, `topClocks`, and `topPlayers` to manage interaction with windows, clocks, and players. For more information, see the section “Managing Windows, Clocks, and Players in a Title” on page 400.

It is good programming practice to insure that an accessory has its own namespace by giving the accessory its own module. You might not know all the titles that would want to use an accessory, so to avoid conflicts in variable names between containers, compile the accessory in its own module. For more information on modules, see the *ScriptX Language Guide*.

Creating an Accessory Container

In general, to create an accessory, subclass `TitleContainer` to override `addAccessory` and `isAppropriateAccessory`, and subclass `AccessoryContainer` to override `getAccessory`. When the accessory is opened, these methods get called automatically or not, depending on how the accessory is opened:

- If the user chooses the accessory from the **Open Accessory** menu command:
  - The `open` method is called on the accessory with `theTitleContainer` as its user.
  - The `isAppropriateAccessory` method is called. If it returns `true`, it calls `addAccessory`. Otherwise, the accessory is not added to the title, but is left open.
- If the user opens the accessory through some user interface action that is built into the program, these methods do not get called automatically, but you can script for them.

The following example describes how to set up an accessory, assuming the user uses the **Open Accessory** menu command, as illustrated in Figure 15-9.

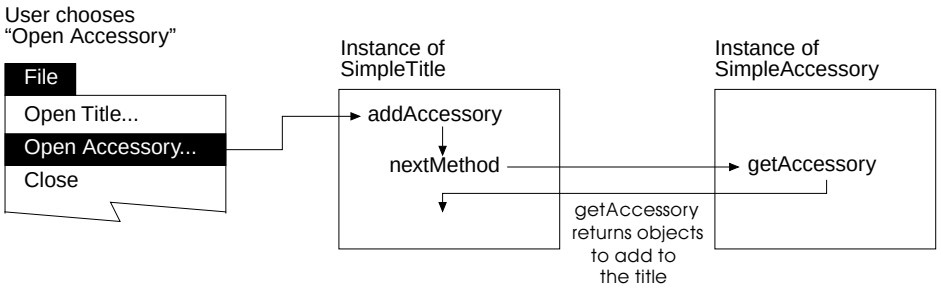


Figure 15-9: Opening an accessory calls a sequence of methods.

This example creates a subclass of the `AccessoryContainer` class called `SimpleAccessory`, which overrides `getAccessory` to return a list of accessory objects to be used by the title container. In this case, it returns a list that contains a circle.

Then the example creates an instance of `SimpleAccessory`. This creates a file on disk, visible to the user in the operating system. The name of the file, `simple.sxa`, is supplied with the `path` keyword. By default, the accessory is saved in the `theStartDir` directory. The user cannot double-click on the operating system file icon named `simple.sxa` to start the accessory; this feature is available only for titles, not accessories.

Note – The extension `.sxa` is the convention for the filename of an accessory container.

```
class SimpleAccessory (AccessoryContainer)
end
```

```
method getAccessory self {class SimpleAccessory} -> (
    -- Create a circle
    local myCircle := new TwoDShape target:(new Oval x2:50 y2:50) fill:blackBrush

    -- Create the list to be returned by getAccessory
    local accList := new HashTable
    add accList @circle myCircle
    accList
)

-- Create an instance of the accessory, then close and save it
global ac := new SimpleAccessory path:"simple.sxa" name:"Simple Accessory"
close ac
```

After executing this script, the accessory container is created. Next, the title container must be able to handle the accessory. The following script creates a subclass of `TitleContainer` called `SimpleTitle`. It overrides the `addAccessory` method to put the object returned by `getAccessory` (which in this case is a circle) into its window.

The first line in the body of `addAccessory` contains `nextMethod`, which calls `addAccessory` on the superclass `TitleContainer`, which calls `getAccessory` on the accessory. The call to `nextMethod` returns the same list that `getAccessory` returns. The `prepend` method uses the key `@circle` to get the circle from the list of accessories and adds that circle to the window.

```
-- Define a subclass of TitleContainer
class SimpleTitle (TitleContainer)
end

-- Get the objects from the accessory
method addAccessory self {class SimpleTitle} accContainer -> (
    local accList := nextMethod self accContainer

    -- Add the @circle item from the accessory to the frontmost window
    prepend theTitleContainer.windows[1] accList[@circle]
)

-- Create the title container and add the window to it
global tc := new SimpleTitle dir:theScriptDir path:"simple.sxt"

-- Create a window and add it to the title container
global myWin := new Window
show myWin
append tc myWin

-- At startup, load the objects in the title container
tc.startupAction := (tc -> forEach tc load undefined)
```

After you execute this title container script, the title container window is open. You can then test the accessory by choosing **Open Accessory** from the File menu and selecting `simple.sxt`, which causes the circle from the accessory to appear in the window.

Opening an Accessory File

This section summarizes the procedure to open an accessory container.

The user selects an option to open a particular accessory, either by choosing **Open Accessory** from the File menu, or by performing some other user interface action in the title that calls `open` on the `AccessoryContainer` class.

If the user did not choose **Open Accessory**, but instead performed some other action in the title, then the title must be responsible for calling `open`, `isAppropriateAccessory`, and `addAccessory`.

The **Open Accessory** menu command:

1. Calls `open` on the `AccessoryContainer` class, supplying the value of the global variable `theTitleContainer` as its user.
   - The `open` class method loads the accessory container into memory, but none of its contents.
   - The `open` class method then calls the function in `preStartupAction`, loads all files listed in its `libraries` instance variable, and makes the accessory a user of each library.
   - If the `user` keyword is supplied with `open`, this method then calls `addUser` on the supplied title container with the library as its new user.
   - The `open` method then adds a reference to the accessory to the list maintained by `theOpenContainers` global variable, and calls the function specified by `startupAction`, as defined by the developer.
2. Calls `isAppropriateAccessory` on `theTitleContainer`.
   - The title container's `isAppropriateAccessory` method checks to see whether the accessory is compatible with the title, to determine whether to load the objects contained in the accessory container.
   - If `isAppropriateAccessory` returns `true` is returned, the system calls `addAccessory`.
   - If `isAppropriateAccessory` returns `false` is returned, the accessory is not added, but is left open.
3. The `addAccessory` method calls `getAccessory` on the accessory. The `getAccessory` method should be specialized to return a collection of objects added to the title. Specialize `addAccessory` to do whatever is needed with the objects from this collection to start the accessory—create new objects, add presenters to windows, or start clocks and players.

Quitting ScriptX

To quit the ScriptX Player, call the global function `quit`. When the user quits the ScriptX Player, the Quit Manager takes control before the entire object system shuts down. The Quit Manager gives a developer an opportunity to run one or more scripted tasks before closing the system.

Quit queries and quit tasks are ScriptX functions that run automatically before the ScriptX Player shuts down. When a user quits, ScriptX does not automatically inform title containers. If your title or accessory needs to be informed of an impending shutdown, you can use quit queries and tasks to notify the open titles to clean up, save data, and close. In addition, they can save preferences or state information, show a credits screen, play an audio message, or release resources of the underlying operating system.

By default, the quit process has no user interaction. If you want to prompt users about saving their work, or if you want to make sure they want to quit, you would do that in a quit query.

Any object or process can create quit queries or tasks, and there are no explicit limits on the number that can be installed. Objects that exist outside the title container framework may need to create a query or task to ensure that they are saved, closed, and disposed of properly. In general, if your title installs quit queries or tasks, it should remove them when it closes.

The user quits the ScriptX Player by choosing **Quit** (MacOS) or **Exit** (Windows and OS/2) from the **File** menu, or by performing some other user interface action that calls the global function `quit`. When the user quits, the system first runs all quit queries. If all queries return `true`, it runs all quit tasks. Both quit queries and quit tasks run in LIFO (last-in, first-out) order. To add or remove quit queries and quit tasks, use the global functions `installQuitQuery`, `installQuitTask`, `deinstallQuitQuery`, and `deinstallQuitTask`.

It is often important to perform a query when closing a title (as opposed to when quitting the ScriptX Player), such as asking the user "Okay to close title?" To do this, you should create a subclass of `TitleContainer` and override the `close` method. For more information, see the section "Saving and Closing a Container" on page 392.

The following two sections describe the differences between quit queries and quit tasks.

Quit Queries

A quit query is a function that determines whether or not it is okay to quit. It should return `true` to continue quitting, or `false` to abort the quit process. A quit query should *not* take any action that is final, such as to purge objects from memory or set useful variables to `undefined`—that should be done later by a quit task.

A typical use of a quit query is to display a dialog that the asks the user, "Okay to quit?" as shown in Figure 15-13. If the quit query returns `false`, shutdown is aborted, and execution returns to the ScriptX Player. A quit query is the appropriate place to ask if the user wants to save changes to a file.

You would probably install a quit query that asks “Okay to quit?” immediately at the startup of a title, so that at any later time when the user chooses **Quit** (MacOS) or **Exit** (Windows and OS/2), a prompt would appear.

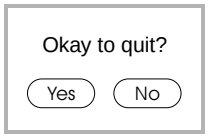


Figure 15-10:A dialog box called by a quit query.

The point of separating quit queries from quit tasks is that only quit queries should allow the user to abort the quit process. If the user answers “No” to the previous question and attempts to quit again, the Quit Manager will repeat all quit queries from the beginning. Thus, a quit query may run several times before the system shuts down. You should write queries so the user can run them repeatedly. Quit tasks should run only once, with no turning back.

The global function `installQuitQuery` allows you to assign a function to be run when the user quits the ScriptX Player. You should write it to take one argument and to return true or false. The following example is taken from the full, working sample script in the section “Quitting ScriptX Gracefully” on page 431.

```
function quitQueryFunc notUsed -> (
    deinstallQuitQuery myQuery
    local myWindow := new YesNoWindow centered:true \
        boundary:(new Rect x2:200 y2:120)
    show myWindow
    false       -- Necessary so ScriptX will not quit yet
)

global myQuery := installQuitQuery quitQueryFunc undefined
```

The global function `installQuitQuery` takes two arguments: the name of the function to call (`quitQueryFunc`) and its first argument (`undefined`). It returns a unique integer that you can use later to deinstall the quit query:

```
deinstallQuitQuery myQuery
```

Quit queries can be challenging to debug because an error in the quit query function may prevent ScriptX from quitting at all. You then have to either remove the quit query to allow you to quit, or quit ScriptX using some technique defined by the operating system or hardware.

Quit Tasks

Quit tasks run only after all quit queries have run successfully and have returned `true`. A typical use of a quit task is to save preference or state information that can be restored when the ScriptX Player starts up again. Once quit tasks begin running, shutdown is imminent and cannot be aborted. Each quit task runs once and has no return value.

The following script creates a function called `putAway` to close a library container. The `installQuitTask` global function adds a quit task that will call `putAway` on a library container called `prefsFile`.

```
function putAway libraryName -> (
    if (isAKindOf libraryName LibraryContainer) do (
            close libraryName
        )
    )
global myPrefs := installQuitTask putAway prefsFile
```

The global function `installQuitTask` returns an integer that uniquely identifies the quit task. This integer can be used anytime after installing the quit task to remove the quit task:

```
deinstallQuitTask myPrefs
```

Quit queries do not suspend the thread system, but quit tasks do. Processes that run in other threads, such as callbacks or animations, remain active until all quit queries return `true`. When the quit queries have finished and the quit tasks begin running, the ScriptX Player suspends the thread scheduler. From that point, only quit tasks can run until the system shuts down. Quit tasks are the logical place to clean up, to discard references, and to let go of hardware resources.

Title Management Examples

You use the Title Management component for the high-level organization and file storage of titles, libraries, and accessories. In the title development process, once you have written the scripts that create the windows, add 2D presenters, add the interactivity, and import the media, you use the Title Management component to store the compiled code as files that can be run by the ScriptX Player.

The following working scripts are included in this section:

- A Simple Title (page 421)
- A Painting Title and Library (page 423)
- A Painting Title and Accessory (page 426)
- Quit Dialog Box (page 431)

A Simple Title

The following two scripts demonstrate how to display the text “Hello, world” in a window, as shown in the following figure, and store it in a title container. (The *ScriptX Quick Start Guide* contains an example very similar to this one.)

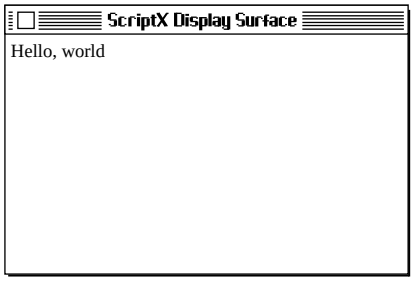


Figure 15-11:A simple title composed of a window with text.

Displaying Text in a Window

First, create a window and add the text “Hello, world” to it.

```
-- Filename: simple.sx
-- Create a window and show it
global myWindow := new Window  boundary:(new Rect x2:400 y2:300)
myWindow.y := 40
show myWindow

-- Create text and add it to the window
global myText := new TextPresenter target:"Hello, world" \
                  boundary:(new Rect x2:200 y2:50)
append myWindow myText
```

At this point you could center the text, or add other objects to the window, if you wanted.

Saving the Title to Disk

The next step is to create a title container, append the window to it, write the startup script that loads the window on startup, and close the title container. Closing a new title container automatically saves it. Notice that the filename ends in `.sxt`, indicating that it contains a title container.

The startup script, `tc.startupAction`, is defined as an anonymous function with one argument, `t`, that represents the title container. At startup, this function is executed, which loads into memory the first item in the title container, which is the window. The window is automatically displayed when it is loaded, since it was showing (`isVisible` was `true`) when the title container was closed.

```
-- Create a title container. Notice the filename ends in .SXT
-- indicating it contains a title container.
global tc := new TitleContainer dir:theScriptDir path:"SIMPLE.SXT"

-- Move the window to the title container (from the scratch title)
```

```
prepend tc myWindow

-- Include the startup script, which is an anonymous function
tc.startupAction := (t -> load t[1])

-- Close the title container
close tc
```

Closing the title container updates the container to include any changes, then closes the file on disk.

Testing the Simple Title

To test that the window has indeed been stored properly, execute the previous script in ScriptX, which creates one file: `simple.sxt`. Then quit ScriptX and do the following:

Open up the file `simple.sxt` in ScriptX. You can do this by selecting the file in the operating system and choosing **Open** from the File menu, or by double-clicking on its icon.

This should start ScriptX, and run `startupAction`, which loads the first item of the title container into memory, making the original window appear.

When you are done, choose **Close** from the File menu to close the title.

A Painting Title and Library

This is a simple example of a title container that depends on a library container. To run this example requires a bitmap file named `Gauguin.bmp` (this can be any bitmap file in Windows `DIB` format).

The title container has a module that defines a window that displays a bitmap, as shown in Figure 15-12. The bitmap is stored separately in a library container file. The title container is set up to automatically open and close the library file when the title is opened and closed.

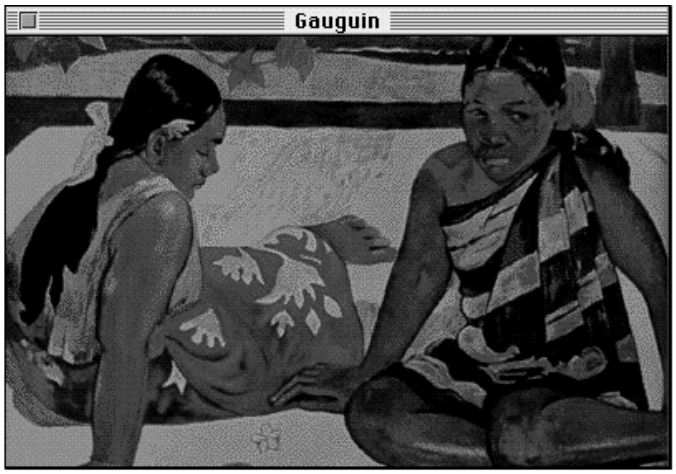


Figure 15-12:The title provides the window; the library provides the bitmap.

Creating the Title File

The following script creates a module that defines a window and puts a 2D shape in the window, to eventually hold a bitmap. Then the script creates a title container file called `painting.sxt`, adds the module to the title container, and creates a startup action script for the title. This startup script loads the module, then gets a bitmap from the library and assigns the bitmap to the `target` instance variable of the 2D shape.

```
-- Filename: painting.sx

-- Create the window
global myWindow := new Window boundary: (new Rect x1: 0 y1: 40 x2: 400 y2: 300)
myWindow.name := "Gauguin"
show myWindow

-------------------------------------------------------------------------------
-- Create and append a TwoDShape to the window
global myShape := new TwoDShape
append myWindow myShape

-------------------------------------------------------------------------------
-- Create the title container and add the window to it
global tc := new TitleContainer dir:theScriptDir path:"painting.sxt" \
    name:"Painting Title"
append tc myWindow

-- Define the startup action
tc.startupAction := (
    -- Load all objects in the title container
    tc -> forEach tc load undefined

    -- Get the library container
    local myLC := chooseOne theOpenContainers \
                  (v a -> v.name = "Painting Library") 0

    -- Assign the first bitmap in the library to the target
```

```
    -- of the 2D shape in the frontmost window
    tc.windows[1][1].target := myLC[1]
)
```

Creating the Library File

The following script imports the bitmap file `Gauguin.bmp` into a bitmap object, which is appended to a library file called `painting.sxl`. The library is then created with the `user:tc` keyword argument, which causes the library container `lc` to be added to the `libraries` instance variable of the title container `tc`. The `libraries` instance variable is saved when `tc` is closed. Later when the title container is opened, this reference is used to automatically open the library container.

For the title container's `libraries` instance variable to save the library, it is important that the title container be open when the library container is created, and that the title container be saved (closed) before the library container is closed.

```
function getPict fileName -> (
    local myStream := getStream theScriptDir fileName @readable
    local myImage := importMedia theImportExportEngine myStream @image @dib \
       @bitmap colormap:defaultColormap
    myImage
)

global p1 := getPict "GAUGUIN.BMP"

-- Create the library, and append the picture to it
-- Note that this library uses the title container tc defined previously
global lc := new LibraryContainer dir:theScriptDir path:"painting.sxl" \
    name:"Painting Library" user:tc
append lc p1

-- Close the title container and then the library container
close tc
close lc
```

Testing the Painting Title and Library

To test that the title and library containers work together properly, execute the previous scripts in ScriptX—two files are created: `painting.sxt` and `painting.sxl`. Then quit and do the following:

Open up the file `painting.sxt` in the ScriptX Player. This should load the title container and library container into memory, and run `startupAction`, which displays the window and the painting bitmap from the library container.

When you are done, choose **Close** from the File menu to close the title along with its associated library.

A Painting Title and Accessory

This example is a modification of the previous example—it includes an accessory that allows the user to scale the bitmap smaller or larger by moving the mouse left or right. To exit the accessory, click the mouse, and the bitmap is left at its scaled size.

The following description emphasizes the differences from the previous example. To run this example requires the same bitmap file named `Gauguin.bmp` from the previous example.

This example creates three files: a title container, the same library container as the previous example, and an accessory container.

Creating the Title File

As in the previous example, the following script creates a window and puts a 2D shape in the window—the 2D shape will later hold a bitmap.

Unlike the previous example, this example creates a subclass of `TitleContainer` called `ExtendableTitleContainer`, in which the method `addAccessory` is overridden. This method first calls `nextMethod`, which calls `addAccessory` on its superclass `TitleContainer`, which calls `getAccessory` on the accessory and returns a list of objects from the accessory, which is assigned to the variable `accList`.

The `prepend` method gets the item in `accList` whose key is `@text`, and adds it to the window in the title container that is holding the accessory container.

The `removeAccessory` method removes the objects that were added in `addAccessory`.

The `terminateAction` instance variable is assigned the function `closeAccessory` that is defined to close the accessory.

The `startupAction` is identical to the previous example.

```
-- Filename: paintacc.sx

--- Create the window
global myWindow := new Window boundary: (new Rect x1: 0 y1: 40 x2: 400 y2: 300)
myWindow.name := "Gauguin"
show myWindow

-- Create and append a TwoDShape to the window
global myShape := new TwoDShape
append myWindow myShape

-- Define a subclass of TitleContainer
class ExtendableTitleContainer (TitleContainer)
```

```
end

-- Get the objects from the accessory
method addAccessory self {class ExtendableTitleContainer} accContainer -> (
    local accList := nextMethod self accContainer

    -- Add the @text item from the accessory to the frontmost window
    prepend accContainer.windowHoldingAccessory accList[@text]
)

-- Remove the objects from the accessory (Undo what addAccessory did)
method removeAccessory self {class ExtendableTitleContainer} accContainer -> (
    local myText := chooseOne accContainer.windowHoldingAccessory \
                      (v a -> isAKindOf v TextPresenter) 0
    if myText = undefined do exit with myText
    deleteOne accContainer.windowHoldingAccessory myText.target
    nextMethod self accContainer
)

-- Create the title container and add the window to it
global tc := new ExtendableTitleContainer dir:theScriptDir path:"painting.sxt" \
    name:"Painting Title"
append tc myWindow

-- Define the title's startup action
tc.startupAction := (tc ->

    -- Load all objects in title container
    forEach tc load undefined

    -- Get the painting library
    local myLC := chooseOne theOpenContainers \
                  (v a -> v.name = "Painting Library") 0

    -- Assign the bitmap to the 2D shape's target in the frontmost window
    tc.windows[1][1].target := myLC[1]
)
```

Creating the Library File

The following library script is identical to the previous example. This script creates the library, then closes both the title and library containers.

```
function getPict fileName -> (
    local myStream := getStream theScriptDir fileName @readable
    local myimage := importMedia theImportExportEngine myStream @image @dib \
       @bitmap colormap:defaultcolormap
    myimage
)

global p1 := getPict "GAUGUIN.BMP"

-- Create the library, and append the picture to it
-- Note that this library uses the title container tc defined previously
```

```
global lc := new LibraryContainer dir:theScriptDir path:"painting.sxl" \
    name:"Painting Library" user:tc
append lc p1

-- Close the title container and then the library container
close tc
close lc
```

Creating the Zoom Accessory File

The following accessory script allows the user to scale the bitmap smaller or larger by moving the mouse left or right.

First, it creates a subclass of `AccessoryContainer` called `ZoomAccessory` so that the `getAccessory` method can be overridden. The class defines instance variables `mouseMoveInterest` and `mouseDownInterest`, which hold the event interests so that they can be removed later with `removeInterest`. The `windowHoldingAccessory` instance variable holds the window, in case the accessory is used on several windows in a given title at the same time.

The `removeAccEventInterests` method removes the two event interests, for mouse move and mouse down. The `callRemoveAccessory` method is needed only to change the number and order of arguments. It is called by the mouse down event, which passes in three arguments.

The `getAccessory` method finds the first 2D shape in the frontmost window, then gets the bitmap from its `boundary` instance variable. The `scaleIt` function scales the bitmap up or down based on the x value of the mouse. The mouse-move event calls the `scaleIt` function on the 2D shape whenever the mouse moves anywhere in the window.

The mouse down event calls the `callRemoveAccessory` function, which calls `removeAccessory` on the title to remove the accessory when the mouse button is pressed.

A text presenter is defined with the string "To zoom, move mouse left or right. Click to stop." This text is added to a hash table named `accList`—this list holds the items to be passed to the title container as a return value from `getAccessory`. A hash table is a keyed, unsorted collection that can be searched quickly. The text presenter is given the key `@text`.

The `terminate` method is overridden to remove the mouse event interests from the `MouseMoveEvent` and `MouseDownEvent` classes. Because `interests` is a class variable, it is not discarded when the title and accessory are closed.

Finally, the last lines of the script create an instance of the accessory container, save it to a file named `painting.sxa`, and close it.

```
--- Create an accessory container which must be added by the user
class ZoomAccessory (AccessoryContainer)
```

```
    instance variables
        mouseMoveInterest    -- Only reason for these ivs is they
        mouseDownInterest    -- hold onto objects so those objects
        windowHoldingAccessory  -- can later be removed
end

-- Remove event interests MouseEvent classes
method removeAccEventInterests self {class ZoomAccessory} interest event -> (
    if (self.mouseMoveInterest != undefined) do
        removeEventInterest self.mouseMoveInterest
    if (self.mouseDownInterest != undefined) do
        removeEventInterest self.mouseDownInterest
)

-- This method is needed only to change the number and order of arguments
method callRemoveAccessory self {class ZoomAccessory} interest event -> (
    removeAccessory theTitleContainer self
)

-- Method called by addAccessory from the title
method getAccessory self {class ZoomAccessory} -> (
    -- Locate the bitmap in the title
    self.windowHoldingAccessory := theTitleContainer.windows[1]
    local myShape := chooseOne self.windowHoldingAccessory \
        (v a -> isAKindOf v TwoDShape) 0
    local myBitmap := myShape.boundary

    -- Scale the image
    function scaleIt theTwoDShape interest event -> (
        if (event.screenCoords.x > 1) do (
            local myScale := 2 * event.screenCoords.x / theTwoDShape.width
            local myMatrix := scale (new TwoDMatrix) myScale myScale
            transform myBitmap myMatrix @mutate
            theTwoDShape.changed := true
            theTwoDShape.x := (theTwoDShape.presentedBy.width \
                - theTwoDShape.width)/2
            theTwoDShape.y := (theTwoDShape.presentedBy.height \
                - theTwoDShape.height)/2
        )
    )

    -- Set up the mouse move event
    self.mouseMoveInterest := new MouseMoveEvent
    self.mouseMoveInterest.authorData := myShape
    self.mouseMoveInterest.presenter := undefined
    self.mouseMoveInterest.device := new MouseDevice
    self.mouseMoveInterest.eventReceiver := scaleIt

    addEventInterest self.mouseMoveInterest

    -- Set up the mouse down event
    self.mouseDownInterest := new MouseDownEvent
    self.mouseDownInterest.authorData := self
    self.mouseDownInterest.presenter := undefined
    self.mouseDownInterest.device := new MouseDevice
    self.mouseDownInterest.eventReceiver := callRemoveAccessory

    addEventInterest self.mouseDownInterest
```

```
    -- Add text to the accessory
    local myText := new TextPresenter boundary:(new Rect x2:400 y2:20) \
           target:" To zoom, move mouse left or right. Click to stop."
     setDefaultAttr myText @alignment @center

    myText.fill := whitebrush

    -- Create the list to be returned by getAccessory
    local accList := new HashTable
    add accList @text myText
    accList
)

-- Remove the event interests when title is closed
method terminate self {class ZoomAccessory} -> (
    removeAccEventInterests self undefined undefined
    nextMethod self
)

global ac := new ZoomAccessory dir:theScriptDir path:"painting.sxa" \
    name:"Painting Accessory"
close ac
```

One minor flaw in this program is that in the unlikely event that two titles opened this accessory, and if the user were to close the first title while the accessory is active, then the first title's window would not be freed from memory even after both titles were closed. This is because the mouse event interests would not have been deleted, since `close` does not call `removeAccessory` on an accessory if another user exists.

Testing the Title and Accessory

To test that the accessory container works properly, execute the previous scripts in ScriptX, which creates three files: `painting.sxt`, `painting.sxl`, and `painting.sxa`. Then quit and do the following:

1. Open up the file `painting.sxt` in the ScriptX Player.

   This should load the title container and library container into memory, and run `startupAction`, which displays the window with the painting in it.

2. Choose **Open Accessory** from the File menu in the ScriptX Player, select `painting.sxa` from the list and press the Open pushbutton.

   This should open the accessory, causing the following text to appear at the top of the window: "To zoom, move mouse left or right. Click to stop".

3. Move the mouse back and forth, left and right. The accessory should cause the bitmap to get smaller as you move left and larger as you move right. When the bitmap is the size you want it, click the mouse—the bitmap will stay at that size and the accessory is removed.

When you are done, choose **Close** from the File menu to close the title along with its associated library (and accessory, if it's not already removed).

Quitting ScriptX Gracefully

By default, when a user closes a ScriptX title or quits the ScriptX Player, the system shuts down immediately. The Quit Manager provides a mechanism for a developer to override this behavior. Quit queries and quit tasks are functions that run before the system shuts down. They can be added and removed dynamically. For a general discussion of the Quit Manager, see the section “Quitting ScriptX” on page 419.

The following example demonstrates a typical quit query. Each title container can install its own quit queries, so that it has a chance to clean up, save data, or even suspend the quit process.

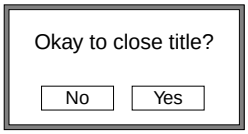


Figure 15-13:The quit query function in this example opens a dialog box.

Flow of the Script

The following diagram gives a general overview of the quit process, which this sample script implements.

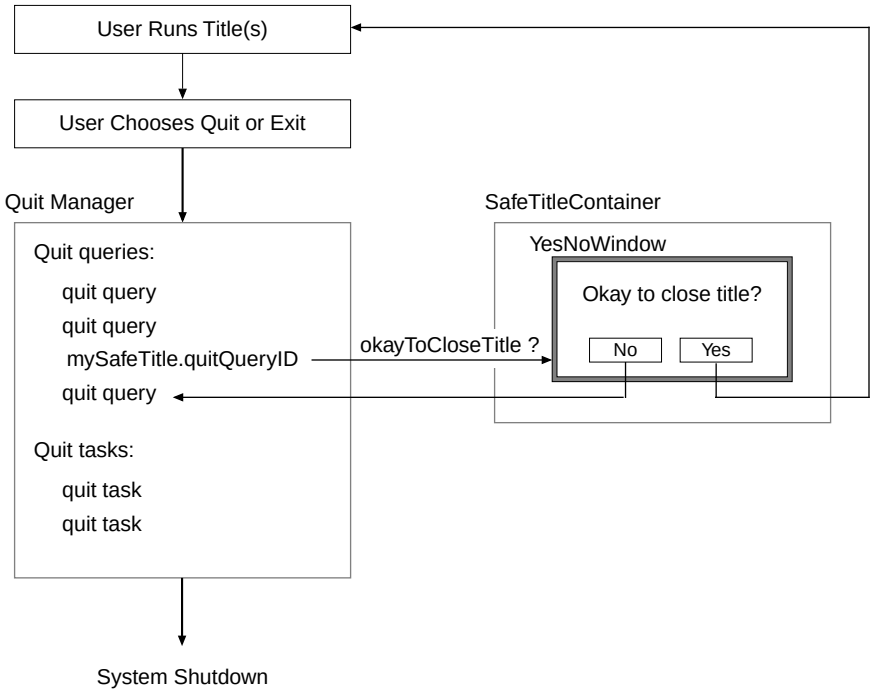


Figure 15-14:Diagram of functions called by the ScriptX Quit Manager.

For the user to quit ScriptX, the global function `quit` must be called. Quitting can be initiated through menu or keyboard commands, or it may be part of the programmatic interface of a title. The global function `quit` initiates the Quit

Manager, which runs a series of functions. Quit queries run before quit tasks, and must return `true`. If any quit query returns `false`, the quit process is suspended. After all the quit queries have returned `true`, the Quit Manager runs quit tasks. Once it begins running quit tasks, shutdown can no longer be aborted. A developer can install or remove both quit queries and quit tasks.

In the following example, the title container installs a quit query, maintaining a record of its unique integer ID so that it can remove the quit query as well, if the title closes before the user quits the ScriptX Player. In addition to defining a quit query, this example also overrides `close`, a method defined by `TitleContainer` and its parent classes, to remove the title's quit query when the title is closed without quitting ScriptX.

The script defines three classes:

- `SafeTitleContainer`, a subclass of `TitleContainer`, that intercepts the user's calls to `close` and `quit` to bring up the "Okay to close title?" dialog box.
- `YesNoWindow`, a subclass of `Window` that queries the user and includes "Yes" and "No" pushbuttons.
- `InterfaceButton` is a user interface object, derived from `PushButton`.

Calling either `close` or `quit` prompts the user, through a dialog box, to indicate whether she wants to close the current title. If the user clicks **No**, then execution returns to the previously active title. This dialog box is one of a number of possible uses for quit queries. Another application of quit queries is to update objects or save state information, such as a current game score.

The following sections describe the script in detail.

Application of Modules

Typical ScriptX programs use a module with each container file that stores code objects. The module itself acts as a container, and it provides namespace protection. Since this example demonstrates how multiple titles share the resources of the Quit Manager, each title is created and compiled in a separate module. Each title container's module uses both the `ScriptX` module and `QuitQueryExample`. The classes which all the titles share are compiled in `QuitQueryExample`.

```
module QuitQueryExample
    uses ScriptX
    exports SafeTitleContainer, InterfaceButton
end
in module QuitQueryExample
```

Defining the Dialog Box

`InterfaceButton`, the first class defined in the script, defines a simple text button for user interaction.

```
class InterfaceButton (PushButton)
    class variables
        buttonBoundary:(new Rect x2:60 y2:20)
    instance variables
        text
    instance methods
        method init self #rest args #key \
                buttonText: -> (
            if isAKindOf buttonText String then (
                self.text := buttonText
            ) else (
                report badParameter #(buttonText, init,
                    self, "not a string")
            )
            apply nextMethod self \
                releasedPresenter:(new TextPresenter \
                    boundary:InterfaceButton.buttonBoundary\
                    target:(self.text) \
                    stroke:blackBrush) \
                pressedPresenter:(new TwoDShape \
                    target:InterfaceButton.buttonBoundary \
                    fill:blackBrush) \
                args
        )
        method afterInit self #rest args #key \
                action: -> (
            setDefaultAttr self.releasedPresenter @alignment @center
            if isAKindOf action AbstractFunction then (
                self.activateAction := action
            ) else (
                report badParameter #(action, afterInit,
                    self, "not a function")
            )
        )
end
```

`YesNoWindow` defines instance variables for the three presenters that it displays: `textQueryPresenter`, `yesButton`, and `noButton`. Other instance variables include a controller and for its buttons: `actuatorController` and `buttonBoundary`. This script presents an instance of `YesNoWindow` in a separate thread, allowing the window to block and wait for a response from the user. To handle synchronization between threads, `YesNoWindow` defines the instances variables `theLock`, a `Lock` instance, and `startPipe` and `finishPipe`, instances of `PipeClass`.

```
class YesNoWindow (Window)
    instance variables
        actuatorController
        textQuery
        textQueryPresenter
        yesButton
        noButton
        theResult:false
        theLock:(new Lock)
        startPipe:(new PipeClass size:1)
        finishPipe:(new PipeClass size:1)
end
```

For performance reasons, this script creates the dialog window on the fly, rather than creating and restoring a persistent object from the object store. The title requires an instance of `YesNoWindow` only intermittently, when `quit` is called, and it must present the window and all the objects it contains quickly.

In its `init` method, `YesNoWindow` sets the value of the type keyword to create a notice window, a modal dialog window that has various names on the different native ScriptX platforms.

```
method init self {class YesNoWindow} #rest args ->
    apply nextMethod self type:@notice args
```

The `afterInit` method sets up the window and the presenters it displays. This `afterInit` method does not actually display the window. It depends on another calling function to make the window visible.

```
method afterInit self {class YesNoWindow} #rest args \
       #key question: xPosition: yPosition: -> (
    apply nextMethod self args
    self.actuatorController := new ActuatorController \
           space:self \
           wholeSpace:true
    if isAKindOf question String then
        self.textQuery := question
    else
        report badParameter #(question, afterInit,
            self, "not a string")
    self.textQueryPresenter := new TextPresenter \
           boundary:(new Rect x2:260 y2:20) \
           target:(self.textQuery)
    self.yesButton := new InterfaceButton \
           buttonText:"Yes" \
           action:(a b -> (
               if a.title != theScratchTitle then (
                   a.theResult := true
               ) else (
                   a.theResult := false
               )); hide a;
               write a.finishPipe a.theResult)
    self.yesButton.authorData := self
    self.noButton := new InterfaceButton \
           buttonText:"No" \
           action:(a b ->
               a.theResult := false
               hide a;
               write a.finishPipe a.theResult)
    self.noButton.authorData := self

    -- center the text, so it looks good
    setDefaultAttr self.textQueryPresenter @alignment @center

    -- position the three presenters, based on window's dimensions
    local buttonHeight := (self.height * 0.75) as ImmediateInteger
    local buttonCenter := (self.width * 0.50) as ImmediateInteger
    self.yesButton.y := buttonHeight
    self.noButton.y := buttonHeight
```

```
    self.yesButton.x := buttonCenter + 10
    self.noButton.x := (buttonCenter - self.noButton.width) - 10
    self.textQueryPresenter.x := buttonCenter -
(self.textQueryPresenter.width / 2)
    self.textQueryPresenter.y := (self.height * 0.30) as
ImmediateInteger

    -- position self
    self.x := xPosition
    self.y := yPosition

    -- add the three presenters
    append self self.yesButton
    append self self.noButton
    append self self.textQueryPresenter
)
```

The instance method `askForYesOrNo` is designed to run as a thread's control function. By running `askForYesOfNo` in a separate thread, the calling function can block and wait for user response without suspending other ScriptX processes. Note that `afterInit` defines an activate action for the yes and no buttons that writes to the windows `finishPipe` instance variable. The reading thread, which is running `askForYesOrNo`, blocks until it receives input from the user. When `askForYesOrNo` is able to read from the pipe, it relinquishes the lock and returns a result to the original calling function.

```
method askForYesOrNo self {class YesNoWindow} -> (
    acquire self.theLock
    write self.startPipe true -- unblocks calling thread
    show self
    local myResult := read self.finishPipe -- block
    relinquish self.theLock -- unblocks calling thread
    return myResult
)
```

Defining the "Safe" Title Container

`SafeTitleContainer` defines a title container that intercepts calls to close the title or quit the ScriptX Player, in order to display the previous dialog box. Its `quitQueryID` instance variable stores the unique ID of a quit query, so that it can be installed and removed as the title opens and closes. A well-behaved title should "clean up" when it closes, removing its quit query and any objects it has installed in the system. In this example, `_quitQueryID` is the physical slot used for storage, and `quitQueryID` is its virtual interface. Note that `_quitQueryID` is defined with the `transient` qualifier, since it contains only run-time information. (A quit query's unique ID applies only to the current ScriptX session.)

```
class SafeTitleContainer (TitleContainer)
    instance variables
        transient _quitQueryID
    instance methods
        method quitQueryIDGetter self ->
            self._quitQueryID
```

```
        method quitQueryIDSetter self value ->
            if getClass value == ImmediateInteger then
                self._quitQueryID := value
            else
                report badParameter #(value, quitQueryIDSetter,
                    self, "not an immediate integer")
        method okayToCloseTitle self -> (
            bringToFront self
            local buildText := "Is it OK to quit the title " +
                self.name + "?"
            local a := new YesNoWindow \
                    boundary:(new Rect x2:300 y2:100) \
                    question:buildText \
                    title:self \
                    xPosition:50 \
                    yPosition:50
            local modalProcess := new Thread \
                    func:askForYesOrNo \
                    arg:a
            read a.startPipe
            acquire a.theLock
            print "I was here five"
            local b := modalProcess.result
            return b
        )
        method close self #rest args -> (
            if okayToCloseTitle self then (
                deinstallQuitQuery self.quitQueryID
                apply nextMethod self args
            )
        )
        method startupActionGetter self -> (
            -- make sure only a single quit query is installed
            if self.quitQueryID = undefined do
                self.quitQueryID :=
                    installQuitQuery okayToCloseTitle self
            nextMethod self
        )
end
```

`SafeTitleContainer` specializes `close`, first calling `okayToCloseTitle` to allow then user to validate his choice. If the user chooses to close the title, `close` removes the query, by calling the global function `deinstallQuitQuery` on its stored ID.

`SafeTitleContainer` specializes the accessor method `startupActionGetter` so that `okayToCloseTitle` is installed as a quit query each time the title opens. Note that it specializes this instance variable as an accessor method in such a way that the `startupAction` can still be set by another script.

`SafeTitleContainer` specializes `close` to remove its quit query when the title closes, but its quit query does not “remove itself” from the queue when the user quits. The title should not assume that the user will continue with the quit process. Each quit query should behave as if it were one of many. A quit query or quit task cannot anticipate that it will run in any particular order with respect to other functions that run at quit time. Since any quit query in the

queue could return `false`, suspending the process, a given quit query might have to run again, the next time the Quit Manager is called. The role of quit queries and quit tasks is limited to putting the container in a safe state before the system shuts down.

Note – A quit query or quit task should be tested in an environment with multiple containers.

The instance method `okayToCloseTitle` acts as the calling function for the quit query. In doing so, it spawns a separate thread. This allows the title to display a "modal" dialog box and suspend user input, without suspending other active processes. Note the use of a lock and two pipes (instances of `PipeClass`) for synchronization. The `startPipe` pipe is necessary to prevent a race condition, and assures that the dialog box and its contents show before the calling function blocks and waits for a response.

Building the Title

This final part of the script puts it all together by creating two title containers, each of which is compiled in a separate module. The title containers "Ross's Peccadillos" and "Wade's Follies" do nothing but open a window and display buttons so that can you test `close` and `quit`.

```
module QuitQueryTest1
    uses ScriptX
    uses QuitQueryExample
end
in module QuitQueryTest1

object mySafeTitle (SafeTitleContainer)
       dir:theStartDir
       path:"quitqone.sxt"
       name:"Wade's Follies"
end

open TitleContainer path:"quitqone.sxt"
object myWindow (Window)
       title:mySafeTitle
       name:mySafeTitle.name
end
object buttonController (ActuatorController)
    space:myWindow, wholeSpace:true, enabled:true
end
object myCloseButton (InterfaceButton)
       buttonText:"Close"
       action:(a b -> new Thread \
           func:(c -> close c) arg:a)
    settings
       x:420, y:360
end
myCloseButton.authorData := myWindow.title
object myQuitButton (InterfaceButton)
       buttonText:"Quit"
       action:(a b -> new Thread func:(c -> quit()))
```

```
    settings
        x:500, y:360
end
myQuitButton.authorData := myWindow.title
append myWindow myCloseButton
append myWindow myQuitButton
add mySafeTitle 1 myWindow
show myWindow
```

Now, create a second title in order to test the quit queries in a multiple environment.

```
module QuitQueryTest2 uses ScriptX, QuitQueryExample end
in module QuitQueryTest2

object mySafeTitle (SafeTitleContainer)
       dir:theStartDir
       path:"quitqtwo.sxt"
       name:"Ross's Peccadillos"
end
open TitleContainer path:"quitqtwo.sxt"
object myWindow (Window)
       title:mySafeTitle
       name:mySafeTitle.name
end
object buttonController (ActuatorController)
    space:myWindow, wholeSpace:true, enabled:true
end
object myCloseButton (InterfaceButton)
       buttonText:"Close"
       action:(a b -> new Thread \
           func:(c -> close c) arg:a)
    settings
       x:420, y:360
end
myCloseButton.authorData := myWindow.title
object myQuitButton (InterfaceButton)
       buttonText:"Quit"
       action:(a b -> new Thread func:(c -> quit()))
    settings
       x:500, y:360
end
myQuitButton.authorData := myWindow.title
append myWindow myCloseButton
append myWindow myQuitButton
add mySafeTitle 1 myWindow
show myWindow
```

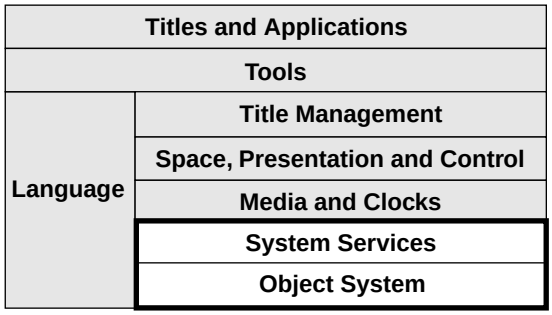
Every ScriptX title should be designed so that it can coexist with other titles, provided that sufficient system resources are available. Although these two titles define a quit button that invokes the Quit Manager directly, as well as a close button, a title's quit button should be defined so that it only closes the title itself. The quit dialog in this example is really protection against quit queries, written by other developers, that may not be well-behaved.

Note – The current version of Script contains a bug that prevents quit queries from receiving mouse or keyboard input if `quit` is called from the `EventDispatchQueue` thread (which receives and processes events from input devices). This script will not work properly if you invoke `quit` from the system menu or keyboard.

To test this example, first compile the script in ScriptX. You can test it by calling the global function `quit` from the scripter, or by clicking on one of the quit buttons. Test every possible combination of quitting and closing, in every possible order. If you use the scripter for testing, note that each title is compiled in a separate module. Each title should be able to close independently, without disturbing other titles, and each title should be able to recover if any other title attempts to close it.

PART TWO

Lower-Level Components



CHAPTER

16

Collections

The Collections component defines objects that store and manage other objects, so that objects can be grouped together, arranged, and sorted. A collection allows a set of objects or data to be processed as a single entity. The collection classes include some of the basic ScriptX data structures.

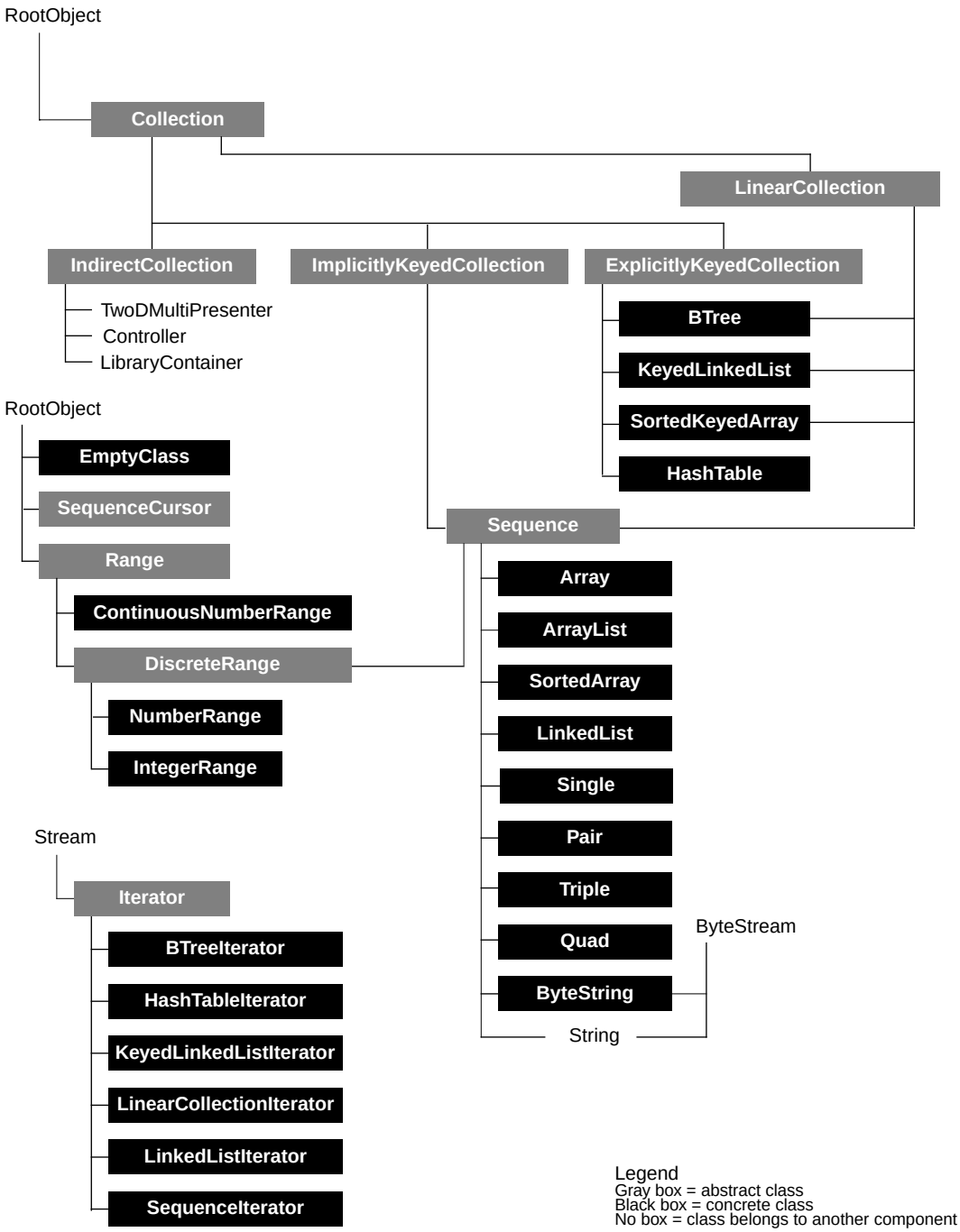
Collections are used by many other components. They are one of the fundamental building blocks of the ScriptX Player. Anywhere a list, array, or sequence of objects is needed, a particular collection class is available to serve the need. For example, `TwoDMultiPresenter` and `Controller` manage collections of objects. The Collections component defines two main categories of classes: collections and iterators.

A *collection* is an object that maintains a set of data, often a list of other objects. Items in a collection are called elements, items, members, key-value pairs, or bindings. Each collection provides methods for managing its elements. Collections are the basic classes that implement protocols for group behavior. They can have a fixed, variable, or unbounded number of items. Items in a collection can be sorted or unsorted. Some collections can hold objects of any class, others restrict the classes of their keys or values, and some collection classes hold elements that are not objects.

An *iterator* is a mechanism for stepping through each item in a collection in an orderly fashion. An iterator is an object that manages elements of a collection as a stream. Many operations on collections require a script to process each element, often in some natural order. By creating an iterator, a script has access to methods in the Streams component that are useful for sequentially processing and manipulating the elements of a collection. If a collection has a natural order, its iterator always processes items in that order. For more information on streams, see Chapter 20, “Streams.”

Classes and Inheritance

Class inheritance for the Collections component is shown in this figure:



Conceptual Overview

The Collections component defines two main categories of classes: collections and iterators.

A *collection* is an object that maintains a set of data, often a list of other objects. Items in a collection are called elements, items, members, key-value pairs, or bindings. Each collection provides methods for managing the data or objects it

contains. Collections are the basic classes that implement protocols for group behavior. Collections can have a fixed, variable, or unbounded number of items. Items in a collection can be sorted or unsorted. Some collections can hold objects of any class, others restrict the classes of their keys or values, and some collection classes hold elements that are not objects.

The ScriptX Player factors the root behavior of collections between two abstract classes: `Collection` and `LinearCollection`. `Collection` inherits from `Container`, and provides the root behavior of all the concrete collection classes. `Collection` defines the basic methods for adding, processing, and removing elements, for gaining access to each key and value. Collections that have a natural order, collections that can be processed sequentially, inherit from `LinearCollection` as well as from `Collection`. `LinearCollection` is a mix-in class that defines methods for processing elements in order, for rearranging elements within a collection.

Each element in a ScriptX collection can be identified by using a *key*. This key may or may not be stored in the collection itself. For collections that inherit from `ImplicitlyKeyedCollection`, this key is simply the element's position within the collection. For example, with the `Array` class, it is the integer offset for each object in the array. Other collections, subclasses of `ExplicitlyKeyedCollection`, store their key explicitly. This key can be any object, but it is usually some object on which the collection can be conveniently and systematically ordered.

An *iterator* is a mechanism for stepping through each item in a collection in an orderly fashion. An iterator is an object that manages elements of a collection as a stream. Many operations on collections require a script to process each element, often in some natural order. An iterator is a stream that is readable, writable, and seekable. By creating an iterator, a script has access to methods in the Streams component that are useful for sequentially processing and manipulating the elements of a collection. If a collection has a natural order, its iterator always processes items in that order.

`Iterator` is an abstract class that inherits from `Stream` and provides the root behavior for all iterators. Iterators can be created explicitly. They are generated automatically by `for` loops in the ScriptX language, and also by generic functions such as `forEach` and `map` that all collections implement. For more information on ScriptX `for` loops and their application to collections, see Chapter 4, "Conditional and Loops," in the *ScriptX Language Guide*. For more information on streams, see Chapter 20, "Streams."

How Collections Work

A collection is an aggregation of objects. (Some collection classes, such as `String` and `NumberRange`, do not actually contain objects, although methods such as `getOne` that retrieve an element from these collections return an object.) The objects within a collection can be of any type of object: numbers, strings, bitmaps, pushbuttons, presenters, or other collections.

Collections are not the only classes that store other objects—it is common for a class to define instance variables that store other objects. In C++ terminology, such variables are called *members*. Collections are distinct in that the base

collection classes defined by ScriptX make no semantic distinction between elements. For example, any of the four elements of a `Quad` can contain any object. This differs from other classes that contain more than one object. For example, in a `Point` object, one item intrinsically represents the x position and another item represents the y position.

The `LinearCollection` mix-in class denotes that items of a collection will always be visited in the same order until a new item is added or an old one deleted. Some collections, such as `Array` and `LinkedList`, have a natural order. Others, such as `HashTable`, do not. `LinearCollection` factors out behavior for ordinal operations—operations that can only be performed on collections that have a natural order.

The ScriptX collection classes define several different abstract types of collections. All of the collection classes are keyed, that is, each item in the collection has a corresponding key. A *key* is any object (often an integer or string) that is paired with, or bound to, a value, either explicitly or implicitly. Thus, we often refer to an item in a collection as a *key-value pair*.

Collections with *explicit keys* are those in which the key is an explicit member of the collection. An explicit key can be any object—such as an integer, a string, a date, or even another collection. As with other collections, a value can be any type of object. In addition, keys need not be the same kinds of object as values.

Note that in unsorted collections with explicit keys, the order of the keys themselves isn't important. For example, if you wanted to keep track of who parks in the parking lot, you could build an explicitly keyed collection in which the keys are the names of drivers, and values the names of cars.

Mike	Harvey	Jim	Karen	Felicia	Greg	Ross
Buick	Integra	Ranger	Cabriolet	Saab	Isuzu	Miata

Collections with implicit keys are those in which the keys are automatically assigned by the collection rather than by the user. Thus, collections that are subclasses of `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 positive integers 1, 2, 3, and so on. In sequences, the key is simply the position of the item in the sequence; you can use that position to access the value. For example, you could use a sequence to represent a queue of tasks:

1	2	3	4	5	6
Feed dog	Eat dinner	Wash dishes	Do laundry	Call Mom	Go to bed

The ScriptX collection hierarchy contains several abstract classes that implement the behavior of these different types of collections.

Every collection is either an explicitly keyed or and implicitly keyed collection, as shown in Figure 16-1. `LinearCollection` defines additional methods that allow a script to access items through their position in the list.

`LinearCollection` is mixed into both implicitly- and explicitly-keyed collections to provide access through position. `Sequence`, a subclass of both `ImplicitlyKeyedCollection` and `LinearCollection`, provides access by position without explicit keys.

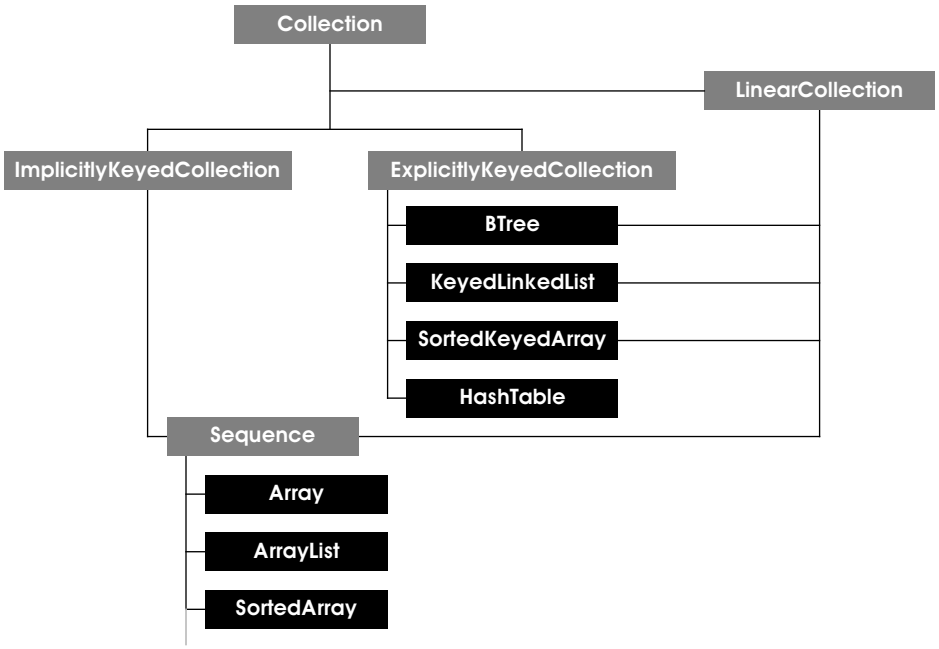


Figure 16-1: Classes that define the Collection protocol

A linear collection has a natural order. Items in a linear collection are generally processed in that order. Most generic functions that depend on the ordering of items are defined by `LinearCollection`, rather than by `Collection`. Note that while a script can have access to the nth item in a linear collection, the `LinearCollection` class does not define methods for adding an item in a particular position, or for changing the position of an item. The `Sequence` protocol defines additional methods that determine the position of an item in a collection, such as `addNth`, `moveToFront`, and `sort`.

Sorted Collections

The sorted collections are `BTree`, `SortedKeyedArray`, and `SortedArray`. Sorted collections are those in which the items in the collection are maintained in sorted order. The order of the sorting is specified by a sort function. During sorting, the collection compares one item against another, using this function to determine the order of the items. This function enables a collection to sort its members alphabetically, numerically, forwards, backwards, or by any criterion you can devise. You specify a “less than” function in the `new` method of `SortedArray` and `SortedKeyedArray` collections.

Note – Developers will generally want to create their own sorting functions for sorted collections that are heterogenous (contain more than one kind of object), since the default sorting function sorts items by class and then by value.

An explicitly keyed collection can be sorted either by key or by value, depending on the sort function that is supplied. However, an implicitly keyed collection can be sorted only by value, because the keys are a result of the sort.

When an item is added to a sorted collection, it is automatically inserted in its proper sorted position. Although you can access the items of a sorted collection by their position in the list (first, second, third), that position is subject to change based on any items that are added or deleted.

Note – Sorted collections insert an item in its proper position automatically. Do not insert items manually, rearrange items by script, or modify keys. To modify the key of an item, remove it from the collection, change the key, and explicitly add it back to the collection.

Comparison Functions

Comparing for “Equal”

Some of the generic functions implemented by collections compare keys or values. For example, `isMember` tests to see if an object is in the collection, and `getOne` compares keys to retrieve the appropriate value. The instance variables `valueEqualComparator` and `keyEqualComparator` return the functions that are used to compare values and keys, respectively. Typically, this is the `ueq` function (universal equal). The function `ueq` can compare any two objects, and is guaranteed to return a value without reporting an exception. However, `ueq` requires that two objects be of the same class to return `true`. You may wish to create collection classes that use different equality tests. Table 16-2 lists three global functions, defined by ScriptX, that can be used as equal comparators.

Table 16-1: ScriptX equal comparators

Function	Comment
`eq`	returns `true` if two objects are identical
`equal`	used when the `isComparable` and `localEqual` methods can perform the equality testing
`ueq`	guaranteed to return a value; returns `true` if two objects are of the same class and pass the `localEqual` test

For example, suppose you want to create an instance of `HashTable` that uses strings as keys. `StringConstant` and `String` are separate classes in ScriptX. The following expression returns `false` because "foobar" is a string constant, while "foo" + "bar" evaluates to String.

```
ueq "foobar" ("foo" + "bar")
⇨  false
```

You might want a hash table in which strings and string constants can be equal. You can specialize `HashTable` to relax the key equal comparator, to use a different equals function, as shown in the following example.

```
myHash := object (HashTable)
    instance methods
        method keyEqualComparatorGetter self -> equal
end
```

Comparing for “Less Than”

When sorting, comparison functions must be used to compare items two at a time to determine if one item is “less than” the other. ScriptX has two mechanisms for sorting items in a collection:

- Use a class that keeps its items sorted, `SortedArray` or `SortedKeyedArray`, and pass in the “less than” function when you instantiate the class with the optional keyword argument `ltFunction`. These classes default to `ult` (universal less than). Here’s an example of a sorted array with a custom sort function:

  ```
  mySortedArray := new SortedArray \
      ltFunction:(x y -> (x as String) < (y as String))
  ```

- Use a subclass of `Sequence` and call the `sort` method on the instance of the collection, which takes a less-than function as its second argument. Here’s an example of a sorted array with a custom sort function:

  ```
  myArray := new Array
  sort myArray (x y -> (x as String) < (y as String))
  ```

The less-than function is used to provide a consistent ordering of items. ScriptX allows you to specify functions as comparators for collections, so that you can either built-in comparison functions, as shown in Table 16-2, or write your own. In either case, a “less than” function should take two values as arguments, and return `true` if *value1* should sort before *value2*. The previous two examples use this function:

```
(x y -> (x as String) < (y as String))
```

You might want to use custom comparison functions in place of built-in functions if you’re putting unusual objects into a sorted collection. For example, how does one sort an array of instances of `PushButton`? Since the PushButton class does not specialize `localLT`, the generic comparator for objects, the built-in comparison functions would report an exception.

If you specify your own “less than” function, that function can determine what aspect of a button you want to compare. For example, you might compare and order buttons according to text in the button’s label, the button’s position on the display, or the order in which the button appears in the title. There are two approaches to creating a comparison function. You can create a new global function, such as the global comparison functions in the *ScriptX Class Reference*. You can also specialize the generic functions in the Comparison protocol, so that existing global comparison functions are able to compare new classes of objects.

Table 16-2: ScriptX less than comparators

Function	Comment
`lt`	use when you want to safely compare comparable objects; reports an exception if you pass in incomparable objects
`localLt`	a method defined for all objects, one of the four primitives from which all comparison functions are defined
`ult`	Use a "universal" less-than comparator when you want to consistently compare anything. Note that `ult` reports an exception if it is used to compare two objects of the same class, and that class does not specialize `localLT`.

The following is a list of variables and methods in the Collections component that explicitly specify a comparison function.

Instance Methods
`SortedArray` `new` *self* `initialSize:`*integer* `growable:`*boolean* `ltFunction:`*ltfunction*
`SortedKeyedArray` `new` *`self`* `initialSize:`*integer* `growable:`*boolean* `ltFunction:`*ltfunction*
`Sequence` `sort` *self* *ltFunction*

Instance Variables
`Collection` *self.*`valueEqualComparator`
`KeyedCollection` *self.*`keyEqualComparator`

For discussion of the Comparison protocol, see "Comparing Objects" on page 633 of Chapter 23, "Object System Kernel." See also the discussion of comparison operators in the *ScriptX Language Guide*.

Choosing a Collection Class

Given that all classes that inherit from `Collection` store and manipulate multiple items, you may have questions about which one is optimal to use for any one situation. Choosing the best collection is a key to performance. Since ScriptX is an object-based system, it is possible to switch one collection for another that shares the same protocols.

One way to determine which collection class to use is by looking at the inheritance tree at the start of this chapter. The following flowchart, Figure 16-2, illustrates the process of exploring the inheritance tree. If you want either a discrete or continuous range, use those classes. Otherwise, start at `Collection` and move down the tree, making a decision at each branch.

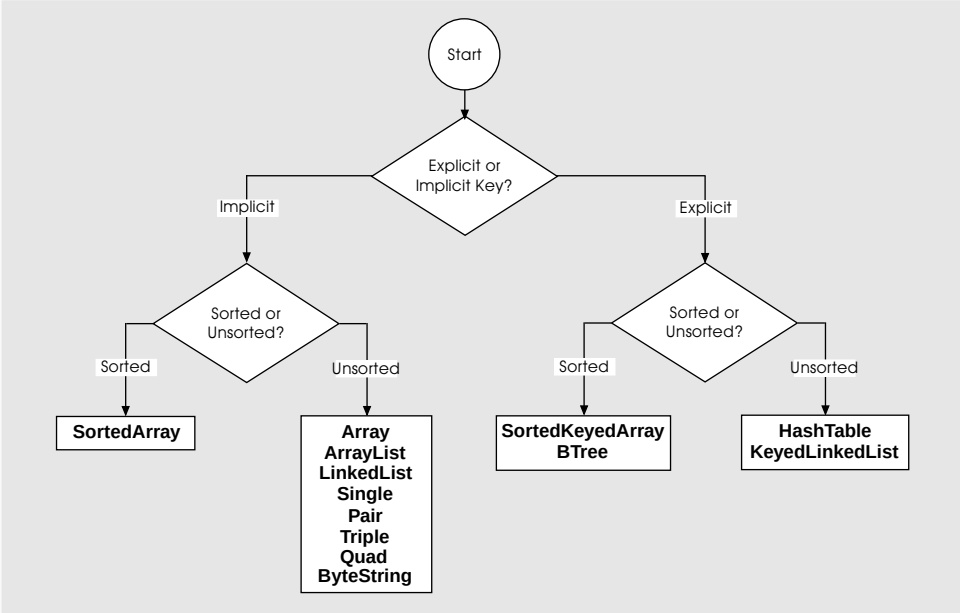


Figure 16-2: Flowchart for selecting a collection class.

Even if you follow the preceding flowchart to an endpoint, you may still be faced with a choice. Each class of collection implements a different data structure. The choice of which collection to use in a particular application involves understanding tradeoffs between performance in various operations and efficient use of memory. Select a collection based upon how you intend to use the collection. In a well-designed object-based system, it is easy to switch from one collection class to another, since all collections implement a common set of methods. The following list summarizes the major issues in choosing a collection.

1. **Inserting items.** How fast can new items be inserted into a collection. For most classes, it is easy to either append or prepend an item to a collection. Some collections must rearrange other elements each time an item is inserted in the middle of the collection.

2. **Deleting items.** How fast can items be removed from a collection? The issues are the same as for inserting items.

3. **Linear access.** How quickly can you access elements of the collection in sequential order?

4. **Linear access in reverse.** How fast is linear access in reverse order?

5. **Random access by key.** How quickly can you obtain access to a randomly choose element, given its key?

6. **Random access by value.** How quickly can you obtain access to a randomly choose element, given its value?

7. **Size and performance.** How does performance vary with size? Does performance of any operations decrease as the collection grows large?

8. **Memory requirements.** How much memory, in addition to what is required for the key and the value, does the collection require for other structures, such as indexes?

Keep in mind that ScriptX is an untyped language, and that assignment is always by pointer. In more structured languages, data structures such as arrays must contain homogeneous elements. ScriptX collections contain pointers to objects. With the exception of immediate objects, the object itself is stored elsewhere. (For a discussion of immediate objects, see page 490 of Chapter 17, "Numerics.")

Since ScriptX collections contain pointers, linear structures of large objects are quite flexible in ScriptX. Programmers who are accustomed to a language that has explicit data types, such as C or Pascal, will want to rethink their strategies for choosing a data structure. For example, the trade-off in ScriptX between using an array and a linked list often favors an array in cases where a developer might choose a linked list in other languages.

Some collection classes give a programmer additional options, determined at initialization, that affect performance and memory requirements. For example, the `Array` class allows a script to determine whether an array is growable, and what its initial size will be. The discussion that follows summarizes the advantages and disadvantages of each of the collection classes, offering strategies for using them efficiently.

Arrays and Sorted Arrays

An *array* stores its items contiguously and uses a simple linear search to locate an item. When an item is inserted anywhere except at the end of the array, existing items must be moved to keep them contiguous. This process can be slow. The bigger the array, and the closer to the beginning it is inserted, the longer this move takes. However, random access to items by key is fast because items in an array are contiguous. The ScriptX Player can retrieve the *nth* item by directly computing its location.

`Array`, `SortedArray`, and `SortedKeyedArray` are linear collections. The methods that these classes inherit from `LinearCollection`, such as `getNth` and `deleteNth`, allow access to items by position. Note that for sorted arrays, that position may change each time an item is inserted or deleted, since the items are sorted after each change.

There are three built-in concrete array classes:

- `Array` – the basic implementation of an array. `Array` inherits from `Sequence` and, therefore, uses implicit keys.
- `SortedArray` – an array in which the values are always sorted. `SortedArray` also inherits from `Sequence`.
- `SortedKeyedArray` – an array of key-value pairs which is always sorted by key. `SortedKeyedArray` inherits from `ExplicitlyKeyedCollection` and `LinearCollection`.

Many of the ScriptX Player's internal data structures use arrays by default. For example, all of the subclasses of `Controller` and `TwoDMultiPresenter` that are defined in the core classes specify an array as a target collection. Although a developer can use a different target collection, an array is usually optimal for operations that require fast linear and sequential access to elements of the collection.

Table 16-3: Performance characteristics of the Array class

Inserting items	slow, except when the element is appended to the collection.
Deleting items	slower, depending on the size of the collection, and the position of the element that is being deleted
Linear access	fastest
Linear access in reverse	fastest
Random access by key	fast, however the implicit key may have no useful meaning
Random access by value	slower, depending on the size of the collection, and the position of the element
Memory requirements	each element requires a pointer. Blocks of memory are added in increments, set by the `initialSize` keyword.
Size and performance	performance on any operation that requires a linear search decreases as an array grows larger

Sorted arrays, like unsorted arrays, offer fast linear access. Inserting items into a sorted array is considerably slower, since the entire collection must be rearranged each time an item is added. However, random access is much faster than with unsorted arrays.

Sorted arrays are ideal for smaller and relatively static collections, where satisfactory performance on random access is required. `SortedKeyedArray` and `SortedArray` objects search for a random element using a "binary chop" algorithm—they divide the collection in half and then divide one subcollection in half, continuing until a matching object is found. The efficiency of this algorithm is proportional to $\log_2 n$, where n is the size of the collection.

Note – An object cannot be used as a key when a key-value pair is added to an instance of `SortedKeyedArray` unless that object implements `localLT`. Similarly, an object cannot be added as a value to an instance of `SortedArray` unless that object implements `localLT`. The default implementation of `localLT`, defined by `RootObject`, reports an exception. For more information, see the discussion of comparison that begins on page 450.

Both `SortedArray` and `SortedKeyedArray` allow a script to set a comparison function at initialization by setting the `ltFunction` keyword.

Table 16-4: Performance characteristics of the SortedArray class

Inserting items	slow, depending on the size of the collection, and the position of the element that is being added
Deleting items	slow, depending on the size of the collection, and the position of the element that is being deleted

Table 16-4: Performance characteristics of the SortedArray class

Linear access	fastest
Linear access in reverse	fastest
Random access by key	fast, however the implicit key may have no useful meaning
Random access by value	faster than `Array`, although still dependent on the size of the collection and the position of the element
`Memory requirements`	same as `Array`
`Size and performance`	performance on any operation that requires rearranging the collection decreases as the array grows larger

`SortedKeyedArray` objects arrange key-value pairs according to an explicit key. Its performance parallels that of `SortedArray`, except that random access to by value is much slower. (The ScriptX core classes do not currently include a class of explicitly-keyed array that sorts itself by value.)

Table 16-5: Performance characteristics of the SortedKeyedArray class

Inserting items	slow, depending on the size of the collection, and the position of the element that is being deleted
Deleting items	slow, depending on the size of the collection, and the position of the element that is being deleted
Linear access	fastest
Linear access in reverse	fastest
Random access by key	fast, although still dependent on the size of the collection and the position of the element
Random access by value	slower, depending on the size of the collection, and the position of the element (same as `Array`)
Memory requirements	each element requires a pointer for the key and a pointer for the value. Blocks of memory are added in increments, set by the `initialSize` keyword.
Size and performance	performance on any operation that requires rearranging the collection decreases as the array grows larger

Linked Lists

A *linked list* is a sequence of items, in which each item is linked to the next. The items are not necessarily contiguous. When a new item is inserted between two items, the link is broken between them. New links are created between the new item and each of the existing items. This technique does not require moving any items, therefore adding items in a linked list can be quite fast.

However, searching for an item in a linked list is slow, because when traversing a linked list, the searching program must follow links from item to item.

There are two concrete linked list classes:

- `LinkedList`: a single-linked list of objects. `LinkedList` inherits from `Sequence`.

- `KeyedLinkedList`: an unsorted linked list of key-value pairs. The order of the key-value pairs is not determined. `KeyedLinkedList` inherits from `ExplicitlyKeyedCollection`.

In other systems, linked lists are a useful data structure when inserting and deleting items is a priority. Since ScriptX collections actually store pointers to elements, linked lists offer very little that the `ArrayList` class cannot do better, and with less overhead for memory allocation and management.

Table 16-6: Performance characteristics of the LinkedList and KeyedLinkedList classes

Inserting items	fast, provided you are already at the position where you want to insert an item
Deleting items	fast, provided you are already at the position where you want to delete an item
Linear access	fast, but slower than arrays
Linear access in reverse	very slow, especially as the collection grows large
Random access by key	very slow, dependent on position in the collection
Random access by value	very slow, dependent on position in the collection
Memory requirements	proportional to size, but each element requires more overhead than for an array
Size and performance	performance on most operations decreases as the array grows larger

If you create a scripted subclass, note that the classes `LinkedList` and `KeyedLinkedList` have recursive data structures. Each value added to the collection causes a new instance of the class to be created. If you create a subclass of `LinkedList` and override the `init` method, you must address the fact that your `init` method will be invoked for each object added to the list. This is not the same behavior for `init` in the `Array` or `HashTable` classes, where `init` is invoked only once for the lifetime of the collection.

Array Lists

Think of the `ArrayList` class as a hybrid of `Array` and `LinkedList`, a class that sacrifices some performance advantages of an array to gain others. Arrays offer excellent performance at linear access and random access by implicit key. But as an array grows large, adding and deleting values becomes more and more cumbersome. An array list can be thought of as a series of subarrays linked together—a linked list of arrays.

A script determines the size of each subarray by setting the `initialSize` keyword when an instance of `ArrayList` is created. Thereafter, a script knows nothing about how elements are arranged between subarrays—subarrays are an implementation detail that is completely transparent to the scripter.

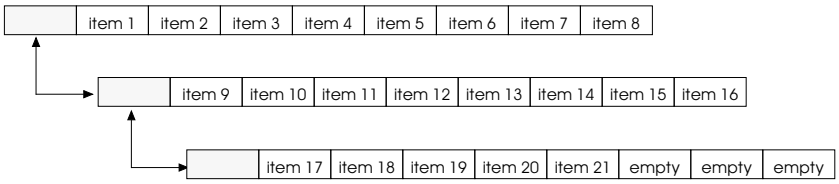


Figure 16-3: An array list containing 21 elements, with initial size 8

As you add and delete elements, the array list rearranges only the subarray that contains the element. It can add and delete new subarrays in the middle as well as at the end of the collection.

Figure 16-4 depicts the array list in Figure 16-3 after several changes. First, the original item 4 has been deleted. Next, two new items have been inserted after the original item 12, which is now item 11. Inserting two new items causes the `ArrayList` object to split its second subarray, putting the elements after the insert into a new subarray. Implicit keys are renumbered automatically, preserving the order of the collection.

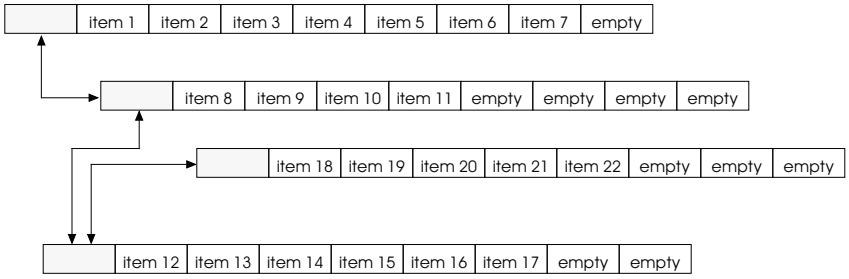


Figure 16-4: An array list after items have been inserted and deleted

Note that these operations do not affect the contents of the third subarray, which becomes the fourth subarray after several items are added. Items 18 through 22, although renumbered, remain in their original positions. Although there is empty space at the end of each subarray in Figure 16-4, this empty space is invisible to the scripter, which knows nothing about how an array list organizes its elements. An array list recovers this empty space when it is saved to the object store.

`ArrayList` places an upper limit on the overhead associated with adding and deleting items in arrays. For large collections, this gives it a performance advantage over `Array` on these operations, for which it sacrifices performance in both linear and random access.

Table 16-7: Performance characteristics of the ArrayList class

Inserting items	depends on the initial size of the array list, slower as the initial size grows large
Deleting items	depends on the initial size of the array list, slower as the initial size grows large
Linear access	slower than `Array`, slightly faster than `LinkedList`

Table 16-7: Performance characteristics of the ArrayList class

Linear access in reverse	slower than `Array`, but much faster than `LinkedList`
Random access by key	slower than `Array`, but much faster than `LinkedList`
Random access by value	slow, depending on the size of the collection, and the position of the element
Memory issues	blocks of memory are added in increments, set by the `initialSize` keyword. As items are inserted and deleted, the number blocks and the amount of empty space increases
Size and performance	insertion and deletion can be much faster than for `Array` in large collections

B-Trees

`BTree`, short for balanced tree, is an explicitly keyed collection whose items are sorted and indexed by key. You gain access to an element of a B-tree by traversing the tree. For large collections, random access to elements in a B-tree generally requires fewer steps than access to elements of an array or linked list of the same size.

The ScriptX Player implements a B-Tree in which all nodes are either branching nodes or terminal nodes, and all terminal nodes are at the same level. Branching nodes store pointers to nodes at the next level down. Terminal nodes store a pointer to the value as well as the key. This facilitates fast retrieval. At any given level, all nodes are double-linked. Since nodes are double-linked, linear access is fast, comparable to `ArrayList`, although not quite as fast as `Array`. (This assumes that all terminal nodes are in memory.)

When you create a new `BTree` object, you can specify a comparison function at initialization (using the keyword argument `cmpFunction`). This function determines the ordering of keys as elements are added to the collection.This function is analgous to the comparator function (specified using the `ltFunction` keyword) for `SortedKeyedArray`, except that it returns one of `@before`, `@same`, or `@after` rather than a `Boolean` value. The default value for `cmpFunction` is `ucmp` (universal comparison). The implications of specifying a comparison function are the same in all ordered collections.

The default comparison function (`ucmp`) is guaranteed to compare any two keys without returning an exception, provided that those keys are objects that implement `localLT`. Of course, the comparison that `ucmp` makes is meaningless, unless the two keys are objects of the same class. As a replacement for `ucmp`, you can specify any function that returns `@before`, `@same`, or `@after` to create a consistent ordering of elements.

Note – An object cannot be used as a key when a key-value pair is added to a `BTree` unless that object implements `localLT`. The default implementation of `localLT`, defined by `RootObject`, reports an exception. For more information, see the discussion of comparison that begins on page 450.

A program should not explicitly change the key of any element in a BTree object, except by removing it from the collection, setting its key, and inserting that element back into the collection. To guarantee the integrity of the sort order, always use the generics defined by `Collection` and `LinearCollection` to add, delete, or modify items in the collection.

B-trees differ from binary trees in that they have a specifiable branching factor that is greater than two. Figure 16-5 depicts a B-tree with a branching factor set to 4. (In most real applications, the branching factor should be much larger than 4.) A script determines the branching factor by setting the `brFactor` keyword at initialization. When a `BTree` object searches for a random key, it must perform a linear search at each node. Thus, the optimal branching factor increases as the size of the collection increases.

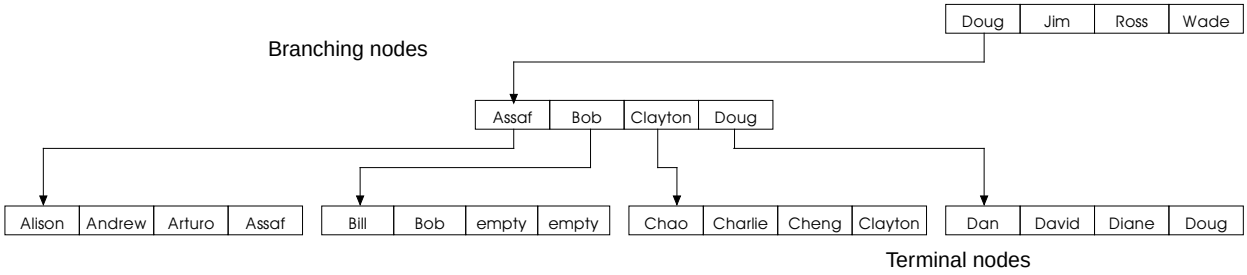


Figure 16-5: A BTree object

Since each terminal node in a `BTree` object is itself a top-level object, a terminal node is brought into memory only when it is needed. Branching nodes are automatically brought into memory as embedded objects. This makes the `BTree` class ideal for use with large, disk-based databases. For more information on top-level and embedded objects, see the section "Collections and Load Management" on page 462.

Table 16-8: Performance characteristics of the BTree class

Inserting items	fast
Deleting items	fast
Linear access	fast, comparable to `ArrayList`
Linear access in reverse	fast, comparable to `ArrayList`
Random access by key	fast, even on large collections
Random access by value	very slow, requires a linear search
Memory requirements	ideal for large collections, since only the branching nodes must be in memory
Size and performance	very little performance degradation as size increases

Hash Tables

The `HashTable` class stores unsorted items sparsely. It uses a hashing technique, which can be fast, to locate an item. The *hash table* is a fixed-size array of "buckets" that contain key-value pairs. *Hashing* provides access to a bucket in the table directly by arithmetically transforming a key that locates the bucket. Therefore, no searching is involved in locating a bucket.

In a large collection, computing the hash value and looking in the bucket is faster than looking through all of the items of an array or linked list. Any number of key-value pairs can be put in a bucket, so some searching may be needed once the proper bucket is located. ScriptX allows you to determine the number of buckets at initialization. Although the items in a hash table have explicit keys, they are unsorted.

Each time you add a key-value pair, the key is "hashed" to determine which bucket that key-value pair goes into. Then, the key-value pair is inserted into that bucket. A good hashing function is one that returns a value quickly, with little computational overhead, and that balances items in the collection evenly between buckets.

Note – If you do not designate a hashing function using the `hasher` keyword, any object used as a key must implement or inherit a `hashOf` method. For more information, see the definition of `HashTable` in the *ScriptX Class Reference*.

Table 16-9: Performance characteristics of the HashTable class

Inserting items	fast, provided that the number of elements in a bucket is small
Deleting items	fast, provided that the number of elements in a bucket is small
Linear access	not a linear collection, no linear ordering
Linear access in reverse	not a linear collection, no linear ordering
Random access by key	fast, even on large collections
Random access by value	very slow, requires a linear search
Memory issues	large, since in the current implementation of `HashTable`, all buckets must be in memory
Size and performance	very little performance degradation as size increases

Single, Pair, Triple, Quad

In addition to the classes listed in previous sections, the Collections component also includes the following classes, which inherit from `Sequence`.

- `Single` – a sequence that always contains one item.
- `Pair` – a sequence that always contains two items.
- `Triple` – a sequence that always contains three items.
- `Quad` – a sequence that always contains four items.

These classes are created with a fixed size. You cannot add or delete elements, but you can set their values. Since the number of elements never changes, an element always retains the same implicit key. Since they are bounded, `Single`, `Pair`, `Triple`, and `Quad` each offer a slight performance advantage over an array of the same size.

Collections and Threads

ScriptX developers should be aware of the possibility for thread conflict when more than one thread has access to a collection. For example, the `Collection` methods `forEach` and `forEachBinding` create an iterator. If a thread is suspended while iterating over a collection, and if another thread modifies the collection before the initial thread has a chance to resume and finish the operation, the results can be disastrous.

One solution is to allow access to a collection only within one thread. But this may be unsuitable for many programs. Another solution is to invoke methods that depend on the integrity of the collection only within code that is not preemptible, by using the global functions `threadCriticalUp` and `threadCriticalDown` to protect the critical code segments. The most general solution is to attach a gate to the thread, as in the `LockedArray` class, which is defined below.

```
class LockedArray (Array)
    inst vars lock
    instance methods
    method init self #rest args -> (
        apply nextMethod self args
        self.lock := new Lock
    )
    method acquire self -> acquire self.lock
    method relinquish self -> relinquish self.lock
    method add self key value -> (
        acquire self
        nextMethod self key value
        relinquish self
    )
    method deleteOne self value -> (
        acquire self
        nextMethod self value
        relinquish self
    )
    -- specialize other methods here
end
```

The `LockedArray` class only specializes a very minimal part of the `Collection` protocol. To create a robust locked array that cannot be accidently modified by two threads at once, specialize every method that can potentially modify the collection. Note that `IndirectCollection` is not suitable for creating a "locked" collection, since the `objectAdded` and `objectRemoved` methods are not called until after the object is added or removed.

For more information, see "Gates" on page 595 and "Preemptibility" on page 599 in Chapter 22, "Threads."

Collections and Load Management

Since ScriptX includes both a real-time incremental garbage collector and an object store, a program is not required to perform explicit memory management. A ScriptX program can reference an object without knowing whether it is in main memory, or on a storage device. If the object is on a

storage device, it is automatically retrieved and brought into main memory. However, given the performance demands of real-time media playback, you might want to manage the loading and purging of objects. An object that is in main memory is accessible almost immediately. Objects on a hard disk, CD-ROM, or network device are available only with some latency.

When you call `load` on a ScriptX collection, only the collection's internal data structure is actually loaded into memory. A collection is really a container that stores references to its member objects. Your program can reference these member objects without being concerned about where they reside—in main memory or on a storage device. Member objects are loaded on demand.

Members of a collection are not loaded when the collection itself is loaded, unless you call the global function `loadDeep`. A call to `loadDeep` causes the collection to iterate through its member objects, bringing those objects into the ScriptX heap, and any objects they refer to as well. Of course, `loadDeep` can have unintended consequences, so it should be used with caution. For example, if a member object contains a reference to a module, any object that has a name binding in that module, or in any other module that is used by that module, is also brought into main memory.

Among the ScriptX collection classes are several that load only necessary internal structures when you access a member object. For the `LinkedList` and `KeyedLinkedList` classes, only the "head" of the list is brought into memory when the collection is referenced. When you search for an item, each internal node is brought in as the program iterates through the list. Figure 16-6 depicts a ScriptX linked list after it has been loaded into memory. Note that only the head of the list is initially in main memory.

A `HashTable` object appears internally as an array of arrays, otherwise known as buckets. Each bucket is an array that contains references to individual objects. Buckets are loaded into memory only on demand. Thus, to load a few items from a hash table requires that only a relatively small part of the entire data structure be brought into memory. Similarly, the `BTree` class loads its branching nodes, but not its leaf nodes. In this way, both `HashTable` and `BTree` provide data storage that requires relatively little main memory until objects are actually needed.

All ScriptX array classes, including `Array`, `SortedArray`, `SortedKeyedArray`, and `ArrayList`, are loaded in their entirety whenever they are referenced in a program. Figure 16-6 depicts a ScriptX array after it has been loaded into memory. Note that only the array's internal data structure is initially in main memory, and not its member objects. A call to `load` (or any other reference to the collection) causes the ScriptX Player to load references to member objects, but not the objects themselves. A call to `loadDeep` brings in both the array's internal structures and the member objects themselves, including any other objects referenced by member objects.

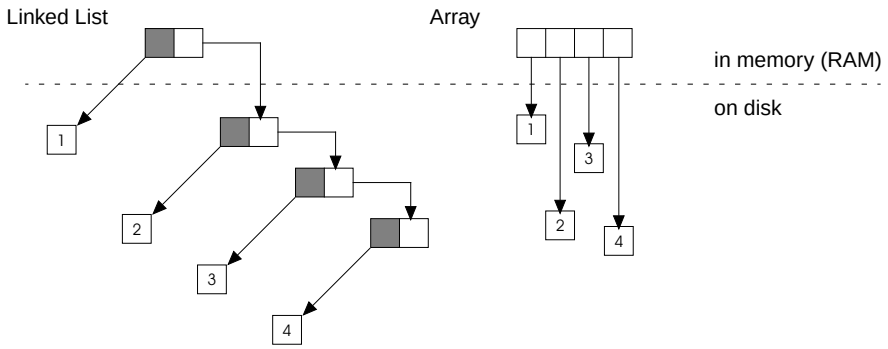


Figure 16-6: An array and a linked list, after each has been loaded but before elements of the collection have been accessed.

Once a ScriptX program has referenced a member object in a collection, internal structures such as buckets (`HashTable`), nodes (`LinkedList` and `KeyedLinkedList`), or leaf nodes (`BTree`) remain in memory until the collection as a whole is garbage collected or purged.

Strings

The `ByteString` class, which inherits from both `Sequence` and `Stream`, stores a series of integers between 0 and 255. In contrast with the `String` class, a byte string can contain an ASCII nul character. `ByteString` objects are useful for manipulating machine-specific structures. For text strings, see the `String` class in the chapter "Text and Fonts."

SequenceCursor

`SequenceCursor` is an abstract mixin class. It is designed to be mixed in with 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 a `SequenceCursor` object keeps its cursor pointing to the same item even when items are added or deleted, and it 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 runs 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 with 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 collection methods directly on the collection (such as deleteNth), rather than limiting you to the smaller set of iterator operations.

SequenceCursor is a true mix-in class, in that it provides all of 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 in to a Sequence subclass (using the class keyword), you should mix SequenceCursor in *ahead* of the Sequence subclass

```
class ArrayCursor (SequenceCursor, Array) end
```

This allows the ArrayCursor class to implement the version of methods such as deleteNth and add that SequenceCursor defines, overriding methods of the same name, defined by Array. Note that ArrayCursor still uses the same keyword arguments as Array.

The following script mixes SequenceCursor with Array to create a new class called ArrayCursor. It then creates an instance, appends three items to it, goes to the third item, returns the cursor and current item, prepends a fourth item and gets the current item again. Notice the cursor continues to point to the same item even as items are added ahead of it.

```
class ArrayCursor(SequenceCursor, Array) end

-- create an instance and add 3 strings to it
seqObj := new ArrayCursor initialSize:6
append seqObj "bear"
append seqObj "cat"
append seqObj "dog"
goTo seqObj 3 -- go to the third item
seqObj.cursor
⇨ 3
getNth seqObj (seqObj.cursor)
⇨ "dog"
prepend seqObj "amoeba" -- add another item at beginning
-- it still points to the same item
getNth seqObj (seqObj.cursor)
⇨ "dog"
```

Ranges

A range is an ordered set of values, usually numbers, ranging between a lower and upper bound. The core classes include three concrete subclasses of `Range` that incorporate numeric values: `ContinuousNumberRange`, `NumberRange`, and `IntegerRange`. The latter two inherit from `Range` through `DiscreteRange`, an abstract class that mixes in `Range` with `Sequence`. A developer is not restricted to ranges that belong to these classes. A scripted subclass of `Range` could specify a range of times or dates. A developer could even define ranges that incorporate non-numeric values, such as strings.

ScriptX implements range literals. When the ScriptX compiler evaluates a range literal, it automatically creates an instance of the appropriate subclass of `Range`. The following are examples of range literals; for more information, see "Range Literals" in the *ScriptX Language Guide*.

```
30 to 100 by 5 -- creates a discrete range
1.4 exclusive to 5.7 inclusive continuous -- continuous range
```

Each range has a value class, which specifies a class to which elements within the range must belong. For example, the value class of an `IntegerRange` object is `Integer`.

Only discrete ranges are collections, but all ranges are similar in how they incorporate their elements. Unlike collections, in which each element is a separate object, a range stores only information about its upper and lower bounds, and its increment. If the value of `increment` is `@continuous`, then a range includes every possible value between `lowerBound` and `upperBound` as an element. A range that is not continuous is discrete. A range may or may not include its upper and lower bounds.

A script can use the `withinRange` method, defined by `Range`, as a test of set membership. For discrete ranges, the `isMember` method, defined by `Collection`, is equivalent to `withinRange`.

The `size` property of a range is implemented as a virtual instance variable that indicates the number of elements it incorporates. Although discrete ranges also inherit a definition of `size` from `Collection`, their behavior as a range takes precedence. Since ranges do not actually contain their elements, their method of determining the value of `size` differs. A collection reports the actual number of elements it contains. A range calculates and reports the number of elements it incorporates. For a range whose elements are uncountable, such as a continuous number range, or a range whose elements are countable but infinite, the `size` property is not defined.

A range is immutable if none of its properties can be changed. Otherwise, it is mutable. Instances of `ContinuousNumberRange` are always immutable. The `DiscreteRange` class inherits a definition of `mutable` from `Collection`, but mutable has a different interpretation. A collection is mutible if its elements can be modified. Since a range does not store every element that it incorporates, to change a range is to change the properties of its boundaries, or the value of its increment.

Continuous Ranges

A continuous range incorporates all values between the lower and upper bounds. Figure 16-7 depicts a continuous number range with a lower bound of 0.8 and an upper bound of 6.6. In this range, the value of `includesLower` is `true`, while the value of `includesUpper` is `false`.

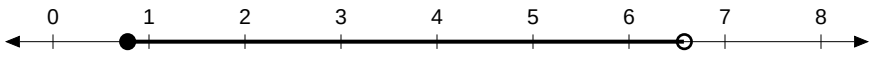


Figure 16-7: A continuous number range includes all possible points between its bounds.

The upper and lower bounds of a range are subject to the imprecision of floating point numbers, as this code example demonstrates. It first creates an instance of `ContinuousNumberRange` that includes the lower bound but excludes the upper bound. It then tests for membership in the range, using the `withinRange` method.

```
myRange := new ContinuousNumberRange \
    lowerBound:2.2 \
    upperBound:7.4 \
    includesLower:true \
    includesUpper:false
-- now perform some tests of membership in the Listener window
withinRange myRange myRange.lowerBound
⇨ true
withinRange myRange myRange.upperBound
⇨ false

-- these tests illustrate floating point imprecision
withinRange myRange 2.19999998
⇨ true
withinRange myRange 7.39999998
⇨ false
```

The ScriptX Player runs on many platforms, with several different processors, in configurations both with and without a floating point coprocessor. The examples above were produced on a Macintosh Quadra 840, a system that implements the IEEE 80-bit floating point standard. Most computer systems in use today use a lower-precision (64-bit) standard. Floating point output may vary slightly when the ScriptX Player runs on different platforms.

The global constants `posInf`, `negInf`, and `nan` are commonly used with continuous number ranges to indicate infinite or uncountable values. They do not have a value or magnitude in the same sense that other numbers do. These constants are useful for tests of comparison and set membership. For more information, see the “Numerics” chapter of the *ScriptX Components Guide*, and the “Global Constants and Variables” chapter of this volume.

Discrete Ranges

A discrete range contains a countable sequence of ordered values, separated by the range's increment. Figure 16-8 depicts a discrete range with a lower bound of 1.2, an upper bound of 6.2, and an increment of 1.25. The size of this range is 5.

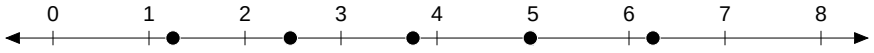


Figure 16-8: A number range is a range with discrete numeric values

A ScriptX `for` loop can iterate over elements of a discrete range. The following code sample creates the range that is depicted in Figure 16-8 and iterates over elements of the range to print out their values.

```
global myNumberRange := new NumberRange \
    lowerBound:1.2 upperBound 6.2 increment:1.25
for i in myNumberRange do print i
```

The increment of a discrete range can be negative. If so, then the value of `lowerBound` should be greater than the value of the `upperBound`. Since discrete ranges are sequences, they are also linear collections. A loop that iterates over the values in a discrete range is guaranteed to access every value in an orderly and sequential manner.

The `NumberRange` class creates a range of elements that belong to a given subclass of `Number`, consistent with the values of `lowerBound`, `upperBound`, and `increment`. In this way, a script can create a range of integers, floats, times, dates, or any other subclass of `Number`. The following script creates a range or sequence of times separated by 20 minutes, collecting the elements into an array.

```
object beginTime (Time)
    settings hours:8, minutes:30 -- set to 8:30
end
object endTime (Time)
    settings hours:17, minutes:30 -- set to 17:30 (that's 5:30 pm!)
end
object timeInterval (Time)
    settings hours:0, minutes:20 -- a 20 minute time interval
end
-- create a number range whose elements are instances of Time
object timeRange (NumberRange)
    lowerBound:beginTime upperBound:endTime increment:timeInterval
end
object myArrayOfTimes (Array) -- an array to collect them into
end
-- now collect the elements of the Range into the array
for i in timeRange collect into myArrayOfTimes i
```

Although the NumberRange class automatically sets the values of includesLower and includesUpper to true, instances of NumberRange sometimes do not technically include their upper bound. The following script creates an instance of NumberRange that illustrates this seeming contradiction.

```
global myNumberRange := new NumberRange \
    lowerBound:3.1 upperBound 6.4 increment:1.0
myNumberRange.includesUpper -- this will return true in listener window
⇨ true
withinRange myNumberRange myNumberRange.upperBound -- yet this returns false!
⇨ false
```

In this script, the upper bound is best interpreted as the highest potential value in the range. The range itself is a sequence of floating point values in which the lower bound is the starting value, but the size of this range is 4 and its increment is 1.0, so its greatest value will be the best floating point approximation of 6.1 on any given platform.

IndirectCollection

The IndirectCollection class is provided to allow you to implement special notification methods that take action when a collection is modified. The notification methods are isAppropriateObject, objectAdded, and objectRemoved. The isAppropriateObject method is invoked when a new object is about to be inserted into the collection. The objectAdded and objectRemoved methods are called after an object has been inserted into or deleted from the collection, respectively.

An IndirectCollection object does not actually store the objects inserted into the collection. Instead, it points to its target, stored in the targetCollection instance variable, which is an instance of a Collection classes. The target collection actually holds the elements of the collection. The IndirectCollection class merely overrides any methods that can potentially modify the collection so that the notification methods can be called. Then, the target collection object is called to do the actual work of the method.

Subclassing Collections

You may need to create a subclass of one of the built-in collection classes for an application you are building. You will need to examine the behavior and inheritance path of your proposed superclass carefully before you proceed, as there are a few pitfalls you might encounter.

The classes LinkedList and KeyedLinkedList are recursive data structures; each value added to the collection causes a new instance of the class to be created. If you create a subclass of LinkedList and override the init routine, you must deal with the fact that your init routine will be invoked for each object added to the list. This is not the same behavior for init in the Array or HashTable classes, where init is invoked only once for the lifetime of the collection.

Another issue is the choice of methods to override. The collection protocol defines a large number of operations, some of which have default implementations in the `Collection` class itself. Others have specific implementations in the collection subclass. One possible problem that can arise is that you may not get complete coverage of the methods if you don't override the appropriate ones.

For example, if you want to create a subclass of `Array` that prints a message every time a new object was added, you might choose to override the `append` method. This would miss any objects added using `addNth`, however. You may want to examine your target superclasses using the `getDirectGenerics` call, to check the actual implementation for each class. Alternatively, you can make use of `IndirectCollection`, which has special case behavior for adding and removing objects.

Iterators

An *iterator* is the mechanism that allows a collection to step through every one of its items in an orderly fashion. Each subclass of the `Iterator` class is designed to iterate over a particular kind of collection—for example, a `LinkedListIterator` iterates over a `LinkedList` object.

An iterator accesses objects in a collection, one item at a time, and makes it possible to access all items of that collection, regardless of their order (even if they have no explicitly-defined order). Iterators are not guaranteed to return the items in any particular order, nor will they necessarily return the items in a repeatable manner unless the collection inherits from the `LinearCollection` class. An iterator is guaranteed, however, to visit every item in the collection exactly once when you step from one end to the other (using `next` or `previous`).

Some collection operations, such as `deleteAll`, are implemented using temporary iterators. An iterator is temporarily created to perform the operation, and is then discarded. If you explicitly create an iterator on a collection, that iterator is invalidated if you modify any items in the collection through the collection's own protocol.

When using collections from the core classes, note that each `Collection` class implements methods, such as `forEach`, `forEachBinding`, and `map`, which operate on all of its member items. If possible, use these methods, rather than creating an iterator to step through the entire collection. These methods in the core classes are often highly optimized for the collection's own particular data structure.

Iterators provide the Stream protocol for use with collections. You can delete and overwrite keys and values, just as with collection protocols, but iterators do not allow you to add members.

Each collection object knows which iterator it works with it. This information is stored in the `iteratorClass` instance variable. When you call `iterate` on a collection object, it creates a new iterator of the appropriate class, setting the iterators `source` instance variable to itself.

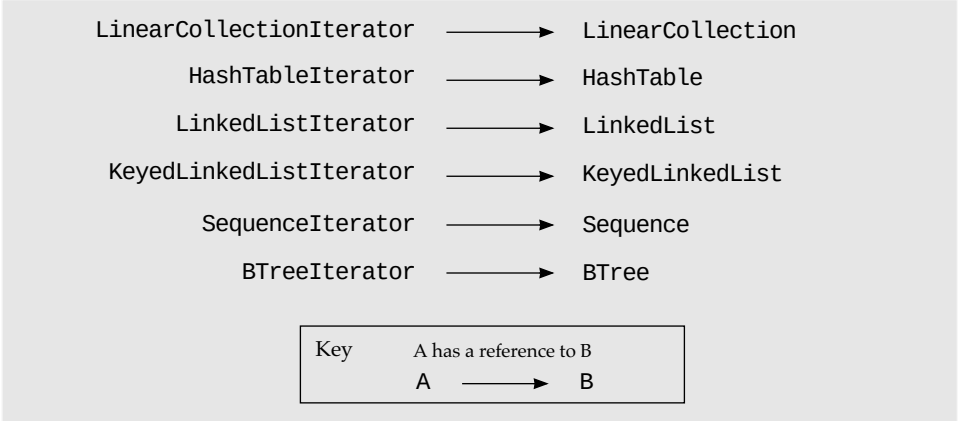


Figure 16-9: Collaboration between iterators and collections.

An iterator has a cursor that keeps track of its current position in the collection by pointing between items, as shown in Figure 16-10. When you call `next` on the iterator, it takes one step, stepping just past the next item in the collection. Even though the cursor technically points between items, for convenience we often say that the cursor "points to" the item containing `key` and `value`, since that item is the one that is currently available and ready to be operated on. As an iterator is a kind of a stream, its methods on the `Stream` class (`atFront`, `cursor` and `pastEnd`) are shown in the figure.

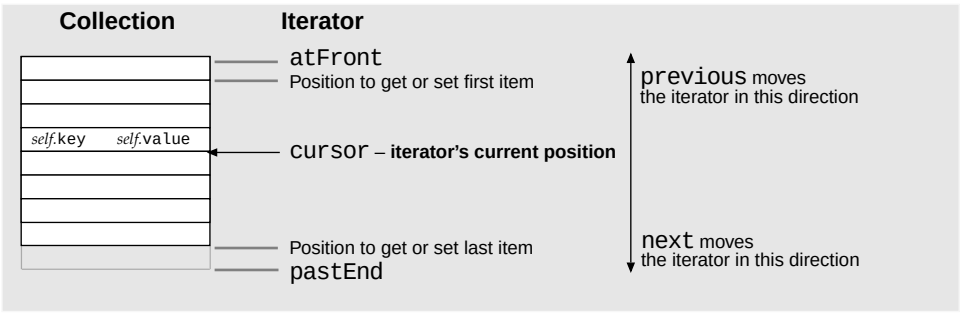


Figure 16-10:An iterator's cursor points to the current item in its collection.

Just because a collection is mutable does not mean that its iterator is writable (with `write` on `Stream`). For example, If you tried to write the value 6 to the sorted array [1, 2, 3, 4, 5], you might end up with [1 ,2 ,6 ,4 ,5], which invalidates the array, since it is no longer sorted.

Using Iterators

Here are some examples that show how to use an iterator in ScriptX. In each example, `c` is any collection, and `i` is its iterator.

The feature described in the previous note enables the following syntax for iteration, which gracefully handles an empty collection:

```
--Define iterator i on collection c
i := iterate c
repeat while (next i) do (
    v := i.value
    k := i.key
    -- do something interesting
)
```

This next script is identical to the previous one in function, except that it doesn't assign the `key` of `i` to `k`: (`read` is defined in `Stream`)

```
i := iterate c
repeat while (v := read i) != empty do (
    -- do something mind boggling
)
```

Similarly, the `excise` method leaves the cursor such that `next` should be called before any other iteration operations are called. Thus, the following loop works well:

```
i := iterate c
repeat while (next i) do (
    excise i
)
```

Most iterators in the system are seekable, so you can run them in reverse:

```
i := iterate c
seekFromEnd i 0
repeat while (cursor i > 0) do (
    v := i.value
    -- do something amazingly interesting
    previous i
)
```

You can even seek through iterators arbitrarily:

```
i := iterate c
repeat while true do (
    val := read i
    if (grottyTest val) then (
        seekFromStart i 3
        -- something profound
    )
    else (
        -- something equally profound
    )
)
```

Using the Collections Component

Since collections underlie many other components of ScriptX, there are many collection examples in other chapters. Here is a list of useful collection scripts in other chapters of this volume.

Implementing a Lookup Table

A function lookup table is a generally useful technique in programming. It is an easy way to implement dynamic behavior, since entries in the table can be modified while the program runs. A program can even switch from one lookup table to another.

In this example, the lookup table can be any explicitly keyed collection, but a hash table is generally the best choice, especially if the table is large, since it offers fast random access. A lookup table is generally not processed in linear order. For efficiency reasons, it uses `NameClass` objects as keys. The values are anonymous functions. Anonymous functions provide a kind of "wrapper" around a generic or global function that is defined elsewhere, allowing it to be stored in a collection and called with a very general syntax. For information about anonymous functions, see the *ScriptX Language Guide*. This script creates a hash table and adds several entries.

```
object lookupTable (HashTable) end
add lookupTable @groink (a b -> processGroink a b)
add lookupTable @ululate (a b -> processUlulate a b)
add lookupTable @giggle (a b -> processGiggle a b)
```

By contrast, the source table must be a linear collection, since it will be processed in order. This script creates a source table, several specialized instances of objects. It stores those objects in the source table.

```
object kiwi (Rect)
    x2:100, y2:100
    instance variables
        name:"evelina", activity:@groink
end

object aussie (PageElement)
    presenter:(new TextPresenter boundary:(new Rect x2:50 y2:100) \
        target:("gudday mate" as String))
    instance variables
        name:"mathilda", activity:@ululate
end

object texan (MouseUpEvent)
    x2:100, y2:100
    instance variables
        name:"georgia", activity:@giggle
end

object sourceTable (LinkedList) end
append sourceTable kiwi
append sourceTable texan
append sourceTable aussie
```

The `ActivityReader` class represents a process rather than a type of data. It defines an iterative method, `readTable`, that processes each object in a source collection. In this example, the `ActivityReader` class also defines simple methods for each activity.

```
global readTable
class ActivityReader (RootObject)
    instance methods
        method readTable self source table -> (
            if not isAKindOf table ExplicitlyKeyedCollection do
                report badParameter #(table, self, readTable,
                    "Expected an explicitly keyed collection")
            if isAKindOf source LinearCollection then (
                local i := iterate source
                repeat while (next i) do (
                    if hasKey table (i.value.activity) then
                        table[i.value.activity] self i.value
                    else
                        report generalError "missing table entry"
                )
            )
            else (
                report badParameter #(source, self, readTable,
                    "Expected a linear collection")
            )
        )
        method processGroink self obj ->
            format debug "Object %1, of class %2, just groinked.\n" \
                #(obj.name, (getClass obj))
        method processUlulate self obj ->
            format debug "Object %1, of class %2, just ululated.\n" \
                #(obj.name, (getClass obj))
        method processGiggle self obj ->
            format debug "Object %1, of class %2, just giggled.n" \
                #(obj.name, (getClass obj))
end
global myActivityReader := new ActivityReader
readTable myActivityReader sourceTable lookupTable
```

The two final lines of this script create an instance of `ActivityReader`, calling the `readTable` method on the reader with the source and lookup tables as parameters.

Specializing IndirectCollection to Enforce Uniformity

This example creates a subclass of `IndirectCollection` that enforces uniformity of keys and values. `UniformCollection` could be used in any application where type validation is required.

The Collection protocol includes a number of “descriptive” instance variables. These variables have no functional behavior, but they indicate properties of a collection that other objects in the system might want to query. They are implemented as virtual instance variables, and can be specialized at either the instance or class level. The `UniformCollection` class specializes four getter

methods that are defined by `Collection`: `uniformityGetter`, `uniformityClassGetter`, `keyUniformityGetter`, and `keyUniformityClassGetter`. By default, a uniform collection uses the generic function `getClass` to test both keys and values that are added to the collection. Specializing `uniformityGetter` or `keyUniformityGetter` in a class or instance to return `@commonSuperclass` changes the acceptance criteria to use the generic function `isAKindOf`.

Note that indirect collections redirect certain methods to the target collection. An indirect collection does not otherwise test whether a method can be called on its target collection. For example, if the target collection is a `Quad` object, a bounded array with 4 elements, the indirect collection itself does not report an exception when a script attempts to add a fifth element. It is up to its target collection to report such errors. However, there is no reason a developer cannot specialize `IndirectCollection` to detect such errors before the method is redirected to the target collection.

The following script creates the `UniformCollection` class:

```
class UniformCollection (IndirectCollection)
    instance vars
        keyType -- class to which keys must belong
        valueType -- class to which values must belong

    instance methods
        method init self #rest args #key keyType: valueType: -> (
            apply nextMethod self args
            if isAKindof self.targetCollection \
                    ImplicitlyKeyedCollection then (
                self.keyType := @implicit
            )
            else (
                if keyType = unsupplied then
                    report keywordRequired @keytype
                else if isAKindOf keyType Behavior then
                    self.keyType := keyType
                else
                    report badParameter #(keyType, init, self,
                        "keyType must be a class.")
            )
            if valueType = unsupplied then
                report keywordRequired @valueType
            else if isAKindOf valueType Behavior then
                self.valueType := valueType
            else
                report badParameter #(valueType, init, self,
                    "valueType must be a class.")
        )
        -- keyUniformityGetter and uniformityGetter can be
        -- specialized at the class or instance level
        method keyUniformityGetter self -> @sameClass
        method keyUniformityClassGetter self -> self.keyType
        method uniformityGetter self -> @sameClass
        method uniformityClassGetter self -> self.valueType
        method isAppropriateObject self addedObject -> (
            case (self.uniformity) of
                @sameClass: (
```

```
                    if (getClass addedObject == self.valueType) then
                        return true
                    else
                        return false
                )
                @commonSuperclass:(
                    if (isAKindOf addedObject self.valueType) then
                        return true
                    else
                        return false
                )
                otherwise:
                    report generalError \
                        "inappropriate value for uniformity"
            end
        )
        method add self key value -> (
            if self.keyType == @implicit then (
                nextMethod self key value
            )
            else (
                case (self.keyUniformity) of
                    @sameClass: (
                        if (getClass key == self.keyType) then
                            nextMethod self key value
                        else
                            report badkey (#(self, key) as Pair)
                    )
                    @commonSuperclass:(
                        if (isAKindOf key self.keyType) then
                            nextMethod self key value
                        else
                            report badkey (#(self, key) as Pair)
                    )
                    otherwise:
                        report generalError \
                            "inappropriate value for keyUniformity"
                end
            )
        )
end
```

Note that in this example, `UniformCollection` specializes only the `add` method, for the sake of brevity. In a more robust implementation, `UniformCollection` could specialize other methods that can add new objects to the target collection, such as `append`, applying the appropriate test to the key.

The first test script tests the `UniformCollection` class by creating an instance of `UniformCollection` with an `Array` object as its target collection.

```
-- test of UniformCollection class
global myArray := new UniformCollection \
    targetCollection:(new Array initialSize:10) \
    valueType:String
method uniformityGetter self {object myArray} -> @commonSuperClass
add myArray empty ("Grok" as String)
add myArray 1 "Voodoo"
```

After the test script runs, `myArray` contains two items, a string and a string constant. Specializing `uniformityGetter` at the instance level relaxes the "appropriate object" criteria, allowing any instance of a subclass of `String` to be added to the collection.

```
myArray
⇨  #<UniformCollection+ over #("Voodoo", "Grok")>
```

In the next test, `myBTree` does not specialize `uniformityGetter`. Only actual `String` objects can be added to the collection.

```
global myBtree := new UniformCollection \
    targetCollection:(new BTree) \
    keyType:NameClass \
    valueType:String
add myBtree @elephant ("Trunk" as String)
-- test that you cannot add a StringConstant value to myBTree
guard (
    add myBtree @pig "Snout"
    print "this line should not print!"
)
catching
    all: (
        print "attempt to give a pig a snout foiled"
        add myBtree @pig ("Tail" as String)
        caught undefined
    )
end
```

Within the `guard` construct, the attempt to add a string constant value (`"Snout"`) fails. The `guard` construct catches the exception, and adds a proper `String` object to the collection.

```
myBTree
⇨  #<UniformCollection over #(@elephant:"Trunk", @pig:"Tail") as Btree>
```

CHAPTER

17

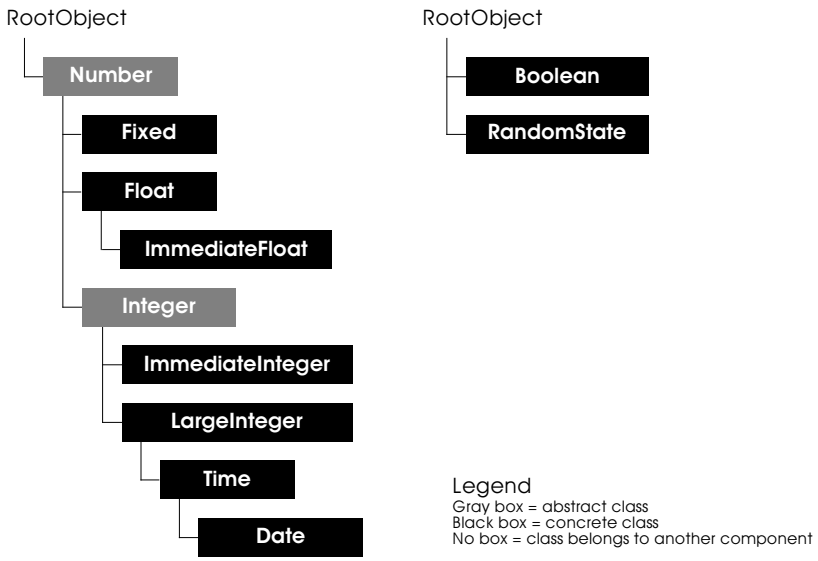
Numerics

The Numerics component contains the classes that represent numbers, Boolean values, and random number generators.

ScriptX is a pure object system, meaning that every expression in the ScriptX language returns an object. In some object systems, the base numeric data types such as integer and float are not themselves objects. In ScriptX, all numeric data types are objects. Developers use the same methods for saving, coercing, comparing, printing, and retrieving numbers that they use for all other classes of objects.

Classes and Inheritance

The class inheritance hierarchy for the Numerics component is shown in the following figure:



The following classes form the Numerics component. In this list, indentation indicates inheritance.

`Number` – Abstract class for representing all types of numbers.

`Fixed` – Numbers to $\pm 2^{15}$ (±32,768) with 16 bits of fractional info ($1/2^{16}$).

`Float` – Floating-point numbers, for numbers larger (either in integer size or decimal precision) than `Fixed`.

`ImmediateFloat` – Floating-point numbers that do not require the range or precision offered by the `Float` class.

`Integer` – Numbers without any fractional or decimal part.

`ImmediateInteger` – Integers to $\pm 2^{29}$ (536,870,912).

`LargeInteger` – Integers to $\pm 2^{63}$.

`Time` – Represents a fixed time in hours, minutes, seconds, and ticks.

`Date` – Represents a fixed date with the day, month, and year.

`Boolean` – Represents the values `true` and `false`.

`RandomState` – A random state generator for random numbers.

Conceptual Overview

The `Number` subclasses (except `Date` and `Time`) are not usually created by calling the generic function `new`. Instead, whenever the compiler encounters a numeric value in a script, it automatically coerces it to the appropriate `Number` object. Utility routines convert numeric values to and from number objects. You must explicitly create `Date` and `Time` objects with `new` or `object`.

Numeric operations are of two types: methods defined in the `Number` family of classes, and operators defined by the language (such as +, –, *, and /) that translate into private methods.

How Numerics Work

ScriptX implements five storage classes for numeric values, offering a range of tradeoffs for precision, storage, and processing requirements. For efficiency reasons, all the numeric classes are sealed and should not be subclassed. (“Sealed” is defined in the Glossary at the end of this manual.)

Table 17-1: The Number Classes

Numerics class	Precision
`ImmediateInteger`	1-bit sign, 29-bit integer (-536,870,912 to 536,870,911)
`LargeInteger`	1-bit sign, 63-bit integer
`Fixed`	1-bit sign, 15-bit integer, 16-bit fractional
`ImmediateFloat`	1-bit sign, 21-bit mantissa, 8-bit exponent
`Float`	1-bit sign, at least 52-bit mantissa, at least 11-bit exponent (64-bit IEEE 754)

Although the ScriptX Player implements five number classes, a developer can usually depend on ScriptX to coerce any value to a numerics class that offers the desired precision and range. The result of an operation is promoted transparently if the class of the operands lacks the precision or range to store the results.

Although there are times when a programmer may want to be aware of how a number is stored, you can generally ignore type at the scripter level. Many programmers may be accustomed to languages which distinguish between integer and floating-point division, such as C and C++. In ScriptX, integers are automatically promoted to floating-point before they are divided.

If you are developing a title that depends on high floating-point precision, you should be aware of the following.

- The ScriptX Player runs on many low-end machines that lack a floating-point coprocessor. Since these target systems must emulate floating-point operations with integer arithmetic, their performance may lag significantly. Developers need to be aware of the performance of their title on a low-end target machine. The `Fixed` class can be used to avoid floating-point operations.
- Although the ScriptX Player currently prefers systems that implement the established 64-bit IEEE 754 floating-point standard, there may be slight differences in precision across platforms.

Coercion of Numbers

A script can coerce numbers between types using the coercion construct `as`, for example: `12.3 as Fixed`. Of course, coercion may result in loss of precision. For more information on coercion, see the *ScriptX Language Guide*.

ScriptX automatically converts numbers to the class that offers the optimal storage for the level of precision that is needed. For example, when you multiply an `Integer` object and a `Float` object, the integer is automatically converted into a float. In other words, for multiplication there is a notion of the class with higher precision that should be used to contain the result.

```
x := 2 as ImmediateInteger
y := 2.75 as ImmediateFloat
z := x * y
⇨ 5.5
getClass z
⇨ ImmediateFloat
```

Implicit conversion matches message parameters to function and method parameters. In the following example, even though it is passed by reference, `y` is automatically promoted to `Float` within the function `cosineSquared`.

```
function cosineSquared x -> (1 - (sin x * sin x))
y := 1 as ImmediateInteger
cosineSquared y
⇨  0.291926581726429
```

Implicit conversion occurs when there is an overflow. The result is automatically promoted to a wider class, that is, a class capable of containing it. For example, `ImmediateInteger` overflows to `LargeInteger`.

```
x := 536870911; y := 536870911 -- the largest immediate integer
z := x + y
⇨ 1073741822 -- it returns a large integer
```

ScriptX automatically promotes a result for greater precision as well as range.

```
a := 1.4652 as ImmediateFloat
b := 2.25 as ImmediateFloat
c := a * b
⇨ 3.29669988155365
```

```
getclass c
⇨ Float
```

When the result of an operation fits in a smaller class, objects can be demoted as well as promoted.

```
x := 5.5 as ImmediateFloat
y := 2.75 as Float
z := x / y
⇨ 2
getClass z
⇨ ImmediateInteger
```

Boolean Operations

ScriptX defines a `Boolean` class. The `Boolean` class is associated with five operations: `and`, `or`, `xor`, `not`, and `eq`. Some of these operations are called with operator syntax, and others with function syntax. Two of these operations, `xor` and `eq`, are defined as global functions, while `and`, `or`, and `not` are defined as operators in the ScriptX language.

```
x := true
y := false
x and y -- operator syntax
false
x or y -- operator syntax
true
xor x (x or y) -- function syntax
false
```

The `Boolean` class has two global instances, `true` and `false`. Since `true` and `false` are system constants, it is never necessary to create a new instance.

Operations on Numbers

Since ScriptX numbers are objects, all operations on numbers are method calls on objects. The ScriptX language contains the following four operators that can operate on objects in the `Number` class. These symbols substitute for substrate methods that are hidden from the scripter level.

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

Other operations on numbers use method syntax. ScriptX defines most numeric operations on the `Number` class, even those that return a meaningless result on some subclasses of `Number`. Values are promoted internally to a class that can perform the operation. For example, a `Date` object is promoted to `Float` and returns a floating-point value when the `sin` method is called on it.

ScriptX defines several numeric constants, including `e`, `pi`, `piDiv2`, and `sqrt2`. In addition, ScriptX defines three infinite constants: `posInf`, `negInf`, and `nan`, where `nan` is interpreted as an "uncountable" number. Operations that cannot return a finite value automatically overflow to return one of these infinite constants. For more information on numeric constants, see the "Global Constants and Variables" chapter of the *ScriptX Class Reference*.

```
exp 11356.6 -- e^11356.6
⇨ posInf
```

Table 17-2 lists mathematical functions that comprise the Number protocol.

Table 17-2: Operations defined by the Number class

`abs`	absolute value	`max`	maximum
`acos`	arc cosine	`min`	minimum
`asin`	arc sine	`mod`	modulus
`atan`	arc tangent	`negate`	negate
`atan2`	inverse arc tangent	`power`	power of
`ceiling`	round up to integer	`radToDeg`	radians to degrees
`cos`	cosine	`rand`	random number
`cosh`	hyperbolic cosine	`rem`	remainder
`degToRad`	degrees to radians	`round`	nearest integer
`exp`	exponent	`sin`	sine
`floor`	round down to integer	`sinh`	hyperbolic sine
`frac`	fractional part	`sqrt`	square root
`inverse`	`inverse (1/x)`	`tan`	`tangent`
`log`	`logarithm`	`tanh`	`hyperbolic tangent`
`ln`	`natural logarithm`	`trunc`	`integer part`

Some operations on numbers may report the result as `Float`, even though it could be converted back to an integer.

```
x := 4 -- an integer value
y := ln (exp x) -- ln and exp are inverse operations
⇨ 4.0
getClass y
⇨ Float
```

Note – There may be differences across platforms in when conversion between integer and floating-point classes takes place.

Operations on Integers

The `Integer` class defines three bit operations: `length`, `lshift`, and `rshift`. Bit operations are defined to be independent of the number of bits in the actual representation of integers. The `length` method returns the number of bits needed to represent an integer in two's complement.

```
length 536870911 -- the largest immediate integer
```

```
⇨  30
length 0
⇨  2
length 2
⇨  3
```

Bit shift operations are logical, not arithmetic. They are defined as if integers were stored in two's complement form. It is likely that every platform on which the ScriptX Player ever runs will use two's complement numbers internally, but all that matters is that these functions operate as if they were two's complement. Note the effect in the following example of shifting an immediate integer to the right, and then to the left, into the sign bit.

```
x := 536870911 -- largest immediate integer
y := rshift x 1
⇨  268435455 -- low-order bit is on; one bit was shifted out and lost
z := lshift y 1
⇨  536870910 -- one bit was shifted out and lost
a := lshift z 1
⇨  -4 -- shifted into the sign bit
b := lshift a 1
⇨  -8
c := rshift b 1
⇨  536870908 -- now the sign bit is off again
```

Note – ScriptX does not have unsigned data types.

The `Integer` class also defines four bitwise logical operations: `logicalAnd`, `logicalOr`, `logicalXOr`, and `logicalNot`. A "bitwise" operation on two integers applies the logical operation to the first bit of the first and second integers, putting the result into the first bit of the resulting integer, then continues to the second bit, and so on.

```
logicalAnd 16 15
⇨  0
logicalOr 16 15
⇨  31
logicalXOr 16 15
⇨  31
```

Immediate Objects

An immediate object is a ScriptX object that "collapses" into the upper 30 bits of its own pointer. ScriptX defines two classes of immediate objects: `ImmediateInteger` and `ImmediateFloat`. These classes can save storage and decrease allocation overhead in applications that do not require the range or precision of `LargeInteger` and `Float`. For technical specifications of the `ImmediateFloat` and `ImmediateInteger` classes, see their respective class definitions in the *ScriptX Class Reference*.

In ScriptX, the two low-order bits of a pointer comprise an object tag, indicating whether the remaining 30 bits contain a pointer to a full-fledged ScriptX object, an instance of one of the two immediate classes, or a non-pointer. Figure depicts both a regular object, in which the high 30 bits points to an object somewhere in the ScriptX Heap, and an immediate object.

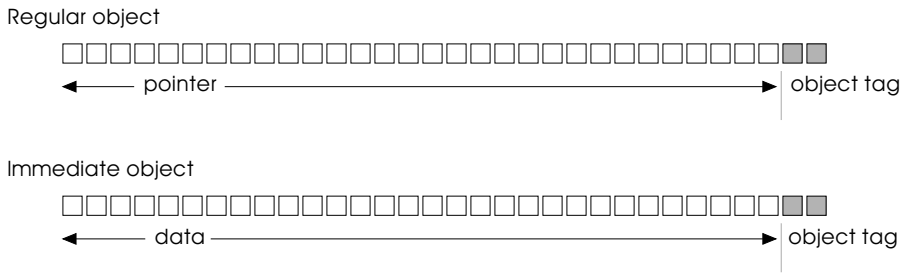


Figure 17-1: ScriptX pointers

Since a regular object requires additional storage for a class tag and for memory management, even the simplest regular objects require at least 16 bytes of storage. In addition, a regular object requires incremental time in each garbage collector cycle for tracing. An immediate object requires only the 4 bytes that would otherwise be occupied by its pointer. Since space for an immediate object is allocated directly within its own pointer, it imposes no overhead on the garbage collector. Of course, `ImmediateInteger` and `ImmediateFloat` are sealed classes, like other `Number` classes. But in every other respect, they are just like any other class in the ScriptX core classes.

For most purposes, developers can ignore the existence of immediate objects. Even though they are not full-fledged objects, immediate objects define all the operations that can be performed on regular objects. ScriptX determines transparently whether an object should be an immediate or a regular object. When a number is assigned to an untyped variable, ScriptX automatically creates an instance of the appropriate class of `Number`. For example, a positive or negative integer that is within range automatically creates an instance of `ImmediateInteger`.

```
getClass ((power 2 29) - 1) -- the largest possible immediate integer
⇨  ImmediateInteger
getClass (power 2 29)
⇨  LargeInteger
```

Similarly, a floating-point constant with six or fewer decimal digits is assigned automatically to an `ImmediateFloat` object. A constant that requires higher precision is automatically assigned to a `Float` object.

```
getClass (3.14159) -- six significant digits of pi
⇨  ImmediateFloat
getClass (3.14159265) -- more than six digits of pi
⇨  Float
```

It is not necessarily better to store a floating-point value as an `ImmediateFloat` object. The advantage of the `ImmediateFloat` class is in memory allocation. An immediate object requires only four bytes of storage. A regular `Float` object requires 32 bytes in memory, not counting the 4 bytes for its pointer. Although immediate float values impose less overhead for allocation, they offer less precision. Calculations with `ImmediateFloat` values also pay a performance penalty, since all floating-point values must be converted into a machine-specific format for calculation. While the `ImmediateFloat` class is preferred for allocation-intensive applications where its lower precision is adequate, the `Float` class may be more efficient for use with calculation-intensive applications. This trade-off is machine-dependent.

Equality comparators work differently for immediate objects. For other classes, the global function `eq` returns `false` if two objects are not the same object. Immediate objects are incorporated into their own pointers, so two immediate objects that have the same value appear to be pointers to the same object. The global function `eq` tests whether two variables point to the same object. It does this by testing whether or not their pointers are identical. That's why `eq` works differently with immediate objects. The following example illustrates this paradox.

```
global a := 7.0 -- a is an ImmediateFloat object
global b := 7.0 -- b is an ImmediateFloat object
a == b -- equivalent to eq a b
⇨  true
```

A `Float` object is a full-fledged object, so `eq` returns `false`.

```
global a := 7.0 as float -- coerce to Float
global b := 7.0 as float -- coerce to Float
a == b -- equivalent to eq a b
⇨  false
```

The global function `eq` is equivalent to `==`, the ScriptX identity operator. For more information on comparison, see "Comparing Objects" on page 633.

Fixed and Floating-Point Precision

Note that the ScriptX Player runs on a variety of platforms, and under a variety of processor configurations, each of which has its own internal representation of floating-point numbers. `Float` and `ImmediateFloat` values are converted into machine-specific values for calculation purposes. For example, 68040-based Macintosh systems use an 80-bit standard internally, while PowerPC-based processors use a 64-bit standard. Developers should be aware that there may be slight differences in the precision of the same process running on different platforms under the ScriptX Player.

The `Fixed` class offers a compromise between the greater range of `Float` and `ImmediateFloat` and the low processor overhead of integer arithmetic. Numeric values that are stored as `Fixed` objects are not really floating-point values. ScriptX performs basic addition, subtraction, and multiplication on

`Fixed` objects using integer routines. These calculations on `Fixed` objects will have identical results on any platform that runs the ScriptX Player, provided that they do not overflow and require a `Float` value.

Since fixed values use 16 bits to encode the fractional part of a number, they are accurate to five significant digits after the decimal point. Since a `Fixed` object is not an immediate object, each fixed value requires a full 16 bytes in memory, plus the four bytes for its body pointer. And because a fixed value is a full-fledged object, it adds incrementally to the garbage collector's overhead.

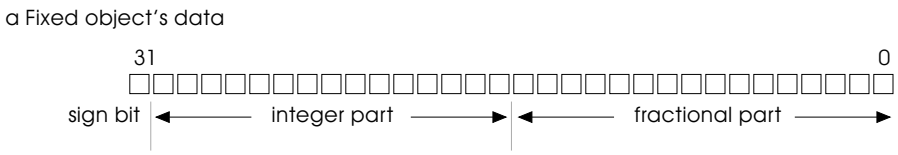


Figure 17-2: A Fixed object stores its data in 32 bits

Only integer arithmetic can be performed on fixed values. For long division, fixed values are converted internally to `Float` and the result is promoted to `Float`. Operations that require floating-point arithmetic, such as the exponential, logarithmic, or transcendental functions, cause a `Fixed` object to be promoted automatically to the `Float` class.

```
global x, y, z, w
x := 5.2 as fixed
y := 5.3 as fixed
z := 5.2/5.3 -- calculation is performed internally on float
⇨ 0.981124877929688
getClass z -- result is promoted to float
⇨ float
w := sin x
⇨  ⇨ -0.8834560855154513
getClass w
⇨ float
```

Dates and Times

The current version of the ScriptX Player uses the date and time facilities of the underlying operating system. On both OS/2 and Windows, the system clock reckons time in ticks since January 1, 1970, with an ending date of December 31, 2105. The Macintosh also maintains a tick count in an integer format, but with a different base date, January 1, 1904, and ending date, December 31, 2039.

The ScriptX Player sacrifices range for compatibility, using the beginning date from OS/2 and Windows, and the ending date for Mac/OS. Thus, ScriptX Player time begins on January 1, 1970, and ends on December 31, 2039. In a future release, the ScriptX Player will probably implement its own date and time package, with a wider format for a much wider range of dates.

Both the Clocks and Players components make use of the `Time` class. When the ScriptX Player starts up, it creates two global instances of `Clock`: `theEventTimeStampClock` returns a `Time` object that is used to timestamp

events as they occur, while `theCalendarClock` is an instance of `CalendarClock`, and it returns a `Date` object which indicates the current date and time.

`Time` objects store time internally as a long integer, together with a scale. The `scale` instance variable, defined by `Time`, specifies the number of ticks per seconds. The following equations demonstrate the relationships between the instance variables of a `Time` object. The example begins by taking the current time from `theEventTimeStampClock`. The instance variable `ticks`, defined by this clock, records the number of ticks since the ScriptX Player started up.

```
global theTicks := theEventTimeStampClock.ticks
global theScale := theEventTimeStampClock.scale
global hours := floor(theTicks/(60 * 60 * theScale))
global minutes := floor((theTicks - (hours * 60 * 60 * theScale)) /
       (60 * theScale))
global seconds := floor((theTicks - (hours * 60 * 60 * theScale) -
       (minutes * 60 * theScale))/theScale)
```

The `Time` class defines several instance methods—`addHours`, `addMinutes`, and `addSeconds`—which add the corresponding number of units to the time, taking into account the scale of the `Time` object. Although these methods accept a non-integer value, they truncate it and operate on only the integer part. A simple addition or subtraction operation adds or subtracts the corresponding number of ticks. `Time` objects cannot be multiplied into or divided by each other. An arithmetic operation on two `Time` objects with different scales returns a `Time` object with a scale equal to that of the first operand.

```
global a := new time
a.scale := 1000
addHours a 500
⇨  500:0:0:0 as Time
global b := new time
b.scale := 100
addHours b 400
⇨  400:0:0:0 as Time
global c := a + b
⇨  900:0:0:0 as Time
c.scale
⇨  1000
```

Although `Date` inherits from `Time`, it has its own internal storage format. A `Date` object stores each element of a date—month, day, hour, and so forth—in a byte, except for the year, which is stored as a short integer. To other ScriptX objects, a date is a large integer. However, when a date is saved to or retrieved from the object store, it uses its specialized internal format to maintain a representation of itself that is platform-independent.

`Date` inherits the instance variables `seconds`, `minutes`, `hours`, and `scale` from `Time`. `Date` also defines `dayOfWeek`, `dayOfMonth`, `month`, and `year`. For the `Date` class, `scale` is read-only and is always 1. In addition to the

instance methods defined by `Time`, the `Date` class defines methods for adding days, months, and years to a date. Note the behavior of `addMonths` in the following example.

```
object myDate (Date)
    year:1996, month:@January, day:31, hours:8
end
⇨  Wed Jan 31 08:28:54 1996 as Date
addmonths myDate 1
⇨  Thu Feb 29 08:28:54 1996 as Date -- leap year
addmonths myDate 1
⇨  Fri Mar 29 08:28:54 1996 as Date
```

A `Date` object can be coerced to a string. It can also be coerced to any other numerics class, including `Time`. The resulting magnitude represents the number of seconds since the base date, and is platform dependent.

```
theCalendarClock.date as String -- platform independent
⇨  "Mon Feb 19 14:31:04 1996"
theCalendarClock.date as Time -- platform dependent
⇨  807662:31:33:0 as Time
theCalendarClock.date as LargeInteger -- platform dependent
⇨  2907585141
```

Certain arithmetic operations are permitted on dates. Subtracting one `Date` object from another results in a `Time` object representing the time difference between the two dates. You can add or subtract a `Time` object from a `Date` object, and the result will take into account the scale of the `Time` object. Adding or subtracting any number from a `Date` object adds or subtracts the corresponding number of seconds. Addition is prohibited for the `Date` class.

Numerics Example

The following example demonstrates the Numerics component.

Net Present Value of a Winning Lottery Ticket

The numeric classes are sealed, meaning that it is impossible to create a subclass or specialize a method. Numeric objects are often elements of collections or instance variables that belong to other objects.

An object-oriented system can create classes of objects that perform mathematical calculations on themselves. Objects can also be designed to perform the range and type checking that a robust mathematical program requires. Other features that can be built into objects include printing and data conversion. In this sample script, a list of individual payments on a winning lottery ticket knows how to print a listing of itself, report its total payout, and calculate its net present value. Individual payments are also objects that perform type checking.

Net present value is one of the fundamental techniques used to value financial assets such as stocks, bonds, and mortgages. The following example calculates the net present value of a winning lottery ticket. The California Lottery, like other state and national lotteries, publicizes the total amount that is paid out in winnings over a period of years. To win a $20 million prize is really to earn $1 million per year for 20 years.

The net present value of the winning ticket is its equivalent value as a lump-sum payment, received today. From the point of view of the California Lottery Commission, it is the amount of money the commission must set aside to meet future obligations on that ticket. The exponential function "discounts" a future payment back to the present. The discount rate could be thought of as an interest rate or growth rate—it is the growth rate of money that is earning continuously compounded interest. The following mathematical formula discounts each payment from time *t* to the present.

$$PV_0 = x_t \, e^{-rt}$$

where r is the discount rate, expressed per units of time, and t is the time at which the payment x_t is received. The following script implements this operation in ScriptX, but in a procedural style. It performs no type or error checking. This example returns the present value of a single payment of $1 million received on July 1, 2000, as of July 1, 1995, discounted at 10%.

```
function presentValue dollars discountRate beginDate endDate -> (
    local conversion := (60 * 60 * 24 * 365.25)
    local x := endDate as LargeInteger/conversion
    local y := beginDate/conversion
    return dollars * exp(negate(discountRate * (x - y)))
)
object date95 (Date) year:1995, month:@july end
object date00 (Date) year:2000, month:@july end
global pv := presentValue 1000000 0.10 date95 date00
⇨  577048.5628947
```

Since numeric values are object-based in ScriptX, we can use ScriptX to create a truly object-oriented program that calculates net present value. First we define the `Payment` class. A payment is a simple collection, an ordered pair of values where the first value must be a `Date` object, and the second a `Number`, representing an amount due. `Payment` specializes the `add` method to check that objects are of the correct type as they are added.

```
class Payment (Pair)
    instance methods
        method add self key myValue -> (
            -- specialize the add method to do type checking
            if ((self.size = 0) and (not getClass myValue = Date)) do (
                format debug "first element not a date\n" undefined @normal
            )
            if ((self.size = 1) and (not isAKindOf myValue Number)) do (
                format debug "second element not a number\n" undefined @normal
            )
```

```
            apply nextMethod self key myValue
        )
        method presentValue self today discountRate -> (
            if not isAKindOf discountRate Number do (
                format debug "third arg not a number\n" undefined @normal
            )
            -- handles dates that have already been converted to large integer
            if (getClass today = Date) then (
                -- there are (60 * 60 * 24 * 365.25) seconds in a year
                local secondsPerYear := 31557600
                local y := (today as LargeInteger)/secondsPerYear
                local z := (self[1] as LargeInteger)/secondsPerYear
                -- return the present value of the payment
                return (self[2] * exp (negate(discountRate * (z - y))))
            ) else (
                format debug "second arg not a date\n" undefined @normal
            )
        )
        method prin self arg stream -> (
            prin "You will receive $" @unadorned debug
            prin self[2] @unadorned debug; prin " on " @unadorned debug
            prin (self[1] as String) @unadorned debug
            prin "\n" @unadorned debug
        )
end -- Payment
```

The `PaymentList` class stores a collection of `Payment` objects. It is implemented as an indirect collection because the `IndirectCollection` class provides a protocol for examining objects when they are added to and removed from a collection. An indirect collection automatically calls `isAppropriateObject` before an object is added to its target collection. The `PaymentList` class specializes `isAppropriateObject` to allow only instances of `Payment` to be added to its target collection.

The target collection is set by script in this example. Since it is a sorted array, members of the collection are automatically inserted in sort order. `Payment` objects are pairs in which the first element is always a date, so members of the collection are sorted by date as they are added to a `PaymentList` object. Although an insertion sort creates overhead when elements are added to the collection, it allows for the faster access and retrieval.

```
class PaymentList (IndirectCollection)
    instance methods
        method init self #rest args -> (
            apply nextMethod self targetCollection:(new SortedArray) args
        )
        -- this method is called automatically before an object is added
        -- object does not get added to the collection unless it returns true
        method isAppropriateObject self addedObject -> (
            -- check that it is the right kind of object
            if not isAKindOf addedObject Payment then (
                format debug "not a pair\n" undefined @normal
                return false
            ) else (
```

```
            return true
        )
    )
    -- this method is called automatically after an object is added
    method objectAdded self key obj -> (
        format debug "Your total is %*\n" (self.total) @normal
    )
    -- total is an example of a virtual instance variable
    method totalGetter self -> (
        local sum := 0
        forEach self (x -> sum := sum + x[2]) undefined
        return sum
    )
    -- generic prin takes care of all printing functions
    method prin self arg stream -> (
        forEach self (x->prin x @unadorned debug) undefined
    )
    method netPresentValue self discountRate -> (
        local sum := 0 -- initialization
        local today := theCalendarClock.date
        -- calls function, defined inline, on each member of collection
        forEach self (x -> sum := \
               sum + presentValue x today discountRate) undefined
        return sum
    )
end -- PaymentList
```

The method `netPresentValue` calls every member of the `PaymentList` collection, passing the discount rate and the current date as arguments. Since the value of `theCalendarClock.date` is continuously being updated, it saves a local copy of the date before it begins. A date is a kind of large integer, but it is stored internally in a more complex format so that it can be saved to the object store as a platform-independent format. For this reason, developers should avoid operations that require a date to be continuously updated or changed. For example, a `Date` object should not be used as if it were a clock.

Now we are ready to test the `PaymentList` class by adding objects. The test script adds 20 payments to a payment list, representing annual payments of $1 million on a winning lottery ticket.

```
-- create an instance, and then add some data to it
object myWinnings (PaymentList) end
for i in 1 to 20 do (
    add myWinnings empty (new Payment \
           values:#(new Date year:(1995 + i) month:@january,1000000))
)
-- these tests are a function of the current date
netPresentValue myWinnings 0.06 -- present value at 6% discount rate
⇨  11088641.504961
netPresentValue myWinnings 0.10 -- present value at 10% discount rate
⇨  7965826.92870522
```

C H A P T E R

Events and Input Devices

18

Events are objects that represent a change in state, a change that occurs at an instant in time. Some events are initiated by the system or the user: a key is pressed, the mouse moves, a colormap is changed. ScriptX also allows for the creation of new classes of events. Any object, script, or function can create an event, or post interests in events and receive them.

The Events component has two roles in ScriptX. One role is to receive and process system events—events that originate with underlying hardware devices. These events are often initiated either directly or indirectly by the user. The event system provides interfaces to input devices such as a mouse or keyboard. Through events, ScriptX receives real-time input from the user. Input devices, and the classes of events that they generate, are covered later in this chapter.

Events have a second role—to serve as a notification system, typically between processes that run in different threads. Of course, objects can communicate without the event system, but the event system allows for much greater indirection. A title can define its own events and set up an arbitration mechanism that delivers them to interested receivers.

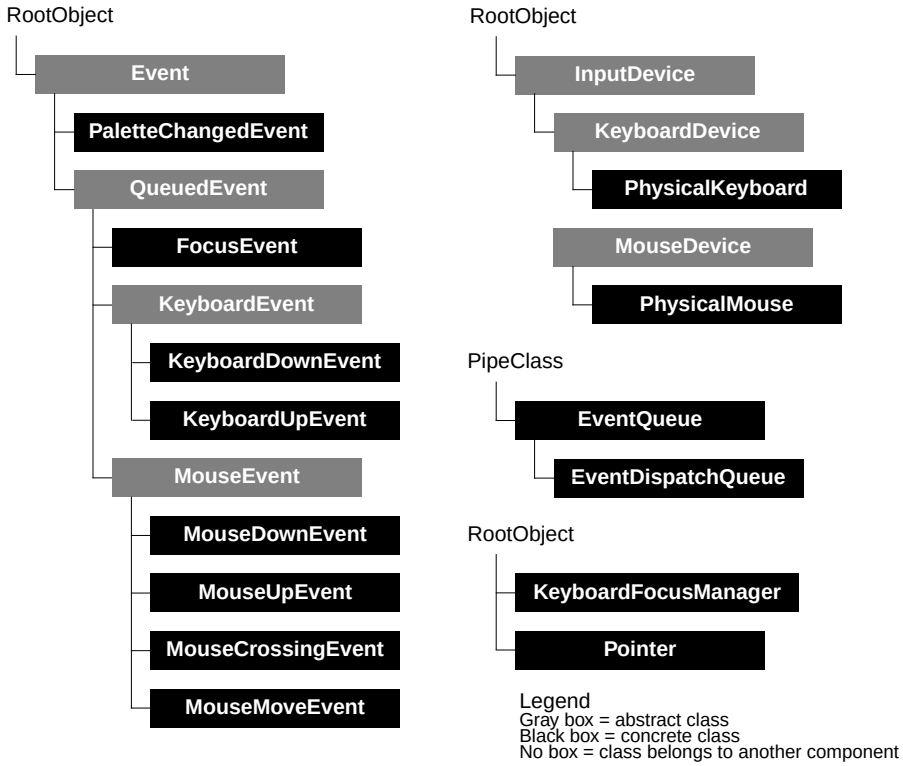
Think of the event mechanism as a tool kit that allows a programmer to set up communication between independent processes. Through the event system, threads can send out messages without knowing which thread will receive them. Furthermore, an event receiver can respond to an event without knowing anything about the process that generated it. Events are an internal messaging system that can be used to create complex simulations and behaviors.

Events are the foundation for many other components of ScriptX. The core classes provide several built-in mechanisms for receiving and processing events. For example, the `TextEdit` class automatically receives and processes keyboard events. The User Interface component defines a variety of presenters and controllers, such as `ScrollBar` and `ActuatorController`, that receive and process mouse events, giving the ScriptX Player common user interface elements such as buttons, menus, and scroll bars. Using these classes, an author can create a title without descending into the underlying event system. The event system is a low-level component of ScriptX, accessible to developers who want to create their own user interface elements.

Time-based messaging is not part of the ScriptX Events component. State changes that are based on reaching a given point in time are handled by the `CallBack` classes, defined by the Clocks component.

Classes and Inheritance

The class inheritance hierarchy for the Events component is shown in the following figure.



The following classes form the Events component. In this list, indentation indicates inheritance.

`Event` – root abstract class for all events

`PaletteChangedEvent` – an event that the system broadcasts to indicate a change in the color map on the underlying hardware.

`QueuedEvent` – describes both system and user-defined events that pass through a primary dispatch queue to be delivered sequentially and chronologically.

`FocusEvent` – an event that the system signals to indicate a change in input focus.

`KeyboardEvent` – an abstract class that represents all keyboard events.

`KeyboardDownEvent` – represents the pressing of a key on a keyboard by a user. The key code is passed as an instance variable.

`KeyboardUpEvent` – represents the release of a key on a keyboard by a user. The key code is passed as an instance variable.

`MouseEvent` – an abstract class that represents all mouse events.

`MouseDownEvent` – represents the pressing of a mouse button.

`MouseUpEvent` – represents the release of a mouse button.

`MouseCrossingEvent` – indicates that the mouse has crossed the boundary of a presenter.

- `MouseMoveEvent` – represents a mouse movement by the user.

`EventQueue` – a pipe that receives and stores events.

- `EventDispatchQueue` – a pipe that receives queued events in a primary event queue so that they can be dispatched sequentially to their final event receivers. It performs its own dispatching in its own thread.

`InputDevice` – abstract class that represents instances of user input devices such as a keyboard or mouse.

- `KeyboardDevice` – abstract class that defines the properties of all keyboard devices.
  - `PhysicalKeyboard` – represents a keyboard by communicating with the keyboard driver, which in turn receives key events from the actual keyboard.
- `MouseDevice` – abstract class that defines the properties of mouse devices.
  - `PhysicalMouse` – represents a mouse by communicating with the mouse driver, which in turn receives mouse-move and mouse button events from the actual hardware mouse.

`KeyboardFocusManager` – manages focus on a keyboard device.

`Pointer` – defines the appearance of the mouse pointer.

Conceptual Overview

ScriptX has an event system because it is a system-level as well as an application-level multimedia development environment. The platforms that ScriptX supports have their own event management systems. For example, in the Windows environment, events are called messages. The `InputDevice` classes translate events, sent by hardware devices in the protocols of the underlying system, into ScriptX events, and sends them to interested parties using methods that ScriptX defines.

Unlike some operating systems, ScriptX does not have a centralized event manager that dispatches events. The event system is really the collective behavior of the event classes, together with related classes like event queues and input devices.

Think of an event as an electronic form letter that is sent from one process to another. You could also think of an event as being a snapshot of some state change in the system. ScriptX supports a hierarchy of events, all of them inheriting from the abstract class `Event`. Each subclass of `Event` creates a specialized version of this form letter or snapshot. The Events component provides a dispatch and delivery mechanism that routes events to interested parties. ScriptX also allows developers to define new classes of events.

Every event has both a sender and a receiver. The sender can be a process that is managed by the system or by an external device, such as a mouse, but it can also be a process that is defined within a program. The receiver is any object or script that needs to be notified of the change in state that the event represents.

Instances of each event class serve two purposes in ScriptX. One is to represent an actual event. The other is to act as an event interest, a template for an event receiver. The following description shows how the class `MouseDownEvent` defines both actual mouse events and interests in mouse events.

1. **Event.** Suppose the user presses a mouse button. The mouse device detects this action and sends a mouse-down event. A mouse event is like a snapshot of the mouse at a particular moment in time. Its instance variables indicate the state of the mouse at the moment when the event was generated. For example, the instance variable `buttons` on all classes of `MouseEvent` stores a list of buttons that were down when the event occurred. The sender of a mouse event is usually an instance of `MouseDevice`, a subclass of `InputDevice`.

2. **Event interest.** Interests, though also instances of an event class, do not represent an actual event. Instead, they serve as a proxy or template for an event receiver. Event interests are matched with events in the delivery process. Suppose that a process is interested in receiving mouse-down events, but only if certain buttons are down. The instance variable `buttons` is used to match mouse buttons on this template with mouse buttons on an actual mouse-down event. In this way, the interested process is notified only of those mouse-down events it is interested in receiving.

An event records a change in state, but is not itself a state change. An event that is sent is not necessarily received. Suppose that a tree falls in the forest. An observer, standing in the forest with a camera and tape recorder, receives the event. Does the tree still fall if the observer was paying too much attention to bears to take notice? Even if nobody sees the tree fall, we expect that another observer, passing through the forest the following day, would see a fallen tree.

What does this say about the ScriptX event system? There can be a change in state without the corresponding event being received, or even sent. If the user presses the mouse button at a time when no process is interested in mouse events, the mouse is still down. An event is an observation or notification of a change in state, a picture of a small part of the system (such as the state of an attached mouse) at the instant when some change occurred. The event system is the mechanism that sends a record of this change to any processes that want to be informed of it.

How Events Work

Events go through a three-stage delivery mechanism in ScriptX. Events are matched with event interests, and a reference to the event itself passes from sender to receiver. The following list summarizes this mechanism, which is covered in greater detail later in the chapter.

1. **Dispatch.** A process that generates an event sends it out by calling one of three methods: `signal`, `broadcast`, or `sendToQueue`. These methods, which are sometimes specialized by particular subclasses of `Event`, provide the rules for event delivery. At this stage in the delivery process, the sender is concerned only with what interests are registered on a particular class of events. If an event is broadcast, it is sent to all interested and satisfied

receivers. If it is signaled, it is delivered to only one receiver, the satisfied receiver that has the highest priority. If it is sent to a particular queue, the interest list is ignored and the event is placed directly in that queue.

2. **Matching.** The second stage applies to events that are delivered with the `signal` method. The event system *matches* the event with potential interests. The `signal` method calls the `isSatisfiedBy` method on the event interest that has the highest priority. This method, which is not called directly from the scripter, is a crucial part of event delivery. `isSatisfiedBy` determines whether the event can satisfy the interest, often by comparing particular instance variables on the template and the event itself. If `isSatisfiedBy` returns `false`, then the `signal` method calls `isSatisfiedBy` on the interest with the next highest priority, until a matching interest is found, or until all interested parties have been checked.

3. **Acceptance.** If an event is signaled asynchronously, the delivery mechanism has a third stage—acceptance or rejection of the event. After an event has been delivered to a matching receiver, the receiver (either a function or queue) is activated immediately. It returns `@accept` or `@reject` to indicate its acceptance of the event. If it rejects the event, the dispatching method regains control and attempts to deliver the event to other interested parties.

As you read the remainder of this chapter, keep in mind that there are always two points of view on the event system—that of the sender and that of the receiver. A script interacts with the event system both to send events and to receive them.

In the ScriptX event system, an event cannot be delivered unless a receiver has registered an interest in receiving it, and a receiver does not receive events unless it has registered an interest in them. But the underlying action that caused the event to be sent out is separate from and distinct from the event system. For example, a `PhysicalMouseDevice` object keeps track of the current state of a mouse that is connected to the system. If the first button on this mouse is pressed and no interest has been registered with the class `MouseDownEvent`, then no event is delivered. But if a program calls the method `isButtonDown` on the mouse device before the user releases the button, the device still reports that the mouse button is down.

Generating Events

Events come from three potential sources in ScriptX, however there is nothing distinct about the three.

1. **User.** Some events originate directly with the user, interacting with the program through an input device. ScriptX defines a set of input device classes, covered later in this chapter. Input devices act as interfaces with hardware devices. These devices send an event to indicate a change in the state of an input device that is controlled by the user, such as a keyboard.

2. **System.** Events also originate with the native operating system. For example, the ScriptX Player uses the `PaletteChangedEvent` class to broadcast changes in the color map associated with a monitor that is attached to system.

3. **Program.** A developer can create new scripted event classes and use them internally in a title or tool. Events can be used to set up communication between threads.

Events, whether generated by the user, the system, or the program, are created and dispatched using the same method calls. Any class of event can be simulated by software. Thus, a script can create and send a mouse event, although an input device would normally do so.

Most events that originate with the user, including all classes of events that are signaled by input devices, belong to a specialized branch of the `Event` class hierarchy. The class `QueuedEvent` is the abstract superclass of `MouseEvent`, `KeyboardEvent`, and other event classes that originate with input devices. Queued events pass through a primary queue before they are sent to a receiving queue or function.

This primary queue insures that queued events are processed in a sequential and orderly manner. For example, a `MouseUpEvent` can never be processed before the `MouseDownEvent` that it is paired with. The primary event queue, an instance of the specialized class `EventDispatchQueue`, is actually of little consequence for title and tool developers. Although a developer can create a scripted subclass of `QueuedEvent`, classes that are created by script can generally be regular events. Regular events, since they are processed in fewer steps, offer a performance advantage over queued events. Queued events are covered in greater detail later in this chapter.

Sources of User Input

ScriptX creates interfaces to hardware input devices through the input device classes. These classes, subclasses of `InputDevice`, store information about a hardware device that is connected to the computer. The input device classes receive events from the native operating system and translate them into ScriptX events.

As shown in Figure 18-1, a *hardware device* is an actual input device connected to the computer that a user can touch and feel. A *device driver* is a component of the native operating system that receives signals from a hardware device and passes them on to the ScriptX Player. For example, the device driver for a mouse passes mouse coordinates and mouse button events from the hardware mouse to the ScriptX Player.

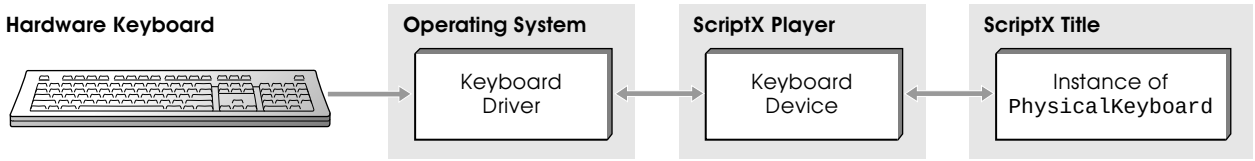


Figure 18-1: Receiving input from a hardware device.

Suppose the user presses a key. The `PhysicalKeyboard` object that represents the device receives a keydown event from the native operating system. It updates its instance variables, generates a ScriptX `KeyboardDownEvent`

object, and signals this event to any event receivers that have posted an interest in such events. `PhysicalKeyboard` is a subclass of `KeyboardDevice`, an abstract class that defines most of the properties and behaviors of keyboard devices. For example, the instance variable `keyModifiers` on the class `KeyboardDevice` stores the current state of modifier keys such as the Shift and Control keys.

Receiving Events

Any function, method, or queue can receive and process any class of event. A script indicates which classes of events a particular receiver is interested in by creating an event instance to act as an interest and calling `addEventInterest` on that instance to register it with the event subclass. Each `Event` subclass maintains a collection of interests. The following code fragment creates a new event interest of the scripted class `SchreklichEvent`, assigns the function `getNewEvent` as its receiver, and posts it as an interest with the `SchreklichEvent` class.

```
class SchreklichEvent (Event) -- class definition, creates a new event class
end
global meinEvent := new SchreklichEvent -- create new event interest
-- set the function getNewEvent as its receiver
-- the receiver must already be defined and it must be an
-- instance of EventQueue or AbstractFunction
meinEvent.eventReceiver := getNewEvent
-- add it to the collection of interests, which is maintained by the class
addEventInterest meinEvent
```

Each event class defines an `interests` class variable, a collection in which it stores a list of interests that are active. The class uses its collection of interests as templates to match events with event receivers. The class itself is responsible for both the interest matching mechanism and the event delivery system.

ScriptX provides for two distinct types of event receivers. The first is a receiver function. (A method can be an event receiver function.) A function receives events directly, since it is called within the thread in which the event was delivered. Events that originate with input devices, including mouse and keyboard events, are generally received by functions for performance reasons.

The other type of receiver is an event queue, a specialized form of pipe. An event queue is useful when the sender and receiver are in different threads. ScriptX is an open environment, one which allows a programmer to add new objects to existing titles. Event queue receivers offer another option for interprocess communication in such titles.

Figure 18-2 illustrates the flow of communication and control as an event passes from sender to receiver. Event classes store and match event interests. Each class defines its own mechanism for delivering events to receivers, either functions or event queues.

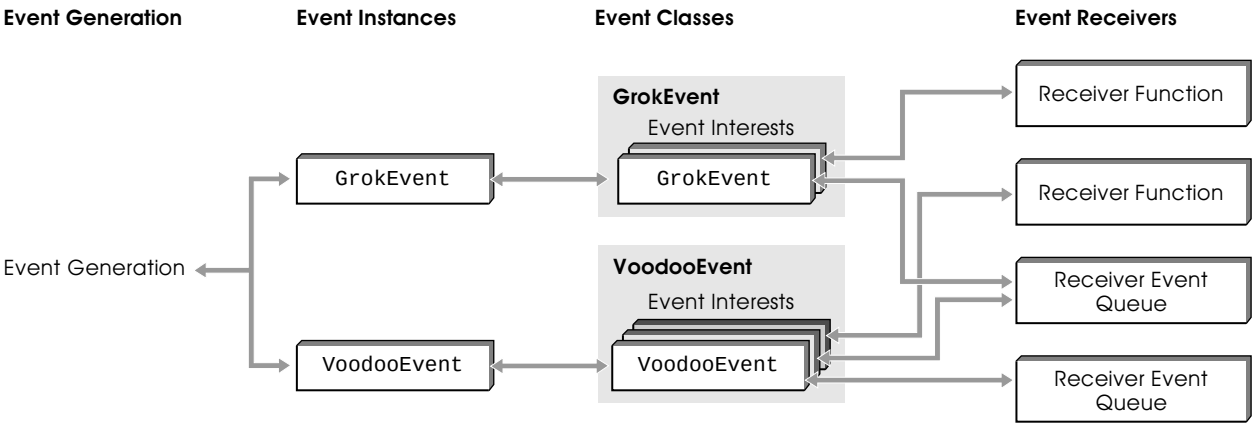


Figure 18-2: Event flow from senders to receivers.

Event Receiver Functions

Generally, a function is the fastest and the simplest event receiver. To receive events through a function, a program must create an event instance that acts as an interest. It sets the value of the `eventReceiver` instance variable on this interest, and any other variables that may apply, and then calls `addEventInterest`, adding it to the interests list that is maintained for that particular event class.

A function that acts as a receiver takes three arguments. The first argument takes the contents of the instance variable `authorData` on the interest that matched the event. This variable can store any object. The second and third arguments receive the matched event interest and the actual event, respectively.

When a function receives events, it is called directly, in the same thread from which it was sent. A function that receives events should run briefly without blocking and return a value. It returns `@accept` or `@reject` to indicate whether or not the event was accepted. If a receiver function initiates a process that requires a long period of execution time, a good strategy is to activate another thread that runs the process. The following code fragment indicates the general format for an event receiver function. Note that `addEventInterest` reports an exception if no receiver has been specified.

```
-- getNewEvent is a typical event receiver function
-- note that getNewEvent calls a function in another thread so that it can
-- execute and return quickly. This is good behavior for an event receiver.
function getNewEvent authorData interest event -> (
       case event of
           SchreklichEvent: (
               -- the next line spawns a new thread using the ”&“ operator
               -- not necessary, but good behavior if it takes a while to run.
               fangenWirBitteAn(authorData) &
               @accept
```

```
            )
         -- insert more event types here
         otherwise:
            @reject -- rejects events it is not familiar with
      end -- case
)
class SchreklichEvent (Event) -- class definition, creates a new event class
end
-- now create an instance of SchreklichEvent to act as an interest
meinEvent := new SchreklichEvent
meinEvent.eventReceiver := getNewEvent -- set the receiver
-- the authorData instance variable is used as an argument
meinEvent.authorData := "Anne has been bitten by Fangenveer!"
addEventInterest meinEvent -- add it to collection of interests

function fangenWirBitteAn data -> (
      print data
)
```

In the previous example, the second argument to the receiver function, which stores an event interest, is ignored, but it could be very useful in a more complex program. The same receiver function can be called on to process different classes of events. Even if it processes events of only one class, a given receiver may be associated with many different interests in that class.

Keep in mind that an event receiver function does not poll; once it is set up to receive events of a given class, the event class itself delivers those events. Under no circumstance should a receiving function block—if the function blocks, it blocks the delivering thread. For more information, see “Blocking” on page 591 of the “Threads” chapter.

In most cases, developers will want to use functions rather than event queues to receive events. Functions offer a considerable performance advantage. Since receiver functions run in the same thread from which the event was sent, there is no inherent difference between synchronous and asynchronous event delivery to a function receiver. For synchronous delivery of a regular event, functions are several times faster than queues. For synchronous delivery of a queued event, a function receiver is many orders of magnitude faster.

Event Queues

An event queue is an alternative mechanism for receiving events. Developers who plan to use an event queue in their scripts should begin by reviewing the section on “Pipes” on page 593 in the “Threads” chapter.

Although function receivers have a performance advantage over event queues, queues can be used to set up interprocess communication between separate threads within a title. ScriptX is an open environment, in which a developer can create a title that accepts new objects. Objects that respond to a common set of events can communicate through the event system using event queues without knowing anything else about each other.

An event queue, like an event receiver function, becomes an event receiver when it is assigned to the `eventReceiver` instance variable. The following script creates a new class of events, called `EarthQuakeEvent`. It creates an event queue called `myQueue` and an instance of `EarthQuakeEvent` called `myQuake` to act as an event interest. It sets the event queue as the event receiver for `myQuake`. Finally, it registers `myQuake` as an event interest.

```
class EarthQuakeEvent (Event) -- class definition
    instance variables magnitude
end
myQueue := new eventQueue -- create an new queue
myQuake := new EarthQuakeEvent -- create an instance of EarthQuakeEvent
myQuake.eventReceiver := myQueue -- set the queue as a receiver
addEventInterest myQuake -- add it to the interest list
```

Although we speak informally of “delivering” an event to an event queue, the event is actually processed by an underlying thread that reads from the queue. The class `EventQueue` defines several methods, but they are not normally called at the scripter level. A queue’s “behavior” is really the logic embodied in its underlying thread function. This function maintains event interests, accepts and rejects events, sets priorities, and responds to the event itself. For an example of a thread function that reads from an event queue, see the `EventDispatcher` sample script, which begins on page 534 of this chapter.

`EventQueue` inherits most of its scripter-visible methods and instance variables from `PipeClass`, in the Threads component. A pipe is a mechanism for communication between threads. When you deliver an event to a queue, you write to a pipe. The thread that wakes up and processes the event reads from the pipe. The class `EventQueue` inherits many powerful features from `PipeClass`. For example, a thread can use the `acquireQueue` method defined by `PipeClass` to gain low-level access to the queue, perhaps to look ahead at objects in the queue. To understand event queues, see the sections “Pipes” on page 593 and “Blocking” on page 591 of the “Threads” chapter.

Event queues are more restrictive than pipes. A `PipeClass` object is a conduit for any kind of object. Any thread can write any object to the pipe, providing there is room in the pipe. An event queue acts as a conduit only for `Event` objects, and the function that reads from it has some actual control, through the interest mechanism, over which events are placed on the queue. (A script can override the event delivery system using the `sendToQueue` method.)

Unlike the `PipeClass` class from which it inherits, an event queue is not a passive conduit. It has all the capabilities of an event receiver function. As a receiver, it is associated with its own event interests. A particular queue can be a receiver for many different event classes. If an event queue does not have event interests, it will not receive events. Event interests can be added or removed at any time, and their priorities can be varied within the framework of the given event class. A thread that reads from an event queue can accept and reject events, if they are delivered synchronously. A function accepts and rejects events by returning the `NameClass` objects `@accept` or `@reject`. An event queue does the equivalent by calling `accept` or `reject` on the event.

An event queue acts as a storage bin that accepts an event, keeping it in order until it can be processed by the underlying thread's function. In most cases, the event is processed immediately when the thread runs, but there is nothing to prevent a developer from using the queue to store events for future use.

Dispatching Events

Event dispatch is the first of three phases in the event delivery mechanism, which is summarized at the beginning of the section "How Events Work" on page 500. In the dispatch phase, the delivery method searches through the collection of event interests, which is managed by the event class. It does not actually examine or compare the event with event interests—this operation comes in the next phase, event matching.

Note – Subclasses of `InputDevice`, such as `MouseDevice` and `KeyboardDevice`, create their own classes of events automatically. Instances of `FocusEvent` and `PaletteChangedEvent` are also generated automatically. You only need to be concerned with dispatch mechanisms on event classes that you create for your own programs.

Dispatch should be considered from the point of view of the event receiver as well as the sender. A receiver can filter out unwanted events at each of the three delivery stages. It can reject or ignore an event that has been delivered, but significant processing time may be involved. Since adding and removing event interests can be done with so little overhead, it is often most efficient for a receiver to actively managing its event interests.

Consider the performance trade-offs in a particular program. In one situation, there may be very few events, and a large number of receivers that are constantly changing their interests. In such a program, it might be more efficient to have these receivers post their interests continuously, using the matching and acceptance phases of the event delivery mechanism to filter out unwanted events. At the other extreme, imagine a program that generates a large number of events while it runs, with event interests that are quite static. In such a program, it would be more efficient to add and remove event interests.

Any object, function, or script can generate and send events. Once created, an `Event` object is the only copy of itself. We say that an event has been delivered and received, but it actually sits in the same fixed memory location from the moment of creation until the moment when the garbage collector reclaims its memory. The function or queue that "receives" an event receives only a reference to it.

To *send* an event is to tell the event to deliver itself. The event places itself in the proper queues, or it calls a matching receiver with itself as one of three arguments. The actual method used to deliver the event will vary, depending on the nature of the event. Three delivery methods are available:

1. Broadcast—the event is placed on event queues for all threads that have registered an event interest in the event that has occurred, using the method `broadcast`.

2. Message-passing—the event is targeted only to a specific event queue, using the method `sendToQueue`. This method applies only to event queues. To target a receiver function, call the function directly. If an event is sent to a specific queue, it bypasses the matching and acceptance phases of the event delivery system.

3. Token-passing—the event is delivered to only one event receiver by using the method `signal`, but that queue or function is not specified. When `signal` is called, the event is delivered either synchronously or asynchronously.

The `signal` method requires an additional argument, a `Boolean` value that represents either synchronous delivery (`true`) or asynchronous delivery (`false`). The receiver does not need to reply if the event is delivered asynchronously. But if an event is delivered synchronously, the receiver needs to accept or reject the event. If an event is not accepted, it goes to the event interest with the next highest priority.

Synchronous delivery to a function event receiver is virtually as fast as asynchronous delivery. Synchronous delivery to an event queue bears a significant performance penalty because the thread that generated the event blocks while waiting for a reply. This penalty is especially large with queued events.

Advanced users can specialize the event delivery mechanism at either the instance or class levels. The instance methods `signal` and `broadcast` call the class methods `signalDispatch` and `broadcastDispatch`. A title developer does not normally call these methods, which define the delivery mechanism at the class level. These methods can modify the event, to some degree, as it is delivered. For example, when an event instance is delivered to a receiver, its `matchedInterest` instance variable, formerly undefined, may be modified to point to the matching event interest.

Since the event delivery mechanism modifies properties of the event itself, an event receiver should never assume that a copy of an event remains the same as the original. The ScriptX Player itself does not make copies of an event—the original event instance is passed by reference from sender to receiver. Suppose that a high priority receiver makes its own local copy of an event and then rejects it, so that the event instance is delivered to another receiver. Once an event has been delivered, a local copy may no longer be valid. For example, when a `MouseEvent` object is delivered, its `localCoords` instance variable is updated to reflect the coordinate system of the presenter which receives it.

Matching Event Interests

Event matching is the second of three phases in the event delivery mechanism, which is summarized at the beginning of the section “How Events Work” on page 500. Events that are sent directly to a queue bypass this phase in delivery. Events that are delivered by the `signal` or `broadcast` methods, however, can be examined in more detail.

In the matching process, an event class can examine the properties of an event, comparing them with the corresponding properties of an event interest. For example, each `KeyboardEvent` subclass can check that an event occurred on a

particular key, or with a particular modifier. Event classes define their own mechanisms for storing interests, matching interests to events, and delivering events to interested receivers. In effect, an event interest matches incoming events with itself, using the rules that are defined by its class.

Each `Event` subclass, by storing and matching its own interests, acts as an arbitration and dispatch mechanism, connecting events that have the desired properties with receivers. This does not mean that the event subclass is the sole determinant of who receives an event. Event receivers set their own `priority`. It is also possible for a process that generates an event to look ahead and examine the collection of interests stored on that event class before it sends an event.

Event interests are held in a data structure that is associated with each `Event` subclass. Implementation of this data structure is not specified in ScriptX; it can be optimized for a particular class. A default version, defined in the `Event` class, is available for event types which do not have critical requirements. Figure 18-3 illustrates the default data structure, stored in the class variable `interests` on objects of the `Event` class and its subclasses.

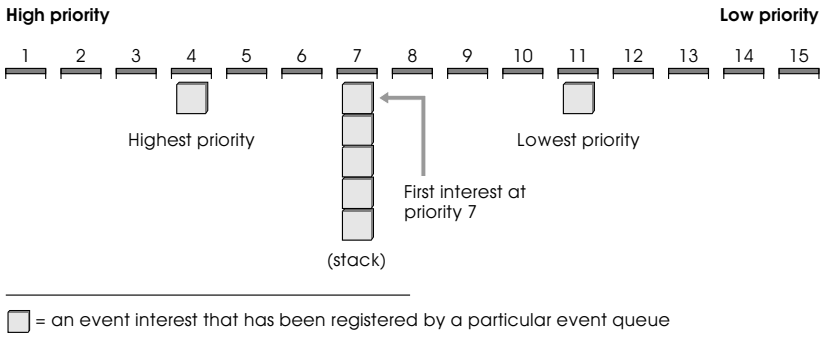


Figure 18-3: Default data structure used to store event interests.

In the default implementation, event interests are stored in buckets. The event class uses the `priority` instance variable to place an event interest in the proper bucket (the default is 7). Each bucket is a collection, with the most recent event interest at the top of the stack.

Interest collections change over time. Event receivers are usually interested only in particular events, and their event interests may change dynamically. For example, an actuator controller might only be interested in instances of `MouseDownEvent`. Once the mouse is down, it becomes interested in `MouseMoveEvent`. When the button is released, it becomes interested once again in `MouseDownEvent`, but is no longer interested in `MouseMoveEvent`.

The `signal` and `broadcast` methods, covered in the previous section, call the instance method `isSatisfiedBy` to search for an interest that matches a given event. By overriding `isSatisfiedBy`, a developer can modify the interest matching process. By overriding other methods on the `Event` class, a developer can modify the entire interest storage and matching system for a subclass of `Event`.

When is a more specialized data structure required? Some interest collections are designed to avoid a linear search through all possible interests. For example, the class `MouseEvent` has specialized techniques for interest management. `MouseEvent` subclasses search through interests based on the presentation hierarchy, in depth-wise, front-to-back order. Thus, a mouse-click is received first by the presenter that sits directly under the mouse pointer. In addition, the class `MouseUpEvent` specializes the instance variable `matchedInterest`, which is defined by `Event`, to insure that an instance of `MouseUpEvent` is delivered to the same presenter that received the most recent instance of `MouseDownEvent`. For more information on how presenters store mouse events, see the section "Storing Interests in Mouse Events on Presenters" on page 521. For more information on the `matchedInterest` instance variable, as specialized by `MouseEvent`, see "Matched Interests" on page 525.

A program might need to know about all mouse events. ScriptX provides a mechanism for trapping every mouse event before it reaches this top-most presenter. If a `MouseEvent` interest does not set its `presenter` instance variable, it has priority over other interests that do. This allows objects such as tools to intercept mouse events. Presumably, such a high-priority receiver will register and react to the event, and then reject it, so that the top-most presenter receives it.

Accepting an Event

Acceptance is the final phase in the event delivery mechanism, which is summarized at the beginning of the section "How Events Work" on page 500. It applies only to events that are signaled synchronously.

When an event is signaled, the event's class searches through its collection of interests to find the matching interest that has the highest priority. If it finds a match, then the event is delivered to the function or event queue associated with that event interest. If the event was delivered asynchronously, delivery is complete, and no reply is needed. If the event was delivered synchronously, the `signal` method waits for a reply.

Note – If an event is delivered synchronously and `isSatisfiedBy` returns `true`, event delivery is not necessarily final. Calling `signal` on the event activates the thread that reads from the receiving queue, or it executes the receiving function. The receiver then accepts or rejects the event.

With synchronous delivery, the function or queue that registered the matching interest has an opportunity to examine the event, and perhaps reject it, passing it on to another interest with lower priority. A function receiver is activated immediately, in the same thread. It accepts or rejects the event by returning `@accept` or `@reject`.

The acceptance mechanism for synchronous events is more complex when the receiver is a queue. When an interest that is registered by an event queue states that it is satisfied by an event, `signal` asks for the reject queue of the event. Reject queues are created automatically by the event class. This creates an event queue with a single interest, a reply interest for the event. The `signal`

method places the event on the satisfied interest's event queue and reads from the reply queue. The sending thread now blocks waiting for a reply. The receiver's thread is awakened because there is an event in its queue. If the event has been signaled as rejectable, the receiver may choose whether or not to accept the event.

An event queue receiver accepts this event by calling `accept`, with the event as the first argument. This method call signifies that this interest will take the event and that the event should not be placed in any other queue. If the receiver calls `reject` on the event, then the event class searches for the next best match for the event (in its interest list) and delivers the event. This continues until some event queue calls `accept` or until all interests have been processed. If no interest is found, or if no interest accepts the event, the event is discarded and the space it occupies in memory is reclaimed by the garbage collector.

Flow Diagrams for Events

Figure 18-4 illustrates event handling for regular events. A regular event is one that inherits from `Event`, but not from `QueuedEvent`. Regular events are delivered directly to the final receiver, a queue or function, after being matched with event interests.

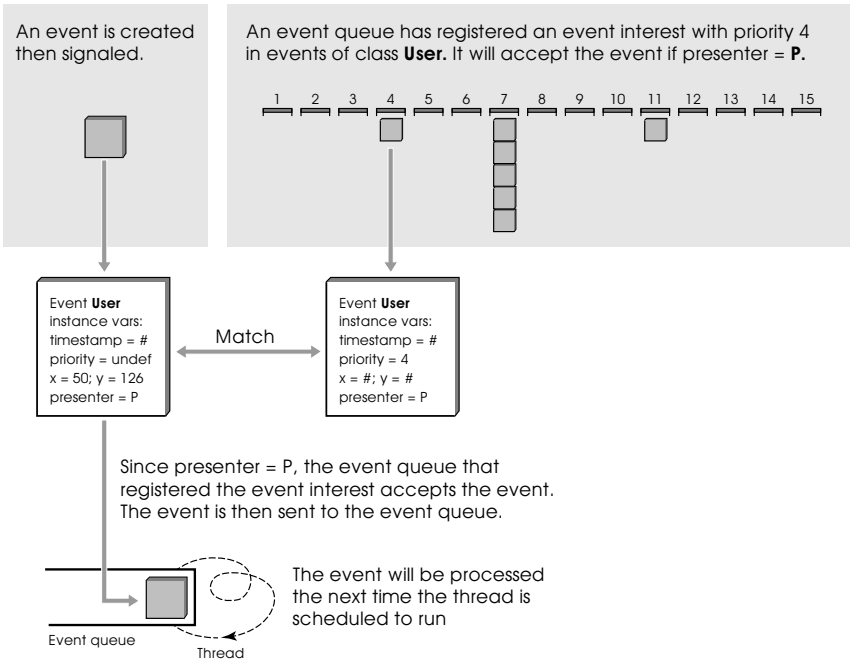


Figure 18-4: Flow diagram of event handling with an event queue.

A queued event differs from an event in that all queued events are funneled through the event dispatch queue to assure orderly handling.

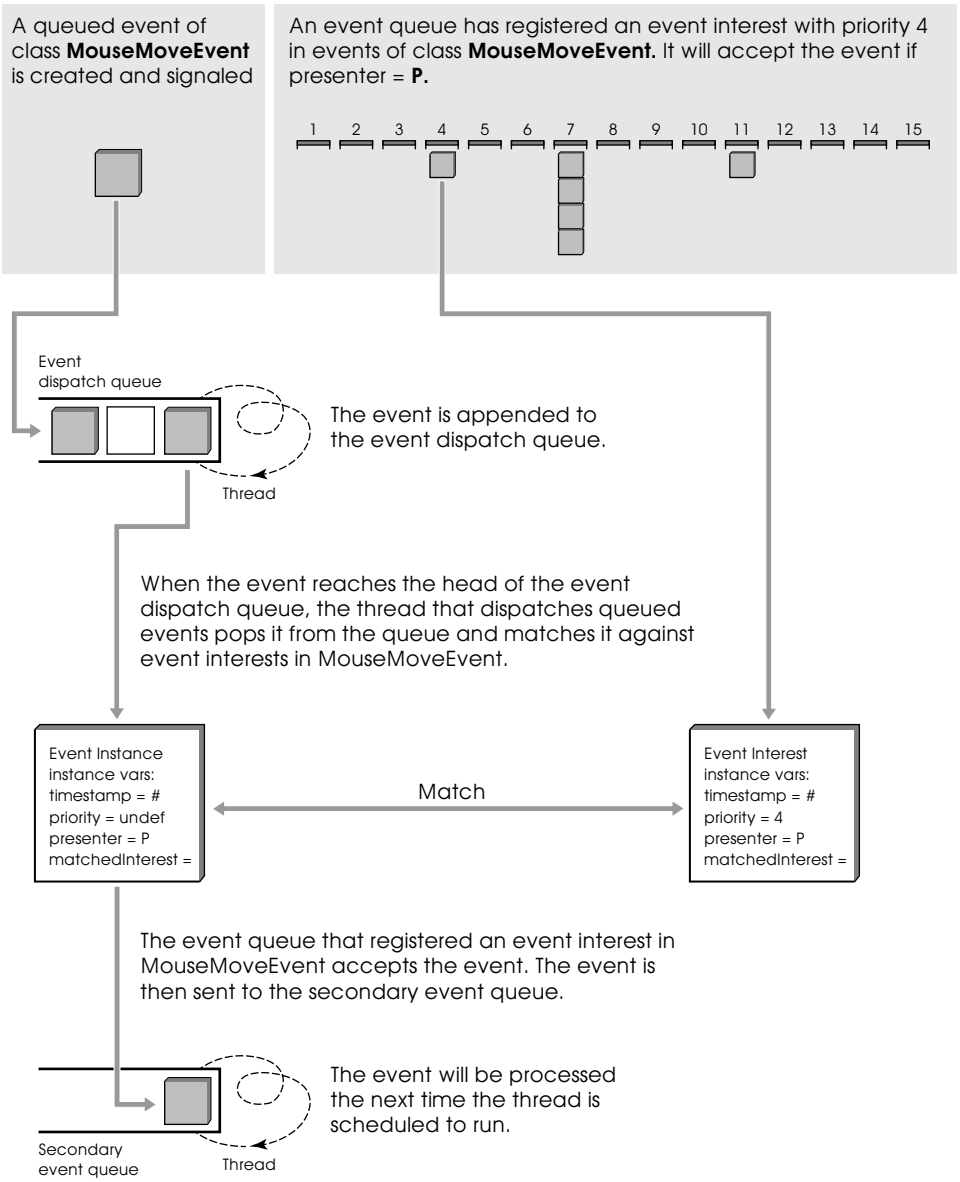


Figure 18-5: Flow diagram of queued event handling with an event queue

Events and Event Interests—Creating New Classes

This section is for developers who want to create their own classes of events. As a developer, you create objects and scripts that approach the ScriptX event system as both a sender and a receiver. If your script creates events, you do not want to overload the system, and any potential receivers, with unwanted messages. If you create an event receiver, you don't want to intercept messages that should be left for other receivers.

All of the visible mechanisms defined by the event system, such as the `signal`, `broadcast`, and `isSatisfiedBy` methods, are really default mechanisms. A scripted subclass of `Event`, or of any particular event subclass, is free to override these methods to provide additional functionality.

Since a given event class creates both events and event interests, a script that creates a new subclass of `Event` automatically creates a mechanism for sending and receiving that class of event.

Event classes serve a dual role in ScriptX. Many of the instance variables defined by `Event` and its subclasses are overloaded—they have one meaning if the object represents an event, another if it represents an event interest. For example, you can designate a `presenter` when any mouse event is used as an interest. But when the event represents an actual event instance, `presenter` is maintained automatically, by the `MouseEvent` class. Many properties of events apply only when the object is registered as an interest. ScriptX reports an exception if an event is used as both an interest and an event.

Suppose that a script creates a new instance of some subclass of `Event`. What makes an `Event` object represent an event, and what makes it represent an interest? Any instance of `Event` can potentially become either an event or an event interest. A developer must approach the event system from both directions.

If the `Event` instance is to be used as an actual event, it must be sent out to interested receivers. The methods `signal`, `broadcast`, and `sendToQueue` and provide alternative delivery systems. Sending an event to interested receivers is what makes an event an actual event.

If the `Event` instance is to be used as an interest, it must be registered as an interest with its class. When an event acts as an interest, it acts as a template, belonging to an event receiver. This template is registered with the event class, allowing the receiver to be matched with and receive actual events.

Many instance variables on the event classes are read-write. Setting a new value makes sense only if the event is used as an interest, not if it serves as an event. When the event represents an actual event, these variables are maintained by the class itself. They should not be modified.

If an event is to become an interest, its `eventReceiver` and `authorData` instance variables are set. An event interest must have a receiver—otherwise, it cannot be delivered. `authorData` is an optional parameter. It is used as an argument when an actual event instance is delivered to the event receiver. Finally, the event itself is registered as an interest by calling the method `addEventInterest` on the event instance. This posts the interest on the class's interest list.

Events are generally matched and processed quickly, existing as objects only momentarily. Event interests, on the other hand, can be stored and saved almost indefinitely while a program is running. Instances of `Event` can be saved to and retrieved from the object store, allowing a program to manage a set of predefined interests. Event interests can be added to or removed from interest lists as a program runs with very little overhead.

In the `Event` class, the `interests` class variable represents a collection that is publicly visible. A script can examine this collection before posting new interests. It is also easy to check whether there are interests posted, and at what priorities, before it sending an event.

How Input Devices Work

Input devices are objects that serve as interfaces between hardware and other ScriptX classes. Input devices are event generators. They detect events that are initiated by the user, such as mouse and keyboard events. They translate these events, which originate with the underlying operating system, into `Event` objects—objects that the ScriptX Player can receive and process.

This section is for developers who need direct access to input devices and the classes of events that they generate, perhaps to create new user interface elements. Most title developers will find that existing classes meet their needs.

ScriptX provides a range of presenters and controllers that receive keyboard and mouse events automatically. The classes `ScrollBar`, `ActuatorController`, `DragController`, and others that are defined by the User Interface component, receive and process mouse events automatically, creating a range of objects that implement common user interface elements. The `TextEdit` class, defined by the Text and Fonts component, acts as a presenter that can receive and process keyboard events. Each of these classes creates an `InputDevice` object, if one is not already present, to act as an interface with the native operating system.

Before you create a new user interface element, consider the objects that are built into the core classes. For example, a `PushButton` object is a really a group of simple presenters (a `TwoDMultiPresenter` object) with an attached controller (an `ActuatorController` object). This architecture makes it possible to separate appearance from implementation. Modify its presenters, and you change the appearance. Modify the controller, and you change the behavior.

Creating Input Devices and Processing Events

To receive events directly from a keyboard, mouse, or other input device, you must first create an instance of the corresponding input device class. ScriptX does not automatically create instances of input devices. The `InputDevice` object creates an instance of the appropriate event for each event that it receives from the operating system. It signals those events to the appropriate receivers.

When you create a new input device, the ScriptX Player resolves the new device instance with existing instances of that class. For example, if a `TextEdit` object is already set up to receive and process keyboard events, creating a new `KeyboardDevice` object will be resolved to create a reference to the existing keyboard device.

ScriptX is capable of supporting multiple instances of input devices; however, this feature has little practical application for current titles. As ScriptX encompasses joysticks and other input devices, title developers will be able to use the instance variables `enabled` and `deviceID`, both defined on `InputDevice`, to create interactive titles for two or more users. Each `InputDevice` subclass maintains a list of instances; a script has access to this list through the class method `getDeviceFromList`.

To receive events from an input device, a title must first create an object that will act as an interface to the device driver. This object is an instance of the class that represents the device. In a ScriptX title, an input device object is always an instance of one of the subclasses of `InputDevice`. For example, a hardware mouse is represented by an instance of `PhysicalMouse`.

Next, the title must create an event receiver and post interests for that receiver in the classes of events that are associated with the device. As event receivers, functions enjoy a major performance advantage over event queues in processing queued events, especially if those events are delivered asynchronously.

A process can also poll an input device directly. For example, a process can ask the mouse device for its current coordinates. Polling of input devices can only be done from a thread that can block. For example, it would not be appropriate to call a function that polls for keyboard activity from a callback thread.

Receiving mouse events requires more memory than polling a mouse device, but it is a better way of handling mouse movements when a script requires that instances of `MouseMoveEvent` be processed asynchronously with other queued events. The event dispatch queue insures that events are processed in an orderly manner. If a script is recognizing a gesture, or if it is engaged in some operation that does not require tight integration with the compositor, then it is easier and more efficient to receive mouse events.

Keyboard Devices and Keyboard Events

`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.

An upcoming release of ScriptX will allow the ScriptX Player to substitute a virtual device for a physical input device that is not available. Although they are not implemented in the current release of ScriptX, developers need to be aware of virtual devices. For more information on virtual devices, see the section beginning on page 529 entitled “Input Devices of the Future.”

Although `KeyboardDevice` is an abstract class, you create a new instance of any `KeyboardDevice` subclass by calling new on the `KeyboardDevice` class itself. The class automatically creates an instance of the appropriate concrete subclass. Since the current release of ScriptX is directed primarily to desktop computers, a title should assume for now that a physical keyboard is connected to the system.

Key Codes

Each key has an associated key code. A complete table is given with the definition of the class `KeyboardDevice` in the *ScriptX Class Reference*. When a `KeyboardDevice` object 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.

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 section "String Encoding and Unicode" on page 284 in Chapter 12, "Text and Fonts."

An event interest in keyboard events can specify a range of key codes that it is interested in receiving by modifying the instance variables `maxKeyCode` and `minKeyCode`, defined by the class `KeyboardEvent`.

Modifier Keys

Modifier keys, which include shift keys and state keys, fall into a special category. State keys, such as CapLock, NumLock, and ScrollLock, are toggled on and off. Shift keys, such as the Shift, Option or Alt, Control, and Command keys, are considered active when they are being actively pressed.

Keyboard devices generate an event when a modifier key is pressed. They also continuously monitor the state of modifier keys. Whenever a keyboard event is generated, the state of the seven modifier keys is recorded in the `keyModifiers` instance variable for that event.

Note the distinction between the instance variable `keyModifiers`, defined by `KeyboardDevice`, and the instance variable `keyModifiers`, defined by `KeyboardEvent`. The former represents the state of the modifier keys at present. The latter represents a snapshot of the state of the modifier keys when an event was generated, when a key was pressed or released.

For each key name there is a corresponding name token, that is, a `NameClass` object. Use `getKeyName`, a method defined by `KeyboardDevice`, to get the name token for any given key code. The system name is always the key name with an "@" symbol before it. System names provide a convenient syntax for checking the values of the modifier keys.

```
myDevice := new KeyboardDevice
```

```
isModifierActive myDevice @shift -- returns true or false
myDevice.keyModifiers -- returns an array
```

On an event interest, `keyModifiers` designates a state that will satisfy that interest, so that the keyboard event can be delivered to the associated receiver. The `isSatisfiedBy` method, specialized by the `KeyboardEvent` class, matches `keyModifiers` on an event with `keyModifiers` on an event interest before delivering the event to a receiver.

Focus Events

Suppose that a user is typing at the keyboard, and that several different `TextEdit` presenters that are attached to a ScriptX window. Each of these presenters has an interest in receiving keyboard events. The presenter that is the current target of keyboard events is said to have *focus*. If our title was a database application, and if each `TextEdit` presenter was a field on a database layout, we might say that the presenter with focus is the current field.

Focus events are generated by a focus manager. A focus manager is an invisible class that is closely associated with an input device, such as a keyboard device. For the `KeyboardDevice` class, focus events are used to direct input to a particular event receiver, usually associated with an instance of `TextEdit`. A `FocusEvent` object is an event with several specialized instance variables that are visible to the scripter. The instance variable `presenter`, which must be an instance of `TwoDPresenter`, is set by event interests. The instance variable `focusType`, used only on focus events, takes on three possible values: `@loseFocus`, `@gainFocus`, and `@restoreFocus`.

The focus manager for the keyboard device maintains a private record of which event receiver currently has focus. When another receiver requests focus, it informs the current receiver that it is losing focus. It does this by sending this presenter a `FocusEvent` object with the value of `focusType` set to `@loseFocus`. Upon losing focus, a receiver modifies its presenters to show that it is inactive, and removes its interests in receiving keyboard events.

The keyboard device also informs the new event receiver that it is gaining focus. It sends this new receiver a `FocusEvent` object too, setting `focusType` to either `@gainFocus` or `@restoreFocus`. The distinction between these two values is minor. Some programs might want to make a distinction as to how focus was obtained. If a receiver gains focus through user manipulation, the value of `focusType` is `@gainFocus`. If the change in focus is initiated by a third party, such as another script or object, then the value of `focusType` is `@restoreFocus`. Upon gaining focus, a receiver might perform an initialization, modify its presenters to show that it is active, and add its interests in receiving keyboard events.

Figure 18-6 demonstrates the focus mechanism, while hiding some of the underlying detail. The programmer's objective is to have one and only one `TextEdit` object have an interest in events of the `KeyboardDownEvent` class at any given time. In this diagram, a ScriptX window contains several `TextEdit` objects, presenters that are interested in gaining focus and receiving keyboard events. Roger Kaputnik, a user from Global Village, is entering personal information into these presenters. As he types, the underlying ScriptX

program allows only one presenter at a time to have focus. Roger finishes entering his city, and he wants to move on to state—Global Village is in World Federation. Here is a step-by-step summary of what happens.

1. The keyboard device sends a focus event to the presenter that currently has focus to tell it that it is losing focus.
2. The `FocusEvent` class delivers the focus event to the `Addr:` presenter.
3. The `Addr:` presenter responds by removing its interests from the interests list maintained by the `KeyboardDownEvent` class.
4. The keyboard device sends a focus event to notify the `City:` presenter that it is gaining focus.
5. The `FocusEvent` class delivers the focus event to the `City:` presenter.
6. The `City:` presenter adds its interests to the interests list maintained by the `KeyboardDownEvent` class.

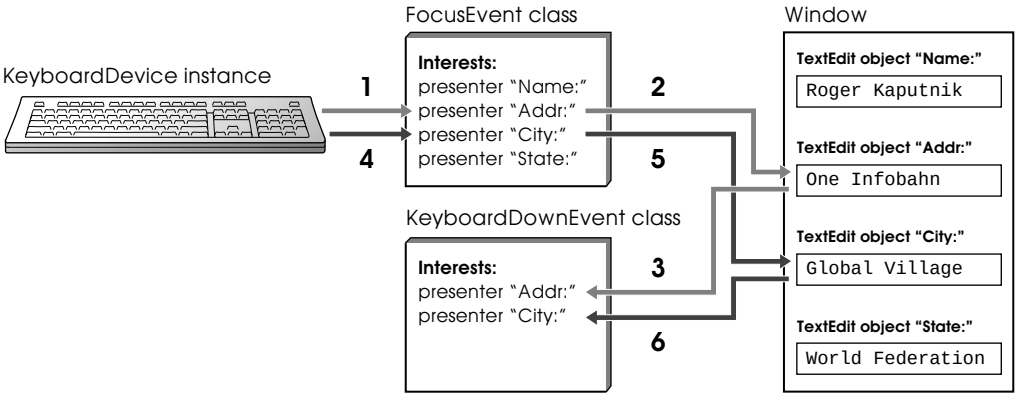


Figure 18-6: Focus events notify presenters that they are gaining or losing focus

A sample script that demonstrates the focus mechanism is available at the end of this chapter, on page 538.

Note – A focus event is not like a valve—it does not direct a flow of events to a particular receiver. It is a notification device. Event receivers are responsible for adding and removing their own event interests. Well-behaved receivers might be expected to add and remove interests on cue, but it is up to the program to supply this logic. (The `TextEdit` class does this automatically.) Focus events are only delivered to interested receivers. If a new receiver adds an interest in keyboard events, but does not add an interest in focus events, it can receive keyboard events even though it is not registered with the focus event mechanism.

Mouse Devices and Mouse Events

`MouseDevice` is an abstract class that acts as an interface between the ScriptX Player and the native operating system. A `MouseDevice` instance receives mouse events from the device driver and sends them to other classes as ScriptX events. As with keyboard events, developers should note that a future version of ScriptX will allow for virtual devices.

Mouse events are geographic in that they occur at a given location on the screen as well as at a point in time. The `MouseEvent` class adds several instance variables to handle the location of an event. Each mouse event maintains its own location in two different coordinate systems. The instance variable `surfaceCoords` stores a point representing the Kaleida Media Player coordinates where the event occurred. The instance variable `localCoords` expresses the location of the event in the local coordinate system of the presenter associated with the interest that receives the event. For information on screen coordinates, see the section "Transform, Position and Size" on page 68 in Chapter 3, "Spaces and Presenters".

The delivery system modifies a mouse event as it is being delivered. Until a mouse event is delivered to a receiver, usually a presenter or a controller that is managing objects in some presenter, the `localCoords` instance variable points to the origin and has no meaning. Each time an event is delivered to a receiver, its `localCoords` instance variable is revised. If the receiver accepts the event, then its local coordinates are cast in stone. They will remain unchanged until the event is eliminated by the garbage collector. This means that a copy of the mouse event will no longer be valid. If a mouse event is delivered to an interest that is not associated with a presenter, the class sets `localCoords` in the coordinate system of the top presenter. In many cases, the top presenter does not coincide with the coordinates of the display surface where the event occurred.

Note the distinction between the instance variable `currentCoords`, defined by `MouseDevice`, and the instance variable `surfaceCoords`, defined by `MouseEvent`. The former represents the location of the mouse at present. The latter represents the actual location of the mouse when a mouse event was generated.

Mouse Pointer

The mouse pointer is a movable bitmap image, with its associated mask, that represents the position of the mouse on a display surface. ScriptX allows a program to use the standard mouse pointers that are defined by the underlying operating system. Alternatively, a ScriptX mouse device can be associated with its own custom pointer, an instance of `Pointer`, and this pointer can be changed dynamically as the program runs. The value of `pointerType`, an instance variable defined by `MouseDevice`, determines the appearance of the mouse pointer.

To create a custom mouse pointer, define an instance of `Pointer`, setting the values of `bitmap`, `mask`, and `hotSpot` to create a pointer of the desired appearance. Set the value of `pointerType`, which is defined by `MouseDevice`, to reference the custom `Pointer` object.

Note – Distinguish between the mouse pointer and the text cursor. The mouse pointer refers to a movable bitmap that represents the position of the mouse on a display surface. The text cursor, also known as the insertion caret, represents the current insertion point in text.

When a ScriptX program uses pointer types that are defined by the underlying system, the appearance of the mouse pointer varies across platforms. For example, on a Windows or OS/2 system, the mouse pointer appears as an 16 x 16 hourglass icon when the value of `pointerType` is `@wait`. When the value of `pointerType` is `@wait` on MacOS, the pointer is a 16 x 16 watch icon.

Mouse Buttons and Modifier Keys

A mouse device supports events on three different buttons, each of which has a system name: `@mouseButton1`, `@mouseButton2`, and `@mouseButton3`. The mouse event classes specialize `isSatisfiedBy` to match events and interests to a particular button or set of buttons, if the interest specifies them. Implementation of buttons by the mouse event classes is similar to the implementation of modifier keys by the keyboard event classes.

Note the distinction between the instance variable `buttons`, defined by `MouseDevice`, and the instance variable `buttons`, defined by `MouseEvent`. The former represents the current state of the mouse buttons, as maintained by the device. The latter represents the state of the mouse buttons when an event was generated. If the mouse event is used as a template, it represents a desired state.

The handling of modifier keys by the mouse device and mouse event classes really parallels that of the corresponding keyboard classes. Mouse devices, like keyboard devices, track which modifier keys are active or inactive. Mouse events, like keyboard events, report which modifier keys were down when an event occurred. An event receiver can specify that it is interested only in events that occurred while certain modifier keys were pressed or released by setting the `keyModifiers` instance variable.

Mouse devices, and the gestures that are associated with them, are not standard across platforms. Since the ScriptX Player is designed to function with many operating systems, consider the needs and habits of users in different computing environments in designing a user interface. For example, many ScriptX users do not have a second or third mouse button. Modifier keys can be used to define alternative gestures for clicking on these buttons.

Presenters and Interests in Mouse Events

The class `TwoDPresenter` incorporates all presenters that can be displayed on a display surface, including 2D shapes, video and animation players, user interface objects, and text presenters. Every `TwoDPresenter` instance has an `eventInterests` instance variable, used to store interests in mouse events.

In effect, a `TwoDPresenter` object can be programmed to behave like a button and receive mouse input. Interests are attached to the presenter itself, not to the event class. In this way, interests in mouse events follow the presenter when it is moved around, even when it is moved to a new space.

Although presenters maintain their own interests in mouse events, the storage mechanism is transparent to the scripter. `addEventInterest` is a generic that every event class understands. A script calls `addEventInterest` on a mouse event interest, just as on any other interest. On a mouse event, if the `presenter` instance variable is not `undefined`, then the interest is attached to the presenter itself.

Interests in mouse events can leave their `presenter` instance variable undefined. If they define no presenter, they are stored with the event class itself, using the default mechanism for events. When a mouse event is sent by the `signal` method, the event class first searches for matching interests in its own interest collection. This allows a program to define an interest that has a higher priority than any presenter. If no interests that are stored by the class are satisfied by and accept this event, it continues by searching the presentation hierarchy.

Storing Interests in Mouse Events on Presenters

Every instance of `TwoDPresenter`, when it is displayed in a window, is associated with a region on that window's display surface. If a mouse event occurs within that region, the presenter can receive and process that event. A presenter can advertise an interest in a particular class of mouse events. Then, when the user performs that mouse action within the boundary of the presenter, only that presenter responds. In other words, when the event matches the interest that the presenter has advertised, the presenter receives the event and performs a specified action.

You can design presenters to be sensitive to only certain events. Clicking on, moving over, or dragging a presenter could cause it to change colors, change images, start an animation, bring in a new scene, or whatever you can imagine.

Setting up a Presenter to Receive Mouse Events

To make a presenter receive mouse events requires that you create an instance of a subclass of `MouseEvent`, fill in its instance variables as an interest, and add this interest to the presenter's list of interests, as shown in the following steps. A complete example is shown later in this chapter in the section "Selecting Presenters with a Mouse" on page 530.

1. **Create the mouse event interest** – Make a new instance of the appropriate `MouseEvent` subclass and fill in its instance variables, as follows:

```
myMD2 := new MouseDownEvent
```

   Assign values to the instance variables of the mouse event interest:

```
myMD2.presenter := myShape
myMD2.device := myMouse
```

```
myMD2.eventReceiver := selectFunction
```

This fills in the values of `myMD2`, as shown in the lower right corner of Figure 18-7. Possible instance variables that can be assigned for a mouse event interest include `presenter`, `device`, `eventReceiver`, `buttons`, `priority`, `matchedInterest`, and `authorData`. Note that as with all the event classes, the meaning of several of these instance variables differs for events and event interests.

2. **Add the interest to the interest list** – Add the interest to the appropriate interest list by calling `addEventInterest` on the interest itself:

```
addEventInterest myMD2
```

The `MouseEvent` class specializes `addEventInterest` to examine its `presenter` instance variable to determine where the interest should be stored. Since `myMD2` specifies a presenter (`myShape`), `addEventInterest` stores the event interest in the collection defined by the presenter's `eventInterests` instance variable, as shown in Figure 18-7. The `TwoDPresenter` class defines `eventInterests`, an instance variable, as a read-only list of event interests associated with a given 2D presenter. It can store any instance of `MouseEvent` that is defined as an interest.

Now the presenter is set up to receive a mouse-down event.

Matching a Mouse Event

When the user presses the mouse button down on `myShape`, the `MouseDownEvent` class goes through the three steps shown in Figure 18-7 to find an interest that matches the particular mouse-down event. These steps are as follows:

1. **Match in Class** – The `MouseDownEvent` class first looks in its `interests` class variable for a mouse-down event interest that matches the event. Interests in this list have `presenter` set to `undefined`, which means the class will try to match them first before the events are sent to presenters in the presentation hierarchy. A match occurs if the interest and event have the same mouse device and the same specified mouse buttons

   These two conditions are considered the interest's "template." If there's no match, the `MouseDownEvent` class continues looking for a match with the next interest. If there is a match, one of two things happens:

   - If `eventReceiver` contains a reference to a function, it runs, performing the operation you envisioned. If the function returns `true`, the event is "swallowed" and the rest of these steps are skipped. If it returns `false`, the event continues on to the next interest looking for another match.
   - If `eventReceiver` refers to an event queue, the thread that reads from the queue awakens and its function immediately accepts or rejects the event.

2. **Search Presenters** – If no interest that is stored in the `MouseDownEvent` class's own interest list accepts and "swallows" this event, the class searches through the presentation hierarchy (as described in Figure 18-7), seeking the front-most presenter whose boundary contains the display coordinates of the event. Front-most means that if presenters overlap at the point the mouse event occurred, they are tested in order of front to back.

3. **Match in Presenter** – When the class finds a presenter to test, it compares the event to interests held in the presenter's `eventInterests` instance variable. Once again, a match occurs if the interest and event specify the same mouse device and the same mouse buttons

   In this example, an interest in `myShape` satisfies the event if the user clicked with the proper mouse button within the boundary of `myShape`. The function referred to by `eventReceiver` is called and the matching process continues, exactly as in the first step.

   If this presenter does not swallow the event, the process alternates between searching in step 2 and testing for a match in step 3, until the event is accepted and "swallowed," or until the entire presentation hierarchy has been traversed.

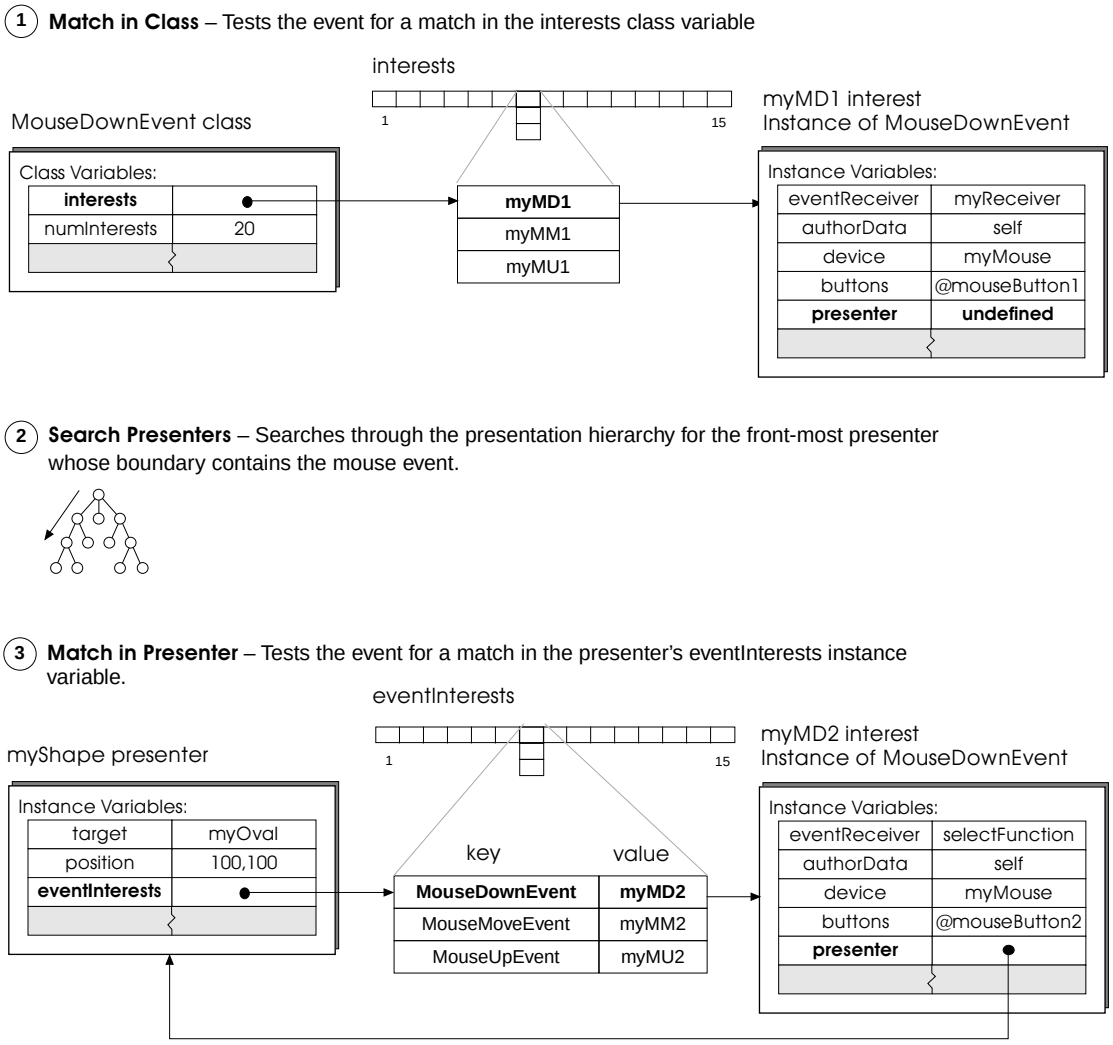


Figure 18-7: Three steps to search for and match mouse event interests.

`MouseEvent` classes search through the presentation hierarchy in a very specific manner to find an interest whose template matches the current event. Figure 18-7 shows the search order for a hierarchy that has 9 presenters. (The presenters are arranged left-to-right as they appear in the subpresenters list.)

Suppose the user clicks on presenter 8. The mouse event itself maintains the coordinates of the mouse click. The event system searches through the presentation hierarchy to find the front-most presenter at that location. If the mouse click is not within the boundary of presenter 1, it skips that presenter; otherwise it checks each of its subpresenters. It performs the same test for each subpresenter, going one level deeper each time the click is within a subpresenter's boundary. The search continues in this manner until it finds the deepest presenter for a particular branch that contains the event's coordinates. It then compares the event to the mouse interests in the presenter's `eventInterests` instance variable by calling `isSatisfiedBy` on those event interests.

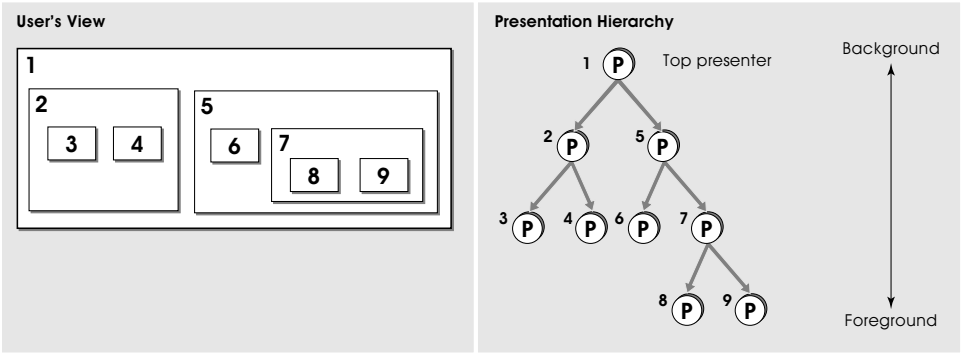


Figure 18-8: Search order for event matching through a presentation hierarchy.

In any presentation hierarchy, when two presenters overlap, one that is deeper down in the tree appears in front of the other. As a consequence, searching in a bottom-up manner allows the `MouseEvent` class to search through presenters from the front of the display surface towards the back. Thus, when a user clicks at a point on the screen where two or more presenters overlap, the presenter in front is the first to process the event. When that presenter is finished processing the event, it can either swallow the event, preventing it from continuing, or let it continue along to the next presenter in the hierarchy.

Note that this "bottom-up" order in which ScriptX traverses the presentation hierarchy to compare events and event interests is roughly opposite to the top-down order that is used when the compositor draws presenters on the display surface.

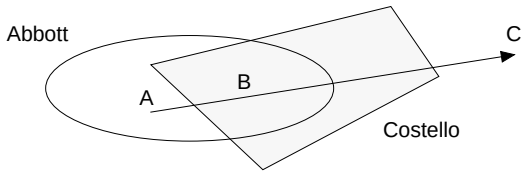
To optimize this search, the presentation hierarchy should be comprised of collections that are easily searched, such as arrays. Instances of `TwoDMultiPresenter` and its subclasses default to such an optimized collection as their target collection.

The `eventInterests` instance variable is normally defined only when a program calls `addEventInterest` on a subclass of `MouseEvent`. `TwoDPresenter` defines a generic function, `createInterestList` which can be specialized by a developer to create and store interests in new kinds of events. For example, a developer could specialize `createInterestList` to support a new kind of hardware device, such as a pen device. You should call `createInterestList` on a 2D presenter only if `eventInterests` is `undefined` on that presenter. Note that `eventInterests` is a read-only instance variable.

Matched Interests

Suppose a given presenter receives an instance of `MouseDownEvent`. What happens if the user drags the mouse away from that presenter before releasing it? The user could drag the mouse outside the boundary of the presenter entirely, or she could drag it into the boundary of another interested presenter that might intercept the event and receive it with higher priority.

The following diagram illustrates the problem of matching interests on mouse events. Suppose the user has just clicked the mouse at point A on Abbott, a `TwoDPresenter` instance that targets an oval, and has an interest in mouse-down events. Costello, another instance of `TwoDPresenter`, partially covers Abbott. If the user drags the mouse to point B and if Costello is interested in mouse-up events, then Abbott's interest will receive the event first. If the user continues dragging the mouse, without releasing the button, to point C, then neither Abbott nor Costello receives the associated event.



If a presenter needs to receive a mouse-up event that is associated with the most recent mouse-down event, the script should set the `matchedInterest` instance variable on its interest in the class `MouseUpEvent` to point to the matching event interest registered with the `MouseDownEvent` class.

This mechanism overrides the normal delivery of mouse-up events—it insures that the same presenter receives matching mouse-down and mouse-up events. Matched interests can be used to allow a presenter, such as a push button or menu, to clean up or undo actions that began when they received a mouse-down event. In the diagram above, this means that if you press the mouse button at point A and then release it at B or C, Abbott still receives the associated mouse-up event.

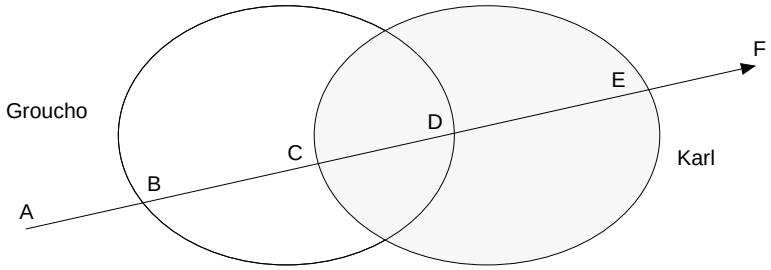
Note – The `matchedInterest` instance variable, although writable, is an event-only instance variable on other `Event` subclasses. This use of `matchedInterest` by the `MouseUpEvent` class is specialized.

Mouse Crossing Events

The `MouseCrossingEvent` class provides an efficient mechanism to register when the mouse enters and exits from a 2D presenter. Every `TwoDPresenter` object has a `boundary` instance variable, which points to a `Stencil` object. Each time the mouse travels across the boundary of a presenter that is visible, the system can potentially generate a mouse crossing event. A typical use of `MouseCrossingEvent` is to allow a presenter to modify its appearance as the mouse travels over it.

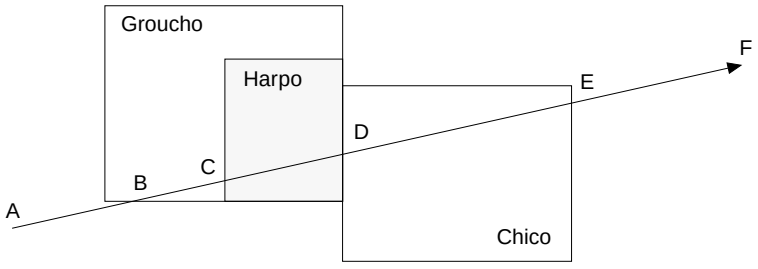
The `crossingType` instance variable has two possible values: `@enter` and `@leave`. A single event is associated with entering a presenter, and another with leaving. If any receiver receives an event with `crossingType` set to `@enter`, then it is guaranteed to receive the matching event with `crossingType` set to `@leave` as soon as the mouse travels back across the presenter's boundary.

In the following diagram, Groucho and Karl are overlapping presenters, siblings in the presenter hierarchy, each of which targets an instance of `Oval`. Karl is in front of Groucho, meaning that Karl is drawn on top of Groucho when presenters in this space are composited. Suppose that Groucho and Karl have each registered interests in receiving mouse crossing events. As the user moves the mouse in a straight line from A to F, the system generates mouse crossing events at points B, C, and E. Note that no event is generated at point D, since Karl is the frontmost presenter.



The following describes what happens at point C in this diagram. Karl is the frontmost presenter, so the event system first generates a `MouseCrossingEvent` object with the value of `crossingType` set to `@leave`. This event passes through the primary dispatch queue and is delivered to Groucho. Next, the system generates another `MouseCrossingEvent` with `crossingType` set to `@enter`. This event passes through the primary queue and is delivered to Karl. The primary queue assures that the mouse cannot enter Karl before it leaves Groucho. However, it does not assure that these two events are delivered atomically. The thread system can switch while these events are being processed.

In the following diagram, Groucho, Harpo, and Chico are overlapping presenters, each of which targets an instance of `Rect`. Harpo is a subpresenter of Groucho. Groucho and Chico are siblings in the presenter hierarchy. The `x2` boundaries of Groucho and Harpo are coterminous with the `x1` boundary of Chico. All three presenters have registered interests in mouse crossing events. As the user moves the mouse in a straight line from A to F, the system generates mouse crossing events at points B, C, D, and E.



The following description applies to points C and D in this diagram. When the mouse reaches point C, it does not leave Groucho in order to enter Harpo. Since Harpo is a subpresenter of Groucho, the event system generates a mouse crossing event with the value of `crossingType` set to `@enter` for Harpo. At this point, two events with `crossingType` set to `@enter` are extant that have not been matched with leave events. When the mouse crosses three presenter

boundaries at point D, it first generates a leave event for Harpo, and then a leave event for Groucho. Only then does the system generate an enter event for Chico. For a script that demonstrates mouse crossing events, see the example "Receiving Mouse Crossing Events" on page 541.

Note – It is possible for the user to jerk the mouse 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 `@enter`, it is guaranteed to generate the matching event with crossing type `@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.

As with all mouse events, any interest not associated with a presenter (its `presenter` instance variable is `undefined`) has priority over interests that are stored by presenters. If there is an interest that is not associated with a presenter, then the `MouseCrossingEvent` class generates events every time the mouse enters or leaves *any* presenter that is visible in the window. (This can have a noticeable effect on performance if the number of presenters is large.)

Polling and Mouse Devices

An alternative to tracking mouse movements by waiting to receive events is to poll the mouse device directly. To poll a device is to query it continuously for some change in state. Since ScriptX is a multi-threaded system, a process that runs in one thread can continuously poll for changes in the state of a device.

Polling of input devices must always be done from a thread that can block. (The `MouseDevice` class defines accessor methods that block while waiting on `theUIEventDispatchQueue`. See the definition of `screenCoords`, an instance variable defined by `MouseDevice`, in the *ScriptX Class Reference*.)

Polling is an alternative to blocking, which is implicit to the event handling system. Blocking places fewer demands on the system, but is constrained by the throughput of the queued event delivery mechanism. (For additional reading on blocking, see the "Threads" chapter of this volume.)

Compatibility Across Platforms

Version 1.0 of the ScriptX Player assumes that a mouse and a keyboard are physically attached to the system. Since this version of the ScriptX Player is designed primarily with desktop computers in mind, developers can count on these basic input capabilities for current titles.

Keep in mind that input devices are not standardized across hardware platforms. For example, Windows can operate without a mouse, but when a mouse is attached to a Windows-based computer, it is generally a two-button mouse. The Macintosh requires a mouse with at least one button. Some vendors sell a Macintosh mouse or trackball with additional buttons that can be customized by the user, but there are no standards or user interface

guidelines that dictate what this extra button should do. Keyboards are close to standardization; however, users of the Windows, OS/2, Macintosh, and Unix operating systems have different expectations for how they should perform.

A carefully designed title reflects the differences in how particular modifier keys or mouse buttons are used across the various hardware platforms on which the ScriptX Player runs. Title developers should create a user interface that is comfortable and accessible to users on all platforms.

Certain keyboards, especially those on computer systems sold for use in homes and classrooms, do not support a full extended keyboard. For example, many keyboards are not equipped with function keys or with cursor control keys such as page up and page down. To determine that a particular keyboard supports a given key, use the `existKey` instance method, which is defined on the class `KeyboardDevice`.

It is always good form to check the system for an input device that meets your needs before you actually use it. A title can call `systemQuery` to assure that it has access to a keyboard or mouse.

Performance is also a compatibility issue. ScriptX receives events from input devices through the underlying operating system and processes them at a speed that is dependent on underlying hardware. What happens when a user flicks a mouse across a large screen, passing over hundreds of pixels along the way? A high-performance system will signal many more mouse-move events than a low-performance system. When the `MouseMoveEvent` class signals events, it actually looks ahead at the primary dispatch queue to see if there are other `MouseMoveEvent` instances ahead of it in the primary dispatch queue. The class does not signal a new event until the old one is removed from the primary event queue. The same issue applies to `autoRepeat`, which is an instance variable on `KeyboardDevice`.

Input Devices of the Future

As the field of multimedia develops, consumers will want multimedia players that do not require a desktop computer. Watch for a future version of the ScriptX Player to run on a console connected to a television set. As the ScriptX Player moves into “form factors” outside the realm of the desktop computer, ScriptX will evolve to encompass new input devices, such as a digital joystick.

Of course, a console that sits on a television monitor may not be attached to a physical keyboard. A future version of ScriptX will get around this problem by creating a virtual keyboard, a software emulation of a keyboard that can be operated with a mouse or joystick. Thus, a title does not need to be concerned about whether a physical keyboard exists.

ScriptX will provide virtual counterparts of a keyboard and mouse so that title developers are assured of basic input capabilities. If an input device does not exist in hardware or if the hardware implementation does not meet the author’s requirements, a virtual device, with identical functionality, will be created for the author’s use.

Although you can create an instance of `PhysicalKeyboard` or `PhysicalMouse`, it is better to create a `KeyboardDevice` or `MouseDevice` object. Although the latter classes are abstract, they determine automatically what hardware devices are connected to the system. In a future release of the ScriptX Player, calling `new` on `KeyboardDevice` or `MouseDevice` will automatically create an instance of the appropriate concrete class, either a physical device or a virtual representation of it.

Events and Input Devices Examples

The following example scripts demonstrate events and input devices:

- Selecting Presenters with a Mouse
- Processing with an Event Queue
- Focus Events
- Receiving Mouse Crossing Events

Event handling is already built into several ScriptX core classes, such as `ScrollBar`, `ActuatorController`, `DragController`, and `TextEdit`. Thus, it is possible to create a ScriptX title without ever descending into the underlying event system. But the events system offers many hooks for specializing existing classes. A common use of the Events component is to create a new subclass of `Event` for communication within a title. Another application is to create a new user interface object, such as an instance of `Controller` that receives user events.

Selecting Presenters with a Mouse

This example creates a mixin class called `Selectable`, which demonstrates how to make specific presenters respond to mouse events. This class is mixed in with 2D shapes to make them “selectable” in a primitive way. When a user clicks on a 2D shape, the shape is moved in front of other presenters in the window and momentarily highlighted (with a heavy bounding box). This is a short example; it would be easy to imagine different and more elaborate ways of highlighting and selecting objects, and extending this script to implement them.

After creating the `Selectable` class, this example creates “event receiver” methods that are called when the event is received and matched. Each mouse event has its own method that it calls. A mouse button down event causes the 2D shape to move to the front and become highlighted. A mouse button up event causes the highlight to disappear. It then creates an initialization method which sets up and adds the mouse event interests to the event system.

Finally, this example creates a window, which is a 2D space, and adds two shapes to the window.

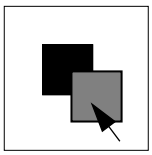


Figure 18-9: Selecting a presenter with the mouse.

The Selectable Class

The `Selectable` class has two instance variables and three methods. The instance variables `mouseDown` and `mouseUp` are declared in this script; the methods are the event receiver and initialization methods described next. These instance variables are used only while setting up the mouse event interests, until the interests are added to the interest lists using `addEventInterest`.

```
-- A mixin class
class Selectable (RootObject)
    instance variables
        mouseDown      -- For setting up an interest in mouse-down events
        mouseUp        -- For setting up an interest in mouse-up events
        selectionBox   -- Appears around the presenter to highlight it
end
```

Event Receiver Methods

This script defines two methods that are called in response to particular mouse events: `mouseDownSelect` and `mouseUpSelect`. The first method moves the shape to the front of the window and highlights it with a bounding rectangle when the mouse button is pressed. This method then removes the interest in `mouseDown` and adds an interest in `mouseUp`. It returns `true` so the event will be swallowed by the event system and not passed on to any other presenters.

The `mouseUpSelect` method deletes the bounding rectangle when the mouse button is released, removes the interest in `mouseUp`, adds an interest in `mouseDown`, then returns `true` for the same reason as the previous method.

```
-- Define the method that is called when mouse-up occurs
method mouseDownSelect self {class Selectable} theInterest theEvent ->
(
    -- Move the 2D presenter to the front of its container
    moveToFront self.presentedBy self

    -- Create a selection rectangle that just fits the 2D presenter
    self.selectionBox := new TwoDShape
        self.selectionBox.target :=(new Rect x2:self.bBox.x2 y2:self.bBox.y2)
        self.selectionBox.stroke := new Brush color:yellowColor
        self.selectionBox.x := self.x
        self.selectionBox.y := self.y
```

```
        self.selectionBox.stroke.linewidth := 2
    prepend self.presentedBy self.selectionBox

    -- Now we're interested only in mouse-up events
    removeEventInterest self.mouseDown
    addEventInterest self.mouseUp
    @accept -- must accept for event to be swallowed
)

-- Define the method that is called when mouse-up occurs
method mouseUpSelect self {class Selectable} theInterest theEvent ->
(
    deleteOne self.presentedBy self.selectionBox

    -- now we're interested only in mouse-down events
    removeEventInterest self.mouseUp
    addEventInterest self.mouseDown
    @accept -- must accept for event to be swallowed
)
```

Initialization Method

This method first calls on its superclass (`RootObject`) to initialize the new object. It then creates instances of a mouse-down and mouse-up event interest. Each event interest has `eventReceiver` and `authorData`, which specifies the method to call (`mouseDownSelect` or `mouseUpSelect`) if the incoming mouse event matches the interest. It also has a mouse device to respond to and a presenter to respond to.

The presenter for `mouseDown` is `self`, which indicates the method is called for a mouse-down only within the boundary of the presenter. The `matchInterest` instance variable for `mouseUp` ensures the shape receives the `mouseUp`, even if the user drags off the shape before releasing the mouse button.

Note that this `init` method must be positioned after the event receiver methods, since this method refers to them.

```
method init self {class Selectable} #rest args ->
(
    apply nextMethod self args        -- call init on superclass

    -- set up the mouse event interests
    self.mouseDown := new MouseDownEvent
    self.mouseDown.eventReceiver := mouseDownSelect
    self.mouseDown.authorData := self
    self.mouseDown.device := new MouseDevice
    self.mouseDown.presenter := self

    self.mouseUp := new MouseUpEvent
    self.mouseUp.eventReceiver := mouseUpSelect
    self.mouseUp.authorData := self
    self.mouseUp.device := self.mouseDown.device
```

```
    self.mouseUp.presenter := self
    self.mouseUp.matchedInterest := self.mouseDown

    -- Initially, we're interested in mouse-down events
    addEventInterest self.mouseDown
)
```

Testing the Selectable Class

The following script creates a window, then creates and prepends black and red rectangle shapes into the window. Note that the shapes are created by mixing together `Selectable` and `TwoDShape`.

Once you've run this script, use the mouse to click on the rectangle that is in background. It should move to the front and be highlighted with a heavy bounding rectangle only so long as the mouse button is down.

```
-- Set up a simple example of black and red rectangles in a window
global myWindow := new Window boundary:(new Rect x2:300 y2:300)
    myWindow.x := 40
    myWindow.y := 40
    show myWindow

-- Create the black rectangle
object myBlackRect (Selectable, TwoDShape)
end
    myBlackRect.target := new Oval x2:100 y2:100
    myBlackRect.fill := blackBrush
    myBlackRect.x := 75
    myBlackRect.y := 75
    prepend myWindow myBlackRect

-- Create the red rectangle
object myRedRect (Selectable, TwoDShape)
end
    myRedRect.target := new Rect x2:100 y2:100
    myRedRect.fill := new Brush color:redColor
    myRedRect.x := 125
    myRedRect.y := 125
    prepend myWindow myRedRect
```

For another script that demonstrates how to receive mouse events, adding and removing interests as appropriate, see "Example – Stencilizer" on page 274 of Chapter 11, "2D Graphics."

Processing with an Event Queue

The following example creates a class called `EventDispatcher`. This script expands upon the `Dispatcher` class, defined on page 607 in the "Threads" chapter. As you study this section, compare `EventDispatcher` with

Dispatcher. The Dispatcher class creates a regular thread that runs at high priority, waiting for function calls. The EventDispatcher class is similar, except that it waits for events to pop through an event queue. Both of these classes maintain a private list of operations they know how respond to. Like the Dispatcher class, EventDispatcher is useful as a quick responder. When an event appears in its queue, it runs briefly at high priority, responding to the event and then blocking. Although it could respond in its own thread, it is really designed so that it can activate some other thread that runs at normal priority.

```
class EventDispatcher (RootObject)
instance variables
    interestList -- KeyedLinkedList
    thread -- a thread for responding to user actions
    eventQueue -- use this queue to receive events
instance methods
    method init self #rest args -> (
        self.interestList := new KeyedLinkedList
        -- note that the event queue replaces the condition
        -- in the Dispatcher class, in the Threads chapter
        -- the queue itself acts as a gate
        self.eventQueue := new EventQueue thread:(self.thread)
        self.thread := new Thread func:dThreadFn arg:self priority:@system
        apply nextMethod self args
    )
end -- (Dispatcher class definition)
```

The Dispatcher class uses a Condition object as a gate. Its thread waits on this condition when it has nothing to process. EventDispatcher does not explicitly define a gate, but an event queue creates a gate internally that performs an analogous role. Dispatcher defines the pleaseDo method, which activates its thread. EventDispatcher defines no pleaseDo method because this behavior is embodied in its event queue. Placing an event in an EventDispatcher event queue is analogous to calling pleaseDo on a Dispatcher object.

EventDispatcher defines an instance variable, interestList, with which it maintains a private list of its event interests, together with associated data. Although this collection is defined as a KeyedLinkedList, it could be any linear collection. In this script, the event interest itself is used as a key, with its authorData instance variable storing a function or method. The value of the key stores an object that is used as an argument for this function whenever the function is called. In ScriptX, KeyedLinkedList is suitable for relatively small collections where frequent additions and deletions are expected. Since events do not implement a method for localLt, they cannot be used as keys for sorted collections such as SortedKeyedArray or BTree.

To maintain this collection, EventDispatcher defines methods for adding and removing event interests. These methods maintain the contents of interestList, as well as adding and removing event interests on the interests collection, a class variable of the event subclass itself. The

`interestList` collection is heterogenous, unlike the collection each event class maintains. It maintains a list of many kinds of event interests, but it may also store several instances of the same class.

```
method addInterest self {class EventDispatcher} eventClass func data -> (
    -- check that func is a kind of function
    if isAKindOf func AbstractFunction then (
        -- check that eventClass is a class, and that it inherits from Event
        if isAKindOf eventClass RootClass and isSub eventClass Event then (
            local newInterest
            newInterest := New eventClass
            newInterest.authorData := func
            newInterest.eventReceiver := self.eventQueue
            newInterest.priority := 2
            addEventInterest newInterest
            add self.interestList newInterest data
        ) else (
            format debug "Second argument must be an event class!/n" \
                   undefined @normal
        )
    ) else (
        format debug "Third argument must be a function or generic!/n" \
                   undefined @normal
    )
)
method removeInterest self {class EventDispatcher} eventClass -> (
    -- check that eventClass is currently defined
    -- and that the event dispatcher has as an interest in it
    if isAKindOf eventClass RootClass and isSub eventClass Event then (
        local i := iterate self.interestList
        repeat while (next i) do (
            if (getClass i.key = eventClass) do (
                removeEventInterest i.key
                excise i -- this deletes it from the collection
                exit -- remove only the first occurrence
            )
        )
    ) else (
        format debug "Second argument must be an event class!/n" \
               undefined @normal
    )
)
```

Like `Dispatcher`, the `EventDispatcher` class creates its own thread. This thread runs a control function that is similar in form to the dispatcher's control function. Structured as an infinite loop, this process blocks while waiting for events to be placed in its queue. When an event is placed in its queue, it immediately becomes active. (A system-priority thread runs immediately when it becomes active.)

```
function dThreadFn dispatcher -> (
    local myEvent := undefined
```

```
    local data := undefined
    repeat while (myEvent := read dispatcher.eventQueue) do (
        -- figure out how the event was delivered and
        -- get the data that goes with the associated interest
        if isAKindOf (myEvent.matchedInterest) Event then (
            -- must have been delivered by signal method
            accept myEvent
            data := dispatcher.interestList[myEvent.matchedInterest]
            -- call the function with event and data as arguments
            (myEvent.matchedInterest.authorData) myEvent data
        ) else (
            -- event could not have been signaled, so see if broadcast
            -- if matchedInterest is undefined, it must have been delivered
            -- using the sendToQueue method
            if myEvent.matchedInterest <> undefined then (
                -- matchedInterest can have one of three values
                -- it can be an event, an array of events, or undefined
                -- it must have been delivered by broadcast method
                -- get an iterator for the array of matching interests
                -- this next line creates a sequence iterator i
                local i := iterate (myEvent.matchedInterest)
                local myClass := getClass myEvent
                -- now iterate through the matchedInterest collection
                repeat while (next i) do (
                    -- make sure it is one of our own interests
                    -- see if the matching interest is in our own list
                    -- if not, then go on to the next matched interest
                    if getOne dispatcher.interestList (i.value) = empty do (
                        continue -- so skip and go on to the next
                    )
                    data := dispatcher.interestList[i.value]
                    -- call the function with event and data as arguments
                    (i.value.authorData) myEvent data
                    -- excise it in case there were other event interests
                    -- in our own private list.
                    -- this only removes it from the collection of interests
                    -- that matched the current event. It is still
                    -- registered as an interest to receive future events.
                    excise i
                    exit
                ) -- end repeat
            ) else (
                -- must have been delivered by sendToQueue method
                format debug "not interested in this event!\n" undefined @normal
                myEvent := undefined
                continue -- ignore this event, go back and wait for another
            )
        )
        -- reset it so the thread will block until the next event comes along
        myEvent := undefined
    )
)
```

This thread function recovers data from both the event and the event interest it was delivered to. If we were only interested in receiving an event, we could get away with a far less complicated thread function. When an event receiver function receives an event, it receives the associated interest as well, as a third

parameter. An event queue does not automatically receive the associated event interest, so it must recover this interest itself, if it needs it. An event, once it has been delivered, stores a pointer to the matching interest in its `matchedInterest` instance variable.

When the function `dThreadFn` receives an event, it first checks whether the event was delivered by the `signal`, `broadcast`, or `sendToQueue` methods. Since it cannot observe the delivery mechanism directly, it looks at the value of `matchedInterest`. If the matched interest is an event, it must have been signaled. If it is an array, it must have been broadcast. Since `sendToQueue` bypasses the interest matching system, it leaves the value of `matchedInterest` as `undefined`. The most interesting case is when an event is broadcast, since the script must determine which element of the array contains the matching interest. The function `dThreadFn` modifies the `matchedInterest` collection, removing its matching interests as it finds them.

To test the `EventDispatcher` class, create an instance of the class and add some event interests to it. Then create some new event classes, some instances of those classes, and some associated functions. Send the event dispatcher some events using the `signal`, `broadcast`, and `sendToQueue` methods.

```
-- now create three new classes of events to test it with
class GrokEvent (Event) end
class VoodooEvent (Event) end
class EarthQuakeEvent (Event) end
-- create an instance of EventDispatcher
global gDispatcher := new EventDispatcher
-- the init method on Dispatcher automatically creates a thread
-- define some functions that will be called
global fn func1 myEvent x -> (
    format debug "Function 1 just got a %* event " (getClass myEvent) @normal
    format debug "and its argument is %*.\n" x @normal
)
global fn func2 myEvent x -> (
    format debug "Function 2 just got a %* event " (getClass myEvent) @normal
    format debug "and its argument is %*.\n" x @normal
)
global fn func3 myEvent x -> (
    format debug "Function 3 just got a %* event " (getClass myEvent) @normal
    format debug "and its argument is %*.\n" x @normal
)
-- now create some interests in the three kinds of events
-- add them to the interest list, with their associated functions and arguments
addInterest gDispatcher GrokEvent func1 "moof"
addInterest gDispatcher VoodooEvent func2 "foo"
addInterest gDispatcher EarthQuakeEvent func3 "7.1"
addInterest gDispatcher EarthQuakeEvent func1 "8.7"

-- create some events and send them
global myGrokEvent := new GrokEvent
global myVoodooEvent := new VoodooEvent
global prittyBigOne := new EarthQuakeEvent
global rillyBigOne := new EarthQuakeEvent
signal myGrokEvent true
signal myVoodooEvent true
```

```
broadcast prittyBigOne
broadcast rillyBigOne
```

To test the event dispatcher thoroughly, we need to create a competing event receiver and some associated interests. This receiver "competes" in the sense that it registers interests in the same event classes, but without using the `addInterest` and `removeInterest` methods defined by `EventDispatcher`. The event dispatcher must be able to sort its own interests out from those of other event receivers when it searches through the `matchedInterest` collection.

```
-- create an event receiver and associated event interests
function eventEater event interest data -> (
    format debug "eventEater just received an event. \n" undefined @normal
    format debug "authorData: %* \n" event @normal
    format debug "Interest: %* \n" interest @normal
    format debug "Event: %* \n" data @normal
)
global shakeInterest := new EarthQuakeEvent
shakeInterest.authorData := "No function at all!"
shakeInterest.priority := 1 -- this one has higher priority
shakeInterest.eventReceiver := eventEater
addEventInterest shakeInterest
global rattleInterest := new EarthQuakeEvent
rattleInterest.authorData := "Definitely not a function!"
rattleInterest.priority := 3 -- this one has lower priority
rattleInterest.eventReceiver := eventEater
addEventInterest rattleInterest

-- now create some more events and signal them
global mightyOne := new EarthQuakeEvent
global awesomeOne := new EarthQuakeEvent
global giantOne := new EarthQuakeEvent
broadcast mightyOne
broadcast awesomeOne
sendToQueue giantOne gDispatcher.eventQueue
```

In the previous segment, the function `eventEater` registered two event interests, one of which was at a lower priority than our dispatcher's event queue. As you experiment with this script, change the relative priorities of the various event interests. Note that the event dispatcher, which runs in its own thread, tends to process events after the function receiver, which runs in the same thread as the process that sent the event.

Focus Events

This script demonstrates how to receive and process focus events. The `TextEdit` class handles focus events automatically, but handling of focus events can easily be added to another presenter class. This script creates the `FocusedPresenter` class, which forces the keyboard focus manager to give it focus whenever it receives a mouse-down event. The script creates three

instances of `FocusedPresenter` and attaches them to the window. Each focused presenter maintains interests in both focus and mouse-down events. The presenters indicate when they gain or lose focus by changing color.

```
class FocusedPresenter (TwoDShape)
    class variables
        focusManager:((new KeyboardDevice).focusManager)
    instance variables
        mouseInterest
        focusInterest
    instance methods
        method init self #rest args -> (
            apply nextMethod self args
            self.focusInterest := new FocusEvent
            self.focusInterest.authorData := self
            self.focusInterest.device := new KeyboardDevice
            self.focusInterest.presenter := self
            self.focusInterest.eventReceiver := processFocus
            addEventInterest self.focusInterest
            self.mouseInterest := new MouseDownEvent
            self.mouseInterest.authorData := self
            self.mouseInterest.presenter := self
            self.mouseInterest.eventReceiver := processMouse
            addEventInterest self.mouseInterest
        )
        -- event receiver for focus events
        method processFocus self interest event -> (
            if (event.focusType = @loseFocus) then (
                -- we lost focus, so show that it is disabled
                event.presenter.fill := whiteBrush
            ) else (
                -- we gained focus, so be colorful!
                local fillcolor := new RGBColor \
                       red:(rand 255) green:(rand 255) blue:(rand 255)
                event.presenter.fill := new Brush color:fillcolor
                @accept -- accept the event
            )
        )
        -- event receiver for mouse events
        method processMouse self interest event -> (
            forceFocus FocusedPresenter.focusManager self
            @accept -- accept the event
        )
end


object firstRect (FocusedPresenter)
    boundary:(new rect x2:200 y2:50), fill:whitebrush, stroke:blackbrush
    settings x:50, y:50
end
object secondRect (FocusedPresenter)
    boundary:(new rect x2:200 y2:50), fill:whitebrush, stroke:blackbrush
    settings x:50, y:150
end
object thirdRect (FocusedPresenter)
    boundary:(new rect x2:50 y2:250), fill:whitebrush, stroke:blackbrush
    settings x:100, y:25
```

```
end

-- now set up a window
object myWindow (Window)
    boundary:(new Rect x2:400 y2:300), fill:whitebrush
    settings x:16, y:40
end
append myWindow firstRect -- add it to the space
append myWindow secondRect -- add it to the space
append myWindow thirdRect -- add it to the space
show myWindow
```

`TextEdit` objects register interest in focus events and respond to changes in focus automatically. The following script creates three `TextEdit` objects and adds them to a window. As you edit in one of the three presenters, it gains focus, and the one that previously had focus automatically loses focus.

```
object myWindow (Window)
    boundary:(new Rect x2:600 y2:300), fill:whitebrush
    settings x:16, y:40
end
object myCursor (Line) x2:0, y2:16 end
global testString1 := new Text string:"edit me, pretty please!"
global testString2 := new Text string:"oh no, edit me!"
global testString3 := new Text string:"please, edit me first!"
object textBox1 (TextEdit)
    boundary:(new Rect x2:384 y2:60)
    target:testString1, stroke:blackBrush
    settings x:128, y:64, cursor:myCursor
end
setDefaultAttrs textBox1 @alignment @flush
object textBox2 (TextEdit)
    boundary:(new Rect x2:384 y2:60)
    target:testString2, stroke:blackBrush
    settings x:128, y:128, cursor:myCursor
end
setDefaultAttrs textBox2 @alignment @flush
object textBox3 (TextEdit)
    boundary:(new Rect x2:384 y2:60)
    target:testString3, stroke:blackBrush
    settings x:128, y:192, cursor:myCursor
end
setDefaultAttrs textBox3 @alignment @flush
append myWindow textBox1
append myWindow textBox2
append myWindow textBox3
show myWindow
```

With the `TextEdit` class, focus is transparent to the scripter. Each time you select one of the three text strings to edit, the `TextEdit` object that is presenting it gains focus, and the one that was previously selected loses focus.

Receiving Mouse Crossing Events

Mouse crossing events come in two flavors. The instance variable `crossingType`, defined by the `MouseCrossingEvent` class, can take on two values: `@enter` and `@leave`. This script sets up a class of presenter that receives and processes mouse crossing events, however its form could be applied to any type of mouse event. It uses a method as an event receiver.

The script that follows builds on an earlier script in that it demonstrates how mouse events can be registered by presenters, and delivered to presenters within a presentation hierarchy. For background, see the section "Selecting Presenters with a Mouse" on page 530.

The program first creates a window in which to demonstrate mouse crossing events, and two instances of `Brush`. It uses `greenBrush` to show that a presenter has just received an event with the value `@enter`, and `redBrush` to show that it has received an event with the value `@leave`.

```
-- first create a window with default settings
object myWindow (Window)
    boundary:(new rect x2:600 y2:300)
    settings x:20, y:40
end
show myWindow

global greenBrush := new Brush color:greenColor
global redBrush := new Brush color:redColor
```

The `EnterPresenter` class is a stock version of `TwoDMultiPresenter`, with the addition of a single property, the instance variable `mci`, to store an event interest. It also adds a method to receive these events, `colorMe`. `EnterPresenter` specializes the `init` method to create an interest in `MouseCrossingEvent`, which it registers as an event interest.

```
class EnterPresenter(TwoDMultiPresenter)
instance variables
    mci -- an interest in mouse crossing events
instance methods
    method init self #rest args -> (
        apply nextMethod self args
        -- create a mouse crossing event and set its properties
        -- so that it can be used as an event interest
        self.mci := new MouseCrossingEvent
        self.mci.presenter := self
        self.mci.authorData := self
        self.mci.eventReceiver := colorme
        self.mci.device:= new MouseDevice
        -- register it as an event interest
        addEventInterest(self.mci)
    )
    -- this method will receive mouse crossing events
    method colorme self match myEvent -> (
```

```
        if (myEvent.crossingType = @enter) then (
            self.fill := greenBrush -- green if you enter
        ) else (
            self.fill := redBrush -- red if you leave
        )
        print myEvent.crossingType
        @accept -- accept the event, since asynchronous
    )
end
```

Next, the script creates six instances of `EnterPresenter` and adds them to the window. The first five of these presenters will be siblings on the presentation hierarchy, since they are prepended to the window at the same level. The final instance, `insideShape`, is prepended to the fifth presenter, so it becomes its child presenter. For more information on the presentation hierarchy, see "Presentation Hierarchy" on page 56 in Chapter 3, "Spaces and Presenters."

```
-- make some overlapping presenters that will receive
-- mouse crossing events and add them to the window
object shape1 (EnterPresenter)
    boundary:(new rect x2:240 y2:260), stroke:blackBrush
    settings x:20, y:20
end
prepend myWindow shape1
object shape2 (EnterPresenter)
    boundary:(new rect x2:340 y2:260), stroke:blackBrush
    settings x:240, y:20
end
prepend myWindow shape2
object shape3 (EnterPresenter)
    boundary:(new rect x2:180 y2:140), stroke:blackBrush
    settings x:80, y:20
end
prepend myWindow shape3
object shape4 (EnterPresenter)
    boundary:(new rect x2:180 y2:120), stroke:blackBrush
    settings x:240, y:20
end
prepend myWindow shape4
object shape5 (EnterPresenter)
    boundary:(new rect x2:200 y2:120), stroke:blackBrush
    settings x:360, y:20
end
prepend myWindow shape5

-- make one more to put inside the last one
object insideShape (EnterPresenter)
    boundary:(new rect x2:100 y2:100), stroke:blackBrush
    settings x:100, y:50
end
-- put insideShape inside shape5 to show how
-- mouse crossing events works with the presentation hierarchy
-- note that it will be clipped by shape5
```

```
prepend shape5 insideShape
```

The window displays the six presenters as overlapping rectangles, drawing their borders with the `Brush` instance `blackBrush`. Initially, they are not filled in. Drag the mouse over these presenters and observe how they change color as they receive mouse crossing events. The `colorMe` instance method, defined by `EnterPresenter`, also prints the crossing type to the Listener window.

CHAPTER

Files and System Services

19

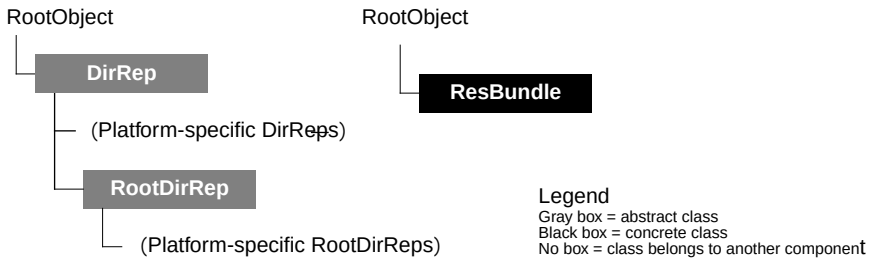
The Files component provides a platform-independent interface for working with files in ScriptX. Two abstract classes form the core of this interface: `DirRep` and `RootDirRep`. Instances of these classes represent the directories of any supporting platform's file system.

Along with the Streams component, the Files component provides access to the data in a file. In general, you access a file using methods defined by the `DirRep` class, then open a `Stream` instance on the file and use `Stream` methods to access the data.

One class in the Files component, `ResBundle`, defines an interface to Macintosh resource files.

Classes and Inheritance

The class-inheritance hierarchy for the Files component is shown in the following figure.



The following classes form the Files component. In this list, indentation indicates inheritance.

`DirRep` – Represents a directory. Each operating system that supports ScriptX has a file-system specific subclass of `DirRep` to handle platform-dependent implementation details.

> `RootDirRep` – Represents the root directories for operating systems that use volumes or drives to represent individual storage devices. ScriptX includes a file-system-specific subclass of `RootDirRep` on each such platform to handle platform-dependent implementation details.

`ResBundle` – Represents a resource file in the Macintosh file system.

Conceptual Overview

Most file systems are organized hierarchically—for example, files are organized by directories on DOS and by folders on the Macintosh. This hierarchical structure provides commonality between file systems: to find a file, you specify

the name of the file and the directory path where that file resides. In ScriptX, this behavior is implemented abstractly through a single class—DirRep—that defines common operations that can be performed on directories. Subclasses of DirRep are implemented in file-system-specific ways to match the specific features of each host.

ScriptX assumes that all host file systems provide certain standard directories. ScriptX defines three global constants to represent these directories: theRootDir, theStartDir, and theTempDir. A fourth global, theScriptDir, is available only in the development system, not in the ScriptX Player. The global constant theRootDir represents the root directory of the host file system. If the operating system requires a volume or drive for each storage device, theRootDir is an instance of the system-specific subclass of RootDirRep. Otherwise it is an instance of the system-specific subclass of DirRep. The global theTempDir represents a directory that scripts can use for temporary file storage. The global constant theStartDir represents the directory used by the ScriptX or ScriptX Player executable.

To represent a specific directory, a script uses one of the global instances of DirRep to create the new representative. Two DirRep instance methods, createDir and spawn, are useful for this purpose. Scripts can use either of these methods to create new instances of DirRep at any time. The difference is that createDir is used to create a new directory, actually changing your disc and returning a DirRep object which represents the new directory. The method spawn, on the other hand, is used to access a directory that already exists; it returns a DirRep object which points to the already-existing directory and does not change your disc. A script can call the same methods on the directory representatives it creates to create yet other instances.

How Files Work

All operations on directories and files are performed through instances of DirRep and its subclasses. When you create an instance of DirRep, ScriptX automatically returns a subclass of DirRep that is appropriate for the underlying platform and its particular file system. This subclass of DirRep handles platform-dependent implementation details transparently, which means that you can operate on directories and files using DirRep methods without worrying about features specific to the host file system.

Access to Directories and Files

DirRep objects represent specific directories in a file system. A DirRep object can represent a directory path, but not a file.

Almost all the DirRep instance methods are of the form methodName(*object*, *path*, ...). To access a file through a DirRep object, you create the object to represent a specific directory, then operate on that directory through the DirRep methods. The *path* argument specifies the file or subdirectory in the directory that you want to work with. This argument can be a String object

or an ordered collection of `String` objects. When an ordered collection is passed as the *path*, the method automatically traverses the directories represented in the collection in order.

Access to Data

To read, write, and seek data in a file, a script interacts with a `Stream` object representing the data. To obtain a `Stream` object for a specific file, a script calls `getStream` on a `DirRep` object. Typically, the `DirRep` object represents the path to the file, while the *path* argument to the `getStream` method is a string specifying the name of the file to access. Alternately, the *path* argument can also be a collection of strings representing a path and filename.

Macintosh Resource Files

The `ResBundle` class provides a platform-dependent representative of Macintosh resource files. `ResBundle` provides a number of methods for accessing the resources contained in a resource file. Data for a resource is represented by a `ResStream` instance. See the "Streams" chapter for more on opening resource files and accessing resources.

Note – Code that uses instances of `ResBundle` and `ResStream` is necessarily platform-specific to the Macintosh. It is intended to support Macintosh-based tools that provide access to media in resource files. It is not recommended for use in a title designed to run on any platform that supports ScriptX.

Using the Files Component

To use a file, you generally start with one of the global `DirRep` instances defined by the ScriptX runtime environment and then use `DirRep` methods to access specific subdirectories and files. The following examples demonstrate the effects of invoking various `DirRep` methods on these global instances.

Path References

Most instance methods of `DirRep` accept sequences of strings to represent paths and filenames. These methods also accept a string that uses the slash character, "/" as a separator. For example, the following two statements use the `createFile` method do the same thing:

```
createFile theTempDir "dirA/dirB/file" @binary
createFile theTempDir #("dirA","dirB","file") @binary
```

The following two statements use `createFile` to create a text file named "a file":

```
createFile theTempDir "a file" @text
createFile theTempDir #("a file") @text
```

Despite appearances, the following two statements are not the same. Both create text files, but the first creates a file `fob` inside the directory `fib`, while the second creates a file called `fib/fob`:

```
createFile theTempDir "fib/fob" @text
createFile theTempDir #("fib/fob") @text
```

Testing Files and Directories

The `DirRep` methods `isDir`, `isFile`, and `isThere` provide quick and easy ways to test for file types and file and directory existence. These will not report exceptions unless something is terribly wrong.

```
isThere theStartDir "testing"
isDir theStartDir #("testing")
isFile theRootDir  "yabba/dabba/doo"
```

The `DirRep` method `getFileType` returns a symbol specifying the type of the object.

```
createFile theStartDir #("example") @text
getFileType theStartDir "example"
=> @text
```

The method `getFileType` will report an exception if the specified path does not exist:

```
delete theStartDir "example"
getFileType theStartDir "example"
```

Directory Paths as Sequences

To retrieve the path to a particular directory, you coerce its `DirRep` to a `Sequence`. For example, you can coerce the global instance `theStartDir` and then use the sequence returned as the path argument to another `DirRep` method:

```
s:= theStartDir as sequence
⇨ #("HardDisk", "ScriptX 1.5 Folder", "Apps")
getFileType theRootDir  s
⇨ @directory -- theStartDir is a directory
```

This set of operations returns `@directory`, since `theStartDir` is a directory

Naming Files

It is highly recommended that you use file names which contain eight characters or less followed by a dot and an extension of three characters or less (xxxxxxxx.xxx). This ensures that your file name will work on all platforms because it follows the most restrictive naming requirements.

Below are examples of file names following the recommended 8.3 convention:

```
myTitleH.sxt -- a title file
myLib.sxl -- a library file
myNeatAc.sxa -- an accessory file
myStats.dat
Chaptr12.txt
```

Converting File Names

To be portable, DirRep objects try to do something reasonable with all possible file names. Every platform has a file name converter that takes any string and turns it into a valid filename for the host platform. For example, if you give a file a name with more than eight characters, that name will have to be converted to a name with eight characters or less in order to be used on a Windows platform. This conversion is done by the instance method fixNameForOS. On the Macintosh, filename conversion is simple (for example, it changes ":" into "-"). Under Windows, the converter performs many character remappings and then abbreviates to arrive at a filename with eight characters and a three-character extension.

As an example of filename conversion, consider the following use of the fixNameForOS method:

```
fixNameForOS theStartDir "This is a test"
```

On the Macintosh, the above call returns the string "This is a test." Under Windows, it returns "THISIATT."

Note – The fixNameForOS method isn't symmetrical. As an example, consider the previous result. The string "This is a test," applied on Windows produces "THISIATT." However, applying "THISIATT" to the same method on the Macintosh won't restore the original string.

The best practice, however, is to simply eliminate the need for fixNameForOS by always using the 8.3 format to name files.

Creating Instances of DirRep

There are two methods that return new instances of DirRep: spawn and createDir.

The createDir method creates a new directory and returns an instance of DirRep which represents its entire path. This method actually changes your disc. Since createDir will create all non-existing directories in a path supplied to it, it may create more than one directory.

For example, the following code returns a DirRep instance for the directory bottom, creating, in the process, the top, middle, and bottom directories:

```
s:=createDir theTempDir #("top","middle","bottom")
```

The method spawn also creates a new DirRep instance, but that instance represents an already-existing directory. Since spawn does not create any new directories, it does not change your disc. If a call to spawn has a pathname containing a directory that does not exist, it will fail.

Given that createDir created the directories top and middle, spawn can be used to create a DirRep instance for either of them. For example, the following code succeeds:

```
u:=spawn theTempDir #("top","middle")
```

On the other hand, given that the directory path left/right/center doesn't yet exist, a call to spawn will fail:

```
t:=spawn theTempDir #("left","right","center") -- fails
t:=createDir theTempDir #("left","right","center") -- succeeds
```

Navigating Directories

There are a number of DirRep methods that help you figure out where you are and how to get to where you want to go.

As mentioned earlier, the class coercion as Sequence returns a full pathname to a DirRep. This may cause problems on systems that allow ambiguous paths, such as the Macintosh.

The parentDir method returns a DirRep instance for the parent of the given DirRep instance.

```
(parentDir theStartDir) asSequence -- succeeds
parentdir theRootDir -- fails because theRootDir has no parent
```

The first statement should return a valid sequence; however, the second statement reports an exception, since the theRootDir global has no parent directory.

The getContents method returns a list of the entries in the passed DirRep.

```
getContents theStartDir
```

Note – ScriptX is UTF based, but currently the Listener window isn't; therefore, filenames that have meta-characters don't show up properly in the Listener window. They are, however, being stored correctly and will work properly.

File Creation

The methods createDir and createFile are the standard ways to create files and directories with DirReps. Note that createFile will create a directory when @directory is given as its third argument. Both createDir and createFile create intermediate directories if needed.

The first example creates a text file `afile` inside the directory `is` inside the directory `this`.

```
createFile theTempDir "this/is/afile" @text
```

The next example creates a directory called `/ouch/`.

```
createDir theTempDir #("/ouch/")
```

The next example creates a directory called `smarts` inside the directory `/ouch/`.

```
createFile theTempDir #("/ouch/","smarts") @directory
getFileType theTempDir #("/ouch/","smarts") -- should return @directory
```

File Deletion

The `DirRep` `delete` method deletes directories or files. If you attempt to delete a directory which contains files or subdirectories, neither the files, nor the subdirectories, nor the directory will be deleted. For example, the following code will not delete anything because the directory `ouch` contains the subdirectory `smarts`:

```
delete theTempDir #("/ouch/") -- no deletion
```

The following invocations of `delete` are successful, however, because the last item is either a file or a directory with nothing in it:

```
delete theTempDir #("/ouch/","smarts") -- deletes the subdirectory
--"smarts”
delete theTempDir #("/ouch/") -- deletes the empty directory "/ouch/"
delete theTempDir "this/is/afile" -- deletes the file "afile"
delete theTempDir "this/is" -- deletes the empty subdirectory "is"
```

On the Macintosh, if a file is in use, the file won’t be deleted.

Note – On Windows systems, due to a limitation in MS-DOS, the `delete` method will delete a file even if it has a stream open on it.

Access to Streams

Creating a file does not give you access to its contents. You must use the `getStream` method to get the *primary stream* for a file. The term *primary stream* applies to the contents of the data fork of a Macintosh file, as well as to the ordinary contents of files under MS-DOS and Unix. The term allows for the fact that alternate interfaces may provide access to other streams for files. Currently, only the `ResBundle` class provides such an interface, by allowing a script to open a stream for each resource when multiple resources are present in a Macintosh resource file.

Note that getStream doesn't create new files. The following code demonstrates how to create files and open streams on them:

```
createFile theTempDir #("myFile") @text
s := getStream theTempDir "myFile" @writable
writeString s "Zoicks! Powee! Kablam!"
s
⇨  "Zoicks! Powee! Kablam!"
streamLength s
⇨  22
plug s -- closes the stream s and also the file "myFile"
delete theTempDir "myFile"
```

The following sample code creates a DirRep instance representing the path "/common/photos/london," then opens a stream on the file "picadilly" (which already exists in the subdirectory "/london") and reads a byte:

```
myRep := spawn theRootDir  "/common/photos/london"
myStream := getStream myRep "picadilly" @readable
myByte := readByte myStream
```

Note that the DirRep methods recognize only the "/" (slash) character as a directory separator. However, they also accept collections of strings, recognizing each entry in the collection as a distinct subdirectory, with the final entry recognized as either a subdirectory or file name. This means that a collection of subdirectory names can be used in place of the string in the previous example:

```
myPath := #("common","photos","london")
myRep := spawn theRootDir  myPath
myStream := getStream myRep "picadilly" @readable
myByte := readByte myStream
```

As another example, a script could create a collection representing an entire path, including filename, then use that collection in the getStream method to open the file:

```
myFilePath := #("common","photos","london","picadilly")
myStream := getStream theRootDir  myFilePath @readable
myByte := readByte myStream
```

In this example, the last entry in the collection is interpreted as the name of the file to open.

On Windows, due to the way that DOS handles files, data written into a file by a writable stream cannot be read unless the file is first closed. A workaround is to use the plug method to close the stream (and the underlying file), and then to use getStream to get a new readable stream for the file. For example, the following code properly writes to, then reads from, the file "newFile.txt":

```
createFile theStartDir "newFile.txt" @text
outStream := getStream theStartDir "newFile.txt" @writablewriteString
outStream "HELLO"
plug outStream
inStream := getStream theStartDir "newfile.txt" @readable
```

Open and Save Dialog Boxes

This section offers a brief discussion of how a title uses the open and save dialog boxes, defined by the underlying operating system, to open and save files in ScriptX. The file types you can open and save include:

`@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 text
`@unknown`	The file type is not specified

The platforms on which the ScriptX Player runs have some standard user interface elements, such as the dialog boxes that are used for opening and saving files. ScriptX allows a developer to use these dialog boxes from within a ScriptX title or tool. Two global functions, `presentOpenFilePanel` and `presentSaveFilePanel` are the means by which a title or tool displays the **Open** and **Save As** dialog boxes defined by the underlying operating system. These functions allow the developer to select a file to open or to name a file to save, but it is up to the title to actually open or save a file.

Open Dialog Box

The following script demonstrates how to open a file from within a ScriptX title using the global function `presentOpenFilePanel`. The single argument is an array containing `NameClass` objects designating the types of files to be displayed as options in the **Open** dialog box. The possible file types are those listed above.

```
presentOpenFilePanel #(@title, @binary)
```

This function call will cause an **Open** dialog box native to the underlying system to be displayed. It will offer as possible selections only files which contain `TitleContainer` objects and files which contain binary data because those are the file types specified in the argument to `presentOpenFilePanel`. If *@unknown* or any unknown value is specified in the argument, the **Open** dialog box allows any file type to be selected. The function returns an array containing the full path of the file selected. If the user clicks **Cancel** (no file is selected), the function returns `undefined`.

The title itself is responsible for opening the file which is returned by `presentOpenFilePanel`. To open a title container, library container, or accessory container, use the `open` method. To open any other kind of file (text or binary), use the `DirRep` and `Stream` methods `getStream` and `read`.

Save As Dialog Box

The **Save As** dialog box is generated by the global function `presentSaveFilePanel` and can save any type of file previously listed. This function takes two arguments, a string giving the prompt, which will appear near the file entry field, and a string which will appear as the default filename in the file entry field when the dialog box first opens. The following script shows how to create a **Save As** dialog box:

```
presentSaveFilePanel "Save As" "filename"
⇨  #("music", "composers", "Mozart")
```

The function displays a **Save As** dialog box with the prompt "Save as" and the default filename "filename" in the file entry field. It returns an array specifying the complete path to the file, including the name of the file. In this example, the user typed in "Mozart" as the filename. If the user clicks **Cancel** instead of **OK** or **Save** (depending on the platform), the function returns `undefined`.

The title itself is responsible for saving the file returned by `presentSaveFilePanel`. To save a title container, library container, or accessory container, use the `update` method. To save any other kind of file (text or binary), use the `DirRep` and `Stream` methods, as follows:

- To save to a new file, use `createFile`, `getStream`, `write`, and `plug` methods
- To save to an existing file, use `getStream`, `write`, and `plug` methods

If your title creates a file, and you want to put that file in the directory where the title container is located, use the title's `directory` instance variable. This helps keep files neatly located in one place.

Filenames

Although file panels are a cross-platform implementation of standard file dialog boxes, they do not impose cross-platform solutions for file naming. Each operating system that runs the ScriptX Player imposes different restrictions on the length of filenames and on the characters they can contain. A call to `presentOpenFilePanel` or `presentSaveFilePanel` returns a filename that is legal in the current environment and for the current directory. It is up to the developer to impose restrictions to make filenames compatible on other platforms.

Note – If your title can create files that ScriptX titles on other platforms may need access to, it should address the issue of filename compatibility. For example, version 3.1 of Microsoft Windows limits filenames to `filename.ext` (8 characters followed by a period and a 3-character-extension for file type), while MacOS limits filenames to 31 characters.

Message Dialog Boxes

The global function `presentMessagePanel` provides the ability to display a dialog box with a developer-supplied message. Its syntax is as follows:

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

- *message* - a `String` object which is the text to display
- *icon* - `NameClass` object, which can be 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 which is the index of the button that is the default (the button selected if the user hits the return key)
- *cancel-ix* - `ImmediateInteger` object which is the index of the button that is to be returned by the cancel key

Example

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

This code 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”. In this example “No” is both the default button and the cancel button.

CHAPTER

20

Streams

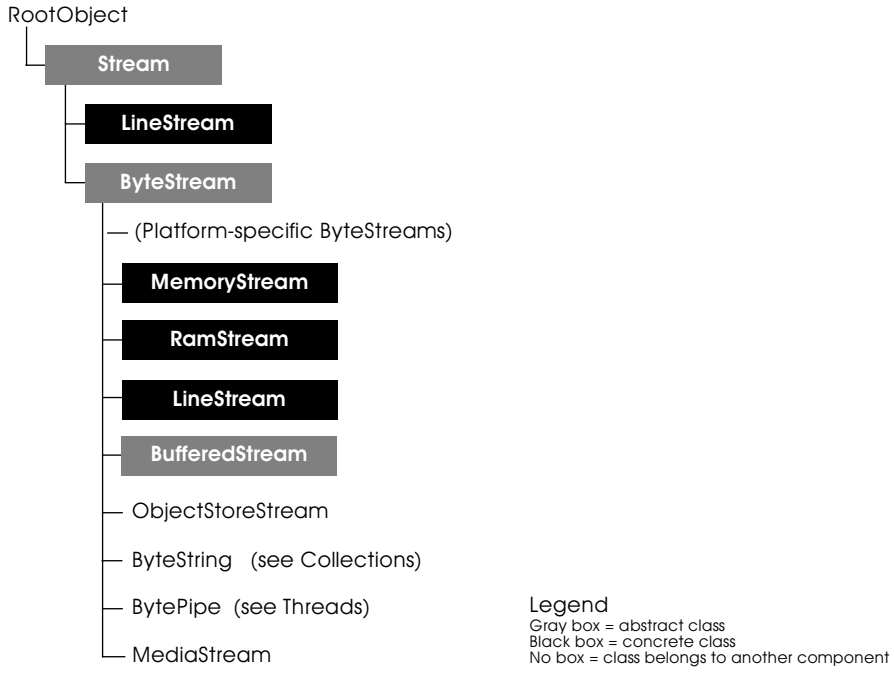
The ScriptX Streams component provides a standardized interface for working with linear sequences of data elements. The abstract class `Stream` provides the protocol for this interface, which applies to linear sequences of objects. Another abstract class, `ByteStream`, extends `Stream` behavior by defining methods for accessing linear sequences of bytes. Other components in ScriptX provide task-specific implementations of the behavior defined by these Stream classes.

For all platforms that support ScriptX, there are platform-specific subclasses of `ByteStream` that provide byte-oriented access to files on the host file system. The `Iterator` class implements streamed access to objects in collections. Classes in other components of ScriptX provide content-specific implementations of stream behavior. For example, the `AudioStream` class provides access to streams of audio data, and `VideoStream` provides access to streams of video data.

In addition to methods defined by `Stream` and `ByteStream`, the Streams component provides a number of functions that are used to format and print objects to streams. These are described more fully in the "Output" section of the chapter "Working With Objects" in the *ScriptX Language Guide*.

Classes and Inheritance

The class inheritance hierarchy for the Streams component is shown in the following figure.



The following classes form the Streams component. In this list, indentation indicates inheritance.

`Stream` – An abstract class representing data as linear sequences of elements and defining behavior to read, write, and seek elements in the stream.

- `ByteStream` – A class of objects that represent streams of bytes and provide the ability to read bytes from or write bytes. Subclasses of `ByteStream` provide platform-specific implementations of this behavior, to enable access to data in files on each platform that supports ScriptX. Subclasses of ByteStream in other components—`MediaStream`, `AudioStream`, `VideoStream`—add format-specific behavior to `ByteStream`.
  - `MemoryStream` – A class of objects representing a variable-size stream of bytes in memory.
  - `RamStream` – A class of objects representing a fixed-size stream of bytes in memory.
  - `LineStream` – A class of objects whose data elements are lines of characters represented in a system-independent way.
  - `BufferedStream` – Defines a general buffered byte stream, allowing ScriptX to handle chunks of data of various types in memory buffers.

Conceptual Overview

A *stream* represents a linear sequence of uniform data elements. Streams in ScriptX are represented by concrete subclasses of the `Stream` class. The data elements in a ScriptX stream are usually objects.

To provide access to individual data elements, most streams provide a *cursor* to keep track of their current position. When a stream is first opened, the current position is at the beginning of the stream, and the cursor is set to 0. In this position, the stream is ready for access to the first data element. As shown in Figure 20-1, the cursor may be thought of as being positioned "just before" the data element ready to be accessed. As you move through the stream, accessing its data, the cursor is updated to reflect the current position in the stream.

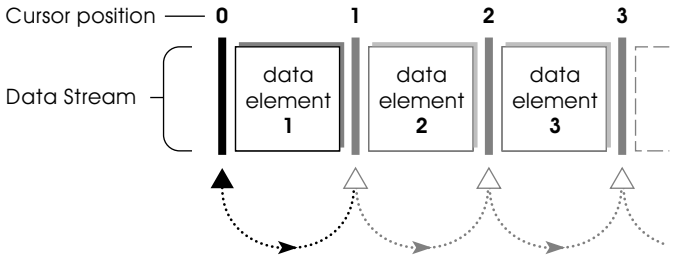


Figure 20-1: Cursor position is just before the current data element

Depending on the type of data and the purpose of the stream, a stream may be *read-only*, *write-only*, or *read-write*. For example, data from an input port would be represented by a read-only stream, data going to an output port would be represented by a write-only stream, and the pixel data for an editable bitmap would be represented by a read-write stream.

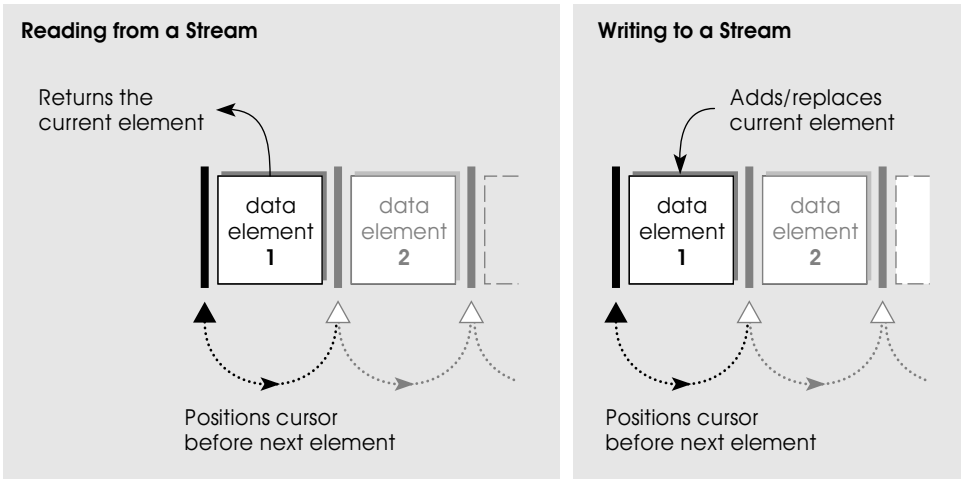


Figure 20-2: Readable and writable streams

The read and write operations, shown in Figure 20-2, give you access to the element just past the cursor. After the element is accessed, the cursor is placed beyond that data element and just before the next element in the stream.

Data in a stream may be intended for access any number of times or just once. For example, a stream representing pixel data for an editable bitmap would allow access to any of its data elements at any time, while a stream representing the input buffer of an I/O port would contain changing data, so reaccessing a particular data element wouldn't be possible.

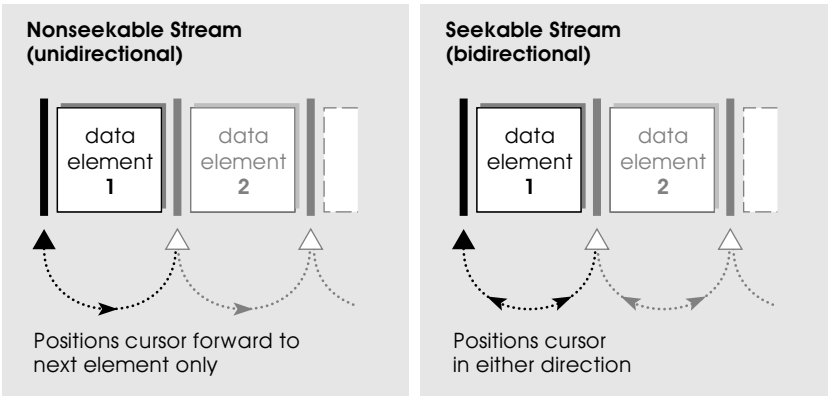


Figure 20-3: Seekable and nonseekable streams

As shown in Figure 20-3, there are two types of streams: *seekable* and *nonseekable*. A *seekable* stream allows you to set the cursor to any position in a stream and access the data element at that point. A *nonseekable* stream provides one-way access to data. The cursor can be moved forward in a nonseekable stream, but not back. Once a particular position in the stream has gone by, you can't access the data element at that position.

How Streams Work

The `Stream` class provides an abstract representation of the standard ways to access streams of data. Subclasses of `Stream` are customized for both a particular data source and a particular data type. For example, the ScriptX `ByteStream` class redefines `Stream` methods for access to streams of bytes. Further, ScriptX defines a number of stream subclasses for a variety of data types and data sources.

Creating instances of these ScriptX-defined streams is discussed in this section, followed by a description of standard techniques for accessing data in streams. Finally, this section lists the methods to override when creating customized subclasses of `Stream` for specific purposes.

Stream Subclasses Defined by ScriptX

Among the streams defined by ScriptX are file-system-specific subclasses of `ByteStream` for each operating system that supports the ScriptX runtime environment. These classes recognize files as their data source and bytes as their data type. ScriptX also defines the `ByteStream` subclasses `MemoryStream`, to represent variable-sized memory buffers, and `RamStream`, to represent fixed-size memory buffers. The `Iterator` class defines stream-oriented access to the objects in a collection. The `ObjectStoreStream` class provides a kind of stream to use for storing arbitrary data in a storage container. The `ResStream` class represents individual resources in a Macintosh resource file.

Streams for Files

To create a stream for a file, you use the `DirRep` method `getStream`. This method returns an instance of a file-system-specific subclass of `ByteStream`.

```
myText := getStream theStartDir "file.txt" @readable
```

The stream returned by this method can be used for byte-oriented access to data in a file. A stream created with the `getStream` method can be opened in one of three modes: `@readable` (read-only), `@writable` (write-only) or `@readWrite`, for both. You use `Stream` methods to read and write data in a file.

Streams for Memory Buffers

Scripts can create streams to represent memory buffers using instances of the `MemoryStream` or `RamStream` class.

To create a fixed-size memory buffer, you create a new `RamStream` object and specify its maximum size.

```
myBuffer := new RamStream maxSize:1012
```

Note – In the current release of ScriptX, optimal buffer sizes are 2^n-12 (as in the example, where n is 10), since the ScriptX memory manager takes 12 bytes of any allocated block for its own uses.

To create a growable memory buffer, you create a new `MemoryStream` object and specify its initial size, grow size, and maximum size.

```
myBuffer := new MemoryStream initialSize:500 \
              growSize:128 maxSize:1012
```

Memory buffers created as instances of either `RamStream` or `MemoryStream` are readable, writable, and seekable.

Streamed Access to Collections

The `Stream` subclass `Iterator` provides streamed access to collections. Subclasses of `Collection` use the `iteratorClass` instance variable to determine the particular `Iterator` subclass appropriate to accessing their data elements. An iterator for a particular collection can be acquired through its `iterate` method.

```
MyIterator := iterate MyCollection
```

Iterators are readable, writable, and seekable.

Streams for Object Storage

An object store stream is intended to store certain types of data in specific storage containers. The `ObjectStoreStream` class provides an abstract stream for data storage containers, with subclasses providing storage-system-specific implementations of this capability.

To store arbitrary data in a container, you first write the data to the stream, then store the `object` referring to that data in the container. To create an object store stream, you use the function `newObjectStoreStream`. This function returns either a buffered or unbuffered storage stream, depending on the size of the buffer you specify.

```
myStorageStream := newObjectStoreStream 1012 myContainer
```

This code example returns a buffered stream with a buffer size of 1012.

```
myStorageStream := newObjectStoreStream 0 myContainer
```

This code example returns an unbuffered stream.

For more on object storage, see the “Object Store” chapter in this guide.

Streams for Macintosh Resource Files

The `ResBundle` class defined in the Files component represents Macintosh resource files. The `ResStream` class defines behavior for streamed access to data for a specific resource. To get a stream for a resource, you first create a `ResBundle` instance for the resource file, then request a particular resource from that file.

```
myBundle := new ResBundle dir:theStartDir path:"MyResources"
myStream := getOneStream myBundle type:"snd " name:"elvisTrak"
```

This code first creates a `ResBundle` instance, `myBundle`, for the resource file `MyResources` in the ScriptX startup directory. It then requests a stream from `myBundle` for the sound resource named `elvisTrak`. In most cases, the next step in this process would be to give this stream to the importer, which would then import the data and return a media stream appropriate to the data.

```
sndStream := importMedia theImportExportEngine myStream @sound @snd
@audioStream
```

Note – The `ResBundle` and `ResStream` are platform-specific to the Macintosh and are probably most useful in platform-specific tools. Titles should save and retrieve data in a platform-independent way. If a title must use the `ResBundle` and `ResStream` classes, it should include alternative code allowing access to corresponding data in a platform-independent way.

Media Streams

The `MediaStream` class, a subclass of `ByteStream`, defines additional methods and instance variables for use in media playback. Subclasses of `MediaStream` implement this behavior for specific types of media, including audio, video, and MIDI. These streams are designed for use with corresponding media players. Media streams and media players are discussed in the "Media Players" chapter of this Guide. The discussion of importers in the *ScriptX Tools Guide* describes how to import media streams.

Access To Streamed Data

To read, write, and seek data in a stream, you interact with the `Stream` object representing that data. Whatever data elements the stream represents—bytes in a file, bytes in memory, objects in a collection—the methods for data access are the same.

Reading and Writing Data

To read data in a stream, you invoke the stream's `read` method. This method positions the cursor just past the current data element and returns that element. You could use the `read` method in a loop to read bytes of information

from the stream for a file, converting the bytes to a string of characters in the process. The following code opens a stream on a file, creates an empty string, then copies all the bytes from the stream to the string :

```
myTextStream := getStream theStartDir "file.txt" @readable
myString := " " as String
for i in (1 to (streamlength myTextStream)) do (
    myString[i] := (read myTextStream)
)
```

A more efficient way to read lines of text from a file is to use the `LineStream` class, as in the following example:

```
myTextStream := getStream theStartDir "file.txt" @readable
myLines := new LineStream source:myTextStream
myString := new String
repeat while ((i := read mylines) != empty) do addmany mystring i
```

Note that the `read` method is implemented by `LineStream` to return lines of text from a file as strings, minus their terminating characters (such as LF, CR or CR/LF). If you wish to add the new line character (`\n`) to each line, you need to explicitly do so while reading from the file. For example:

```
repeat while ((i := read mylines) != empty) do (
    addmany mystring i -- add the unterminated line
    addmany mystring "\n" -- add the new line character
)
```

To write data to a stream, you invoke the stream's `write` method. The `write` method writes one data element at the cursor position, replacing the previous element at that position, if any. It then repositions the cursor just past that element. You could use the `write` method in a loop to write a string of characters to a file using the following code:

```
myString := "some character string"
myTextStream := getStream theStartDir "file.txt" @writable
for i in (1 to (size myString)) do (
    write myTextStream myString[i]
)
```

You would use similar code with an iterator to read objects from or write objects to a collection. You could also use such code to transfer bytes of data between a file and a memory buffer represented by a `MemoryStream` or `RamStream` instance.

When a writable stream is associated with an external device, such as a file or serial port, you need to be sure that any data you've written to the stream is actually transferred to the device. To do so, you call the `flush` method. Plugging a stream, discussed in the next section, closes a stream and implicitly flushes its data.

Note – On Windows systems, due to the way that DOS handles files, data written into a file by a writable stream cannot be read unless the file is closed first. A work around is to use the `plug` method to close the file, then use `getStream` to get a new readable stream. For example, the following code properly writes to, then reads from, the file "newfile.txt":

```
createFile theStartDir "newfile.txt" @text
outStream := getStream theStartDir "newfile.txt" @writable
writeString outStream "HELLO"
plug outStream
inStream := getStream theStartDir "newfile.txt" @readable
```

Seeking Data

As mentioned earlier, a stream's cursor determines the current position in the stream. The cursor is positioned just before the data element that will be affected by the next `read` or `write` operation. When you first open a stream, the cursor's position is 0, just before the first data element. To find the current cursor position for a stream, you call the `cursor` method.

To choose which data element to access, you position the cursor in a stream. Use the `isSeekable` method to test whether or not the cursor can be positioned in a stream. Most ScriptX streams are seekable, including streams for files, streams for memory buffers, and the iterators for collections.

There are three `Stream` methods that explicitly position the cursor within a stream. The `seekFromStart` method positions the cursor relative to the beginning of its stream, while `seekFromEnd` positions it relative to the end, and `seekFromCursor` positions it relative to its current position. To position the cursor using these methods, you specify the stream and the relative position. For example, to write data at the end of a file, you would position the cursor at the end of the stream representing that file as follows:

```
myStream := getStream theStartDir "MyFile" @writable
seekFromEnd myStream 0
```

You can use positive or negative values to specify the position of the cursor. In the following example, the cursor is moved toward the front of the stream, 100 bytes before its current position:

```
seekFromCursor myStream -100
```

To move the cursor a single data element at a time, you use the `next` and `previous` methods. The `next` method moves the cursor forward one data element and returns `true`. If you call `next` with the cursor at the end of the stream, it returns `false`. Similarly, the `previous` method moves the cursor back one data element and returns `true` until it is at the front of the stream, when it returns `false`.

While the `next` and `previous` methods can only place the cursor within the bounds of the data elements, the seek methods can place the cursor at any position, including beyond the beginning or end of the data. The `isAtFront` and `isPastEnd` methods let you test the position of the cursor.

The `streamLength` method always returns the actual number of data elements in the stream, regardless of the cursor position.

Plugging Streams

Streams are often transient, used to transfer data elements from one place to another, then discarded. The `plug` method lets you discard a stream, performing any cleanup necessary to make the stream ready for disposal. For example, streams for data files must flush any data that hasn't yet been written to the file, close the file, and dispose of any file handles acquired from the underlying operating system. This work is performed by the `plug` method. When you call `plug` on a stream, the stream object is invalidated.

Defining Custom Stream Classes

Developers can define subclasses of `Stream` for handling specific kinds of data elements. You may need a customized iterator for a specialized collection subclass, or a customized buffer for storing specific types of data. In some cases, the task of creating a customized stream subclass may be beyond the capabilities of ScriptX alone: for example, to create a stream to represent a serial device or a printer, you will have to use the C-language API described in "Extensions to ScriptX" in the ScriptX Developer's Guide. However, in other cases, you can simply implement the appropriate behavior in ScriptX.

The `Stream` class defines methods that should be implemented to enable different types of access to data. The following tables list the methods that must be implemented by various types of Stream subclasses:

Table 20-1: Methods implemented by all Stream subclasses

Method	Implementation
`isAtFront`	Return `true` if cursor is at 0
`isPastEnd`	Return `true` if cursor is greater than stream length
`isReadable`	Return `true` if stream is readable, false otherwise
`isWritable`	Return `true` if stream is writable, false otherwise
`isSeekable`	Return `true` if stream is seekable, false otherwise
`next`	Position cursor at next data element. Return `true` if cursor is within the stream, `false` if cursor is at end of stream
`plug`	Release stream resources, void stream object and contents

Table 20-2: Methods implemented by readable Stream subclasses

Method	Implementation
`isReadable`	Return `true`
`read`	Return the current data element. Position the cursor at the next data element
`readReady`	Return the number of data elements that can be read without blocking

Table 20-3: Methods implemented by writable Stream subclasses

Method	Implementation
`isWritable`	Return `true`
`write`	Place specified data element at current position. Position the cursor at next data element
`writeReady`	Return number of data elements that can be written without blocking

Table 20-4: Methods implemented by seekable Stream subclasses

Method	Implementation
`cursor`	Return current position
`isSeekable`	Return `true`
`previous`	Position cursor at previous data element. Return `true` if cursor is within the stream, `false` if cursor is at beginning of stream
`seekFromCursor`	Position cursor at offset specified from current position
`seekFromEnd`	Position cursor at offset specified from end of data
`seekFromStart`	Position cursor at offset specified from beginning of data
`setStreamLength`	Set number of data elements in the stream. If the value is less than the current length, remove extra elements. If value is greater than current length, add null elements.
`streamLength`	Return the number of data elements in the stream

C H A P T E R

21

Memory Management

Memory management is a major source of complexity in software. In conventional programming environments, a developer must explicitly deallocate chunks of memory when they are no longer needed. ScriptX is designed to deallocate and reclaim unused memory automatically. Developers need to be aware of memory use when optimizing a title. When you make an object available for garbage collection—by dropping all references to it or calling `requestPurge` on it—the garbage collector then automatically reclaims the memory.

Memory management is built into ScriptX at the lowest level. The Memory Management component uses no classes, but it does offer several global functions. These functions extend internal hooks and diagnostic reports to the scripter level.

Memory management functions are useful for tuning performance, and for diagnosis, testing, and debugging. This chapter covers the theory and use of memory management in ScriptX. For definitions of memory management functions, see the "Global Functions" chapter of the *ScriptX Class Reference*. ScriptX provides additional memory management functions that are not intended for use in titles or tools. These functions are listed in the *ScriptX Tools Guide*.

Conceptual Overview

ScriptX allows developers to work, at least in theory, as if they had an infinite supply of memory at their disposal. This allows for a more natural style of programming. An independent process, the garbage collector, runs asynchronously in the background in its own thread while a title is running. The garbage collector reclaims memory that was assigned to objects.

Of course, no computer system has an infinite memory. As a ScriptX title runs, it continually allocates memory for objects, generally by calling the method `new` or by loading existing objects into memory from the object store. Memory is always a limited resource, in ScriptX as in any operating environment. A well-designed program makes objects available when they are needed, and allows their space in memory to be reclaimed when they are no longer necessary. ScriptX replaces memory management with load management. The concept of load management is discussed later in this chapter.

Think of the rate of memory allocation as a flow over time. The garbage collector works continuously to balance this flow. It must deallocate memory as fast as it is allocated, or the program will eventually run out of memory. If it works too hard, if it collects garbage more than necessary, it wastes processor time that could be used by other threads in the title.

Although the garbage collector is designed to work automatically, the global functions in this chapter allow you to override certain operations. At times, a title developer may want to force garbage collection, or control the amount of time that is allocated to this process.

Garbage collection has evolved in parallel with object-oriented programming. In object-oriented systems, the need for garbage collection is profound because conventional memory management techniques create complex interfaces between objects. ScriptX has a conservative, non-relocating, real-time incremental garbage collector that implements a tracing algorithm. Each of these terms has a specific meaning in computer science that is explained below.

How ScriptX Memory Management Works

This section describes the operation of the ScriptX garbage collector.

Real-time Incremental Activity

The garbage collector is called a real-time, incremental collector because it runs asynchronously with respect to other processes, in its own thread. What is meant by “real-time” and “incremental?” The garbage collector does not actually run at the same time as other processes. (This may change in the future for ScriptX in multi-processing environments.) Instead, the collector’s processing time is interleaved with that of other active threads. In that sense, we can call it an incremental collector. Each time it runs, it does a little more garbage collecting, searching through the ScriptX memory space for objects that are still in use. It completes this search over many increments, and then starts over.

What would it mean to have a garbage collector that does not operate in real time? Such a collector would wait until the system was out of memory. It would then suspend every other process, run to completion, and restore control to the system. And ScriptX will do just that as a last resort if it runs out of memory. Of course, this causes an interruption, such as a jitter in the presentation of sound, video, or animation. Such an interruption could be jarring, though certainly less jarring than allowing the system to crash.

The ScriptX global function `garbageCollect` bypasses the real-time, incremental approach. It runs the garbage collector from its current stage in the cycle to completion, reclaiming all unused memory. Title developers should be aware that the ScriptX system calls `garbageCollect` automatically if it runs out of memory, but this function can be useful at other times.

Each time it runs, the garbage collector runs for a fixed time slice, measured in milliseconds, known as its *increment*. This increment is not the same time slice, determined by the scheduler, that other threads in ScriptX receive. The garbage collector has its own fixed increment, which can be adjusted dynamically.

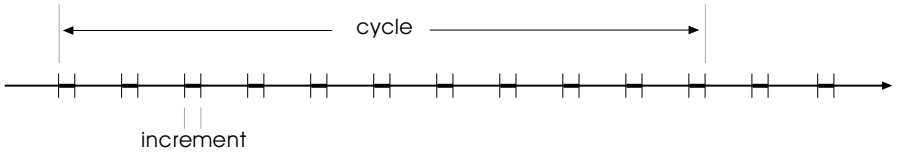


Figure 21-1: The garbage collector runs for an increment, a tiny slice of processor time, and then yields to other processes. Over many increments, it completes a cycle, tracing every object in memory.

The garbage collector's increment can be adjusted by a global function, `setGCIncrement`, which takes a single integer argument that represents time in milliseconds. This function acts, it effect, like a throttle, increasing or decreasing the time that the garbage collector runs each time it is scheduled. Although the target length of an increment is fixed by `setGCIncrement`, there are slight variations in practice, a result of differences in timing. Note that performance also depends on the number and priorities of other threads that are active, and on the speed of the central processor. Developers can use `setGCIncrement` for testing, diagnosis, and performance tuning.

Non-relocating Objects—Organization of Memory

When the Kaleida Media Player starts up, it requests one or more large blocks of contiguous memory from the native operating system. On Mac/OS, Windows 95, and OS/2, it uses a block of memory that lives in the application heap. In Windows 3.1 it is drawn from the global heap, which is shared by all application programs.

The garbage collector does not manage all of ScriptX application memory. A portion is used by the Kaleida Media Player itself to store code and system data. On some operating systems, such as the Apple Macintosh, the operating system also makes use of application memory to provide system services.

The remaining memory—that which is used to store objects—is managed by the garbage collector. We call this zone the ScriptX heap. Some objects stay around as long as the ScriptX Player is running, so the garbage collector doesn't waste time trying to reclaim their memory. These objects are stored in static memory. Other objects are temporary or transitory, used briefly and then discarded. These objects are stored in dynamic memory.

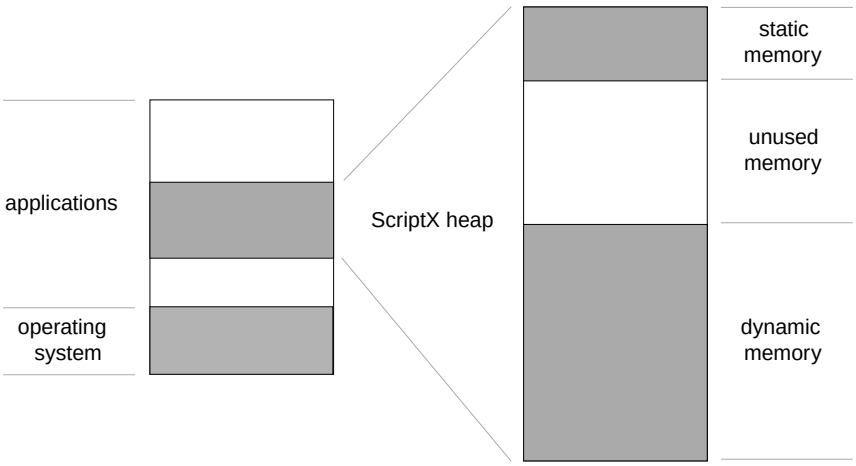


Figure 21-2: A generalized map of how ScriptX uses memory.

Memory management is handle-based. Each underlying object has a fixed master pointer that doesn't move, so the integrity of a reference is guaranteed.

Tracing Collection—How ScriptX Finds Unused Objects

On a 32-bit microcomputer, a handle is a location in memory (4 bytes) that contains an address elsewhere in memory. The garbage collector identifies memory that is not in use by tracing all possible references to objects. Any value that points to a valid address is marked as a possible reference. In this respect, the garbage collector is said to be *conservative*.

The garbage collector begins its search by checking for references to other objects in registers, stacks, and global data. Collectively, these memory locations are known as the *root set*. It will check for references to still more objects in any objects that it identifies. As it executes, the garbage collector traces and marks every possible reference in every possible tree that originates in the root set. Figure 21-3 depicts this process, showing how references that originate in the root set may contain references to other objects in memory. When it finishes its cycle, the garbage collector will have reached every possible live object in memory. Any object that has not been traced is reclaimed.

ScriptX reclaims memory each time the garbage collector completes a cycle. No memory is actually reclaimed until the garbage collector finishes tracing all possible references from the root set.

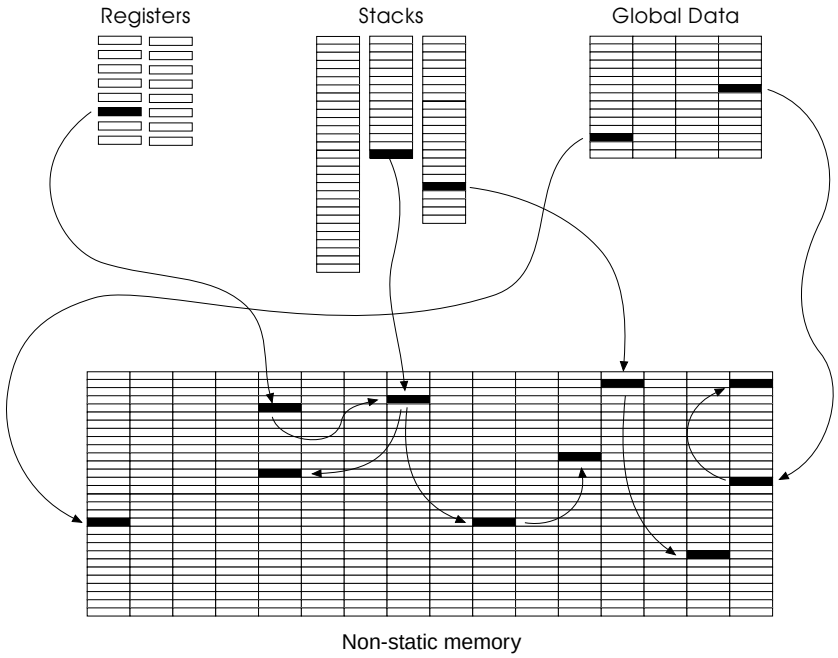


Figure 21-3: Tracing references from the root set.

The garbage collector searches some objects such as stacks conservatively. As a drawback of being conservative, the garbage collector identifies some references that are not really references. For example, a thread's stack might contain values that are easily mistaken for pointers. Although it is possible to create a pernicious program that confuses the garbage collector, this problem is of little consequence in normal programs.

ScriptX does not count references, unlike garbage collectors in some object-oriented environments. This allows the garbage collector to reclaim circular structures, groups of objects in memory that point only to each other.

Although a garbage collector allows a developer to program, at least in theory, as if memory did not matter, the use of a tracing collector has the following implications:

- Variables are references to objects. The garbage collector cannot reclaim space if there is an active reference to an object. When an object is assigned to a variable, assignment creates a reference to that object. Global variables are active for as long as a program runs. Local variables are active only while the block in which they are declared is active. If a variable is declared local, then the garbage collector is able to reclaim its memory when it is no longer needed. It is good programming practice to declare all variables as `local` or `global`, and to use local variables when at all possible.
- Variables can be unassigned, so that the objects they point to in memory can be released, allowing the memory to be reclaimed for other uses. Set the value of a global variable to `undefined` when it is no longer needed.

- Objects are often embedded in other objects. This means that setting a variable to `undefined` does not guarantee that the object it was holding onto will be garbage collected. For example, an event interest that has been added to a collection of interests is a member of a collection. If that interest is assigned to a variable, setting the value of the variable to `undefined` does not remove that event interest from the interests collection. Only if the interest is also removed from the collection can its memory be recovered.

Garbage collection is suited to an object-oriented style of programming. It is possible to write a procedural program using ScriptX. A procedural program generally ends up assigning many global variables. Even though explicit memory management is not required, a procedural program written in ScriptX requires a great deal of implicit memory management. An object-oriented style of programming takes care of many of these tasks.

For information on the scope and extent of variable assignments, see Chapter 2 of the *ScriptX Language Guide*.

Memory Management versus Load Management

In conventional programming environments, a program must explicitly allocate and deallocate every chunk of memory that it uses. A program that fails to release memory as it runs will eventually fail as it uses all available memory. Memory management requires a programmer to be an accountant.

In ScriptX, the analogous process is load management. Although a title does not keep track of every allocation it makes, it must be sure that objects are available in memory when they are needed. Since memory is a scarce resource, a title must also indicate when an object is no longer needed, so that its memory can be reclaimed. Load management requires a programmer to be a choreographer.

ScriptX performs load management through the object store. The `StorageContainer` class provides a virtual container through which groups of otherwise unrelated objects can be moved into and out of memory as a unit. `LibraryContainer` specializes `StorageContainer` to provide for startup and shutdown activities, as well as management by titles. Storage containers are collections of objects. When a storage container is opened, a program has access, by reference, to any objects it contains. Objects can be loaded into memory before they are needed by calling `load`, although this is usually unnecessary.

Multimedia titles are built around an authoring metaphor. Every authoring metaphor is constructed of logical elements, such as characters, scenes, locations, cards, or pages. By building storage containers that correspond to the elements of its authoring metaphor, a title can selectively load objects into memory and release them from memory. For example, suppose that a character is associated with a given set of bitmap images whenever the value of its `mood` instance variable is `@happy`. These objects and their associated presenters can be loaded and unloaded as a unit in that character's "happy mood" storage container.

It is possible to pre-load objects when need is imminent. For example, suppose that the happy character also suffers from arachnophobia, a morbid fear of spiders. When a spider appears on the scene, the value of `mood` must instantly change to `@panic`. In a real-time multimedia title, this change of state must be sudden, or it will not be convincing to the user. A title could preload the objects it needs for this panic state by loading all objects in the character's "panic attack" storage container whenever a spider is about to enter a scene.

A ScriptX program should release objects from memory when they are no longer needed. An object that is purgeable is kept in memory only if there is room for it. This allows a ScriptX program to make optimal use of what memory is available on a given system. For example, if a bird character in the title removes every spider from the scene, the program can call `requestPurge` on all objects in the character's "panic attack" storage container.

For more information, see the `StorageContainer` class in the *ScriptX Class Reference*.

Using Memory Management

This section describes the basic rules for allocating and deallocating memory in ScriptX. For tips and techniques on how to optimize memory use in your title, refer to the chapter "Optimizing for Speed and Memory" in the *ScriptX Tools Guide*.

How to Allocate Memory

Any time a new object is created or type is declared, either explicitly by a script or implicitly as a side effect, memory is allocated. Thus, the following script allocates memory, not only for the window itself, but for structures that the window creates implicitly, such as its boundary rectangle:

```
global w := new Window
show window
```

How to Release an Object From Memory

There are two ways to release an object from memory, depending on whether it's a persistent (stored) or transient (non-stored) object. Releasing a persistent object is called "purging" the object.

To release a stored object from memory, you need to do several things:

- Hide it if it's a presenter.
- Call `requestPurge` on it.
- Stop calling any methods on it.

To release a transient object from memory:

- Drop all references to the object. To drop references, you must also drop references that are implicit within the program. An object's memory cannot be reclaimed if another object is still using it. For example, if the object is a

presenter, hide it so that it is no longer in a presentation hierarchy. If it is an event interest, remove it from any interest lists it may be a member of. Reassign any variables that point to the object so they point to other objects, such as the `undefined` object.

For example, given the window that was created earlier, to free it you must hide it and set the variable `w` to `undefined`:

```
hide w
w := undefined
```

The window `w` will be removed from memory during the garbage collector's next cycle.

Visual Memory

Visual Memory is a tool for examining memory that is managed by the garbage collector. Choose **Visual Memory** from the **File** menu in the ScriptX menus to view a visual map of the ScriptX heap. Visual Memory is described in the *ScriptX Tools Guide*. This tool is not available in the ScriptX Player.

CHAPTER

22

Threads

Every time the ScriptX authoring environment or the ScriptX Player runs, it creates a number of system threads, which run concurrently. Many titles and tools create additional threads. Threads provide a mechanism for managing interaction with the user and presenting media from multiple sources. A title might play a tune, run an animation, search a database, and respond to a mouse event, all at the same time. Each of these processes can be managed by a different thread.

By using threads, a script can manage multiple events and behaviors modularly, rather than through modifications to a main program loop. The ScriptX runtime environment includes a scheduler, which runs invisibly. The scheduler is responsible for allocating time to the various threads.

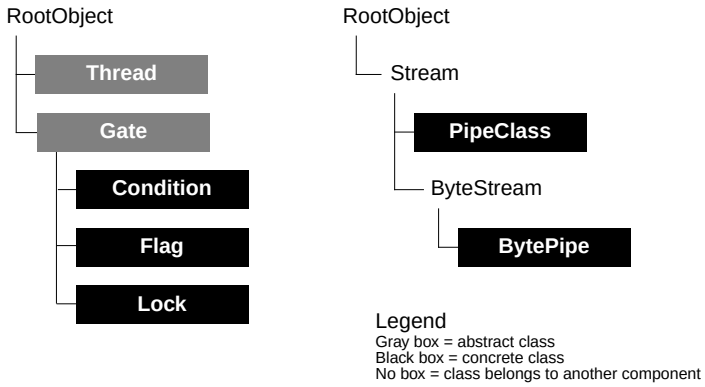
The Threads component includes several classes that are used as communication and control mechanisms. Pipes are structures through which data can be passed between threads. Gates are controls that allow a thread to suspend or block while waiting for an event, for control of a resource, or for another thread to finish some critical task.

A ScriptX title or tool runs using the services of the system threads, and it can create others as it runs. Some of these threads, such as the garbage collector, run asynchronously and invisibly. Others are created automatically to provide system services for objects in other ScriptX components. For example, the two callback threads serve clocks, players, and the compositor. Thus, a developer can create a multi-threaded title without explicitly creating threads.

Although it is possible to develop a title without explicitly creating a thread, the Threads component provides a rich set of tools that developers can use to improve the flow of control in their titles. This chapter is recommended for developers who need to create their own threads, pipes, and gates.

Classes and Inheritance

The class inheritance hierarchy for the Threads component is shown in the following figure.



The following classes form the Threads component. In this list, indentation indicates inheritance. Note that the `Thread` and `Gate` classes, with their subclasses, are sealed. A sealed class is a class that cannot be specialized or subclassed.

`Thread` – an independent thread of execution.

`Gate` – represents any obstacle to the execution of a thread; when a thread is unable to acquire a gate, it blocks until the gate is opened (relinquished).

- `Condition` – represents a condition that becomes instantaneously true and then false again.
- `Flag` – represents a condition or state that remains true over a period of time.
- `Lock` – represents a resource that only one thread can own at any time.

`BytePipe` – a convenient way for threads to pass chunks of data around.

`PipeClass` – a convenient way for threads to pass objects around.

Conceptual Overview

A thread represents an independent process or flow of control in a program. Each thread is associated with a control function, which it calls when it begins running. This control function, specified by the `func` instance variable in the `Thread` class, is a series of expressions which run until the thread yields, finishes executing, or is stopped by the scheduler.

When you create a thread, it is immediately active unless you specify otherwise, but it won't actually begin running until it is scheduled to run, as determined by its `priority`. Technically, only one thread actually runs at a time on current platforms. The current thread is indicated by the global variable `theRunningThread`.

Threads take turns running. A thread's priority determines its share of processing time. Of course, the ScriptX Player manages threads of its own, some of them at system priority, including threads that control such critical processes as clocks and garbage collection (memory management).

When a title starts up, it begins running from the thread indicated by the global variable `theMainThread`. On Macintosh and Windows versions of the ScriptX Player, this thread contains the main event loop. Although a title begins running in the main thread, it may use other threads. For example, a callback runs in a separate thread. The main thread remains in existence, though not necessarily active, for as long as ScriptX is running. To destroy the main thread is to quit the runtime environment.

Note – In current versions of ScriptX, the Kaleida Media Player runs with the services of several system threads. For a list of these threads, run the script, `for x in (allInstances Thread) do print x`. Future versions of ScriptX may not use the same set of system threads. For this reason, ScriptX code should never depend on any property (such as `status` or `priority`) of any system thread. These `Thread` properties are visible to the scripter for testing and debugging, but should not be used in actual tools and titles. For more information, see the `Thread` class in the *ScriptX Class Reference*.

Threads can be created and destroyed at will. The main thread has no special control or authority over other threads. In ScriptX, no central authority is responsible for creation and destruction of threads, and no component manages control mechanisms such as gates and pipes. The ScriptX thread system is really the collective behavior of the thread component classes as a whole. Threads run independently—except for the scheduler, they manage themselves.

Terminology

block – to suspend activity while waiting on a gate. When a thread blocks, the gate acts as an external barrier. When the barrier is removed, the thread can run again.

gate – a control mechanism that represents a global state in a program, used to synchronize activities between threads.

pipe – a data structure that acts as a conduit for exchanging information between threads.

preemptible – interruptable by the scheduler. A thread that is not preemptible yields only voluntarily.

thread – an independent process that runs concurrently with other processes.

How Threads Work

A script can create many threads, which run asynchronously, each one running its own function. The script can manage under what conditions each thread becomes active using gates.

When threads need to wait for some event or state, they suspend themselves. Within a thread—that is, within its control function—a number of functions and methods can be called to make the thread suspend until an event such as a

mouse click or a clock tick. (See sample scripts in the Events chapter of *ScriptX Language Guide*.) Thus, the Thread component is intimately tied with other components, including Events, Clocks, Players, and Controllers.

Much of the discussion in this chapter is really about communication and synchronization between threads. This communication occurs, either directly or indirectly, through gates and pipes. Gates and pipes make it possible to use ScriptX to create very complex simulations. Gates and pipes are also built into or embodied in other ScriptX classes. For example, `EventQueue` (see Events chapter of *ScriptX Language Guide*) is a subclass of `PipeClass`, while `CallBack` (see Clocks chapter of *ScriptX Language Guide*) uses both threads and gates internally.

Programming Guidelines

This chapter places strong emphasis on programming guidelines. ScriptX is permissive. It gives you plenty of rope; you can hang yourself if you want to.

Developers must always be aware, in using the Threads component, of the potential for conflicts between threads. Multi-threaded titles can be very difficult to modify and debug if they are not designed carefully. They have a potential for several kinds of errors that do not exist in programs that execute in a single thread. The following is a list of common execution problems in multi-threaded systems.

1. **Stack overflow.** Every thread runs with a stack. A script determines how much stack space is allocated for a thread when the thread is instantiated. The default size is dependent on the hardware platform. ScriptX has a stack sniffer to detect overflow conditions. Graceful recovery from a stack overflow is not always possible. Most overflows result from too much recursion, often because of a circular reference. Stack overflows are common in single-threaded systems too.

2. **Resource conflict.** What happens when two threads both require the same resources? Threads may compete for access to a peripheral device, for a lock on a global variable, for a set of locks to enable an operation.

   a. **Deadlock.** If a thread is unable to run because another thread has claimed a resource that it needs, the conflict is called a *race condition*. If neither thread is able to run, the conflict is called deadly embrace or deadlock. Multi-threaded titles should be designed to resolve deadlocks.

   b. **Concurrency.** In multi-threaded titles, two or more threads must often share a resource, such as a global variable. What happens when one process is preempted at a critical point in time, while it is working with a local copy of that global variable? Developers need to use pipes and gates in a multi-threaded title to avoid concurrency problems. If a group of expressions must execute as a unit, a script should temporarily suspend preemption using the global functions `threadCriticalUp` and `threadCriticalDown`.

3. **Starvation and thrashing.** Every computer system has at least two limited resources: execution time and memory. From time and memory, an operating system defines other resources, which can also be in short supply.

For example, Microsoft Windows has a limited supply of file handles; the Macintosh, a limited supply of window pointers. Peripherals have a restricted pathway for moving data in and out of memory. If a process is unable to obtain a resource, it is said to be starved. If an entire system makes impossible demands on resources, it is said to thrash. ScriptX has the potential to thrash or starve the system if a title creates too many threads.

Thread Functions

A thread's control function is called just once, when the thread begins running. Two instance variables on the `Thread` class, `func` and `arg`, store the thread's function and its argument. As the thread runs, this function may call other functions and methods, or it may call itself recursively. A thread is an independent process that runs asynchronously with other processes, so it maintains its own stack. Thus, it is able to stop and start.

Some thread functions finish execution and return a value, or yield voluntarily before their time slice ends. A thread that has finished execution can be restarted and run again. Often, a thread's function is in the form of an infinite loop, so that it calls a series of expressions continuously, never running to completion.

A `Thread` object can return a value in one of three ways. The global function `threadExit` is the usual technique for exiting from the running thread. This function is equivalent to calling `threadReturn` (an instance method of `Thread`) on the running thread. If the thread is running in the top level of its control function, the ScriptX `return` expression and the method `threadReturn` are equivalent.

The return value can be any object—often it is a Boolean value or `undefined`. For example, it may be the result of a database search that was processed by the thread, in the background. This value remains available in the thread's `result` instance variable until the next time the thread returns a value.

When a thread is finished executing, the value of its `status` instance variable is `@done`. The thread will remain in existence as long as there is an active reference to it, such as by assignment to a global variable. A thread that has finished executing can be restarted by another thread. If there are no active references to a thread, the garbage collector eliminates it and reclaims its memory.

Thread Status

A thread will always be in one and only one of the following states, as given by the `status` instance variable defined by the `Thread` class:

Table 22-1: Thread status in ScriptX

Status	Meaning
`@starting`	the thread is in the process of starting for the first time
`@active`	the thread is runnable, but not running
`@inactive`	the thread is not runnable, but it isn't waiting on anything

Table 22-1: Thread status in ScriptX

Status	Meaning
`@running`	the thread is currently executing
`@waiting`	the thread is not runnable; it is waiting on a gate
`@done`	the thread is finished executing
`@killed`	the thread was killed
`@restart`	the thread is in the process of restarting

It is helpful to think of the eight possible values of the instance variable `status` defined by `Thread` as states through which a thread passes as a program is running. Figure 22-1 illustrates how the status of a thread changes with method and function calls. (The diagram depicts only calls that constitute good programming practice in a multi-threaded environment.)

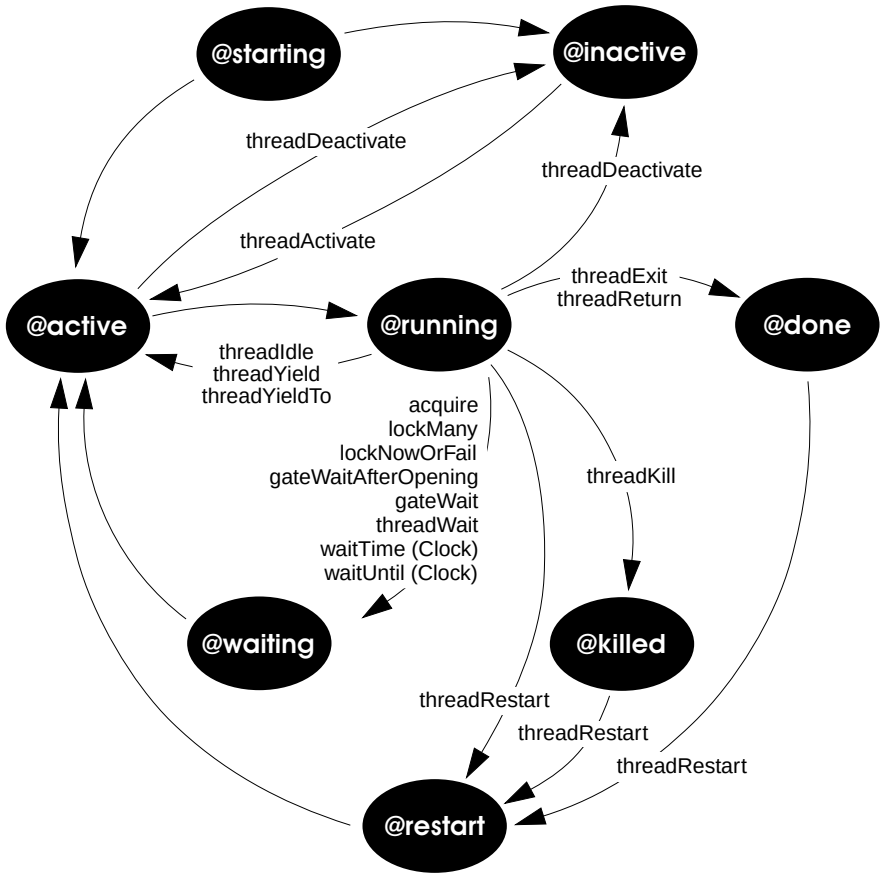


Figure 22-1: States of a Thread object.

A script cannot directly set the `status` instance variable on a thread. In some cases, a function or method call explicitly determines a thread's status. In other situations, status can only be changed indirectly. For example, if a thread is waiting on a gate, the only way to change its status is to open the gate. Of course, the gate may be under the control of another thread. Once the gate is

opened, the thread becomes active, but only the scheduler can make an active thread actually run. The following list is a more detailed summary of each of the eight states.

1. **Starting.** If you create a thread, its status has the initial value `@starting`. This status is transitory, for initialization only—you will rarely observe a thread in this state. While a thread is starting, it is allocated memory, including a stack in which to run. Its `result` instance variable is initially set to `undefined`. After a `Thread` object is started, it becomes immediately active, unless you set the value for the keyword argument `startInactive` to `true` when you create the thread.

2. **Active.** A thread's status must be `@active` before it can run. An active thread has a stack, and it may have current references to other objects. The `result` instance variable, defined on the `Thread` class, remains `undefined` until it is explicitly changed, usually by the thread itself when it exits and returns a value. If the thread has been restarted, it retains the result from its previous execution until the next time it returns a value. Any number of threads can be active, limited only by available memory. Keep in mind that when an active thread actually runs, it may be preempted at almost any stage of execution. Thus, an active function may be virtually anywhere in its function. Never assume that preemption will occur at any particular point in the execution cycle. If the logical flow of control does not allow for preemption at some point, use the global functions `threadCriticalUp` and `threadCriticalDown`.

3. **Inactive.** An active thread's status can be made `@inactive` at any time, or a running thread may make itself inactive. A thread that is inactive retains its stack and all of its properties, but it cannot actually run until it has been made active again. Think of the stack, and any references to objects, as being frozen until the thread is made active again. Any number of threads can be inactive, limited only by available memory.

4. **Running.** Only the scheduler can change a thread's status to `@running`. On current versions of the ScriptX Player, only one thread is actually running at any point in time. A thread's status must be active before it runs. Each time a thread runs, it executes its function from where it left off the last time it ran. The script determines how often an active thread runs relative to other threads by using the `priority` instance variable, but a script should never assume that active threads will run in any particular order. If complex synchronization is needed, it can be achieved using gates and pipes. If a thread is preemptible, it can be interrupted at any stage while it is running. When the thread is preempted, its status becomes active, and remains so until the scheduler runs the thread again (or until the status of the thread is explicitly changed by another thread). The running thread can always refer to itself using the global variable `theRunningThread`. As a rule, a running thread manages itself. Thus, a running thread can move itself into any other status (except `@starting`, which is of no real consequence). A running thread can yield to any active thread by calling the global function `threadYieldTo`.

5. **Waiting.** If a thread is blocked, then its status is `@waiting`. A thread is said to block when it is stopped by a barrier to execution, such as a gate or pipe. The waiting thread is quite similar to an inactive thread in that its stack and

its properties are frozen, suspended in time. However, there is no explicit function or method call on the thread itself that will make a waiting thread active. The only way to activate a waiting thread is to unblock it by removing the barrier on which it is blocked. This barrier could be a gate, or the thread could be trying to read from an empty pipe, or write to a pipe that is full. When the thread becomes unblocked, it becomes active again, but it does not actually execute until its turn comes up in the scheduler. In other words, a thread that has been unblocked does not necessarily run immediately; its turn may be scheduled before or after other active threads.

6. **Done.** If a thread exits or returns, then its status is `@done`. When the thread exits or returns, it throws away its stack. Thus, the thread whose status is `@done` can only be restarted. When it is restarted, it will retain its properties, but it will start over again with a new stack. The return value from the thread's function is put in its `result` instance variable. This value can be any object. Keep in mind that ScriptX does not reinitialize this variable if the thread is restarted. A thread that is done will persist for as long as there is a reference to it elsewhere. For example, if a thread is assigned to a global variable, that constitutes an active reference. This reference can be cleared by assigning the value `undefined` to the variable. When there is no longer a reference to a thread, the garbage collector will remove it and recover any memory it was using.

7. **Killed.** A thread that has the status `@killed` is similar to a thread whose status is `@done`, except that it has no new return value. (Actually, the thread may still have a return value left over from a previous execution.) The thread can still be restarted, and it will persist until there is no longer a reference to it.

8. **Restart.** If you restart a thread, its `status` is briefly `@restart` before it becomes active. Like starting, restart is a transitory state that is rarely observed. These two states are almost identical; in a future version of ScriptX, they may even be merged. Restarting is equivalent to starting in that the thread starts over with a new stack. When it runs again, the thread will begin execution at the beginning of its controlling function. However, the `result` instance variable is not reinitialized; it may retain a value left over from a previous execution.

Think of the state-flow diagram above as representing good programming practice. ScriptX allows you to call many functions and methods that are outside the scope of this diagram, operations that would be illegal in many other programming environments. For example, a developer can call `threadReturn`, `threadKill`, and `threadRestart` on a thread that is not running. If these operations are used incorrectly, they are an invitation to deadlock, data corruption, and disaster.

A good rule of thumb for programmers is that threads should manage themselves. Of course, a thread must be started or activated by another thread. But once a thread is active, it should be allowed to finish. If another thread kills or restarts a thread that is active, it may throw away the only reference to a global variable, or leave data in an inconsistent state. If a thread has to be killed, build logic into it so that it knows how and when to kill itself.

ScriptX will let you live dangerously. If you would rather not live on the edge, follow these guidelines when using threads.

- Never kill, exit, or return from a thread that is not running. These operations throw away the thread's stack, and they can eliminate references to other objects.
- Never restart another thread unless it has finished execution.
- The only method you should call on a thread that is active, but not running, is `threadDeactivate`. The only method that you should call on an inactive thread is `threadActivate`.

Blocking

A thread that blocks is in suspended animation. It retains its stack, but it is unable to run until it becomes active again. It can be activated only by unblocking it, by opening the gate that it is waiting on. This gate may be one that is explicitly defined by a script, or one that is defined internally by another ScriptX class, such as `CallBack`, `EventQueue`, `PipeClass`, or `BytePipe`.

Think of blocking as an alternative to *polling*. In single-threaded operating systems, an application waits for events by polling. Each time an application runs through its main program loop, it polls the operating system to see if any new events have been generated. In the Mac/OS operating system, applications poll for events by calling the functions `getNextEvent` or `waitNextEvent`. In the Windows environment, the equivalent functions are `GetMessage` and `PeekMessage`. (In OS/2, which features fully preemptive multitasking, a process sleeps until a message is delivered to it.)

To create a more efficient program, use blocking instead of polling. Polling wastes execution time because the process has to keep asking the system for new messages or events. Blocking means that a process sleeps until it is awakened externally.

There are several situations in ScriptX where you do not want to use a method or function that can cause a thread to block.

1. Instances of `Callback`, defined in the Clocks component, are scheduled by one of two threads that the system defines, the system callback thread and the user callback thread. If an operation that is initiated by a callback blocks, every other callback that is scheduled in the same thread is unable to run.
2. Instances of `ScriptAction` and `TargetListAction`, defined in the Animation component, should not schedule any operations that might block a thread. If a function that was scheduled on an action list blocks, it suspends every action that follows.
3. In subclasses of `Controller` and `TwoDPresenter`, the `tickle` method must not block.
4. In subclasses of `TwoDPresenter`, the `draw` method must not block.

Table 22-2 lists operations that can cause a thread to block.

Table 22-2: Methods and functions on which a thread can block.

Operation	Type
`closeStorageContainer`	global function
`fastForward`	method on `Player`, blocks with MIDI/Interleaved Stream Player
`gateWait`	global function
`gateWaitAfterOpening`	global function
`goToBegin`	method on `Player`, blocks with MIDI/Interleaved Stream Player
`goToEnd`	method on `Player`, blocks with MIDI/Interleaved Stream Player
`loadAllObjects`	method on `StorageContainer`
`lockMany`	global function
`lockNowOrFail`	global function
`openStorageContainer`	global function
`pause`	method on `Player`, blocks with MIDI/Interleaved Stream Player
`play`	method on `Player`, blocks with MIDI/Interleaved Stream Player
`playUntil`	method on `Player`, blocks with MIDI/Interleaved Stream Player
`read`	method on `Stream`, can block with a pipe
`restoreLockState`	global function
`rewind`	method on `Player`, blocks with MIDI/Interleaved Stream Player
`signal`	method on `Event`, blocks if `rejectable` is `true` and event receiver is an event queue
`stop`	method on `Player`, blocks with MIDI/Interleaved Stream Player
`storeAllObjects`	method on `StorageContainer`
`threadIdle`	global function
`threadInterrupt`	method on `Thread`
`threadProtect`	method on `Thread`
`threadRestart`	method on `Thread`
`threadReturn`	method on `Thread`
`threadUnProtect`	method on `Thread`
`threadWait`	method on `Thread`
`threadYield`	method on `Thread` or global function
`threadYieldTo`	method on `Thread` or global function
`waitTime`	method on `Clock`, creates a callback
`waitUntil`	method on `Clock`, creates a callback
`write`	method on `Stream`, can block with a pipe

The following list gives a heuristic explanation of why some of these operations can fail.

1. When an event is delivered synchronously, when it is rejectable and the event receiver is an event queue, the thread that delivered the event must block while waiting for a response.

2. Both the MIDI and interleaved stream players create two threads to read from one stream of data. The activities of these threads are synchronized by a lock. Note also that players are often synchronized in a clock hierarchy, so that a call to `goToBegin` on a master clock could block another player indirectly.

3. Loading an object into memory, or saving an object that is already in memory, can block a thread. If it is necessary to refer to an object in memory, check that it is available first, using the global function `isInMemory`.

Note – Methods and functions from the list above may block intermittently. In some circumstances, a given operation may be completed without any thread having to block. In other circumstances, perhaps with a different processor or memory configuration, the same operation may fail. An operation that creates only a jitter on one system may create a jarring interruption on another. Avoid using these operations with callbacks and script actions even when they *seem* to work.

Polling of input devices can only be done from within threads that can block. For example, a process that continuously monitors and reacts to the position of the mouse should not start running within a callback thread. Normally, a program should spawn a new thread to run such a process.

Pipes

A pipe is a conduit of data that is used to exchange information between threads. Think of a pipe as a first-in, first-out queue. Any threads that use the pipe will always write to one end and read from the other. ScriptX provides two classes of pipe. Objects of the `BytePipe` class are used to pass chunks of data. Objects of the `PipeClass` class are used to pass objects.

Pipes also act as a control mechanism. Threads often block while waiting for some external event or object, such as a mouse click or a piece of information. Although this logic could be coded into the thread itself, without the use of pipes, pipes provide a built-in mechanism that is both efficient for the system and convenient for the title developer.

Pipes allow the scheduler to run a thread only when it has data to run. If a thread is waiting to read from a pipe, its `status` instance variable will be set automatically to `@waiting` until the pipe is no longer empty.

In Figure 22-2, two threads are set up to communicate through a pipe. The writing thread has a function, written as a loop, that calls `write` on the pipe, putting objects into the pipe. The reading thread has a function, written as a loop, that calls `read` on the pipe.

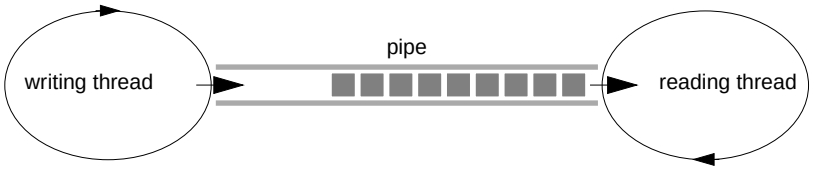


Figure 22-2: The reading thread runs as its function processes data from the pipe. The writing thread is also runnable.

As long as the pipe is not broken, if there is nothing for the reading thread to read, it blocks and will wait for the next item, as shown in Figure 22-3. In this way, the writing thread can continue sending objects to the reading thread without any exceptions being reported.

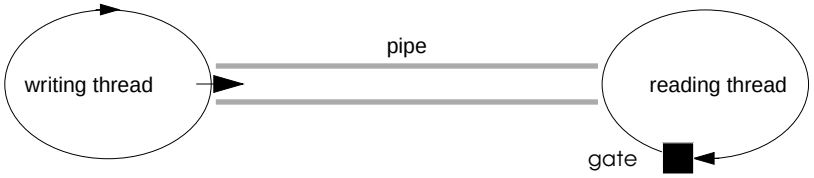


Figure 22-3: The reading thread runs out of data and it blocks. The writing thread is still runnable, and it continues to fill the pipe each time it runs.

When a thread blocks while reading from or writing to a pipe, as in Figure 22-4, it is waiting on a gate. Pipes use gates internally. For information on gates, see the next section of this chapter.

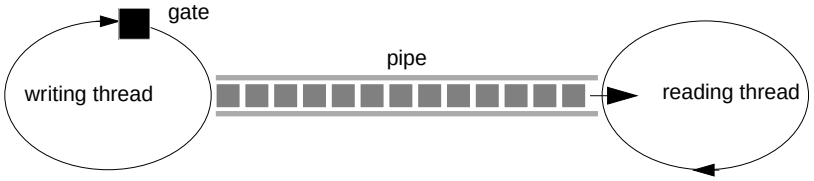


Figure 22-4: The pipe is full, so the writing thread is blocked. The reading thread is runnable and it continues to empty the pipe each time it runs.

A pipe can be broken, as in Figure 22-5, by calling its `breakPipe` method. Once the pipe is broken, a writing thread can no longer write to it. If any thread attempts to write to the pipe, the pipe will report an exception.

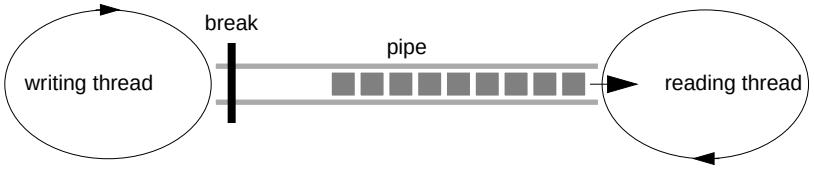


Figure 22-5: The pipe is broken, so the writing thread cannot write to it. The reading thread is runnable until the pipe is empty.

The reading thread can continue to read from the pipe until it is empty. If the reading thread attempts to read from a pipe that is empty and broken, it will report an exception. Note that the writing thread itself is not blocked. Although it cannot write to a broken pipe, it could write to some other pipe.

Events and threads interact through pipes via the `EventQueue` class, which is a subclass of `PipeClass`. An event queue is like an electronic mailbox, used to notify a thread of an action elsewhere in the system to which the thread must respond. A thread blocks while waiting for an event to pop through an event queue.

Gates

Gates represent global states in a ScriptX program. Gates are controls that allow a thread to block while waiting for an event, for control of a resource, or for another thread to finish some critical task. Many ScriptX titles, even those with multiple threads, do not require a developer to explicitly define gates. Other ScriptX components, including Clocks, Players, Events, and User Interface Objects, make use of gates internally. For example, callbacks in the Clocks component can be used to trigger very complex animation sequences.

Gates are useful for the title developer who requires specialized access to the Threads component, perhaps to create simulations that require synchronization of several concurrent processes.

The `Gate` class and its subclasses are sealed. This means that it is impossible to add instance variables and methods to the gate classes, or to create subclasses. To attach a gate to some other object, add the gate as a specialized instance variable of that object, or create a new subclass with the gate as an instance variable.

The `Gate` class itself is abstract; gates are instantiated as `Lock`, `Condition`, or `Flag` objects. Threads interact strongly with gates, and pipes make use of gates internally. Threads may suspend themselves based on the state of gates that they use.

Table 22-3: Gates in ScriptX

Gate class	Possible states	Applications
`Lock`	`@open`, `@closed` the `thread` instance variable stores the lock's current owner.	represents a resource or operation that only one thread can own at once.
`Condition`	`@open`, `@closed`	represents a transitory state that becomes instantaneously true and then false again.
`Flag`	`@open`, `@closed`	represents a state that holds over a period of time

Locks

The `Lock` class represents a resource that only one thread is permitted to work with at a time. A thread must `acquire` a lock, which it can do only if any previous owner that acquired the lock has called `relinquish` on the lock. A thread may acquire a lock any number of times. When it relinquishes the lock, each call to `acquire` must be balanced by a call to `relinquish`.

ScriptX provides the global function `lockMany` to prevent resource conflicts. In the following sample script, Socrates and Plato are sharing a dinner at the First Circle restaurant, but they only have one pair of chopsticks. Each chopstick is attached to a lock. If Socrates and Plato try to acquire the chopsticks in a different order from each other, they deadlock. (In the classic "Dining Philosophers" example, Socrates and Plato are sitting at a round table, and they each attempt to acquire the chopstick on their left.) The solution is to use the function `lockMany` to acquire both chopsticks, allowing the two philosophers to share both utensils and food.

```
c1 := new Lock label:@chopstick_left
c2 := new Lock label:@chopstick_right
function fresser who -> (
       repeat while true do (
           lockMany c1 c2
           format debug "%* is eating now.\n" who empty
           relinquish c1; relinquish c2
           threadYield()
       )
)
thread1 := callInThread fresser "Socrates" @user
thread2 := callInThread fresser "Plato" @user
```

One common application of locks is to protect the integrity of data. Without an explicit protection mechanism, it is possible for two threads to modify a set of data at the same time. For an example of attaching a lock to a collection, see "Collections and Threads" on page 462 in Chapter 16, "Collections."

Conditions

The `Condition` class represents a global state that is temporary or transitory, a state that becomes instantaneously true and then false again. Use a condition to coordinate or synchronize two or more threads. A thread waits on a condition with either the `acquire` method or the `gateWait` function. When the condition becomes open, any threads that are waiting on it are made active. In contrast with a lock, which can only be owned by a single thread, multiple threads can be made active when a condition is opened.

Conditions are like automatic doors—they close immediately after they have opened. Suppose two threads are waiting on a condition at once. When that condition becomes `@open`, both become active. The condition is then automatically reset to `@closed`. Any subsequent threads that try to `acquire` the condition must wait until it is again `@open`.

In this code sample, the main thread creates and manages a condition called door. Dick, Jane, and Sally are threads that each attempt to `acquire` the condition. Since door is a condition, it remains open only momentarily, until waiting threads have been made active.

```
door := new Condition label:@automatic
function user who -> (
    acquire door -- wait on the gate
    format debug "%* is out the door\n" who empty
)
thread1 := callInThread user "Dick" @user
thread2 := callInThread user "Jane" @user
```

Before opening the door, check the status of these two threads.

```
thread1.status
⇨  @waiting -- Dick is waiting to get through the door
thread2.status
⇨  @waiting -- Jane is waiting to get through the door
```

Now, open the door.

```
relinquish door -- relinquish momentarily opens the door
⇨  "Dick" is out the door
⇨  "Jane" is out the door
```

If you create another thread, the gate is already closed.

```
thread3 := callInThread user "Sally" @user
thread3.status
⇨  @waiting -- Sally is still waiting! The door is shut.
```

Sally's thread is not created until after the main thread has relinquished the door. Dick and Jane make it through the door because they were already waiting when it opened. Sally must wait until the door is again opened.

Flags

The `Flag` class is similar to `Condition`, except that it represents a state which remains true over a period of time. Whereas a condition closes automatically, a flag remains open until it is explicitly closed.

Flags are like traffic lights. Suppose two threads are waiting on a flag. When that flag becomes `@open`, both threads become active. Since the flag remains open until it is explicitly closed, any subsequent threads that attempt to `acquire` the flag remain active.

In the following code sample, Calvin and Hobbes both make it through the light. And so does Suzy, even though she was not waiting when the light changed.

```
light := new Flag label:@trafficLight
function redLight -> light.state := @closed
```

```
function greenLight -> light.state := @open
function driver who -> (
    acquire light
    format debug "%* made it through\n" who empty
)
redLight() -- the flag is closed
car1 := callInThread driver "Calvin" @user -- Calvin thread
car2 := callInThread driver "Hobbes" @user -- Hobbes thread
```

At this point, both the Calvin and Hobbes threads are waiting on the traffic light. Open the gate by calling `greenLight`.

```
greenLight() -- the flag is open, and remains open
⇨  "Calvin" made it through
⇨  "Hobbes" made it through
```

Create another thread that runs the `driver` function. The flag is still open, so this driver will make it through also. A flag remains open until it is explicitly closed.

```
car3 := callInThread driver "Suzy" @user -- Suzy thread
⇨  "Suzy" made it through
```

Priority

Threads run for fixed time slices, as determined by the scheduler. The scheduler is a neutral arbiter. Its role is to apportion fixed time slices to active threads, based on their priority. If a thread yields or blocks, perhaps to wait on a pipe or gate, the scheduler passes control to another thread, which begins to run with a new full time slice. The length of a time slice varies with the underlying hardware platform.

How often a thread is scheduled to run—what precedence it has in the scheduler—is determined by its `priority` instance variable. All threads have a priority level, and this level can be dynamically changed. The two priority levels are `@user` and `@system`. Within the `@user` priority level, runnable threads are executed in round robin fashion. A newly created thread at system priority runs immediately—in effect, the `@system` priority level is an interrupt priority. User priority threads run less often than system priority threads, but they are not starved.

If you create a thread by calling the `new` method on one of the thread classes, its priority will be set to `@user` by default. When the main thread in a title is initialized, its priority is also `@user` by default, but the developer is free to change its priority level. The global function `callInThread` requires the script to pass a priority level as its third argument.

Title developers should be aware that many critical system tasks run in their own threads. Some of these threads, such as those that manage the main clock and elements of the user interface, must run at a system priority level. ScriptX gives the developer complete freedom to manage thread priorities.

To keep a title running smoothly, it is a good idea to use system priority threads sparingly, and only for tasks that will quickly block or run to completion. Attempting to run too many threads at system priority will "thrash" the system. (Too many threads at *any* priority level can thrash the system.)

For example, ScriptX has a garbage collector, a process that automatically deallocates memory when an object is no longer in use. The ScriptX garbage collector runs in its own thread, which runs at `@system` priority in the current version of the system. If a title developer creates a large number of threads with that run at system priority, these threads could conceivably exhaust all available memory before the garbage collector has a chance to run.

To assure that a multi-threaded title runs smoothly follow these guidelines:

- Use the `@system` priority level only for threads that perform critical tasks, and that finish executing or block quickly. If possible, leave the `@system` priority level to the ScriptX Player.
- An excellent strategy for creating a system that is both responsive to the user and conservative in its use of processing cycles it to define a single system-priority thread that manages the title's interface. It can communicate with other threads that are set to run at user priority.

Preemptibility

A thread that is preemptible can be forced to yield to another thread before it runs to completion. A thread that has been preempted returns to the list of active threads until its next turn to run. A thread that is not preemptible has no fixed time slice—it runs until it blocks or yields control to another thread voluntarily, perhaps by exiting and returning a value.

The scheduler is not concerned with preemptibility when it schedules an active thread to run—it runs threads based on their priority. Preemptibility determines whether a running thread can be interrupted if it has not returned or blocked. If the thread is preemptible, it will be interrupted at the end of a regular time slice to allow some other thread to run.

Developers need to be concerned with preemptibility at two different levels. First of all, a thread as a whole can be nonpreemptible, insuring that it will never be interrupted. Secondly, a developer might want to make a block of expressions within a function be non-preemptible, even though the thread as a whole is preemptible, to insure that the scheduler does not interrupt a critical operation.

A thread's preemptibility is set on instantiation. `Thread` objects have three preemptibility levels. The global function `callInThread`, which is a convenient shorthand for calling `new` on the `Thread` class, creates a thread that is fully preemptible.

Table 22-4: Preemptibility levels of threads

Level	Implementation	Meaning
`@nonPreemptible`	all systems	yields only cooperatively
`@genericPreemptible`	all systems	may be preempted during generic function dispatches
`@fullyPreemptible`	not guaranteed	may be preempted at any time

When does a thread need to be non-preemptible? In general, a task that must always run to completion each time it runs should never be preempted. Commonly, a thread calls certain functions or executes blocks of expressions that must run atomically—which is to say, they must run without interruption. Preemption can be prevented by using the global functions `threadCriticalUp` and `threadCriticalDown`.

Full preemptibility is not guaranteed. When the ScriptX Player runs on an operating system that does not support preemptive multitasking, a thread that is fully preemptible is the same as generic preemptible. At the scripter level, the full and generic preemptibility are equivalent. Macintosh System 7.5 and Microsoft Windows 3.1 do not support preemptive multitasking. Both Apple Computer and Microsoft have announced that future releases of their operating systems will offer full preemptive multitasking.

Developers should use caution in creating threads that are not preemptible. If possible, tasks should be divided to create processes that run to completion in a short time slice. Threads that are not preemptible can potentially starve other threads, including threads that are managed by the ScriptX Player itself.

And what a potential for conflict there is! Consider the garbage collector, which runs in its own thread at a system priority level. Since the garbage collector runs in its own thread, it cannot recover memory while another thread is running. If a thread is `@nonPreemptible`, it could conceivably run for a very long time, consuming all available memory.

Why would a title developer want to create a thread that is non-preemptible? The answer is that preemption can potentially create other conflicts. Imagine a thread function that must relinquish a series of gates while conducting a sequence of complex and state-dependent operations. Under normal circumstances, this thread runs until it blocks, so that the potential problem is never detected. What happens when the program runs with a slower processor, or with unusual memory constraints? If the thread is interrupted in the wrong place, it might create a deadlock. This kind of bug is treacherous because it fails to show up repeatedly, making debugging very difficult.

A thread that is preemptible can be interrupted virtually anywhere. If the thread manages a process that is visible to the user, it might create a jarring interruption, much like suspending a conversation in the middle of a sentence. If the interruption occurs within a block of expressions that should normally be executed together, the result can be inconsistent behavior.

In general, you should use a lock to delineate a critical section of code. Use `threadCriticalUp` and `threadCriticalDown` only if no other thread can execute while your code is running. Although such situations are common in systems programming, they should be very rare in titles.

The following code example shows how to use a lock to assure that a segment of code is executed atomically. In this example, `users` is a closure function that keeps track of the number of users. (For information about closures, see the *ScriptX Language Guide*.) Assuming that `logout` and `login` can execute in parallel, you want to protect them with a critical section so that `login` doesn't add a new user while `logout` is in the middle of testing whether `users` is equal to zero.

```
-- keepTrack is a function that returns a closure
function keepTrack y -> (local x:0; (y -> x := x + y))
global users := keepTrack 0 -- creates a closure
global busy:false

fn logout -> (
    -- we don't want to swap after the test but before the value of
    -- busy is set to false
    if (users -1) = 0 do busy := false
)

fn login -> (
    busy := true
    users 1
)
```

You could accomplish this task with `threadCriticalUp` and `threadCriticalDown`, but that would be overkill, and it would be very bad for performance, since it would lock out other threads that have no interest in `login` and `logout`. The solution is to use a lock.

```
critical := new Lock
fn logout -> (
    acquire critical
    if (users -1) = 0 do busy := false
    relinquish critical
)
fn login -> (
    acquire critical
    busy := true
    users 1
    relinquish critical
)
```

In summary, to assure that a multi-threaded title runs smoothly follow these guidelines:

- Always set the value of `preemptibility` to `@fullyPreemptible` unless there is a good reason not to.

- Determine which blocks of expressions must execute atomically, and which blocks cannot run while any other thread is running. Use `threadCriticalUp` and `threadCriticalDown` to prevent the scheduler from preempting the running thread while these expressions are executing. Allow the thread to be preempted whereever possible.
- Use `threadCriticalUp` and `threadCriticalDown` only as a last resort. They can often be avoided by using gates as a synchronization device. The global functions `lockMany` and `gateWaitAfterOpening` can be used to execute code atomically where there are multiple locks.

Note – In the future, your ScriptX title or tool may run on systems that use more than one processor. When the ScriptX Player is implemented to run with multiple processors, a nonpreemptible thread will suspend other processors every time it runs, as will a call to the global function `threadCriticalUp`. To assure that a title runs smoothly on future systems, use only threads that are fully preemptible.

Protection

Protection does not actually affect scheduling; rather, it protects a thread from operations, initiated by other threads, that could be destructive. When its protection level is non-zero, certain operations on a thread (`threadKill`, `threadInterrupt`, `threadRestart`, `threadDeactivate`, and `threadReturn`) are deferred.

Why would a thread need protection? Keep in mind that ScriptX allows a developer to create new tools, called intermediate objects, and add them to an existing title. Imagine a tool that must operate in an environment that knows nothing about it, or a title that does not know what tools are in use. Threads can protect themselves from actions, whether unfriendly or inadvertent, that could have destructive consequences. For example, a thread can protect itself to guarantee that data structures are left in a consistent state when a title closes.

When a thread is protected, certain operations are deferred until the thread becomes unprotected. If one of these critical operations has been deferred, it will be stored in the thread's `pendingAction` instance variable. Query `pendingAction` for debugging, but do not assume that it will remain the same in future versions of ScriptX.

Use the methods `threadProtect` and `threadUnProtect` much as you would used `threadCriticalUp` and `threadCriticalDown` to bracket and protect critical segments of code.

```
threadProtect
-- do something meaningful here
threadUnProtect
-- now let preemption occur
```

The following example illustrates a strategy for detecting and reporting potentially harmful function and method calls by another thread. The function `criticalProcess`, which might be the controlling function of a thread, runs in an infinite loop. Each time the loop is repeated, it checks for pending actions that might destroy its stack, and then reports before yielding. This function will run to completion of the loop each time it runs.

```
function criticalProcess theParam -> ( -- only for debugging!
   guard -- in case there is an exception reported
      repeat while true do (
         case (theRunningThread.pendingAction) of
            @killed: -- report thread was killed by another
            @restart: -- report thread restarted by another
            @done: -- report thread was forced to finish
            @inactive: -- report thread was deactivated
            otherwise:
               -- body of the function goes here
               continue -- start repeat loop over
         end -- case
         threadUnProtect theRunningThread; threadYield()
      )
   catching
      all: -- report interrupt or exception here
   end -- end of guarded code
)
```

The entire body of the function `criticalProcess` is enclosed in a `guard` construction. If another thread calls `threadInterrupt` on this thread, which reports an exception, the `catching` construct allows the function to respond. When the function jumps to this section, it retains the values of variables from where it jumped off. For more information on exception handling, see the *ScriptX Language Guide*.

A thread's protection level is stored in its `protection` instance variable as a non-negative integer. A call to `threadProtect` increases the protection level by 1, and can be balanced by a call to `threadUnProtect`. If there is a pending action in the sample script above, the thread will continue to run, executing the repeat loop, until its protection level reaches 0.

ScriptX permits a script to change the protection level of any thread. As a rule, a script should only change the protection level of a running thread—that is, a thread should manage its own protection level. To do otherwise is an invitation for data corruption, deadlock, and disaster.

Using the Threads Component

Threads can be created by calling the `new` method within a script. As part of the `new` method, you pass in the function the thread executes and other information, including stack size, a priority, and an optional label. Among the

ScriptX core classes, threads are unusual for the number of keyword arguments that can be set on instantiation. For convenience, the global function `callInThread` creates a `Thread` object with default parameters.

```
-- create a thread (priority @user) to compute primes to 1000000
myThread := callInThread computePrimesTo 1000000 @user
```

Only three arguments are required: the thread's function, the function's argument, and the thread's priority with the scheduler. With its simple syntax, `callInThread` will be the technique of choice for many developers. A script still needs to call the `new` method to set more keyword arguments on instantiation of a thread.

ScriptX provides a shorthand construct for creating threads, similar to that in the Unix shell languages. The ampersand character (&) at the end of a line of code spawns the expression that preceded it as a separate thread.

```
myThread := computePrimesTo 1000000 &
```

A thread that is created in this manner has a default priority of `@user`, but its priority, and all of its other instance variables, can be reset by script. The thread operator has the lowest precedence of any operator in the ScriptX language other than the assignment operator.

Thread Examples

The following examples demonstrate the ScriptX Threads component.

Asynchronous Processing

Threads allow for a very interactive style of programming. A thread is a process that can be dynamically altered by a script, even while it is active. The script can kill or restart a thread, modify its instance variables, or force it to wait on a gate. In ScriptX, a developer can create a new thread and activate it without modifying any existing threads. Of course, when a new thread is added to a system, it uses a share of the available processing cycles. This may modify the behavior of a multi-threaded program in ways that are hard to predict.

In a single-threaded environment, a program is deterministic and outcomes are in some sense repeatable. Even a random number generator creates a seemingly random but repeatable sequence of numbers from a seed. In a multi-threaded environment, programming takes on a new dimension, with outcome dependent on timing and interaction.

In the following code example, two threads write to a pipe, while a third thread reads from the pipe. To run this program, enter it into the ScriptX Listener, select it, and evaluate it. Output appears in the ScriptX Listener window. In this example, two threads compete to write to the same pipe,

creating a race condition. The output of this script depends on which thread runs first, on whether the reading thread is able to clear the pipe between writes, and on the size of the pipe itself. The dynamic interaction of processes is always an issue in multi-threaded programming.

```
object myWindow (Window)
    boundary:(new Rect x2:300 y2:50), fill:whiteBrush
    settings x:20, y:40
end
show myWindow
-- create text string global
object myTextBox (TextEdit)
    boundary:(myWindow.boundary), target:("musician" as Text)
    fill:whiteBrush, stroke:blackBrush
end
setDefaultAttr myTextBox @alignment @center
append myWindow myTextBox
show myWindow

-- now set up the threads
function classical thePipe -> (
    local composerList := #("Beethoven","Bach","Mozart","Haydn",
        "Stravinsky","Mahler","Debussy","Ravel","Sibelius","Lutoslawski")
    repeat while true do (
        write thePipe (getAny composerList)
        threadYield()
    )
)

function heavyMetal thePipe -> (
    local heavyMetalList := #("Aerosmith","Bon Jovi","Motley Crue",
        "Megadeath","Twisted Sister","Judas Priest","AC/DC","Metallica")
    repeat while true do (
        write thePipe (getAny heavyMetalList)
        threadYield()
    )
)

function listenTo thePipe -> -- this is the reading thread
    repeat while true do \
        myTextBox.target := (read thePipe) as Text
object myPipe (PipeClass) maxSize:3 end

-- create the three threads.  They will start active.
global classic := new Thread func:classical arg:myPipe priority:@user
global metal := new Thread func:heavyMetal arg:myPipe priority:@user
global listen := new Thread func:listenTo arg:myPipe priority:@system
```

A script can change thread properties dynamically. Shift the relative priorities of these threads, and the output of the program changes accordingly. The example above gives user priority to the writing threads and system priority to the reading thread. The thread `listen` receives many more turns from the

scheduler, so it quickly processes any objects that are placed in the pipe. If relative priorities were reversed, the writing threads would tend to block while waiting for the reading thread to empty the pipe.

ScriptX is an interactive programming environment. In the example above, the threads `classic`, `metal`, and `listen` run independently in the background. All three thread functions run in infinite loops. From the ScriptX Browser, they can be suspended by calling `threadDeactivate`, reactivated by called `threadActivate`, or terminated by calling `threadKill`.

```
classic.priority := @system
threadKill metal -- not good programming practice, but very tempting
```

A developer can create a new thread dynamically and make it interact with threads that are already active. For example, the following thread can be created and started up from the Listener while the three existing threads continue to process in the background.

```
function playJazz thePipe -> (
       local composerList := #("Armstrong","Ellington","Parker","Hawkins",
           "Young","Monk","Mingus","Coltrane","Davis","Carla Bley")
       repeat while true do (
           write thePipe (getAny composerList)
           threadYield()
           )
)
playJazz:= new Thread func:jazz arg:myPipe priority:@user
```

A good way to appreciate the limitations of programming in a multi-threaded environment is to create enough threads to thrash the system. Use the examples above as a basis for adding more active threads: country, rap, salsa, klezmer, gospel, blues, soul, reggae, folk, punk, elevator, etc. As you add additional threads, performance will gradually decline. Since the Listener runs in its own thread, you will notice a slow degradation in Listener response times. Your system will thrash much more quickly if you add system priority threads. System priority threads quickly crowd out essential system services, such as garbage collection.

A Thread Dispatcher

Operations that require many CPU cycles, such as rendering a screen or searching a database, should be managed in a separate thread.

The following code sample creates a `Dispatcher` class that manages a thread and an associated condition. It can be used to activate other threads. We use this approach because threads and gates are sealed classes in ScriptX. Note that both the thread and its associated condition are instance variables of the class `Dispatcher`.

`Dispatcher` has an instance variable that is a `SortedKeyedArray`, one of the classes in the Collections component. This dispatch list will store a list of actions and associated function calls as key-value pairs.

```
class Dispatcher (RootObject)
instance variables
    dispatchList -- SortedKeyedArray
    dThread -- a thread for responding to user actions
    dCond -- a condition that prevents us from trying to
    -- respond to a new action until the old one is dispatched
    action -- name of action we are currently responding to
    args -- list of arguments that goes with action
instance methods
    method init self #rest args -> (
        self.dispatchList := new SortedKeyedArray
        self.dCond := new Condition label:self
        self.dThread := new Thread \
            func:dThreadFn arg:self priority:@system
        self.action := undefined
        self.args := undefined
        apply nextMethod self args
    )
    method pleaseDo self act argument -> (
        if (hasKey self.dispatchList act) then (
            self.action := act
            self.args := argument
            relinquish self.dCond -- open the gate
        ) else (
            format debug "action not defined here" @user
        )
    )
end -- (Dispatcher class definition)
```

When a `Dispatcher` object is created, the thread `dThread` and the condition `dCond` are also instantiated. The thread will call the function `dThreadFn`, which is defined below, each time it runs. Note that the thread blocks until it acquires the condition `dispatcher.dCond`. This condition is opened when the method `pleaseDo` on is called on a `Dispatcher` object. Since `pleaseDo` sets a value for the `Dispatcher` object's `action` instance variable, the `else` block of expressions in `dThreadFn` is finally executed.

```
-- create the control function for Dispatcher.dThread
function dThreadFn dispatcher -> (
    repeat while true do (
        if dispatcher.action == undefined then (
            -- thread blocks here until the flag is acquired
            acquire dispatcher.dCond -- block, don't poll!
            -- a condition activates any waiting threads and
            -- is immediately closed again.
            continue -- now execute the loop again
        ) else (
            -- work with a local copy and set action to undefined
```

```
            local actionCopy := dispatcher.action
            dispatcher.action := undefined
            if hasKey dispatcher.dispatchList actionCopy then
                apply (dispatcher.dispatchList[actionCopy]) dispatcher.args
            else
                format debug "undefined action" @user
        )
    )
)
```

The script creates a `Dispatcher` object. When `Dispatcher` is instantiated, the thread `dThread` and its associated condition `dCond` are also instantiated. This thread will be activated when another thread calls `pleaseDo`, specifying the action and its arguments. Since it runs briefly and blocks, it is a suitable use of the `@system` priority level.

```
global gDispatcher := new Dispatcher -- create an instance
-- the init method on Dispatcher automatically creates a thread
-- define the functions that will be called
global fn func1 x -> format debug "Function 1 with %*\n" x @user
global fn func2 x -> format debug "Function 2 with %*\n" x @user
global fn func3 x -> format debug "Function 3 with %*\n" x @user
-- now add key-value pairs, addMany is a method on Collection
addMany gDispatcher.dispatchList #(@act1:func1, @act2:func2, @act3:func3)
pleaseDo gDispatcher @act1 ("Larry" as String)
pleaseDo gDispatcher @act2 ("Curly" as String)
pleaseDo gDispatcher @act3 ("Moe" as String)
```

Think of the `Dispatcher` class as a container for a thread, an associated gate, and a list of actions. It allows one thread to process many different actions, perhaps in response to user input. The thread waits for an action that it knows how to respond to. The method `pleaseDo` with its associated argument acts like a handler. The `Dispatcher` class provides a mechanism so that this handler can be processed in the background.

Note that `Dispatcher.dThread` blocks if one of the functions it dispatches blocks. This could block other functions that a `Dispatcher` instance might want to respond to. The solution is to have the `Dispatcher` class itself activate other threads. Presumably, these threads could be at user priority levels, while `Dispatcher.dThread` runs at system priority.

For a script that builds logically on the `Dispatcher` class, see the section “Processing with an Event Queue” on page 533 of Chapter 18, “Events and Input Devices.” This sample script creates the `EventDispatcher` class, similar to `Dispatcher`, except that it receives events through an event queue. `EventQueue` is a specialized subclass of `PipeClass`.

CHAPTER

Object System Kernel

23

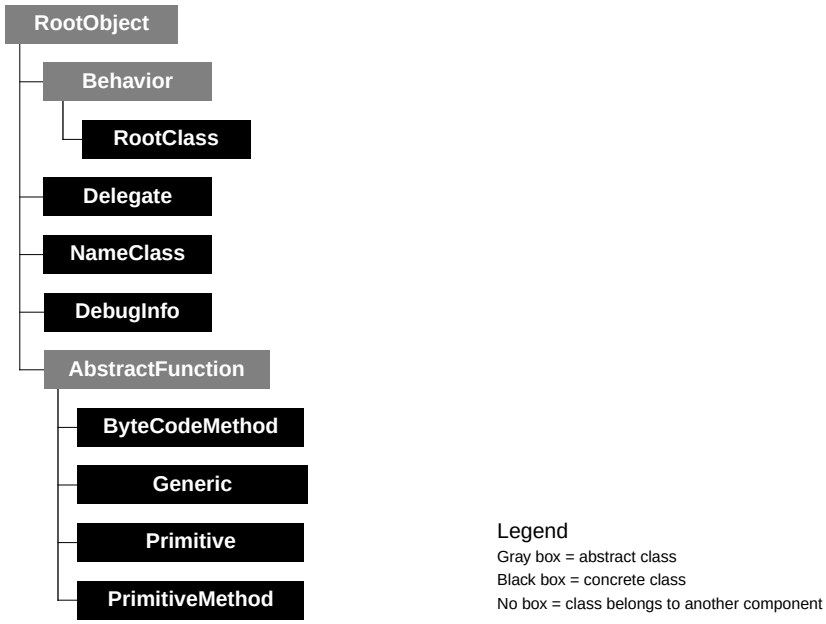
The Object System Kernel is a set of system services that are used by all ScriptX objects. Some of the services this kernel provides are implemented through classes and objects, while others are implemented in the substrate, and hidden from the scripter.

The Object System Kernel is the combination of three parts:

- A set of system classes that define the root behavior of all classes and objects in ScriptX. These system classes include the metaclasses, which allow for the "object" behavior of classes themselves.
- A set of auxiliary and language classes that support the behavior and naming of ScriptX objects.
- A set of system services that is provided for all ScriptX objects, such as object store, garbage collection, and the interface to the scripting language. Many of these system services, though aspects of the "kernel" architecture, are documented elsewhere.

Classes and Inheritance

The class inheritance hierarchy for the Object System Kernel is shown in the following figure.



The following classes form the Object System Kernel component. In this list, indentation indicates inheritance.

`RootObject` – contains methods common to all objects, and the method `new` for creating new objects.

`Behavior` – provides methods common to all classes and metaclasses.

`RootClass` – contains methods common to all metaclasses, and the method `new` for creating new classes.

`Delegate` – a class that redirects method calls to other objects.

`NameClass` – represents names, which are generally interned in the system name table.

`DebugInfo` – provides a template for storing debugging information.

`AbstractFunction` – the root class from which all functions inherit.

`ByteCodeMethod` – An object that represents a function or method that is implemented in the scripter.

`Generic` – An object that represents a generic function, a function that is associated with a particular class or object.

`Primitive` – A function that implements a global function, defined either in the substrate or in the External ScriptX API.

`PrimitiveMethod` – A function that implements a method for a generic function, defined either in the substrate or in the External ScriptX API.

Note – The classes belonging to the metaclass network, except `RootObject`, `Behavior`, and `RootClass`, are not documented in the class descriptions. They operate transparently to someone scripting in ScriptX. These include `MetaClass` and `metaMetaClass`.

Conceptual Overview

Many fundamental features of ScriptX classes and objects are determined by inheritance. The focus of this chapter is on those aspects of ScriptX that are defined by inheritance through the root system classes and the metaclass network. Some portions of this chapter are essential to understanding ScriptX—especially the discussions of the Initialization protocol and of instance variable access. Other topics are advanced or specialized.

How Classes and Objects Work

The root system classes—`RootObject`, `Behavior`, and `RootClass`—define protocols that every class and object in the system inherits, and sometimes specializes. These protocols include initialization, coercion, copying, comparison, and printing to a debugging stream. The metaclasses, which inherit from `Behavior`, allow every class to behave as an object, so that class variables and methods are really instance variables and methods on class

objects. The function classes represent both scripted functions and functions that are defined in the substrate. Access to instance variables, which can only contain other objects, is through accessor functions.

Metaclass Network Introduction

The metaclass network is transparent to authors creating titles, and makes for advanced reading. It is the basis for defining class behavior. It is not necessary to understand the metaclass network in order to create and use class methods.

The ScriptX substrate was developed using Objects-in-C (OIC), a set of extensions to ANSI C that support object-oriented development. Key features of OIC include class variables and methods, global polymorphism, multiple inheritance, and classes as first-class objects. The ScriptX object model is implemented through a metaclass network similar to that of CLOS and Smalltalk.

The metaclass network consists of the classes named `RootClass`, `Behavior`, `RootObject`, and other metaclasses.

Figure 23-1 shows how the `RootObject`, `Behavior`, and `RootClass` classes belong to the metaclass network. When you create a class named `MyClass`, a corresponding metaclass named `metaMyClass` is automatically created by the system. Every class in the class hierarchy has a metaclass. Thus, classes come in pairs: `Event` and `metaEvent`, `QueuedEvent` and `metaQueuedEvent`, `MouseUpEvent` and `metaMouseUpEvent`.

Every class is the sole instance of its metaclass. Every metaclass is an instance of `MetaClass`. The metaclasses, together with the classes `RootClass`, `Behavior`, and `RootObject`, make up the metaclass network.

Metaclass Network Details

The OIC model supports class variables and class methods. This is implemented by using a metaclass scheme similar to that employed in Smalltalk™. Each class is itself an instance of a class, its so-called metaclass, and the class methods are defined on that metaclass. The class methods are in fact instance methods for metaclass instances.

In the OIC system, metaclasses are created automatically whenever a class is created. So that each class can have its own specific class methods, each class is an instance of a separate metaclass. There is one metaclass for each class, and each class is the only instance of this metaclass.

All the metaclasses are themselves instances of another class, namely the distinguished class `MetaClass`, which is an instance of `metaMetaClass`, which is itself an instance of `MetaClass`, thus ending the regression.

In summary, a class is an instance of a metaclass, a metaclass is an instance of `MetaClass`, and `MetaClass` is an instance of `metaMetaClass`.

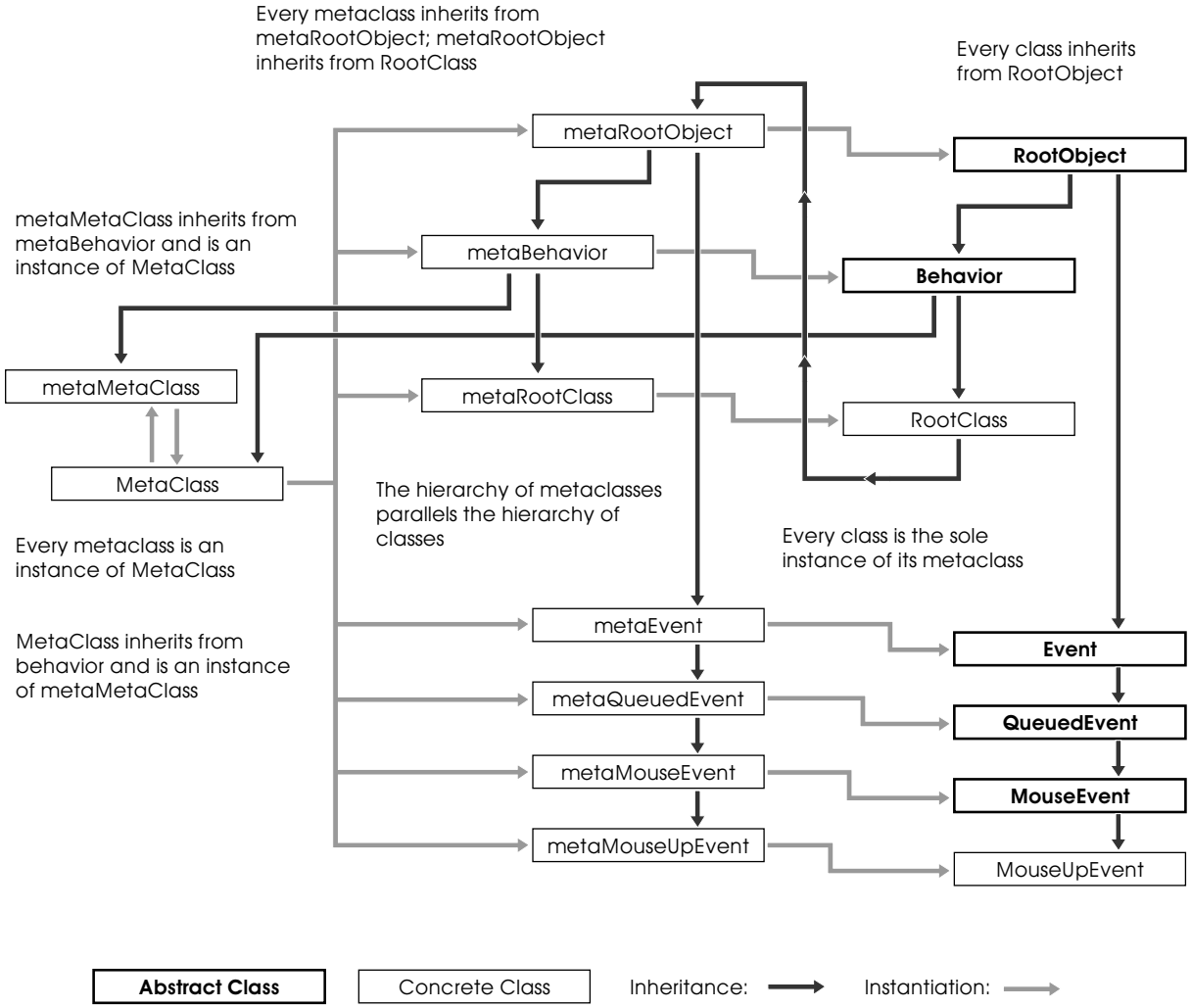


Figure 23-1: The complete metaclass network for MouseUpEvent.

Although we want different classes to have different class methods, we also want to be able factor class methods using inheritance. This is achieved by having each metaclass of a class inherit from the metaclasses of the classes that the class inherits from, forming two parallel chains of inheritance: class to class and metaclass to metaclass. The root superclass for all metaclasses is the distinguished class `RootClass`. So, near the end of the inheritance chain, the metaclass of `RootObject` inherits from `RootClass`.

By convention, the distinguished class named `RootClass` represents all classes. Its class method `new` is used to create new classes.

To be consistent, metaclasses have the same class behavior as classes. So, this common behavior is factored out into the `Behavior` class which is the superclass of both `RootClass` and `MetaClass`. The default class methods, such as `new`, `getSupers`, `getSubs`, and so forth, are implemented as instance

methods by Behavior. Finally, Behavior inherits from RootObject because all classes and metaclasses are objects and share the common behavior of all objects.

If you create a new class called SpecialClass, using the class definition construct in the ScriptX language, the compiler generates a call to the new method in RootClass. This also creates the metaclass, metaSpecialClass, of which SpecialClass is the only instance. Creating this instance of metaSpecialClass invokes the new method in Behavior.

Keep in mind that a class is itself an object, the sole instance of its metaclass. The new method for creating instances of a class is defined by Behavior. When you create a new class, the init method on RootClass is called. It first creates a metaclass, which is an instance of MetaClass, and then creates the class itself, the sole instance of that metaclass.

Quite a substantial part of the OIC system is implemented by the methods in these classes. This is not only a pleasing reflection on the expressivity of the system (it being able to implement large parts of itself in itself), but it also means that the advanced user can specialize the basic OIC model for his own requirements using standard OOP techniques. There is nothing to prevent the development of specializations of these kernel classes that implement, say, different inheritance semantics or object layout for some tuned purpose and having all the schemes coexist in one system.

Initialization

The Initialization protocol comprises three generic functions—new, init, and afterInit. Of these three generics, only new is ever called directly. The generic function new is defined by Behavior as a class method, and can be called on any concrete class in the system. Programmers can create most objects by calling new directly, supplying the appropriate keyword arguments, but many objects are generated automatically, or through syntactic shortcuts in the ScriptX language. The object definition expression provides an alternative syntax for calling new.

As a rule, you specialize initialization behavior by specializing init and afterInit. Never call init or afterInit directly from the scripter, since they are called automatically by new. Many of the core classes define an init method, which initializes the class's internal structures. Scripted classes can override init to supply keyword arguments. Relatively few of the core classes specialize afterInit. Scripted classes should use afterInit rather than init to set the values of class and instance variables, or to add objects to a collection. afterInit is called only after the object itself has been created and initialized.

The *ScriptX Class Reference* demonstrates, for each concrete class, how to create (instantiate) and initialize instances of that class under the heading "Creating and Initializing a New Instance." Creation and initialization 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` for post-initialization. (Note that few core classes have an implementation for the `afterInit` method.)

Creating an Instance of a Class

To create an instance of any concrete class, call the class method `new`, specifying the class name as the first parameter, followed by the keyword arguments from the `init` and `afterInit` methods (described in the next section). You must include required keywords. (In the *ScriptX Class Reference*, optional keywords are indicated with brackets.) You can enter keyword arguments in any order. The `new` method does not process keyword arguments itself—it passes them along automatically to `init` and `afterInit`.

The `new` method creates an empty instance of the class, and then it initializes the instance automatically by calling `init` and `afterInit` with the keyword arguments you supply. The `new` method returns the initialized instance.

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

```
myBitmap := new Bitmap \
       colormap:(new Colormap) \
       data:myStream \
       bBox:(new Rect x2:100 y2:100)
```

The resulting variable, called `myBitmap`, refers to the initialized `Bitmap` object. Its `colormap` instance variable is set to an instance of `Colormap`, its `data` instance variable is set to `myStream`, and the size of the bitmap is set to 100 x 100 pixels.

Initialization Syntax

As stated previously, the `new` method passes its arguments along for `init` and `afterInit` to use. Since most of the core classes do not implement `afterInit`, the syntax for `new` and `init` is usually identical. For example, the syntax for the `init` method of `Bitmap` is shown in Figure 23-2. Note that required keyword arguments in Figure 23-2 don't have default values, since the script must always provide them.

`init` *self* [`data:`*byteString*] [`colormap:`*colormap*] `bBox:`*rect* ⇨ *self*

self	`Bitmap` instance
`data:`	`ByteString` object containing pixel data
`colormap:`	`Colormap` object representing pixel depth

The `Stencil` class uses the following keyword:

`bBox:`	`Rect` object representing height and width of the bitmap

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, and `bBox` sets the area of the bitmap.

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

`data:`storage allocated based on `bBox` size and `colorMap`
`colormap:(new Colormap)`

Figure 23-2: Initialization syntax for a `Bitmap` object

Both the `init` and `afterInit` methods can take two different kinds of arguments: positional arguments and keyword arguments. The initial argument *self* is a positional argument, so it takes no keyword. Other arguments to `init` are all keyword arguments, each in the form of a keyword-value pair, as shown in Figure 23-3.

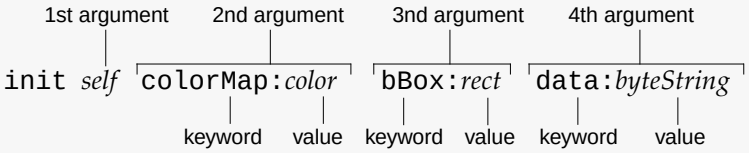


Figure 23-3: The calling sequence for an `init` method

How the new Method Works

When you call `new` on a class, the `new` class method creates an instance of that class, then calls `init` and `afterInit` on that instance, as shown in Figure 23-4.

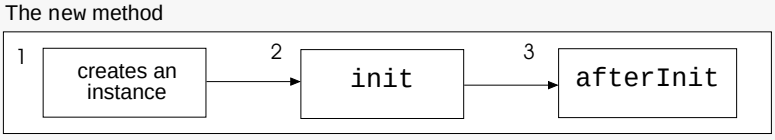


Figure 23-4: The three phases of initialization.

4. Calling `new` on a concrete class creates an instance of that class, allocating the appropriate amount of memory for the specified class.

5. The `new` class method calls the `init` method on that instance, which initializes variables for the instance and invokes any `init` methods that are defined by its superclasses.

6. Once initialization is complete, the `new` class method calls `afterInit` on the instance. The `afterInit` method performs any post-initialization work, such as building internal structures that depend on settings derived from `init`, or assigning additional user-supplied values to the initialized instance. Most of the ScriptX core classes do not specialize `afterInit`, but `afterInit` is commonly specialized in scripted classes.

Note 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 keyword arguments. These arguments are ignored by `new`, 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 insure that inherited structures are properly initialized.

Notes on init and afterInit

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 – Do not call `init` or `afterInit` directly on any instance; these initialization methods are called by automatically `new`. They are visible in the scripter so that they can be specialized.

For most of the core classes, the `afterInit` method is empty and performs no function. The `init` method performs the standard initialization tasks of applying values to instance variables and setting other private internal states.

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

Overriding Initialization in a Subclass

Note – Do not modify any methods in any of the core classes directly, including `new`, `init` or `afterInit`. Specializing any of the generics in the initialization protocol directly can override important internal initialization. Create a subclass and then override the method in that subclass. You can add new methods to a core class that do not override existing methods; however, instance- or class-level specialization is a safer strategy, given that multiple titles often run concurrently.

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.

A scripted class should override the `new` method only in very special cases; for example, to limit the number of instances created. In the ScriptX substrate, the `MouseDevice` class overrides `new`. In a system that has only one hardware mouse, when two ScriptX titles that are running concurrently ask for a mouse, they should both get the same device. When the second title calls `new`, the method does not create a new `MouseDevice` object—instead it returns the device already created.

You would override `init` to change how values are assigned to the new instance through keyword arguments. Most other tasks should be done at post-initialization time, by the `afterInit` method.

If you override `new`, `init`, or `afterInit` 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:

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

For more information on the Initialization protocol, and for examples of how to specialize `init` and `afterInit`, see the discussion of `init` and `afterInit` in the *ScriptX Language Guide*.

Function Dispatch

ScriptX functions, including global and generic functions, are objects that can be saved, stored, bound to names, and assigned as values. The function classes are sealed, and are generally created through syntactic constructs in the ScriptX language. The underlying implementation of functions and methods is designed to be transparent to the ScriptX programmer. However, it is useful to have some familiarity with function dispatch.

All ScriptX functions are instances of one of the concrete subclasses of `AbstractFunction`. In the *ScriptX Class Reference*, many methods and variables return a function. Officially, the ScriptX Engineering Team does not

specify which type of function is returned by any of these methods and variables, and developers should be aware that the ScriptX function classes are subject to future changes. However, developers can assume that all functions will always inherit from `AbstractFunction`. The following test can be used in type verification.

```
isAKindOf draw AbstractFunction -- draw is a generic function
⇨  true
isAKindOf ueq AbstractFunction -- ueq is a global function
⇨  true
```

Global functions that are defined in the substrate are instances of `Primitive`. Global functions defined in the scripter are instances of `ByteCodeMethod`, as are local functions, anonymous functions, and closures. In all respects, global functions behave identically, whether they are defined in the substrate or in the scripter.

ScriptX methods are implementations of a generic function for a particular class or object. There is a one-to-many relationship between generic functions and the methods that implement them for a particular class or object. For example, `draw` is a generic function which many ScriptX classes, including `TwoDShape`, `TextPresenter`, and `ScrollBar` define methods for.

ScriptX has many implementations of the `draw` method, but only a single generic function named `draw`. Each `TwoDPresenter` class defines or inherits a `draw` method. These methods share a single name binding for the ScriptX name `draw`, which is associated with a `Generic` object.

In the scripter, the lexical name `draw` is bound to a `Generic` object. (The name `draw` is exported by the `ScriptX` module, but is defined in the `Substrate` module.) However, the actual `draw` method, as it is defined by `TextPresenter`, is a `PrimitiveMethod` object. `PrimitiveMethod`, like `Generic`, is a class that inherits from `AbstractFunction`. There is no way to define a `PrimitiveMethod` object from the scripter. (The External ScriptX API, documented in the *ScriptX Tools Guide*, allows a programmer to define both `Primitive` and `PrimitiveMethod` objects in C.)

```
getClass draw
⇨  Generic
methodBinding TextPresenter draw
⇨  #<PrimitiveMethod for draw on TextPresenter>
```

When the compositor calls `draw`, it supplies the object to be drawn as the first argument. ScriptX maintains a generic function dispatch table. The bytecode compiler looks in this table to find the particular implementation of `draw` that is associated with the object. Use of a dispatch table is generally faster than traversing a tree (the technique used in some other object-oriented languages), and ScriptX optimizes this dispatch mechanism by caching recent method calls.

Although `draw` is a behavior that is associated with presenters, it is possible to define a `draw` method for any unsealed class. When a new method is defined, it is associated automatically with a `Generic` object. If no `Generic` object with the given name exists in the current module, a new one is created

automatically. This association of methods with `Generic` objects explains why any new `draw` method, implemented by any class or object in any module that uses the `ScriptX` module, must be called with the same "signature" of three arguments:

- `draw` *self surface clip*

Suppose we specialize `TextPresenter` by implementing a specialized version of `draw`. The scripted method cannot be a `PrimitiveMethod` object, since primitive methods cannot be defined in the scripter. The `draw` method, as implemented by the specialized subclass, is also a function, but it is an instance of `ByteCodeMethod`.

```
-- create a specialized class
class SpecialTextPresenter (TextPresenter) end
-- specialize the draw method in some trivial manner
method draw self {class SpecialTextPresenter} surface clip -> (
    nextMethod self surface clip
    print "I just drew myself"
)
methodBinding SpecialTextPresenter draw
⇨  #<ByteCodeMethod anonymous of 3 arguments for draw on \
    SpecialTextPresenter>
```

In ScriptX, scripted global functions, anonymous functions, closures, and methods all compile to `ByteCodeMethod` objects. Thus, a call to a generic function (a `Generic` object) is redirected to an instance of `PrimitiveMethod` or `ByteCodeMethod`, depending on how the `draw` method is implemented by a particular class or object.

The following generic function, which specializes `Behavior`, determines whether any class implements a given method in the substrate, or as a scripted method. Since it is defined by `Behavior` in this example, `definesScriptedMethod` could be called as a class method on any class.

```
method definesScriptedMethod self {class Behavior} meth ->
    if getClass (methodBinding self meth) = ByteCodeMethod then
        true
    else
        false

definesScriptedMethod SpecialTextPresenter draw
⇨  true
definesScriptedMethod SpecialTextPresenter inside
⇨  false
```

Using the generic function `methodBinding`, it is possible to get a binding to a superclass's implementation of a generic function. The following example demonstrates how to override the generic dispatch mechanism. In this example, `primitivePrin` is bound to the version of `prin` that is defined by `RootObject`, rather than the specialization of `prin` that is normally called on a `Text` object.

```
global myText := "ping" as Text
⇨  "ping" as Text
prin myText @normal debug
⇨  "ping" as TextOK
-- now get a binding to prin, as defined by RootObject
global primitivePrin := methodBinding RootObject prin
⇨  #<PrimitiveMethod for prin on RootObject>
primitivePrin myText @normal debug
⇨  Text@0x14ff894OK
```

This example is provided only to illustrate the generic dispatch mechanism. This technique might be used to reveal more about the design of a library of scripted classes. In special cases, it might be used to bypass the normal dispatch mechanism, perhaps for efficiency, within a library of scripted classes. (ScriptX maintains a cache of recent generic dispatch calls—so that in effect, ScriptX already does this internally.) Developers who use this technique in the core classes are living dangerously. Subverting the normal dispatch mechanism can easily cause a system crash, or other unintended results.

Note – Many undocumented features of functions are visible in the scripter. Developers should always assume that undocumented features and interfaces are subject to change in future versions of ScriptX.

Redefinition of Global and Generic Functions

Any ScriptX function can be assigned to a variable. Within the core classes, functions are often assigned to instance variables so that they can be called automatically, in response to some event or change in state. Examples include the `valueAction` of a `ScrollBar` object, the `activateAction` of a `PushButton` object, the `script` of a `Callback` object, and the `eventReceiver` of an `Event` object.

Although any function can be assigned to a variable, global and generic functions differ in what happens to an existing assignment when a function is redefined. The following example illustrates this. First, we define both a global and a generic function, and set up callbacks to each.

```
-- create a global function
function sayCheese x y ->
    print "The best North American cheese comes from Wisconsin"

-- create the cheese class and define a method
class Cheese () end
method frommage self {class Cheese} x ->
    print "The best European cheese comes from France"

-- create an instance of Cheese to call frommage on
object myCheese (Cheese) end

-- now, set up two callbacks
addTimeCallback theCalendarClock sayCheese undefined \
    undefined (theCalendarClock.time + 5) true
```

```
addTimeCallback theCalendarClock frommage myCheese \
    undefined (theCalendarClock.time + 5) true
```

The callbacks are invoked, as expected, after five seconds.

```
"The best North American cheese comes from Wisconsin"
"The best European cheese comes from France"
```

Now, suppose we set up two more callbacks. This time, we leave a few extra seconds to redefine both the global function `sayCheese` and the `Cheese` class's `frommage` method.

```
addTimeCallback theCalendarClock sayCheese undefined \
    undefined (theCalendarClock.time + 60) true
addTimeCallback theCalendarClock frommage myCheese \
    undefined (theCalendarClock.time + 60) true
-- redefine both the global function and the method
function sayCheese x y ->
    print "The best North American cheese comes from Ontario"
method frommage self {class Cheese} x ->
    print "The best European cheese comes from Denmark"
```

The callbacks are invoked, as expected, after 60 seconds. The second callback invokes the redefined version of the `frommage` method. But the first callback still invokes the original version of the global function `sayCheese`.

```
"The best North American cheese comes from Wisconsin"
"The best European cheese comes from Denmark"
```

The generic dispatch mechanism provides, in effect, an extra layer of indirection. Thus, a method can be redefined on the fly—the callback invokes the new version of the method. The global function cannot be redefined in the same manner. The callback's script continues to invoke the old version of the function.

This difference in function dispatch has implications for many classes. A useful workaround, so that a global function can be redefined like a method, is to use an anonymous function as a "wrapper" function. In effect, this adds an extra layer of indirection.

```
addTimeCallback theCalendarClock (a b -> sayCheese a b) undefined \
    undefined (theCalendarClock.time + 30) true
function sayCheese x y ->
    print "Wisconsin still makes the best cheese in North America"
```

With the use of an anonymous function, `sayCheese` is called indirectly. The new and redefined version of the global function is called.

```
"Wisconsin still makes the best cheese in North America"
```

Access to Variables

ScriptX provides access to instance and class variables through getter and setter methods, a design feature that is modeled on the object-oriented language Dylan. A class can be specialized by overriding its getter and setter methods. Two types of instance and class variables are visible to the scripter. Real instance variables are allocated a "slot," a physical memory location within the contiguous memory that is allocated to an object on the ScriptX heap. Virtual instance variables do not have such a slot. To the scripter, they appear just like real variables, but they are calculated on the fly.

In notation, the two types of variables are identical. They use either language notation for instance variable access, or the getter and setter generics. Those instance variables that are read-only have only a "getter" method, while those that are also writable have both "getter" and "setter" methods. When you retrieve a value, you actually call its getter method. Assign a new value, and you call the setter method. Both types of variables are set and retrieved through getter and setter methods. You need to be concerned about how instance variables are implemented only when you are defining or specializing instance and class variables. For example, if were saving an object, you might want to save objects that it contains references to as well.

Note that the generic functions `ivNames` and `ivTypes` return lists of real instance variables—virtual instance variables are excluded. Similarly, `getDirectMethods`, `getAllMethods`, `getDirectGenerics`, and `getAllGenerics` return lists of methods or generics defined by a class, including the getters and setters that are used for access to virtual instance variables.

Using Getters and Setters

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

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

You can also call the equivalent methods:

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

The two styles have identical access times—neither one is more direct or faster than the other. Choose one style or the other based on which syntax you prefer. The ScriptX Engineering Team has adopted the instance variable style rather than the method style.

Note – In the *ScriptX Class Reference* and in other volumes of the ScriptX Technical Reference Series, we document instance variables, but not their setters or getters. Assume that any instance variable has an "undocumented" getter method and setter method (if writable), which you are free to use. The names of these accessor methods are always the instance variable name, with `Getter` or `Setter` added as a suffix.

When you want an instance variable to behave differently, such as to perform some other action whenever you set its value, you can override its 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 specialize the method there.)

A "slot" is a particular location in memory where the value of the instance variable is stored. ScriptX does not provide direct access to instance variable slots in the scripter; you have access to slots only through accessor methods.

Some instance variables store values in slots while others do not. For example, the instance `valueEqualComparator`, defined by `Collection`, has no slot. Instance variables without slots call functions to set or get the values from the environment. The instance variable `presentedBy`, defined by `Presenter`, is an example of a variable with an actual slot to hold its value.

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 instance variable `includesLower` is read-only for the immutable class `NumberRange`, but if you create a mutable subclass of `NumberRange`, you can implement an `includesLowerSetter` method to make it writable.

Note – 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 or unpredictable behavior.

Delegation

In object-oriented programming, it is a common technique for a class or object to "delegate" some action to another object. Delegation is often the key to writing code that is general, and can be applied to many objects. With dynamic binding, multiple inheritance, generic function dispatch, and accessor methods, ScriptX is an ideal system for implementing delegation.

Delegation is built into certain ScriptX classes. The most widely used delegation protocol in ScriptX is the `IndirectCollection` class. `IndirectCollection` is often used as a mix-in class, although it is not strictly a mix-in class, to add collection properties and behavior to other objects. An indirect collection redirects each method call to its target collection. By specifying that an object inherits from `IndirectCollection`, rather than a specific collection class, a developer can write code for a collection without

being concerned with the physical implementation of the collection. In this way, a program could determine, even at runtime, whether a given data structure should be an array or a linked list.

`CostumedPresenter` implements simple delegation for presenters. A costumed presenter has a target presenter, to which its `target` instance variable stores a reference. By using a costumed presenter, a program can implement general connections with a presenter, and determine the appearance of that presenter dynamically, at runtime. The document template classes could also be used to implement delegation. Each template is a presenter that can be used to present or lay out other presenters. `OneOfNPresenter` is yet another presenter class that can be used to implement delgation. `OneOfNPresenter` embodies a one-to-many relationship in which a single presenter from a collection of presenters is the target. For information on `CostumedPresenter` and `OneOfNPresenter`, see Chapter 3, "Spaces and Presenters." For information on document templates, see Chapter 13, "Document Templates."

The `Delegate` class can be used to implement delegation in any unsealed class. `Delegate` really does not belong to any ScriptX component; it is more a utility class than a part of the Object System Kernel. `Delegate` defines a single instance variable, `delegate`, that specifies an object to which method calls are directed. `Delegate` is a concrete class, but it is often specialized or mixed in with other classes to add delegation properties to a scripted class. `Delegate` provides a simple mechanism for redirecting calls at runtime.

Copying Objects

The Copy protocol comprises two generics, `copy` and `initCopy`, that a class can implement to make itself copyable. A script makes a copy of an object by calling the `copy` generic function on that object. `RootObject` defines the default implementation of `copy`, which most classes do not override.

In a sense, the Copy protocol is analogous to `new` and `init`. Just as you create a new object by calling `new` rather than `init`, you copy an object by calling `copy` rather than `initCopy`. A script calls `init` only indirectly, by calling `new`. In the same way, a script calls `initCopy` only indirectly, since `copy` calls `initCopy` automatically, if it is defined by the class.

You specialize copying by specializing `initCopy`. Although you are allowed to, a script generally does not specialize the `copy` method itself. (Among the core classes, the one exception is `NameClass`. For more information, see the definition of `NameClass` in the *ScriptX Class Reference*.)

The default version of `copy`, defined by `RootObject`, creates a new instance of the target class. It then calls `initCopy` on itself, and on each of its superclasses, in turn. If any of these superclasses is a scripted class that does not define an `initCopy` method, it invokes the default action for a scripted class, which is to make a shallow copy of any instance variables that class defines. If any of the superclasses is a member of the core classes, it looks for an `initCopy` method. If that class is not copyable, it reports an exception.

Most core classes that implement the Copy protocol create a "shallow" copy of themselves. A shallow copy is a copy of the class's internal structures, but not of member elements or embedded objects. For example, to copy a mouse event

does not mean to make a new copy of the device that the event is associated with. The initialized copy has its own pointer to a mouse device, but it points to the same device as the original. The device itself does not get copied.

The following example demonstrates the difference between a shallow copy and an assignment. In the first example, `myArrayCopy` is a copy of `myArray`. Since it is a shallow copy, `newText` has its own internal structures. Initially, it points to the same data.

```
global myArray := #("dog", "hog", "frog", "bog")
⇨  #("dog", "hog", "frog", "bog")
-- assign this array to anotherArray, which does not create a new array
global anotherArray := myArray
⇨  #("dog", "hog", "frog", "bog")
-- create a copy of myArray; this creates a new array
global myArrayCopy := copy myArray
⇨  #("dog", "hog", "frog", "bog")
```

While `myArray` and `anotherArray` are the same object, `myArray` and `myArrayCopy` are not. Think of `anotherArray` as just another name binding for the `Array` object that the name `myArray` is bound to.

```
myArray == newArray
⇨  true
myArray == myArrayCopy
⇨  false
```

Whereas `anotherArray` is just a new name for `myArray`, `myArrayCopy` is actually a separate object that shares the same member elements. We see this when we change one element of `myArray`.

```
myArray[3] := "toad"
⇨  "toad"
newArray
⇨  #("dog", "hog", "toad", "bog")
myArrayCopy
⇨  #("dog", "hog", "frog", "bog")
```

A class can implement the Copy protocol only if all of its superclasses in the core classes define an `initCopy` method. The following script prints a list of classes that implement the `initCopy` method:

```
global myArray := new Array
for i in (getSubs RootObject) \
       select i into myArray if canClassDo i initCopy
for i in (myArray as SortedArray) do print i
⇨  Array
   ArrayList
   Btree
   . . .
```

```
SortedKeyedArray
String
Triple
```

For brevity, the middle of this list has been omitted. If you try this yourself, you will find that `initCopy` is defined by most collections, events, and ranges. Other substrate classes report an exception if you attempt to copy them.

In this script, `canClassDo` is an instance method defined by `Behavior`, and is thus a class method for all ScriptX classes. The script asks each class that is a subclass of `RootObject`, that is, all the classes in the core classes, whether it implements `initCopy`. The script then prints out a sorted list of classes.

You can write your own `initCopy` method for a scripted class. By default, scripted classes are copyable. ScriptX makes a shallow copy of each instance variable that is defined by a scripted class. You can specialize this behavior. Since ScriptX automatically calls `initCopy` on all superclasses, you should not include a `nextMethod` call in your own `initCopy` method.

Most of the core classes do not implement `initCopy`. Use the generic function `canClassDo` to determine if a class is copyable.

Creating a Deep Copy

In most cases, you want to create only a shallow copy when you copy an object. A deep copy requires additional memory and processing. Two shallow copies of the same object share internal structures, until one of the copies changes as new values are assigned.

You might want a deep copy of an object that has other objects embedded within it. For example, suppose you wanted to create a class of pushbutton that is copyable. Although instances of `PushButton` are not copyable, a subclass of `PushButton` could be made copyable by defining an `initCopy` method for that class.

Suppose you have created a subclass of `PushButton`, the `SpecialButton` class. Like `PushButton`, `SpecialButton` defines three subpresenters, stored in the instance variables `pressedPresenter`, `releasedPresenter`, and `disabledPresenter`. In shallow copy of a `SpecialButton` object, these three instance variables store handles to the same three presenters as the original object. Of course, since the copy is a shallow copy, when the value of `fill` or `stroke` changes on one of its subpresenters, the change affects both the copy and the original object. To make a deep copy of a `SpecialButton` object, specialize `initCopy` to assigned completely new structures when the object is copied.

Coercing Objects

Object coercion is the conversion of an object from one class to another. The Coercion protocol comprises two generics, `morph` and `newFrom`, that classes can specialize to provide for coercion through the `as` operator in the ScriptX language and the `coerce` function in the substrate.

When an object is coerced to another class, the original object remains in existence. ScriptX creates a new instance of the target class and uses the original object to set the contents of the new instance. Suppose that *x* is the name of some object and *y* is the name of a target class. The following expressions are equivalent in ScriptX.

```
x as y -- ScriptX language, equivalent to the function coerce
coerce x y -- global function
```

The first expression, which uses the ScriptX operator `as`, is translated by the compiler into the second expression. The global function `coerce`, which is defined in the substrate, is equivalent to the following:

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

The function `coerce` first calls the generic function `morph`. If there is no implementation of `morph` on the original class or one of its parents, or if the `morph` method that is available does not know about class *y*, then `morph` reports the `cantCoerce` exception. The `coerce` function catches this exception and attempts to coerce by calling `newFrom`, a generic that can be defined by the class.

This mechanism allows any scripted classes to specialize coercion from any existing class in the core classes without modifying the existing class. By defining a `newFrom` class method, a scripted class can allow for the coercion of objects that are instances of the core classes.

This means that a developer has two mechanisms for converting an object from one class to another. Although the `morph` generic function is the normal mechanism, the `newFrom` class method provides a work-around for certain cases where it is not practical to specialize `morph`. (A script should normally cast an object from one class to another using the `as` operator or the `coerce` function, but it is possible to call the underlying generics directly.)

The following example defines the `PolarPoint` class, which implements both `morph` and `newFrom`. Polar coordinates are used to solve certain problems in analytical geometry and complex analysis—in computer science, they are especially useful in graphics. In the polar coordinate system, position is expressed in terms of radius (*r*), or distance from the origin, and orientation (*theta*), or angle from the y axis. Orientation is expressed in radians.

In two-dimensional computer graphics, it is conventional to show points on a Cartesian surface, with the origin in the upper left corner. (This differs from the standard mathematics textbook treatment only in the placement of the origin.) We can use the `Point` class, one of the core classes, to store Cartesian coordinates.

```
object myPoint (Point) x:6, y:6 end
```

A `PolarPoint` object represents an ordered pair of polar coordinates. Figure 23-5 depicts both Cartesian and polar coordinates for `myPoint`.

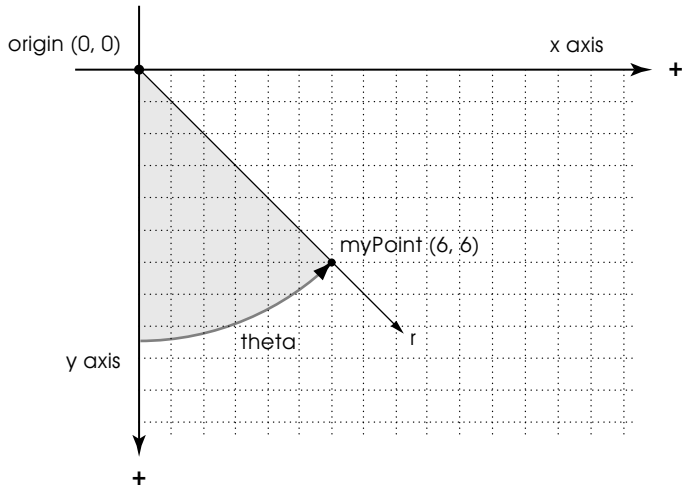


Figure 23-5: Cartesian and polar coordinate systems

We want to be able to freely coerce objects from `PolarPoint` to `Point`, and vice versa. By defining a `newFrom` class method for the `PolarPoint` class, we can accomplish this without specializing the `Point` class, which is one of the core classes. Here is the definition of the `PolarPoint` class:

```
class PolarPoint ()
    instance variables
        r, _theta
    instance methods
    method init self #rest args #key r: theta: -> (
        if not isAKindOf r Number do
            report invalidNumber r
        if r < 0 do
            report badParameter \
                #(r, init, self, "r must be > 0")
        if not isAKindOf theta Number do
            report invalidNumber theta
        apply nextMethod self args
    )
    method afterInit self #rest args #key r: theta: -> (
        apply nextMethod self args
        self.r := r
        self.theta := theta
    )
    method thetaGetter self -> self._theta
    -- normalizes theta between 0 and (2 * pi)
    method thetaSetter self value -> (
        local constant twoPI := (2 * pi)
        if value >= twoPI then
            self._theta := mod value twoPI
        else if value < 0 then
            self._theta := mod (twoPI + value) twoPI
        else
            self._theta := value
    )
end
```

The `PolarPoint` class performs the appropriate range and type checking. It checks that values supplied for `r` and `theta` are instances of `Number`, and that `r >= 0`, reporting the appropriate exceptions if necessary. This validation of arguments is performed in the `init` method, while the assignment of values to instance variables is performed after initialization, in the `afterInit` method. The value of `theta` is normalized internally so that `0 ≤ theta < (2 * pi)`, and the original value is not saved.

To implement the Coercion protocol, define the class method `newFrom` and specialize the instance method `morph`. Note that this class method, defined using the free method syntax, compiles to create an instance method on `metaPolarPoint`, the `PolarPoint` class's metaclass.

```
class method newFrom self {class PolarPoint} other -> (
    if (isAKindOf other Point) then
        new self r:(sqrt (other.x * other.x + other.y * other.y)) \
            theta:(atan (other.y/other.x))
    else
        report cantCoerce #(other, self)
)

method morph self {class PolarPoint} cls style -> (
    if not isAKindOf cls Behavior do
        report badParameter \
            #(cls, morph, PolarPoint, "Expecting name of Class")
    if (isSub cls String) then
        "(" + (self.r as String) + " x " + (self.theta as String) + ")"
    else if (isSub cls Point) then
        new cls x:(self.r * (sin self.theta)) \
            y:(self.r * (cos self.theta))
    else
        report cantCoerce #(self, cls)
)
```

To test the `PolarPoint` class, create a `PolarPoint` object that has the same position or value in a two-dimensional coordinate system as `myPoint` (defined on page 629), only expressed in polar coordinates.

```
object myPolarPoint (PolarPoint) r:(6 * sqrt 2), theta:(pi/4) end
-- check that conversions results are correct
myPolarPoint.r
⇨  8.48528137423857
myPolarPoint.theta
⇨  0.7853981633974483
global convertedPoint := myPoint as PolarPoint
convertedPoint.r = (6 * sqrt 2)
⇨  true
convertedPoint.theta = (pi / 4)
⇨  true
```

As objects are coerced from `Point` to `PolarPoint` and back, `morph` is invoked when we coerce a `PolarPoint` object to a `Point` object, and `newFrom` when we reverse the operation.

PolarPoint also provides a conversion to String. Of course, this conversion depends on the conversion of numbers to strings, which is defined by the String class in the core classes.

```
myPolarPoint as String
⇨  "(8.48528137423857 x 0.7853981633974483)"
```

Note – The following rules apply to specializing morph and defining a newFrom class method. If morph succeeds in coercing an object to the target class, it should not call nextMethod. If morph fails to coerce, it should either call nextMethod, which allows one of its superclasses to handle the coercion, or report the cantCoerce exception. The newFrom class method should never call nextMethod. If newFrom fails to convert the object to the target class, it should report the cantCoerce exception. This insures that if both a morph instance method and a newFrom class method exist for a given class, the morph method takes precedence over newFrom.

Coercion to a Superclass or Subclass

In the Coercion protocol, a special case arises when the original object belongs to a subclass of the result class. If you attempt to coerce an object to a superclass of its class, and the superclass is concrete, nothing happens. This is because the original object, as an instance of a subclass, is already an instance of all of its concrete superclasses. For example, a StringConstant object is also a String object, while an instance of RoundedRect is considered to be an instance of Rect.

```
greeting := "Hello" -- creates a StringConstant object
coerce greeting String -- coerce it to String
getClass greeting
⇨ StringConstant -- still an instance of StringConstant
```

If the original class is a superclass of the result class, the result of coerce depends on the implementation of morph on the original class, just as it does for any two ScriptX classes. Thus, the String class specializes morph so that strings can be coerced freely between the three string classes: String, StringConstant, and Text. For this reason, an instance of String can readily be coerced to an instance of StringConstant, but an instance of EventQueue cannot be coerced to an instance of EventDispatchQueue.

Suppose a developer creates a scripted subclass of StringConstant called NameConstant. She wants to be able to coerce an instance of String to an instance of NameConstant. To do this, she overrides the morph method on String to allow for NameConstant as a special case, and then calls apply nextMethod to allow the String class to handle other cases.

Coercion and Numerics

An object represents information about some entity, both its properties and its behavior. To coerce an object to another class is to represent information about that entity through another template or model. Coercion may involve loss of information or precision. For example, the `Number` classes can readily be coerced, say from `ImmediateInteger` to `Float`, or vice versa, but there is loss of precision with certain operations.

```
3 as Float
⇨  3.0
pi as ImmediateInteger -- Hoosier pi
⇨  3
```

The usual precautions about floating point precision and integer truncation apply to operations in ScriptX. For example, the value of `pi` cannot be simplified, despite the good intentions of the Indiana state legislature, which once debated a member's proposal to set the value of `pi` to 3. As an alternative to coercion, it may be efficient to define a method that represents the value of the object in a different way:

```
trunc pi
⇨  3
```

Note that ScriptX runs on a range of system software and hardware platforms, including some without floating point processors. Internally, these platforms may store floating point or extended precision values in different ways. After a long series of type casts and computations, slight differences in precision may be noticeable. This is not really a coercion issue *per se*—it is an issue of precision and magnitudes that applies globally to any floating point computations in ScriptX.

Comparing Objects

The Comparison protocol, a set of generic functions defined by `RootObject`, is the foundation for the comparison of all objects in ScriptX. For certain classes, it makes sense to compare and order instances. The basis for ordering differs by class. For example, strings can be ordered lexicographically, integers or floats can be ordered numerically, and linear collections can be compared element by element.

Basic comparison operations are all defined in terms of the three generic functions in the Comparison protocol. Developers implement the Comparison protocol for a scripted class by specializing methods for these generics, by indicating how instances are ordered. Override these methods as necessary, and all of the other comparison functions will operate correctly for a scripted class.

The first of these generics is `isComparable`, which indicates whether it is correct to directly compare two objects. By default, *x* and *y* are comparable if `(getClass x == getClass y)`, but some classes override this. For example, the subclasses of `Number` are comparable, as are the subclasses of `String` and `Sequence`.

```
isComparable Array LinkedList
⇨  true
```

If two objects pass the test of comparability, then the generics `localLt` and `localEqual` can be applied, where `localLt` is the less-than comparison and `localEqual` is the equality comparison. The following script prints a list of classes that implement a method for `localLT`, meaning that instances of these classes can be compared with comparable classes:

```
global myArray := #()
forEach (allInstances Behavior) \
   (a b -> if (canClassDo a localLT) do (append b a)) myArray
myArray | print
```

The global function `eq`, which is not part of the Comparison protocol, tests whether *x* and *y* are exactly the same object. Two objects are the same object if they share the same master pointer—that is, they share a handle to the same object in the ScriptX heap. Immediate objects are an exception. 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 on immediate objects, see “Immediate Objects” on page 486 in Chapter 17, “Numerics.”

Defining Global Functions

Each comparison operator has a global function that is its counterpart. From the four generics of the Comparison protocol—`isComparable`, `localLt`, `localEqual`, and `eq`—all other comparison operations are defined as global functions. For example, the equality functions `equal` and `nequal`, defined in the substrate, could be defined as follows in ScriptX. (See the “Global Functions” chapter of the *ScriptX Class Reference* for the equivalent ScriptX definition of each of the comparison functions.)

```
function equal x y -> (isComparable x y) and (localEqual x y)
function nequal x y -> not (equal x y)
```

Comparison functions are defined as global functions, and are documented in the “Global Functions” chapter of the *ScriptX Class Reference*. See that source for an equivalent ScriptX definition of each of the comparison functions. To make instances of a given class comparable, you need only specialize the generic functions of the Comparison protocol. Objects cannot be compared unless they implement this protocol, and they can only be compared with other objects to which they are comparable.

Some ScriptX operations require objects that implement the Comparison protocol. For example, only objects that are comparable with one another can be stored in an ordered collection such as a sorted array or a B-tree. (A developer might specialize `BTree`, `SortedArray`, or `SortedKeyedArray`, to detect whether objects that are added to the collection are comparable.)

The following example demonstrates how to extend the Comparison protocol for scripted classes. In a previous section of this chapter ("Coercing Objects" on page 628), we defined the `PolarPoint` class, a scripted class that implements a polar coordinate system. The core classes define the `Point` class, which implements a point in a Cartesian coordinate space. The substrate does not implement the Comparison protocol for `Point`.

Suppose you want to compare `Point` and `PolarPoint` objects, both with themselves and with each other. To make comparisons in both directions, the program must specialize `isComparable`, `localEqual`, and `localLT` in both classes. First, we implement `isComparable`, `localEqual`, and `localLT` on the `PolarPoint` class:

```
method isComparable self {class PolarPoint} other -> (
    if isAKindOf other Point or isAKindOf other PolarPoint then
        true
    else
        false
)

method localEqual self {class PolarPoint} other -> (
    local comparator
    if getClass other = PolarPoint then
        comparator := other
    else if isAKindOf other Point then
        comparator := other as PolarPoint
    else -- pass it on to RootObject, which reports exception
        nextMethod self other
    if self.r = comparator.r and self.theta = comparator.theta then
        return true
    else
        return false
)
```

There are several possible ways to order a set of points. Any function that produces a unique ordering, so that the comparison operators obey the associative and commutative laws, can be used as an ordering function. If the only purpose of ordering is to store objects in some kind of ordered collection, for speedy retrieval, a lexicographic ordering function is a speedy solution. This version of `localLT` sorts first on the basis of distance from the origin. The final `else` clause is called only if the two objects are equal distances from the origin.

```
method localLt self {class PolarPoint} other -> (
    local comparator := undefined
    if getClass other = PolarPoint then -- the same class
        comparator := other
    else if isAKindOf other Point then -- a comparable class
        comparator := other as PolarPoint
    else
```

```
        report generalError \
            "Cannot compare PolarPoint and %* objects." \
            (getClassName other)
    -- now do the actual comparison
    if self.r < comparator.r then
        return true
    else if self.r > comparator.r then
        return false
    else (
        if self.theta < comparator.theta then
            return true
        else
            return false
    )
)
```

ScriptX allows for the specialization of existing methods in the core classes, but doing so is an invitation for collisions when multiple storage containers and name spaces are open in a single session. Developers should regard the ability to specialize the core classes in this manner as a convenience for debugging and analysis, not as a means of adding new features to titles. For this reason, we do not specialize the `Point` class here.

Instead, we define the `CartesianPoint` class, a subclass of `Point`, in which to implement the Comparison protocol. This example extends the Comparison protocol in both directions between `CartesianPoint` and `PolarPoint`. `CartesianPoint` specializes both `isComparable` and `localEqual`, calling the superclass's version for cases it cannot resolve.

```
class CartesianPoint (Point) end

method isComparable self {class CartesianPoint} other -> (
    if isAKindOf other Point or isAKindOf other PolarPoint then
        true
    else
        nextMethod self other
)

method localEqual self {class CartesianPoint} other -> (
    if getClass other = PolarPoint then
        localEqual other self
    else if isAKindOf other Point then
        if self.x = other.x and self.y = other.y then
            true
        else
            false
    else
        nextMethod self other
)
```

`CartesianPoint` also specializes `localLT`. In this particular implementation, the `localLT` method simply converts objects to `PolarPoint` objects and calls on the `localLT` method that is defined by `PolarPoint`. Of course, `localLT` could be implemented any number of ways, depending on the requirements of the program. Unlike `localEqual`, `localLT` does not call `nextMethod` for cases it cannot handle. Instead it reports an exception.

```
method localLt self {class CartesianPoint} other -> (
    local comparator := undefined
    local cls := getClass other
    if cls = PolarPoint then
        localLt other self
    else if isAKindOf other Point then
        comparator := other as PolarPoint
    else
        report generalError "Cannot compare CartesianPoint and %*." cls
    if self.r < comparator.r then
        return true
    else if self.r > comparator.r then
        return false
    else (
        if self.theta < comparator.theta then
            return true
        else
            return false
    )
)
```

ScriptX makes it possible to choose any ordering function, and to switch functions dynamically at runtime, simply by redefining `localLT`. In the case of points, there are several valid approaches. Points could also be ordered based upon a primary axis in the Cartesian coordinate space, such as the *x* or *y* axis. Another possible ordering function would treat each `CartesianPoint` or `PolarPoint` object as a sum of vectors, and sort them based on the scalar product of those vectors.

Specializing localEqual or localLT

Of the three generics in the Comparison protocol, `RootObject` actually defines a method for only two. Since `RootObject` does not implement `localLT`, `CartesianPoint` and `PolarPoint` do not call `nextMethod` for cases they cannot handle—instead, both classes report an exception.

All ScriptX global functions call `isComparable`, either directly or indirectly, before any call that results in a call to `localEqual` or `localLT`. The generic function `isComparable` qualifies a call to `localEqual` or `localLT`. Thus, there is no reason, within the body of a method for `localEqual` or `localLT`, to call `isComparable`. (Scripter-defined global comparison functions should follow this same prototype.) Within the body of `localEqual` or `localLT` methods, it should be assumed that the two objects are known to be comparable.

Global comparison functions may call other comparison functions, but ultimately, all comparison functions must be defined in terms of the three generics in the Comparison protocol, avoiding any circular definitions and dependencies. A program can also declare new global comparison functions, making them apply automatically to any existing classes that implement the Comparison protocol.

Except within a comparison function, scripts should not call the three generics of the Comparison protocol directly (other than for testing and debugging). They should be called indirectly through global functions or through comparison operators in the ScriptX language, which compile into calls to global functions. The generic functions `canClassDo` and `canObjectDo` are useful for testing whether a class or object implements `localLT`.

Global functions can be defined either in the scripter, so that they compile to `ByteCodeMethod` objects, or using the Extending ScriptX API, so that they compile to `Primitive` objects. The latter are indistinguishable from global functions defined in the substrate in any modules in which they are visible.

For more information on comparison operators in the ScriptX language, see the *ScriptX Language Guide*. For information on comparison functions and ordered collections, see "Sorted Collections" on page 449 and "Comparison Functions" on page 450 in Chapter 16, "Collections." See also the discussions of `BTree`, `SortedKeyedArray`, and `SortedArray` in that chapter, and the definitions of those collection classes in the *ScriptX Class Reference*.

Printing Objects

This section demonstrates the Printing protocol, which allows ScriptX to print information about objects to an output stream, such as the `debug` stream. The Printing protocol comprises two generic functions, `prin` and `recurPrin`, that are used to define all global printing functions. (The Printing protocol is unrelated to the Printing component, a loadable extension to ScriptX. The Printing component, which provides printing services for ScriptX titles and accessories, is documented in the "Printing" chapter of this volume.)

Every ScriptX object can print an ASCII representation of itself to an output stream. This stream must be a byte stream, and it must be writable. The generic function `prin`, implemented by `RootObject`, is the basis for printing objects in ScriptX. As implemented by `RootObject`, `prin` indicates an object's class and the location of its master pointer in memory. Any unsealed class or object can implement a specialized version of `prin`. The usual reason for specializing `prin` is to provide additional debugging information.

The generic function `prin` is the basis for several global printing functions, including `prinln`, `print`, `printString`, and `format`. Most printing is done through one of these functions. By specializing `prin`, a developer can change the way all global printing functions operate on a given class of object. For example, suppose you have defined a scripted class that contains several "member" objects—objects that are assigned to instance variable "slots." You could print an instance of this class by specializing the `prin` method to call `prin` on each of those objects individually. Indeed, that is exactly what the `Point` class does. When you call `prin` on a point, it prints out the values of `x` and `y`.

```
prin (new Point x:6 y:6) @normal debug
⇨  [6, 6] as PointOK
```

Of course, `prin` does little to improve the readability of the stream it prints to. As this example shows, it does not insert an end-of-line character before the return value of the expression, which is the `OK` object. Most actual printing is through global functions, which provide a more readable representation as well as a simpler calling sequence.

The core classes specialize `prin` with several different printing modes. The four standard modes are `@normal`, `@complete`, `@unadorned`, and `@debug`. The `PolarPoint` class, defined in previous sections of this chapter, is similar to `Point`, so a developer might want the debugging stream to yield similar information about a `PolarPoint` object. The following example implements `prin` for `PolarPoint` objects, using the `prin` method that is defined in the substrate by `Point` as a model:

```
method prin self {class PolarPoint} formatStyle printStream -> (
    case formatStyle of
        @normal:(
            format printStream "[%n1, %n2] as PolarPoint" \
                #(self.r, self.theta)
        )
        @complete:(
            format printStream "[r=%c1, theta=%c2] as PolarPoint" \
                #(self.r, self.theta)
        )
        @unadorned:(
            format printStream "[%u1, %u2]" #(self.r, self.theta)
        )
        @debug:(
            (methodBinding RootObject prin) self @normal printStream
            writeString printStream ": ["
            writeString printStream (getClassName self.r)
            writeString printStream ": "
            writeString printStream (self.r as String)
            writeString printStream ", "
            writeString printStream (getClassName self.theta)
            writeString printStream ": "
            writeString printStream (self.theta as String)
            writeString printStream "]"
        )
        otherwise: (
            nextMethod self formatStyle printStream
        )
    end
)
```

This implementation of `prin` is quite wasteful of memory, because it adds many `StringConstant` objects to the memory footprint of the `PolarPoint` class. (A simpler version of the `@debug` mode could be defined with a single string using the `format` global function.) For this reason, it is useful to define `prin` in the free method syntax, binding it to the `PolarPoint` class only for authoring, and not when the title is played back in the ScriptX Player.

Printing Recursive Objects

Recursive structures present a special problem. Certain objects, such as linked lists, contain recursive structures, structures that point to an instance of the same class. For recursion, ScriptX defines the generic function `recurPrin`, also

implemented as a method by `RootObject`. `recurPrin` is not meant to be called directly from a script. With most objects, `recurPrin` and `prin` are the same—a call to `prin` calls `recurPrin` by default.

In an object that contains recursive structures, redefine the `prin` method to call the global function `printRecursively`, passing the same arguments that were used to call the `prin` method. The function `printRecursively` then calls `recurPrin`, adding an additional argument to track the recursive print state. Specialize the method `recurPrin` to print a representation of the object to a stream. To print out an object within the `recurPrin` method, call the global function `safeRecurPrin`, passing on the recursive print state as the final argument. This final argument, which is used to resolve references to objects in recursive structures, is of no real consequence to developers.

CHAPTER

24

Exceptions

The Exceptions component provides facilities for reporting and handling exceptions (also known as errors.)

Each kind of exception in ScriptX is represented by a global instance of the class `Exception` or one of its subclasses. To report (or signal) an exception, call the `report` method on the appropriate global exception instance. For example, to report an error during an attempt to divide a number by 0, the report method is called on the `divideByZero` instance.

When an exception is reported, the system checks if it can be caught. If it can be caught, the action that has been specified as the response to such an exception is performed. If it is not caught, the default exception handler takes over, which means that either a fatal error occurs or the thread that caused the error died.

To catch and handle exceptions, you must use the `guard` and `catching` syntax in the ScriptX language. See the *ScriptX Language Guide* for a discussion of catching exceptions.

This chapter briefly describes the Exceptions component. A complete listing of system-defined subclasses and instances of Exception is included in Appendix A of the *ScriptX Class Reference*. For more information on how to write exception-handling code, see the *ScriptX Language Guide*.

Classes and Inheritance

The class inheritance hierarchy for the Exceptions component is shown in the following figure.

The following classes and groups form the Exceptions component. In this list, indentation indicates inheritance. The superclass `Exception` is first described; then the Exception classes and groups are described in detail in alphabetical order.

`Exception` – the superclass of all exceptions.

- `ClockException` – the class of clock exceptions.
- `CollectionException` – the class of collection exceptions.
- `ContainerException` – the class of container exceptions.
- `DevicesException` – the class of devices exceptions.
- `DirRepException` – the class of dir rep exceptions.
- `EventException` – the class of event exceptions.
- `GeneralException`– the class of general exceptions.

`ImportExportError` – the class of exceptions related to importing and exporting.

`LoaderException` – the class of loader exceptions.

`MathException` – the class of math exceptions.

`MemoryException` – the class of memory exceptions.

`ObjStoreException` – the class of object store exceptions.

`PlayerException` – the class of player exceptions.

`ScriptError` – the class of exceptions that can occur during script compilation.

`SpaceException` – the class of space exceptions.

`StreamException` – the class of stream exceptions.

`SystemError` – the class of system exceptions.

`TextException` – the class of text exceptions.

`ThreadException` – the class of thread exceptions.

Note that there are also space exceptions, general errors, and text and font exceptions that do not have their own subclasses.

How Exceptions Work

The system-defined exception instances, such as `notObject`, `noMethod`, `divideByZero` and so on, are created in the ScriptX source code. These exceptions are all listed in Appendix A of the *ScriptX Class Reference*.

Users may also create their own exception classes and instances. To create a new exception instance, call the `new` method on the appropriate class of exceptions.

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.

ThrowTag and ThrowArg Global Variables

The global variable `throwTag` is always bound to the most recently reported exception, and the global variable `throwArg` is bound to the argument that was passed to the report method for that exception.

For example, suppose you attempt to divide 21.3 by 0:

```
b := 21.3 / 0
```

This expression causes the `divideByZero` exception to be reported. The `throwTag` global variable becomes bound to `divideByZero`, and `throwArg` becomes bound to 21.3.

These global variables can be useful if you want to find out information about which exception was most recently reported.

Using Exceptions

To catch exceptions, use the `guard` and `catching` syntax described in the *ScriptX Language Guide*.

To report an exception, call its `report` method and pass arguments appropriate to the exception's format string.

For example, the following statement creates a `cantBePurple` exception as an instance of the user-defined class `ColorException`.

```
new ColorException name: "cantBePurple" \
       format: "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
)
```

Exceptions in the Kaleida Media Player

When a ScriptX title is running in the Kaleida Media Player, uncaught exceptions open a dialog box warning the user of the problem. The user's only option is to to click **OK** to close the dialog box.

Note – In the current release, certain exceptions reported in the ScriptX Listener may cause the Kaleida Media Player to hang without displaying a dialog box.

C H A P T E R

24

Import and Export

The Import and Export component provides classes that allow ScriptX to import and export various types of data, such as audio files, video files and bitmaps. This component includes predefined importers for common data types.

Classes and Inheritance

The class inheritance hierarchy for the Import and Export component is shown in the following figure.

The Import and Export component consists of the following classes.

`Exporter` – converts an internal data object to a known external data format

`Importer` – converts a stream from an external data format to an internal data object

`ImportExportEngine` – provides the `import` and `export` methods that invoke concrete importers and exporters in this component

How Import and Export Work

When the ScriptX system starts up, it creates a single instance of `ImportExportEngine` and stores that object in the global variable `theImportExportEngine`. The global variable `theImportExportEngine` then uses a `Loader` object to load `Importer` and `Exporter` objects into the system. You can create multiple instances of `ImportExportEngine`, but generally need not do so.

When importing external data into ScriptX, `theImportExportEngine` selects the appropriate `Importer` object to convert the data, then returns an internal data object that represents the imported data.

Rather than interacting with `Importer` or `Exporter` objects directly, call the `importMedia` or `exportMedia` methods on `theImportExportEngine` global instance, which in turn calls the appropriate methods on `Importer` or `Exporter`.

Registering Import and Export Modules

During the ScriptX system's startup process, `theImportExportEngine` requests a `Loader` object to load groups of

- import modules – importers that reside in a directory (folder) called `importrs` (notice the "`e`" is omitted to make it 8 characters)
- export modules – no exporters are available in the current release.

See the *ScriptX Tools Guide* for information about importers supplied with this release.

Using the Import and Export Component

See the *ScriptX Tools Guide* for detailed information on the import modules available in ScriptX and how to use them to import media files, complete with examples.

CHAPTER

25 Loader

The Loader component provides facilities for loading modules into the runtime environment to extend and update existing code. The Loader component allows developers to integrate patches to existing code in a seamless fashion and to add C extensions to the substrate.

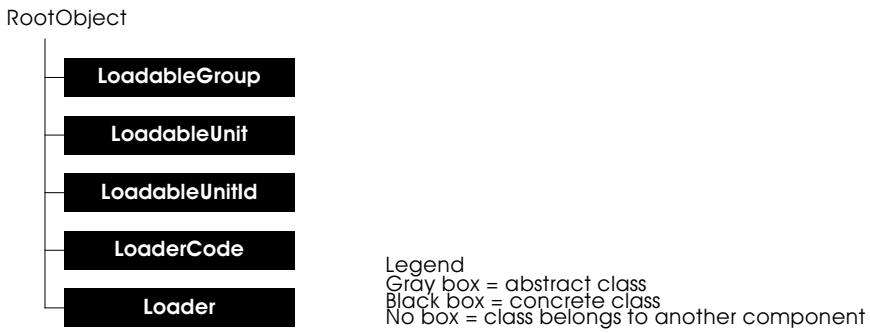
The Loader component links its targets into the current environment transparently, so that users cannot distinguish them from the original system. Users can access the Loader component from ScriptX.

This chapter describes the Loader component and the programmatic tools associated with it. It includes descriptions of the incremental linking and load process as well as instructions for writing code that uses Loader facilities.

Note – A loadable unit can be loaded by a script-level program. However, the loadable unit itself must be written in C. This manual covers material only for the script-level ScriptX programmer. Writing a loadable unit is discussed in the chapter "Extending ScriptX" in the *ScriptX Tools Guide*.

Classes and Inheritance

The class inheritance hierarchy for the Loader component is shown in the following figure.



The following classes form the Loader component.

`LoadableGroup` – Contains a group of loadable units that can be loaded by the Loader component.

`LoadableUnit` – Defines the atomic units for the Loader component, including how to be loaded and version number.

`LoadableUnitId` – A class whose instances are used to identify loadable units. Constants of this class are used to identify errors when a unit can't be loaded.

`Loader` – Maintains the tables of currently loading and successfully loaded units, drives the load and unload operations.

`LoaderCode` – A class whose global constants are used as return values to identify the whether the code in a loadable unit can be used.

Conceptual Overview

The Loader component is responsible for loading and linking the following types of data:

- replacement code, either indirect functions or class/instance methods
- new device drivers and utility functions
- new classes

The runtime environment uses the Loader component for *dynamic loading* to optimize system resources. With dynamic loading, the runtime system loads an object only when it is needed as a resource. When the resource is no longer required, the Loader component unbinds it and frees any associated memory. The result is a runtime environment that optimizes memory usage.

Modules loaded by the Loader component are housed in loadable unit persistent objects, which are created and kept within loadable group persistent objects. These loadable modules can contain any combination of class definitions, class and instance methods, indirect functions, and data. Each class has initialization code that bootstraps the class definitions and registers whatever was loaded into the system.

The focal point for the Loader component is a class called `Loader`. When using the Loader component, the first step is to create an instance of the class `Loader` that manages the loading process.

How the Loader Works

This section describes loadable objects, entry point code, and loader lists.

Loadable Objects

From a high-level standpoint, the Loader component operates on two types of objects:

- A *loadable unit* is the atomic unit of the Loader component. A loadable unit contains a single binary code/data file, a pointer to entry point code, an environment variable, as well as system and unit version identifiers.
- A *loadable group* is a collection of loadable units logically bundled together. A loadable group includes platform-specific device and handle information that uniquely identifies it.

Both loadable units and loadable groups are persistent objects that are available to the system through the Persistent Object System.

The following diagram shows the layout of a loadable group containing three loadable units:

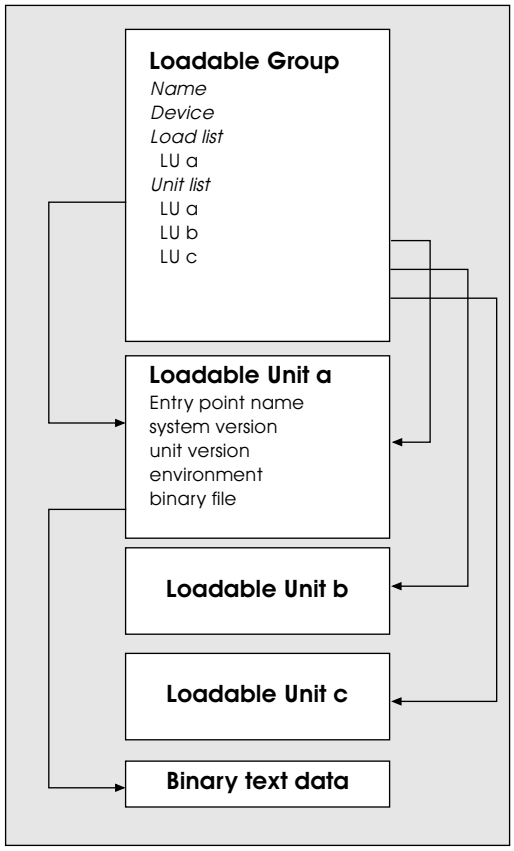


Figure 25-1: Layout of a loadable group.

A loadable group maintains two lists:

- a *load list* specifying which loadable units to load when a group is processed
- a *unit list* of all loadable units in the group

When processing a group, the Loader component loads the units specified by the load list and returns a list of opaque `LoadableUnitID` objects representing the successfully loaded units. The `LoadableUnitID` class is a private class not visible from the ScriptX environment but used internally by the Loader component. `LoadableUnitID` objects contain a unique identifier for each loadable unit, known as an LID, along with other information.

Loadable units are always associated with a group. There are two types of loadable units:

- *Ephemeral units* contain code that is loaded, executed, and then unbound. The ephemeral unit's initialization code does not link its code and data into the system, so the system is unaware of them. An ephemeral unit can call exported code in the system, however.

- *Linkable units* contain the actual target code for a load. Once the Loader component has successfully loaded a linkable unit, its code can be called by other parts of the runtime system. It, in turn, can reference any part of the system that is exported.

Entry Point Code

Both ephemeral and linkable units are associated with *entry point code* that is run after the unit is loaded but before a `LoadUnitID` object is returned. Entry point code can invoke the Loader component recursively to load other units or can set up the environment for linkable units that will be dynamically loaded at some future time.

For a linkable unit, entry point code typically performs the following initialization procedures:

- It invokes the class initialization routines for any classes it creates.
- It registers all new classes and methods with the runtime system.
- If the module contains replacement indirect functions or replacement class/instance methods, the initialization code links these into actual system functions and methods using system utilities.

Ephemeral units normally consist entirely of entry point code. Useful applications for ephemeral units include checking for the presence of certain facilities before loading linkable units.

Loader Lists

The Loader component maintains two internal lists to prevent redundant load operations:

- Whenever a `Loader` object starts the load process on a unit, it places the unit's LID on the *loading list*, a list of all units currently being loaded.
- When a module is successfully loaded, the `Loader` object removes the LID from the loading list and enters it on the *loaded list*, a list of all units successfully loaded and still present in the system.

Before a `Loader` object attempts to load a unit, it first checks the loaded list:

- If the LID of the current unit matches a LID on the loaded list, and the current unit's version number is less than the version number of the loaded unit, the `Loader` object simply returns the current unit's LID.
- If the version of the current unit is greater than or equal to the version of the existing unit, the `Loader` object continues to process the current unit, since it is more recent.

If the unit's LID is not on the loaded list, the `Loader` object checks the LIDs and version numbers on the loading list. If the LID is on the loading list and the version of the matching unit is greater than or equal to the version being loaded, the `Loader` object returns the special code `LoadableUnitIdLoading` to the caller to indicate that the load is in process.

If the LID is not on either list, the `Loader` object loads the specified unit.

Whenever a `Loader` object removes a unit from the system through an `unbind` operation, it deletes its LID from the loaded list.

Using the Loader

Using the Loader component requires only a few steps:

1. First, make an instance of the class `Loader`:

   ```
   deviceLoader := new Loader
   ```

   Multiple instances of the `Loader` class an exist concurrently in the system. Users typically create a private `Loader` instance for each load operation, although entry point code in a linkable unit might use a single `Loader` object recursively in successive load operations.

2. Invoke the `process` method to load all units on a loadable group's load list from the specified media:

   ```
   process deviceLoader deviceName handle
   ```

   This method returns a list of `LoadableUnitID` objects for all units in the group that were loaded.

   Alternatively, you can use the `loadModule` method to load a particular loadable unit from a specified group:

   ```
   loadModule deviceLoader groupInstance unitName
   ```

   The `Loader` method `getGroup` returns a group instance to use in this case. The `loadModule` method returns the `LoadableUnitID` object representing the successfully loaded unit.

How the Loader Works (continued)

This section describes the load process, symbol accessibility, relinquishing, and exception handling.

The Load Process

Processing Groups

The Loader component processes a loadable group by iterating through the group's load list and invoking the `loadModule` method on each loadable unit in the list. The `getgroup` method provides a group instance to use for the process.

Symbol Accessibility

The private class `LoaderHelper` uses the `nameToAddress` instance methods in both `LoaderHelper` and `Loader` classes to resolve references to symbols in the running system so that the newly loaded code can access static symbol definitions. When writing code that uses the Loader component, you can make symbols available to the rest of the run-time environment at load time through any of the following mechanisms:

- When writing the initialization routines to be called by entry point code, use the standard system facilities for linking newly loaded functions and methods into the system:
  - `indirectSet` associates the name of an indirect function with the actual system function definition.
  - `methodFor`, `methods`, and `classMethods` bind an instance or class method to a generic function. Use these methods as you would normally in base system code, such as in an `initClass` method.
- You can export arbitrary symbols through the `Loader` class `export` method.

While these operations are particularly important during entry point initialization, they can be performed at any time in the system, including at system initialization time.

The following table shows what symbols are available to the static system and to units loaded into the static system dynamically:

Symbols	System	Dynamic Module
global (not class or generic function)	Yes	No
exported via `Loader` class	Yes[1]	Yes[1]
available via `indirectSet`, `methodFor`, `methods`, `classMethods`	Yes	Yes
class	Yes	Yes[2]
generic function	Yes	Yes
interned name symbols	Yes	Yes[3]
exception objects	Yes	Yes[4]

1. Available through `Loader` class method `nameToExport` and as a symbol while being loaded.
2. A copy is made of the symbol at the time of the load. Subsequent changes to the original symbol will not be reflected in code that references the symbol. You should export the symbol via the `Loader` class `export` method if you anticipate that its value might change.
3. The Interned name string must be the same as the symbol name, without any leading "SXi".
4. The Exception name string must be the same as the symbol name, without any leading "SXe".

The following table shows what symbols are available within a dynamically loaded unit and to units that are loaded at a later time:

Symbols	System	Current Dynamic Module	Dynamic Module Loaded Later
global (not class or generic function) in current dynamic module	No	Yes	No
exported via `Loader` class	Yes[1]	Yes[1]	Yes[1]
available via `indirectSet`, `methodFor`, `methods`, `classMethods`	Yes	Yes	Yes
class in current dynamic module	No	Yes	Yes[2]
generic function in current dynamic module	No	Yes	Yes
interned name symbol	No	Yes	Yes[3]
exception objects	No	Yes	Yes[4]

1. Available through `Loader` class method `nameToExport` and as a symbol while being loaded.
2. A copy is made of the symbol at the time of the load. Subsequent changes to the original symbol will not be reflected in code that references the symbol. You should export the symbol via the `Loader` class `export` method if you anticipate that its value might change.
3. The Interned name string must be the same as the symbol name, without any leading "SXi".
4. The Exception name string must be the same as the symbol name, without any leading "SXe".

Duplicate Symbols and Shadowing

As the preceding tables indicate, the Loader component cannot relocate certain symbols in dynamically loaded code. However, you can duplicate facilities that do not require sharing, such as string-handling and copy functions, for a loadable unit to use as its private copy. In such cases, the private copy *shadows* the base system version of the facility; that is, the loadable unit invokes the private copies of the functions rather than looking for the functions in the base system. Note, however, that data shared between the base system and the dynamically loaded unit cannot use the private facilities.

Relinquishing

The `Loader` class `releaseLoadableUnit` method removes loadable units from the system. When passed a `LoadableUnitId` object, `releaseLoadableUnit` removes the unit's LID from the loaded list and arranges for the loaded resources (code, data, and any created classes) to be freed when they are no longer referenced. The Loader component works with the system garbage collector to ensure that as long as there are references to classes and/or instances of classes associated with the loadable unit, the class

objects and code will not be freed. If newer loadable units reference classes from a loadable unit that is unbound, the code and classes will not be freed until the newer loadable unit is also freed.

It is not safe to release a loadable unit that contains code not associated with any class unless you are sure that the code or data is not being used by the system. The Loader component does not track the use of normal C functions or data.

The Loader component also does not track references to symbols that have been exported through the `Loader` class `export` method. You must explicitly ensure that you have unexported any symbols from a loadable unit, and that no code is referencing them.

Typically loadable units that are candidates for unbinding simply advertise a specific class (or classes) to the rest of the system. Code that exports symbols or contains C functions used by the system is not usually unbound.

Cleanup During Relinquishing

The Loader component automatically takes care of cleanup for any dynamically loaded class that defines an `unbind` class method. Each loadable unit should include a `relinquish` method that cleans up during the unloading of the module. Note that this `relinquish` method should not do anything detrimental to class operation, since the class, instances of the class, and associated method code remain active in the system until the last instance of the class is disposed.

To do a full cleanup, the class should implement a finalization method, which is invoked at garbage collection time when there are no remaining instances of or references to the class.

Exception Handling

The `Loader` component's exceptions are instances of an `Exception` class called `LoaderException`:

- If a loadable unit reports the `loaderInitError` exception during the running of entry point code, the Loader component automatically unbinds the unit. Usually the `loaderInitError` exception is reported when `assertClass` cannot find an initialized class definition for classes contained in the unit.
- If the Loader component cannot resolve a symbol during relocation, it assigns the symbol the address of `loaderNullFunction`, which reports the `loaderUnresolvedError` exception when any code attempts to call the unresolved function.
- When general errors occur during the process of loading a class, the `cantLoadClass` exception is reported.

Loader Examples

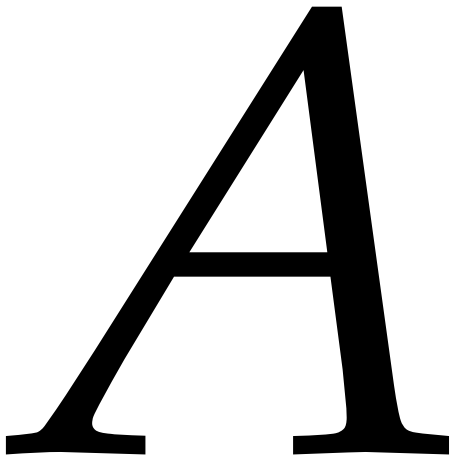
The following ScriptX examples demonstrate how to use the Loader component for the most common tasks: dynamically loading new classes and methods and loading patches to existing code. All of these are ScriptX examples and require no C code.

```
-- The following examples contain two separate LoadableGroups:
--   "PlugIns" is a group of new classes that can be plugged
--        into the system
--   Patches contains replacement code

-- The first example loads the "Foo" class three different ways:
-- directly (Snippet 1), recursively via loading another class
-- (Snippet 2), and by "processing" the group containing "Foo"
-- (Snippet 3). After each load, the example uses the class and
-- then unbinds it before loading it again.

-- The first snippet loads "Foo" twice in a row to show that
-- the Loader component is smart enough to only load it once.
-- Until it is unbound, it will not get loaded again.

--
-- Snippet 1 - load directly
--
fooLoader := new Loader
plugInGroup := getGroup fooLoader "PlugIns"
fooUnit := getLoadableUnit plugInGroup "Foo"
fooId := loadModule fooLoader plugInGroup fooUnit

-- Second loadModule will do nothing, since it's already in the
-- system.
fooId := loadModule fooLoader plugInGroup fooUnit

-- Note: the following code assumes the module containing the
-- Foo class has been imported to this module
aFoo := new Foo
writeString aFoo "This is a test"

-- Nothing will actually be freed until no more refs to
-- Classes loaded ("Foo"), and instances of the classes.
relinquish fooId

-- Still safe ... (See above)
writeString aFoo "This is a test"

--
-- Snippet 2 - load recursively through an ephemeral unit
-- named barUnit
--
barLoader := new Loader
plugInGroup := getGroup barLoader "PlugIns"
barUnit := getLoadableUnit plugInGroup
barId := loadModule barLoader plugInGroup barUnit
fooId := loaderValue barLoader barId
aFoo := new Foo
```

```
writeString aFoo "This is a test"
relinquish fooId

--
-- Snippet 3 - load implicitly via Process method
--
fooLoader := new Loader
idList := process fooLoader "PlugIns"
aFoo := new Foo
writeString aFoo "This is a test"
foreach idList (m -> relinquish m)

-- This example loads the "Replace" unit, which contains
-- a replacement method for the Foo class' writeString method
-- and a replacement indirect function for
-- loaderIndirectFunction. Since the class Foo is dynamically
-- loaded (see above), it might not be in the system when the
-- init code is processed. If the Foo class does not exist,
-- the Loader component won't try to replace the method.

patchLoader := new Loader
patchesGroup := getGroup patchLoader "Patches"
unit := getLoadableUnit "Replace"
id := loadModule patchLoader patchesGroup unit
foo := new Foo
writeString foo "This is a test"

-- Don't relinquish this unit - it will cause problems since it replaces
-- an Indirect function and a method (see "Relinquishing" earlier)
```

APPENDIX

Loadable Extensions



This section contains an overview of the loadable extensions that can be loaded and run in the ScriptX Player. Other extensions that run only in the ScriptX Development Environment, such as the tool framework, are described in the *ScriptX Tools Guide*.

Summary of Loadable Extensions

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 ScriptX 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 this document.

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 this document.

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.

The `QuickTimePlayer` class 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 `VFWPlayer` class works only on a Windows machine with Video For Windows installed. It can play AVI files directly, without having to import them into ScriptX. 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 `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`. 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.

Classes: `QuickTimePlayer`, `VFWPlayer`, `CDPlayer`

For more information: See the Media Player chapter in this document.

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 that follows later in this section.

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 this document.

External Command Interface Extension

This section describes the loadable extension that provides an external command interface to the ScriptX runtime environment. There is one such loadable extension: The MCICMD extension provides access to Multimedia Command Interface commands for Microsoft Windows systems.

Loading the External Command Extension

The External Command Extension is contained in the directory `LOADABLE` in the ScriptX directory. The Microsoft Windows extension is in a subdirectory named `MCICMD`. To load the extension for your platform, start ScriptX, then select **Open Title** from the **File** menu. Select the "loadme.sx" file in the `mcicmd` subdirectory to load the appropriate extension.

MCI Interface

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 MCI interface is provided through a single function, `mciCommand`. The function `mciCommand` loads the MCI command represented by the string *cmdString*. This function is documented in the functions chapter of the *ScriptX Class Reference*.

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 alternate functionality for systems other than Windows.

A P P E N D I X

Glossary

B

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 built-in 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 B-1). 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*.

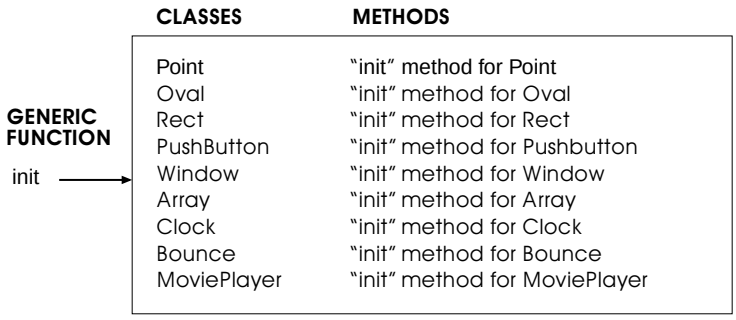


Figure B-1: When the generic function `init` is called on an instance, it calls the `init` method corresponding to that class.

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– A namespace, 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 ScriptX 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

Numerics

A

B

C

D

E

F

G

H

I

M

N

Q

R

S

T

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 • Howard Metzenberg, Douglas Kramer, Jocelyn Becker, Maydene Fisher, Alta Elstad

Illustrations • Graham Metcalfe

Book production • Sandra Ware, Jacki Dudley, Diana Harwood, Beth Delson

This book was created electronically using Adobe FrameMaker on Macintosh Quadra computers.